FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Darstellung inferierter Abhängigkeiten zwischen Java-Paketen

Abschlussarbeit im Studiengang Master of Science im Fach Informatik

vorgelegt am 27.09.2008 von
Sebastian Hauf
Tannenhöhe 1B
85298 Scheyern

Matrikelnummer: 7155832
sebastian.hauf@googlemail.com

Inhaltsangabe

Durch die starke Typisierung von Java ergeben sich Kopplungen zwischen den Klassen, Interfaces und Enums von Java-Projekten: damit eine Quelltextdatei von einem Compiler ohne Fehlermeldung akzeptiert wird, ist im Allgemeinen das Vorhandensein verschiedener anderer notwendig. Weil sie üblicherweise nach thematischen Gesichtspunkten Paketen zugeordnet werden, lassen sich aus diesen Abhängigkeiten zwischen einzelnen, in Java-Dateien deklarierten Typen auch Abhängigkeiten zwischen Paketen herleiten.

Diese Arbeit beschreibt, durch welche Code-Konstrukte verschiedene Arten von Abhängigkeiten zwischen Java-Paketen entstehen und stellt das Eclipse-Plugin Padev vor, das deren grafischer Darstellung dient und auf Basis des Infer-Type-Refactorings Möglichkeiten bereitstellt, Abhängigkeiten umzukehren.

Inhaltsverzeichnis

1 Einleitung

Jedes größere Java-Projekt besteht aus zahlreichen Typen[1], die im Allgemeinen nicht vom Compiler übersetzt werden könnten, würde man sie getrennt von den anderen behandeln. Grund dafür sind Abhängigkeiten von anderen innerhalb oder außerhalb des Projekts deklarierten Typen, die dadurch entstehen, dass z. B. eine Klasse ein Feld besitzt, dessen Typ in einer anderen Compilation Unit (das ist eine Datei, die Java-Quelltext enthält; üblicherweise besitzt ihr Name die Endung .java) deklariert ist. In den Paketen eines Projekts sind thematisch verwandte Typen zusammengefasst, wodurch Abhängigkeiten zwischen Paketen entstehen: Ein Paket `a` ist von einem anderen `b` abhängig, wenn Typen `A` aus `a` und `B` aus `b` existieren und `A` von `B` abhängig ist. Die Betrachtung dieser Paketabhängigkeiten statt derer zwischen einzelnen Typen kann bei Software-Entwurf und -Wartung sinnvoll sein:

- Besonders bei größeren Projekten ist es erstrebenswert, Pakete getrennt voneinander zu entwickeln und zu testen. Ist ein Paket `a` von einem Paket `b` abhängig, muss für Tests von `a` auch `b` vorhanden sein.
- Eine Menge von Paketen, die von Paketen außerhalb der Menge abhängen, kann nicht als getrenntes Projekt veröffentlicht werden. Das verhindert die einfache Wiederverwendbarkeit von Code.
- Abhängigkeiten erschweren die Wartbarkeit von Projekten, da Änderungen an einem Paket Änderungen an anderen, abhängigen Paketen erfordern können.

Natürlich lassen sich Abhängigkeiten nicht auf sinnvolle Weise vollkommen vermeiden. Entscheidend für gutes Softwaredesign ist es allerdings, an welcher Stelle sie auftreten. Hängt ein Paket `a` von einem Paket `b` ab, dann können Änderungen an `b` auch Änderungen an `a` erfordern. Abhängigkeiten zu stabilen Paketen sind daher weniger problematisch als solche zu Paketen, deren Inhalt sich mit größerer Wahrscheinlickeit ändert oder die eventuell komplett ersetzt werden. Betrachtet man die hierarchische Anordnung von Paketen[2], dann erhöht sich deren Instabilität entlang eines Pfades im Hierarchiebaum, wie folgendes Beispiel veranschaulicht:

Abbildung 1 zeigt die Pakethierarchie eines Projektes, das eine graphische Benutzeroberfläche besitzt. Sollen deren Elemente, also z. B. Schaltflächen und Menüeinträge, nicht länger von den in `someProject.ui.graphical.elements` deklarierten Klassen dargestellt werden, weil sie durch die Verwendung ähnlicher Typen aus `java.awt` nicht mehr benötigt

[1] Interfaces, Klassen und Enums definieren Typen (Ausnahme: innere, statische Klassen), weswegen im Folgenden die Bezeichnung „Typ“ verwendet wird, wenn ein Interface (auch die spezielle Form Annotation), eine Klasse oder eine Enum gemeint ist.

[2] Die Hierarchie ergibt sich aus den Paketnamen: beispielsweise ist `org.eclipse.draw2d` ein Unterpaket von `org.eclipse`. Dennoch dient diese Anordnung hauptsächlich der Übersichtlichkeit; für den Compiler spielt es keine Rolle, ob ein Paket Unterpaket eines anderen ist, weil – anders als bei Typen und deren Subtypen – keine Eigenschaften an Unterpakete vererbt werden.

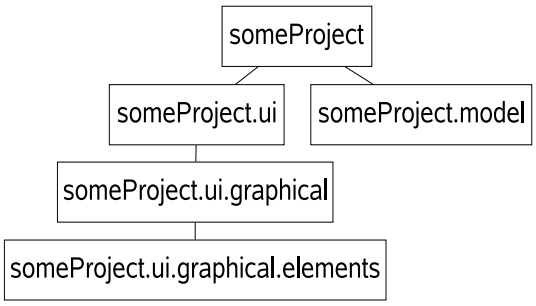

Abbildung 1: Pakethierarchie eines Beispielprojekts.

werden, dann müssen alle von `someProject.ui.graphical.elements` abhhängigen Pakete angepasst werden. Wird die graphische Benutzeroberfläche durch eine textbasierte ersetzt, dann ist nicht nur das Paket `someProject.ui.graphical` überflüssig, sondern auch deren Unterpakete, also wiederum `someProject.ui.graphical.elements`. Ein Paket, das von `someProject.ui.graphical.elements` abhängig ist, muss in beiden Fällen geändert werden; eines, das nur von `someProject.ui.graphical` abhängig ist, nur im zweiten.

Man kann Abhängigkeiten zwischen Paketen als Graph darstellen, wobei die Knoten Pakete repräsentieren und eine gerichtete Kante zwischen zwei Knoten bedeutet, dass zwei Pakete voneinander abhängig sind. Abbildung 2 zeigt ein Beispiel:

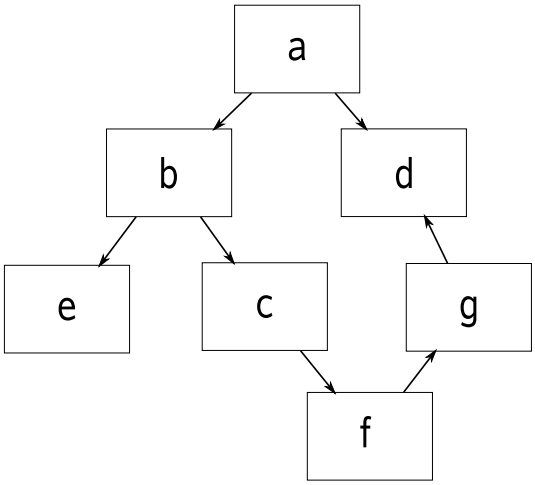

Abbildung 2: Abhängigkeitsgraph ohne Zyklen.

Offensichtlich enthält der Graph keine Zyklen. Ändert sich beispielsweise der Inhalt des Paketes `e`, dann kann das Auswirkungen auf `b` haben, was wiederum Anpassungen im von `b` abhängigen Paket `a` erfordern kann; die übrigen Pakete sind von den Änderungen nicht betroffen. Durch Zurückverfolgen der eingehenden Kanten können so die eventuell zu

ändernden Pakete gefunden werden.

Welche Konsequenzen hätte es, wenn nicht mehr `a` von `d`, sondern `d` von `a` abhängig wäre?

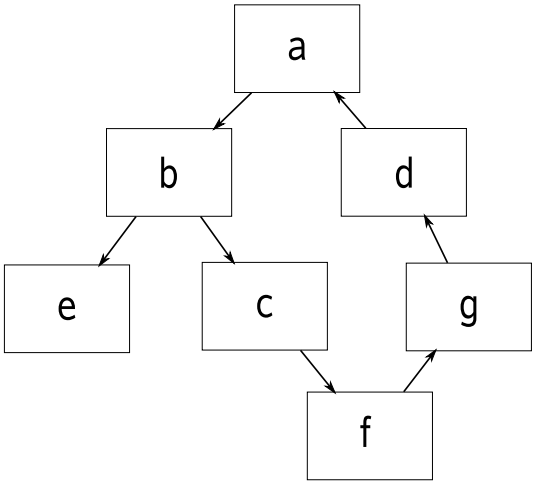

Abbildung 3: Abhängigkeitsgraph mit Zyklus.

Änderungen an `e` können sich jetzt auf alle anderen Pakete auswirken. Allgemein gilt bei Zyklen im Abhängigkeitsgraph: Änderungen an einem im Zyklus beteiligten Paket können Änderungen an allen im Zyklus beteiligten Paketen erfordern. Unter Umständen muss der Zyklus beim Anpassen nach einer Änderung sogar mehrmals durchlaufen werden, weil das ursprünglich geänderte Paket abhängig von seinem Vorgänger im Zyklus ist und dieser geändert wurde. Aus diesem Grund sind zyklische Abhängigkeiten zu vermeiden („The Acyclic Dependency Principle (ADP)“, [MAR05], Seite 18): „*The dependency structure between packages must be a Directed Acyclic Graph (DAG). That is, there must be no cycles in the dependency structure.*“

Die obigen Beispiele zeigen, dass Wissen über die Beziehungen zwischen Paketen von Bedeutung sein kann – das Durchsuchen des Quelltextes „per Hand“ ist sicherlich kein probates Mittel, um es zu erlangen. In dieser Arbeit wird deswegen in Kapitel 3 zunächst untersucht, wodurch genau Abhängigkeiten zwischen Typen und damit zwischen Paketen entstehen können. Im Anschluss daran werden einige Methoden zum Auflösen bestimmter Abhängigkeiten vorgestellt. Darauf aufbauend wurde ein Programm entwickelt, das Abhängigkeiten zwischen Java-Paketen findet und darstellt sowie den Anwender bei deren Auflösung unterstützt. Es wird in Kapitel 4 vorgestellt, in Kapitel 5 evaluiert und in Kapitel 6 bewertet; hier werden auch bereits existierende Werkzeuge vorgestellt, die einen ähnlichen Zweck erfüllen. Kapitel 7 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab.

2 Problemstellung

Das Hauptaugenmerk dieser Arbeit liegt in der Entwicklung eines Programms, mit dessen Hilfe Abhängigkeiten zwischen Paketen dargestellt werden können. Dafür nötig ist eine genaue Betrachtung der Java-Syntax um festzustellen, durch welche Sprachkonstrukte sie entstehen können. Anforderungen an das zu entwickelnde Programm sind:

- Finden und Anzeigen aller Abhängigkeiten zwischen vom Benutzer ausgewählten Paketen (auch aus verschiedenen Projekten),
- Integrierung in die Entwicklungsumgebung Eclipse,
- leicht verständliche grafische Darstellung,
- intuitive Bedienbarkeit,
- Bereitstellen von Refactorings zur Auflösung von Abhängigkeiten.

Zwar existieren bereits einige Programme zur Analyse von Abhängigkeiten (eine Übersicht findet sich in Kapitel 6.1), sie stellen aber anders als das im Rahmen dieser Arbeit entwickelte Werkzeug keine Mittel bereit, um den Benutzer bei deren Auflösung zu unterstützen. „Auflösen“ bedeutet dabei meist, die Richtung von Abhängigkeiten umzukehren, wozu das Eclipse-Plugins „Infer Type“ ([KEG07]) verwendet wird, das auf Basis von [STE06] und [STE07] entwickelt wurde. Infer Type kann für Deklarationselemente (das sind lokale Variablen, Felder, Parameter oder Methoden mit Rückgabewert) neue Typen bestimmen, die nur die unbedingt nötigen Methoden deklarieren und damit maximal verallgemeinert sind.

3 Abhängigkeiten zwischen Java-Paketen

Zunächst seien Abhängigkeiten zwischen Typen, Compilation Units und Paketen durch ein anschauliches Kriterium definiert:

Definition (Abhängigkeit): Sei $\mathcal{V}$ eine Menge von Typen, Compilation Units oder Java-Paketen; seien p, $q \in \mathcal{V}$ verschiedene Elemente dieser Menge. Dann heißt p abhängig von q (Notation: $p \rightarrow q$) genau dann, wenn nach dem Löschen von q Änderungen in p vorgenommen werden müssen, damit p kompilierbar ist[3]; p wird auch als Quelle der Abhängigkeit bezeichnet, q als deren Ziel.

Die Relation $\rightarrow: \mathcal{V} \times \mathcal{V}$ ist offensichtlich nicht reflexiv, nicht transitiv, nicht symmetrisch und nicht antisymmetrisch. Der *Abhängigkeitsgraph* $G(V, \rightarrow)$ eignet sich zur Darstellung der Abhängigkeiten. Eine gerichtete Kante von einem Knoten $v \in V$ zu einem anderen Knoten $u \in V$ existiert genau dann, wenn $(v, u) \in \rightarrow$, also v abhängig von u ist.

Das beschriebene Kriterium bietet allerdings kein brauchbares Mittel, um Typen, Compilation Units oder Pakete auf Abhängigkeiten zu untersuchen. Wodurch sie im Einzelnen entstehen können, wird durch die folgende Analyse der Java-Syntax (wie in [GOS05] beschrieben) betrachtet.

3.1 Entstehung von Abhängigkeiten

Ein Paket `p` ist genau dann Quelle einer Abhängigkeit `p` $\rightarrow$ `q`, wenn eine Compilation Unit `P` in `p` existiert, die Quelle einer Abhängigkeit `P` $\rightarrow$ `q` ist. Compilation Units bestehen aus Importanweisungen und Typdeklarationen. Nichtstatische Importanweisungen beziehen sich entweder auf einen bestimmten Typ (`import packagepath.SomeType;`), auf alle sichtbaren Typen eines Paketes (`import packagepath.*;`) oder auf alle inneren Klassen eines Typs (`import packagepath.SomeType.*;`). Statische Importanweisungen referenzieren statische Methoden (`import static packagepath.SomeType.someStaticMethod;`) oder Felder (`import static packagepath.SomeType.someStaticField;`) oder beziehen sich auf alle in einem Typ deklarierten statischen Variablen, Felder und Methoden, die sichtbar sind (`import static packagepath.SomeType.*;`)[4].

Importanweisungen referenzieren also einen konkreten Typ, außer wenn sie die Form `import packagepath.*;` haben; hier entsteht eine Abhängigkeit, deren Ziel ein Paket ist. Im Gegensatz dazu können innerhalb eines Typs keine Abhängigkeiten auftreten, die ein Paket ohne Angabe eines bestimmten Typs ansprechen. Auch eine Compilation Unit kann nicht referenziert werden, erlaubt sind nur Verweise auf die Typen einer Compilation Unit. Zusammenfassend: Abhängigkeiten mit einer Compilation Unit als Quelle entstehen genau

[3] „Kompilierbar“ bedeutet, dass ein aktueller Java-Compiler einen Typ, eine Compilation Unit bzw. ein Paket ohne Fehlermeldung akzeptiert.

[4] siehe [GOS05], Kapitel 7.5

dann, wenn

- die Compilation Unit eine Import-Anweisung enthält, die alle Typen eines anderen Paketes importiert oder
- die Compilation Unit einen Typ enthält, der von einem in einer anderen Compilation Unit deklarierten Typ abhängig ist.

Um zu ermitteln, ob ein Paket `p` abhängig von einem anderen `q` ist, muss also überprüft werden, ob `p` eine Compilation Unit enthält, die `q` importiert oder ob `p` einen Typ enthält, der von einem in `q` deklarierten Typ abhängig ist. Im Folgenden werden nur noch Abhängigkeiten zwischen Typen betrachtet; daraus entstehende Abhängigkeiten zwischen Paketen ergeben sich sofort, wenn die Typen in verschiedenen Paketen deklariert sind. Anonyme Typen können nicht in anderen Typen referenziert werden; Abhängigkeiten von anonymen Typen werden als Abhängigkeiten vom ersten äußeren, nicht-anonymen Typ betrachtet.

In Java werden Typen durch Angeben des einfach Namens („Simple Name“, z. B. `String`) oder durch den Namen mit vorangestellter Pfadangabe („Fully Qualified Name“, z. B. `java.lang.String`) referenziert. Das Entfernen eines referenzierten Typs erfordert offensichtlich auch Änderungen am referenzierenden Typ:

Referenzabhängigkeit: Wird in einem Typ `P` ein anderer Typ `Q` referenziert, so entsteht dadurch eine Abhängigkeit `P` → `Q`.

Beispielsweise verursachen die folgenden Codefragmente jeweils eine Referenzabhängigkeit zum Typ `A`:

- `class B extends A ...`
- `path.to.A.someStaticMethod();`
- `new A();`
- `if (foo instanceof A) ...`
- `public A[][]<String> getMatrix() ...`

Das Aufrufen einer Methode in einem Typ `P`, die in einem anderen Typ `Q` deklariert ist, führt offenbar dazu, dass `P` nicht mehr kompilierbar ist, wenn `Q` gelöscht wird. Dass es jedoch nicht immer ersichtlich ist, welche Methode tatsächlich aufgerufen wird, zeigt folgendes Beispiel:

```
1  class A {
2    public void doSth() {..}
3    public void m() {..}
4  }
5
6  class B extends A {
7    public void doSth() {..}
8  }
9
10 class C {
11   void foo() {
12     A[] aArray = new A[2];
13     aArray[0] = new A();
14     aArray[1] = new B();
15     for (int i = 0; i < 2; i++) {
16       aArray[i].doSth();
17     }
18   }
19
20   void bar(B someB) {
21     someB.m();
22   }
23 }
```

Durch die dynamische Bindung wird im ersten Schleifendurchlauf in `foo()` die in `A` deklarierte Methode `doSth()` aufgerufen, im zweiten die in `B` überschriebene. Unabhängig davon, ob `doSth()` in `B` überschrieben wird, besteht offenbar eine Abhängigkeit $\texttt{C} \rightarrow \texttt{A}$, weil `doSth()` auf Objekte des Typs `A` aufgerufen wird und das Löschen von `A` Änderungen in `C` nötig macht.

Abhängigkeit durch Aufrufen einer nichtstatischen Methode: Wird in einem Typ `P` eine Methode eines Objektes aufgerufen, dessen in der Typhierarchie letzter mit Sicherheit bekannter Typ `Q` ($\texttt{P} \neq \texttt{Q}$) ist, so entsteht dadurch eine Abhängigkeit $\texttt{P} \rightarrow \texttt{Q}$.

Um der Tatsache Rechnung zu tragen, dass auch die Methode eines Subtyps von `Q` aufgerufen werden könnte, werden nicht näher definierte **indirekte Abhängigkeiten** (Notation: $\texttt{P} \rightarrow_i \texttt{Q}$) eingeführt; in obigem Beispiel entsteht durch den Aufruf von `aArray[i].doSth();` die indirekte Abhängigkeit $\texttt{C} \rightarrow_i \texttt{B}$. In `bar(B)` wird die Methode `m()` eines `B`-Objektes aufgerufen, die in `A` deklariert ist. Änderungen an `A` können sich auf `C` auswirken: Wird etwa die Methode `m()` in `A` gelöscht, ist `C` fehlerhaft. Allerdings sind Anpassungen in `C` nicht zwingend erforderlich, um `C` wieder kompilierbar zu machen: Der Fehler kann behoben werden, indem `m()` in `B` eingefügt wird. Durch den Aufruf `someB.m()` entsteht damit eine indirekte Abhängigkeit $\texttt{C} \rightarrow_i \texttt{A}$.

Indirekte Abhängigkeit durch Aufrufen einer nichtstatischen Methode: Eine Abhängigkeit $\texttt{P} \rightarrow \texttt{Q}$ zwischen Typen `P` und `Q`, die durch das Aufrufen einer nichtstatischen

Methode entstanden ist, hat eine indirekte Abhängigkeit $\texttt{P} \rightarrow_i \texttt{R}$ zur Folge, wenn

- die Methode in einem Subtyp R ($\texttt{P} \neq \texttt{R}$) von Q überschrieben wird oder
- R ($\texttt{P} \neq \texttt{R}$) ein Supertyp von Q ist, der die Methode deklariert, und kein weiterer Typ existiert, der sich in der Hierarchie zwischen R und einschließlich Q befindet und die Methode überschreibt.

Anders bei statischen Methoden: da sie ohne Referenz auf ein bestimmtes Objekt aufgerufen werden, ist immer klar, in welchem Typ sie deklariert sind:

Abhängigkeit durch Aufrufen einer statischen Methode: Wird in einem Typ P eine statische Methode aufgerufen, die in einem anderen Typ Q deklariert ist, so entsteht dadurch eine Abhängigkeit $\texttt{P} \rightarrow \texttt{Q}$.

Analoges gilt beim Zugriff auf Typvariablen:

Abhängigkeit durch Zugriff auf ein nichtstatisches Feld: Wird in einem Typ P auf ein Feld eines Objektes zugegriffen, dessen in der Typhierarchie letzter, mit Sicherheit bekannter Typ Q ($\texttt{P} \neq \texttt{Q}$) ist, dann ist P von Q abhängig ($\texttt{P} \rightarrow \texttt{Q}$).

Abhängigkeit durch Zugriff auf ein statisches Feld: Wird in einem Typ P auf ein statisches Feld zugegriffen, das in einem anderen Typ Q deklariert ist, so entsteht dadurch eine Abhängigkeit $\texttt{P} \rightarrow \texttt{Q}$.

Feldzugriffe verursachen wiederum indirekte Abhängigkeiten, wenn ein Feld in einem Supertyp deklariert ist:

Indirekte Abhängigkeit durch Zugriff auf ein nichtstatisches Feld: Eine Abhängigkeit $\texttt{P} \rightarrow \texttt{Q}$ zwischen Typen P und Q, die durch das Zugreifen auf eine Objektvariable entsteht, hat eine indirekte Abhängigkeit $\texttt{P} \rightarrow_i \texttt{R}$ zur Folge, wenn R ($\texttt{P} \neq \texttt{R}$) der Supertyp von Q ist, in dem das Feld deklariert ist.

Je nach Typ eines Deklarationselementes können beim Aufruf bzw. Zugriff weitere Abhängigkeiten entstehen. Man betrachte dazu den folgenden Codeausschnitt:

```
1  import java.util.*;
2  class A {
3    B getB() {...}
4    B someB;
5  }
6
7  class B extends LinkedList {...}
8
9  class C {
10   void foo(A someA) {
11     someA.getB();
12     List l1 = someA.getB();
13     List l2 = someA.someB;
14     inspectList(someA.getB());
15     inspectList(someA.someB);
16   }
17
18   void inspectList(List l) {...}
19 }
```

Entstehen hier Abhängigkeiten zwischen `C` und `B`? In Zeile 11 wird die Methode `getB()` eines Objektes vom Typ `A` aufgerufen, der Rückgabewert wird allerdings ignoriert; das Löschen von `B` macht keine Änderungen in `C` notwendig. Dagegen wird in Zeile 12 der Rückgabewert von `getB()` einer lokalen Variablen zugewiesen. Wird `B` gelöscht, ist eine Änderung des Rückgabewertes in `getB()` nötig. Wird er z. B. auf `java.util.ArrayList` geändert, muss `C` nicht weiter angepasst werden, da `ArrayList` auch vom Typ `List` ist. Ist der neue Rückgabewert von `getB()` dagegen vom Typ `String`, dann muß auch `C` in Zeile 12 geändert werden, weil `String` das Interface `List` nicht implementiert. Analoges gilt bei der Zuweisung einer Variablen in Zeile 13. Solche Abhängigkeiten werden ebenfalls zu den indirekten gezählt:

Indirekte Zuweisungsabhängigkeit: Ist die rechte Seite einer Zuweisung in einem Typ `Q` ein Deklarationselement vom Klassentyp[5] `R` (`Q` $\neq$ `R`), entsteht dadurch eine indirekte Abhängigkeit `Q` $\rightarrow_i$ `R`.

In Zeile 14 ist das von `getB()` zurückgegebene Objekt Argument eines Methodenaufrufs, in Zeile 15 die Variable `someB`. Auch hier gilt: Eine Änderung von `C` kann nach dem Löschen von `B` erforderlich sein.

Indirekte Methodenargumentabhängigkeit: Wird in `Q` ein Deklarationselement vom Klassentyp `R` (`Q` $\neq$ `R`) einer Methode als Parameter übergeben, dann entsteht eine

[5]Der Klassentyp $|T|$ eines Typs T ergibt sich rekursiv, wobei $T_1, T_2, ...$ Typen sind:
1. Hat T die Form $T_1[]$, so ist $|T| = |T_1|$.
2. Hat T die Form $T_1 < T_2, T_3, .. >$, dann ist $|T| = |T_1|$.
3. Sonst: $|T| = T$.
Zum Beispiel ist `List` der Klassentyp von `List<Object[]>[]`.

indirekte Abhängigkeit `Q` $\rightarrow_i$ `R`.

Abhängigkeiten können ihren Ursprung in verketteten Methodenaufrufen oder Feldzugriffen haben; insbesondere entstehen sie durch Verletzung des Gesetzes von Demeter (siehe [LIE89]): Zum Beispiel ist ein Typ `B`, der `someLocalVariable.getA().doSth();` enthält, abhängig von `A`, wobei `A` ein von `B` verschiedener Typ ist, der die Methode `doSth()` deklariert. In EBNF-Notation (nach ISO/IEC 14977) ausgedrückt haben solche Ketten die folgende Form:

Chain = [DeclarationElement "."] {AccessOrInvocation "."}- AccessOrInvocation;

Sie bestehen also aus einem optionalen Deklarationselement, das von mindestens zwei Feldzugriffen oder Methodenaufrufen gefolgt wird. Die Abhängigkeiten zu den Typen, in denen die AccessOrInvocation-Deklarationselemente (mit Ausnahme des ersten) deklariert sind, werden „verkettete Abhängigkeiten“ genannt.

3.2 Auflösen von Abhängigkeiten

Idealerweise wird bereits beim Software-Entwurf darauf geachtet, die Anzahl der Abhängigkeiten zu minimieren und nicht an ungeeigneten Stellen auftreten zu lassen. Doch auch an bestehenden Projekten können viele von ihnen aufgelöst werden, wie die folgenden Beispiele zeigen.

- Wie erwähnt entstehen Abhängigkeiten zwischen Paketen aus Abhängigkeiten zwischen Typen aus diesen Paketen. Referenziert ein Typ `a.A` einen anderen Typen `b.B` dann läßt sich die Abhängigkeit zwischen `a` und `b` dadurch beseitigen, dass man `B` in das Paket `a` verschiebt oder `A` nach `b`. Das ist natürlich nur dann sinnvoll, wenn die beiden Typen auch thematisch in das selbe Paket passen und durch das Verschieben nicht neue, noch „schlimmere“ Abhängigkeiten entstehen. Letzteres kann unter Umständen verhindert werden, indem `A` bzw. `B` nicht verschoben, sondern teilweise oder komplett kopiert werden. Da dadurch redundanter Code entsteht, ist diese Lösung aber im Allgemeinen nicht sinnvoll.

- Die folgenden Klassen `A` und `B` seien in verschiedenen Pakete `a` und `b` deklariert:

```
1  package a;
2  class A extends java.util.LinkedList {
3    public void addRandomElements() {...}
4  }
5
6  package b;
7  class B {
8    void foo(a.A list) {
9      list.clear();
10   }
11 }
```

Durch die Referenzierung von `A` ist `B` abhängig von `A` und damit `b` abhängig von `a`. In `B` wird allerdings die in `A` deklarierte Methode `addRandomElements()` nicht aufgerufen, weswegen der Typ des Parameters von `foo(a.A)` verallgemeinert werden kann. Geeignet ist jeder Supertyp von `A`, der die Methode `clear()` enthält, also zum Beispiel `java.util.LinkedList` oder `java.util.AbstractCollection`, in dem `clear()` deklariert wird.

Durch Verwendung von Supertypen können wie in obigem Beispiel Abhängigkeiten aufgelöst oder verschoben werden. In Eclipse stehen dazu die Refactorings „Use Supertype Where Possible...“ und „Generalize Declared Type...“ zur Verfügung, die den Programmierer beim Vornehmen von Typverallgemeinerungen unterstützen.

- Obiges Beispiel sei um einen Aufruf von `addRandomElements()` in `B` erweitert:

```
1 package b;
2 class B {
3   void foo(a.A list) {
4     list.clear();
5     list.addRandomElements();
6   }
7 }
```

Da jetzt in `B` eine in `A` deklarierte Methode aufgerufen wird, kann der Typ des Parameters von `foo(a.A)` nicht mehr durch einen Supertyp ersetzt werden. Führt man jedoch ein Interface `I` ein, das die Methoden `clear()` und `addRandomElements()` deklariert und von `A` implementiert wird, dann kann der Parametertyp auf eben dieses Interface geändert werden:

```
1  package a;
2  class A extends java.util.LinkedList implements I {
3    public void addRandomElements() {...}
4  }
5
6  package b;
7  class B {
8    void foo(I list) {
9      list.clear();
10     list.addRandomElements();
11   }
12 }
13
14 interface I {
15   void addRandomElements();
16   void clear();
17 }
```

Offenbar ist `A` jetzt nicht mehr von `B` abhängig, `A` und `B` allerdings von `I`. Noch nicht beantwortet ist die Frage, in welchem Paket `I` abgelegt werden soll. Nach Hinzufügen

zu `a` ist `b` immer noch von `a` abhängig. Das Einfügen in `b` kehrt diese Abhängigkeit um, was von Vorteil sein kann, wenn dadurch z. B. ein Zyklus im Abhängigkeitsgraph aufgebrochen wird. Die dritte Möglichkeit besteht im Ablegen von `I` in einem weiteren Paket `c`. Das kann beispielsweise dann sinnvoll sein, wenn `a` und `b` getrennt voneinander entwickelt werden sollen und die entstehenden Abhängigkeiten zu `c` weniger problematisch sind, weil `c` ein stabiles Paket ist, also mit geringer Wahrscheinlichkeit zukünftig geändert wird.

Nicht immer ist die Bestimmung, welche Methoden in einem auf die beschriebene Weise verallgemeinerten Typ mindestens enthalten sein müssen, eine so einfach lösbare Aufgabe wie in obigem Beispiel. Aus diesem Grund wurde das Programm „Infer Type“[6] im Rahmen von [KEG07] entwickelt, das sie dem Programmierer abnimmt.

- Angenommen, ein Projekt nutze die Klasse `someProject.util.ErrorNotifier`, um den Benutzer über das Auftreten eines Fehlers zu informieren:

```
1 package someProject.util;
2 class ErrorNotifier {
3   public void error(String message) {
4     ErrorDialog errorDialog = new ErrorDialog();
5     errorDialog.handleError(message);
6   }
7 }
```

Die Darstellung der Fehlermeldung auf dem Bildschirm übernimmt die Klasse `someProject.ui.ErrorDialog`:

```
1 package someProject.ui;
2 class ErrorDialog {
3   public void handleError(String message) {
4     //create GUI dialog with the given message
5   }
6 }
```

Offensichtlich ist hier `someProject.util` abhängig von `someProject.ui`. Soll eine Fehlermeldung dem Benutzer allerdings nicht per Dialog übermittelt werden, sondern z. B. in ein Logdatei geschrieben werden, kann die Methode `error(String)` nicht verwendet werden. Die Wiederverwendbarkeit von `ErrorNotifier` ist dadurch eingeschränkt. Angenommen, die Klasse `someProject.logging.ErrorLogger` soll Fehlermeldungen in eine Logdatei schreiben. Um auch dafür die Methode `error(String)` verwenden zu können, bietet sich folgende Lösung an:

Man führt ein Interface `someProject.util.ErrorHandler` ein, das die Methode

[6] http://intoj.org/

`handleError(String)` deklariert und von `ErrorDialog`, `ErrorLogger` sowie eventuell weiteren Klassen, die die Bearbeitung von Fehlermeldungen übernehmen, implementiert wird. Der Methode `error(String)` in `ErrorNotifier` muss dann ein `ErrorHandler`-Objekt übergeben werden:

```
1  package someProject.util;
2  class ErrorNotifier {
3    public void error(String message, ErrorHandler handler) {
4      handler.handleError(message);
5    }
6  }
7
8  interface ErrorHandler {
9    public void handleError(String message);
10 }
11
12 package someProject.ui;
13 class ErrorDialog implements ErrorHandler {
14   public void handleError(String message) {
15     //create GUI dialog with the given message
16   }
17 }
18
19 package someProject.logging;
20 class ErrorLogger implements ErrorHandler {
21   public void handleError(String message) {
22     //write error message to log file
23   }
24 }
```

Die Klassen `ErrorDialog`, `ErrorLogger` und `ErrorNotifier` sind jetzt abhängig von `ErrorHandler` und damit die Pakete `someProject.ui` und `someProject.logging` von `someProject.util`; die Abhängigkeit `someProject.util` → `someProject.ui` wurde umgekehrt und dadurch die Wiederverwendbarkeit von `ErrorNotifier` erhöht. Hier wurde dem von Robert C. Martin formulierten „Dependency Inversion Principle“ (DIP, siehe [MAR96]) Rechnung getragen:

A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions.

Das beschriebene Verfahren ist eine Form des sogenannten Dependency Injection Patterns (siehe [FOW04]). Im Rahmen von [MON08] wurde ein Werkzeug entwickelt, das dessen Anwendung automatisiert.

- Der folgende Programmausschnitt beschreibt verschiedene Medien wie Bücher oder

CDs. Die Klasse `media.util.Statistics` enthält Methoden, in der die Anzahl der Bücher oder CDs aus einer Menge von Medien ermittelt wird:

```
1  package media;
2  abstract class Medium {
3    ...
4  }
5
6  package media.records;
7  class CD extends Medium {
8    ...
9  }
10
11 package media.paper;
12 class Book extends Medium {
13   ...
14 }
15
16 package media.util;
17 class Statistics {
18   public int countBooks(Set<Medium> databaseEntries) {
19     int booksCount = 0;
20     for (Medium m : databaseEntries) {
21       if (m instanceof Book) {
22         booksCount++;
23       }
24     }
25     return booksCount;
26   }
27
28   public int countCDs(Set<Medium> databaseEntries) {
29     //analog zu countBooks(Set<Medium>)
30   }
31 }
```

Den dazugehörigen Abhängigkeitsgraph zeigt Abbildung 4.

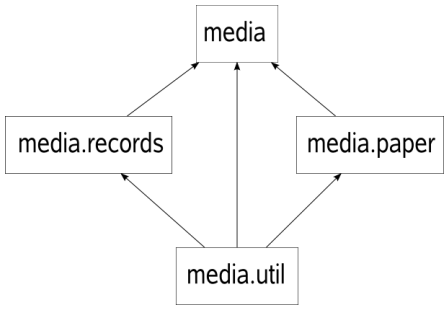

Abbildung 4: Abhängigkeitsgraph der Pakete zur Medienverwaltung.

Durch den Typtest der `Medium`-Objekte mit `instanceof` in Zeile 21 und in allen weite-

ren Methoden, die die Anzahl einer bestimmten Medienart ermitteln, ist `media.util` nicht nur von `media` abhängig, sondern auch von den anderen Paketen, die Medien deklarieren. Statt des `instanceof`-Tests ist auch die folgende Lösung möglich: `Medium` definiert Konstanten zu allen vorhandenen Medienarten und deklariert eine Methode `getKind()`, die für ein `Medium`-Objekt die entsprechende Konstante zurückgibt.

```
1 package media;
2 class abstract Medium {
3
4   public final static int BOOK = 1;
5   public final static int CD = 2;
6   ...
7
8   public abstract int getKind();
9 }
```

Alle Subtypen von `Medium` implementieren die abstrakte `getKind()`-Methode, wodurch der Test auf die Art eines Mediums in `Statistics` ohne Referenz auf die Subtypen von `Medium` möglich ist:

```
1 package media.util;
2 class Statistics {
3   public int countBooks(Set<Medium> databaseEntries) {
4     int booksCount = 0;
5     for (Medium m : databaseEntries) {
6       if (m.getKind() == Medium.BOOK) {
7         booksCount++;
8       }
9     }
10    return booksCount;
11  }
12 }
```

`Statistics` ist jetzt nur mehr von `Medium` abhängig, was zu dem in Abbildung 5 gezeigten Abhängigkeitsgraphen führt.

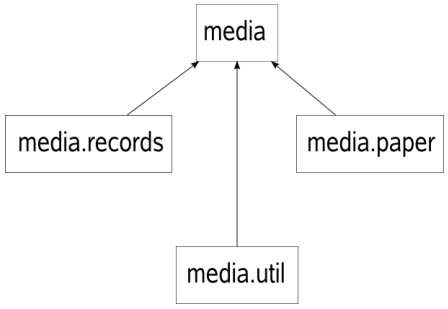

Abbildung 5: Abhängigkeitsgraph nach dem Einführen von Typkonstanten.

Wesentlich eleganter können jetzt außerdem die weitgehend redundanten Methoden `countBooks(Set<Medium>)`, `countCDs(Set<Medium>)` etc. zu einer Methode `countMedium(Set<Medium>, int)` zusammengefasst werden, der eine der Konstanten aus `Medium` übergeben wird, um die Anzahl der Medien der entsprechenden Art zu ermitteln:

```
1  package media.util;
2  class Statistics {
3    public int countMedium(Set<Medium> databaseEntries, int kind) {
4      int booksCount = 0;
5      for (Medium m : databaseEntries) {
6        if (m.getKind() == kind) {
7          booksCount++;
8        }
9      }
10     return booksCount;
11   }
12 }
```

Allgemein lassen sich viele Abhängigkeiten, die durch `instanceof`-Vergleiche entstehen, auf ähnliche Weise vermeiden: Soll ein Objekt vom Typ `T` darauf überprüft werden, ob es vom Typ `S` ist (`S` ist sinnvollerweise ein Subtyp von `T`, darf allerdings auch ein Supertyp von `T` sein), dann können Abhängigkeiten zu dem Paket, in dem `S` deklariert ist, vermieden werden, indem in `T` auf die oben beschriebene Weise Konstanten für mögliche Subtypen eingeführt werden. Allerdings ist das nicht immer möglich, wenn beispielsweise `T` dem Typ `java.lang.Object` entspricht oder einem anderen außerhalb des Projekts deklarierten Typ, der nicht verändert werden kann oder soll. In diesen Fällen kann möglicherweise die Einführung eines Interfaces Abhilfe schaffen, das `getKind()` und die Typkonstanten deklariert und von den Subtypen von `T` implementiert wird.

- Auf ähnliche Weise können sich manche Abhängigkeiten auflösen lassen, die durch Casts entstehen. In obigem Beispiel sei `Statistics` um die folgende Methode erweitert, die die Zeit ermittelt, die für das Konsumieren aller in der Datenbank enthaltenen Medien benötigt wird:

```
1    public int getConsumeTime(Set<Medium> databaseEntries) {
2      int time = 0;
3      for (Medium m : databaseEntries) {
4        if (m.getKind() == Medium.Book) {
5          time += ((Book) m).getPageCount() * 5;
6        } else if (m.getKind() == Medium.CD) {
7          time += ((CD) m).getTotalPlayTime();
8        }
9      }
10     return time;
11   }
```

Je nach Art des Mediums berechnet sich die Zeit auf verschiedene Weise, weswegen auf in `Book` und `CD` deklarierte Methoden zurückgegriffen werden muss und damit wieder Abhängigkeiten zu deren Paketen entstehen. Auch hier bietet sich eine Lösung an, in der die Klasse `Medium` um eine abstrakte Methode `getConsumeTime()` erweitert wird, die von den Unterklassen implementiert wird und die Zeit zum Konsumieren eines Mediums liefert. Damit kann `getConsumeTime(Set<Medium>)` folgendermaßen vereinfacht werden:

```
1  public int getConsumeTime(Set<Medium> databaseEntries) {
2    int time = 0;
3    for (Medium m : databaseEntries) {
4      time += m.getConsumeTime();
5    }
6    return time;
7  }
```

Durch die Delegierung der Berechnung an die verschiedenen `Medium`-Unterklassen ist `Statistics` wieder unabhängig von `Book` und `CD`, zwischen deren Paketen befinden sich keine Abhängigkeiten mehr. Auch dieses Beispiel lässt sich verallgemeinern. Durch Casts entstehende Abhängigkeiten können so in vielen Fällen vermieden werden.

3.3 Die Invert-Dependency-Methode

Die beschriebenen Abhängigkeiten können in zwei Gruppen eingeteilt werden: Implementiert eine Klasse ein Interface oder erweitert ein Typ einen anderen, entstehen Abhängigkeiten zu den in den `implements`- bzw. `extends`-Anweisungen angegebenen Typen. Diese Abhängigkeiten entsprechen den UML-Konstrukten „Realization“ ([UML07], Seite 129 f.) und „Generalization“ ([UML07], Seite 71 ff.) und werden einer Gruppe namens „Super-Abhängigkeiten“ zugeordnet. Die andere Gruppe „Uses-Abhängigkeiten“ enthält alle anderen Abhängigkeiten; sie entsprechen der „Usage“-Beziehung in UML ([UML07], Seite 137).

Uses-Abhängigkeiten von einem Paket `a` zu einem Paket `b` existieren dann, wenn ein Deklarationselement in einem in `a` deklarierten Typ vom Typ `B` ist, wobei `B` in `b` deklariert ist. Wie eine Uses-Abhängigkeit in eine Super-Abhängigkeit in umgekehrter Richtung umgewandelt werden kann, hat das Beispiel auf Seite 13 gezeigt. Im folgenden Quelltext ist es um eine Klasse `b.B2` erweitert:

```
1  package a;
2  class A extends java.util.LinkedList {
3    public void addRandomElements() {...}
4  }
5
6  package b;
7  class B {
8    void foo(a.A list) {
9      list.clear();
10     list.addRandomElements();
11   }
12 }
13
14 class B2 {
15   void bar(a.A list) {
16     list.addRandomElements();
17   }
18 }
```

Nicht nur `B` enthält jetzt eine Uses-Abhängigkeit mit Ziel `A`, sondern auch `B2`. Man kann sie wiederum in Super-Abhängigkeiten umwandeln, indem man, wie beschrieben, ein Interface `b.I` einführt, das die Methoden `clear()` und `addRandomElements()` deklariert, von `A` implementiert wird und als neuer Typ für den Parameter `list` in `B` angegeben wird. Analog dazu wird ein Interface `b.I2` eingeführt, das nur die Methode `addRandomElements()` enthält und ebenfalls von `A` implementiert wird. Setzt man es in `B2` als Typ des `list`-Parameters ein, ist `b` wieder unabhängig von `a`. Stellt man sich allerdings vor, dass in einigen weiteren in `b` deklarierten Klassen zahlreiche Deklarationselemente den Typ `A` besitzen, dann ist es wenig sinnvoll, für jeden einzeln einen maximal verallgemeinerten Typ zu inferieren, der dann als Interface in `b` eingefügt und von `A` implementiert wird. In obigem Beispiel ist die Einführung von `I2` zur Auflösung der Abhängigkeit `a` → `b` nicht unbedingt nötig, stattdessen hätte auch das Interface `I` als Typ von `list` in `B2` verwendet werden können.

Weniger Redundanz und übersichtlicheren Code erhält man, wenn man nicht für jedes in einem Paket `p` vorhandene Deklarationselement eines Typs `T`, der in einem von `p` verschiedenen Paket deklariert ist, einen maximal verallgemeinerten Typ bestimmt, sondern wenn ein Typ berechnet wird, der für *alle* Deklarationselemente in `p` vom Typ `T` geeignet ist. Dieser ist dann innerhalb von `p` maximal verallgemeinert. Erweitert man die Methode `bar(a.A)` aus obigem Beispiel um einen Aufruf der Methode `size()` des `list`-Objektes (`size()` ist in `LinkedList` deklariert), dann muss ein innerhalb von `b` maximal verallgemeinertes Interface die Methoden `clear()`, `addRandomElements()` und `size()` enthalten. In `b` eingefügt, von `A` implementiert und jeweils als neuer Typ der beiden `list`-Parameter angegeben bewirkt es, dass die Abhängigkeit `a` → `b` „auf einen Schlag“ aufgelöst wird:

```
1  package a;
2  class A extends java.util.LinkedList implements b.I {
3    public void addRandomElements() {...}
4  }
5
6  package b;
7  class B {
8    void foo(I list) {
9      list.clear();
10     list.addRandomElements();
11   }
12 }
13
14 class B2 {
15   void bar(I list) {
16     list.addRandomElements();
17     int size = list.size();
18   }
19 }
20
21 class I {
22   void clear();
23   void addRandomElements();
24   int size();
25 }
```

Weil von dem Objekt, das an die Methoden `foo(I)` und `bar(I)` übergeben wird, jetzt verlangt wird, dass es das Interface `I` implementiert, bezeichnet man `I` als ein **Required Interface** von `B` und `B2`. Allgemein: Existiert in einem Typ `T` ein Deklarationselement vom Typ `I` und ist `I` ein Interface, dann ist `I` ein Required Interface von `T` ([UML07], Seite 87). Als die Required Interfaces eines Paketes werden all jene zusammengefasst, die Required Interfaces eines in diesem Paket deklarierten Typs sind.

Dadurch, dass `A` in obigem Beispiel `I` implementiert, liefern `A`-Objekte die in `I` deklarierten Methoden; `I` wird deswegen **Provided Interface** von `A` genannt. Allgemein sind alle von einem Typ `T` implementierten Interfaces die Provided Interfaces von `T` ([UML07], Seite 87). Die Provided Interfaces eines Pakets ergeben sich wiederum aus jenen der in dem Paket deklarierten Typen.

Das beschriebene Verfahren wird **Invert-Dependency-Methode** genannt. Sie kann auf eine Uses-Abhängigkeit $\texttt{p} \rightarrow \texttt{T}$ zwischen einem Paket `p` und einem Typ `T` dann angewendet werden, wenn in `p` ein Deklarationselement vom Typ `T` existiert. Weil allerdings Uses-Abhängigkeiten ihren Ursprung nicht immer in Deklarationselementen haben, ist das Umwandeln aller Uses- in Super-Abhängigkeiten nicht immer vollständig möglich. Dies ist dann der Fall, wenn ein in `p` deklarierter Typ eines der folgenden Sprachkonstrukte enthält:

- Constructor-Aufruf: `Object someObject = new T()`,

- Typtest mit `instanceof`: `if (someObject instanceof T) {...}`,
- Cast: `((T) someObject).foo()`,
- Aufruf einer statischen Methode: `T.someStaticMethod()`.

Das Umkehren von Abhängigkeiten mit Hilfe der Invert-Dependency-Methode ist eine der Hauptaufgaben des im Rahmen dieser Arbeit entwickelten Programms, das im nächsten Kapitel vorgestellt wird.

4 Das Eclipse-Plugin Padev

Im Rahmen dieser Arbeit wurde ein Plugin für die Entwicklungsumgebung Eclipse entworfen und implementiert, das Abhängigkeiten zwischen Paketen darstellen und das Entkoppeln von Paketen mit Hilfe von Infer Type unterstützen soll. Es trägt den Namen „Padev“ (**Pa**ckage **De**pendency **V**iewer)[7]. In den folgenden Abschnitten werden zunächst die Bedienung[8] und der Funktionsumfang des Plugins beschrieben. Im Anschluß werden Details zu dessen Implementierung erläutert.

4.1 Installation und Bedienung

Padev ist unter Eclipse[9] ab Version 3.2.2 lauffähig. Zusätzlich müssen das Graphical Editing Framework (GEF)[10] (Version 3.2.2 oder höher) sowie mindestens die Version 1.0.8[11] von Infer Type[12] installiert sein. Nach dem Kopieren der Datei `org.intoJ.padev_1.1.0.jar` (siehe beiliegende CD) in das Plugin-Verzeichnis und einem Neustart von Eclipse ist Padev verfügbar.

Bei der Entwicklung des Plugins wurde auf eine möglichst intuitive Bedienbarkeit geachtet. Viele Elemente der Benutzeroberfläche werden durch Tooltips genauer erklärt, die erscheinen, wenn der Mauszeiger länger auf ihnen verweilt. Durch Klicken mit der rechten Maustaste auf ein Projekt (im Folgenden „Primärprojekt“ genannt) im Package Explorer von Eclipse öffnet sich ein Kontextmenü, das unter anderem den Eintrag „Package Dependencies...“ enthält. Wird er ausgewählt, erscheint der Einstellungsdialog von Padev (siehe Abbildung 6). Er ermöglicht dem Benutzer eine genauere Spezifikation der zu suchenden Abhängigkeiten: Normalerweise, wenn also die beiden Radiobuttons „Search for dependencies from all packages“ und „Search for dependencies to all packages“ ausgewählt sind, werden Abhängigkeiten

- von allen Paketen des Primärprojekts zu allen Paketen des Primärprojekts und
- von allen Paketen externer Projekte (siehe unten) zu allen Paketen des Primärprojekts und
- von allen Paketen des Primärprojekts zu allen Paketen externer Projekte

gesucht. Nach Auswählen von „...selected packages“ kann jedoch eine Einschränkung des Suchraumes vorgenommen werden: nicht ausgewählte Pakete werden nicht als Quelle (linke

[7] Padev ist wie Infer Type unter der Lizenz „Eclipse Public License - v 1.0“ veröffentlicht. Damit sind u. a. die Veränderung, Weiternutzung und -verbreitung (auch zu kommerziellen Zwecken) von Padev erlaubt. Der genaue Lizenztext ist auf der beiliegenden CD zu finden.

[8] Eine Beschreibung in englischer Sprache findet sich unter http://www.fernuni-hagen.de/ps/prjs/PDV/

[9] http://www.eclipse.org/

[10] http://www.eclipse.org/gef

[11] Die Infer-Type-Version, die momentan (August 2008) auf der IntoJ-Webseite zum Download bereit steht, genügt nicht den Anforderungen; Version 1.0.8 befindet sich auf der beiligenden CD.

[12] http://intoj.org/

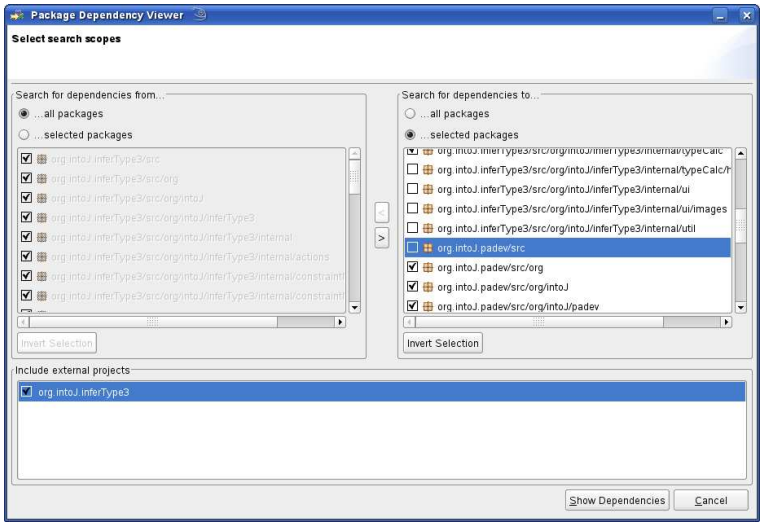

Abbildung 6: Der Dialog zum Festlegen der Suchräume.

Liste) oder Ziel (rechte Liste) in Betracht gezogen. Betätigen des Buttons „Invert Selection“ kehrt die Auswahl der jeweiligen Liste um, die beiden Schaltflächen zwischen den Listen erlauben die Übernahme der Auswahl von einer auf die andere Seite. Um die Suche nach Abhängigkeiten zwischen den Paketen des Primärprojekts und denen anderer Projekte zu ermöglichen, können in der Liste „Include external projects“ Projekte ausgewählt werden, die sich momentan im Package Explorer befinden. Deren Pakete werden dann den zuvor beschriebenen Listen hinzugefügt.

Nach Betätigen des Buttons „Show Dependencies“ wird die Suche gestartet, wobei der Benutzer über deren Fortschritt informiert wird. Im Anschluss werden die Pakete der ausgewählten Projekte und die gefundenen Abhängigkeiten zwischen ihnen angezeigt, wofür zwei verschiedene Arten der Visualisierung zur Verfügung stehen, die sogenannten „Views“. Sie unterscheiden sich in der Darstellung der Pakethierarchie:

Die „Tree View“ verwendet dazu Bäume, deren Wurzeln den Verzeichnissen im Dateisystem entsprechen, in denen die Pakete eines Projekts vorliegen.[13] Für jedes Paket eines Projektes, das als Quelle oder Ziel einer Abhängigkeit erkannt wurde, wird ein Baumknoten erzeugt; der Vater eines Knotens v_p, der das Paket `p` repräsentiert, ergibt sich auf folgende Weise: Existiert ein Knoten v_q, der dem Paket `q` entspricht, ist `p` ein Unterpaket von `q` und existiert kein weiterer Knoten, dessen Paket in der Pakethierarchie zwischen `p` und `q` liegt, dann ist v_q der Vater von v_p. Andernfalls ist die Wurzel der Vater von v_p, die jenem Verzeichnis entspricht, in dem sich `p` befindet.

[13] In den meisten Fällen besitzt ein Projekt genau ein solches Verzeichnis, das häufig den Namen „src“ trägt; es ist allerdings auch möglich, dass ein Projekt mehrere Verzeichnisse mit Quelltext enthält.

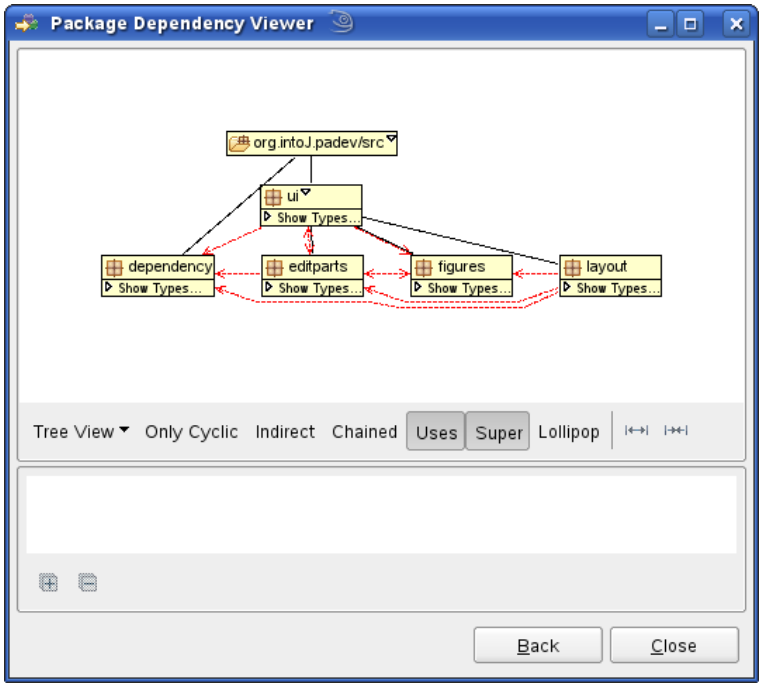

Abbildung 7: Die Tree View.

An jedem inneren Knoten können die Unterbäume ausgeblendet werden. Nach Klicken auf „Show Types...“ werden alle im jeweiligen Paket deklarierten, nicht-anonymen Typen angezeigt. Vater und Sohn werden durch schwarze Linien miteinander verbunden, Knoten, die Paketen mit identischer Tiefe in der Pakethierarchie entsprechen (deren Namen also gleich viele voneinander durch Punkte getrennte Segmente besitzen), in einer Reihe angeordnet.

Die „Box View“ stellt die analog zur Tree View erzeugten Knoten mittels geschachtelter Kästen dar. Ein Sohn wird dabei innerhalb seines Vaters dargestellt (siehe Abbildung 8). Kinder eines Kastens und die Anzeige der deklarierten Typen können wie bei der Tree View ein- und ausgeblendet werden.

Beide Views erlauben die Auswahl, welche Abhängigkeiten wie angezeigt werden sollen:

- Uses-Abhängigkeiten werden eingeblendet, wenn der Button „Uses“ betätigt ist. Deren Darstellung erfolgt mittels gestrichelter, roter Pfeile zwischen den beteiligten Paketen.
- Mit Hilfe des Buttons „Super“ können Super-Abhängigkeiten ein- und ausgeblendet werden. Sie werden von Pfeilen mit grünem, durchgezogenen Schaft zwischen voneinander abhängigen Paketen repräsentiert.

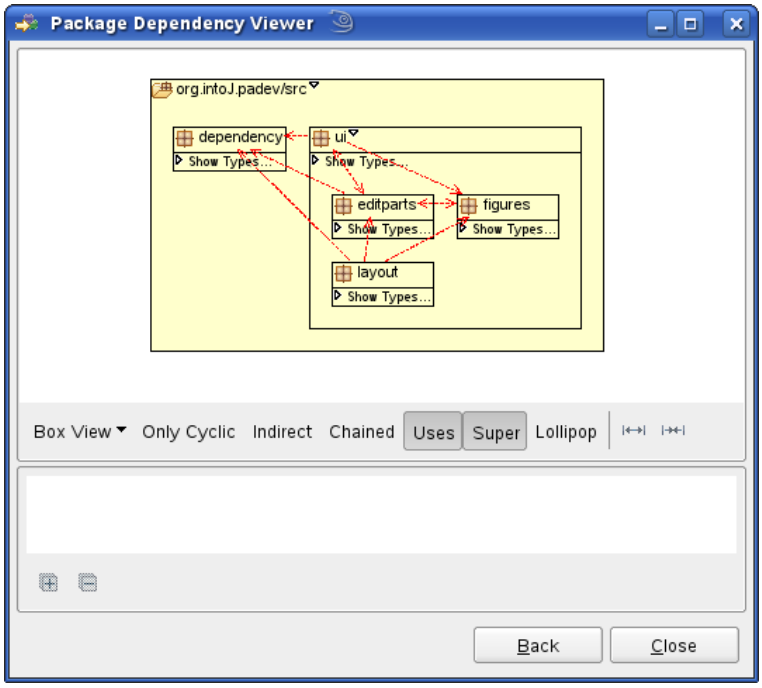

Abbildung 8: Die Box View.

- Durch Auswahl von „Only Cyclic“ kann bestimmt werden, dass nur jene Super-/Uses-Abhängigkeiten angezeigt werden, die Teil eines Zyklus im Abhängigkeitsgraph sind.
- Ob auch indirekte Uses-Abhängigkeiten (siehe Kapitel 3.1, Seite 9 ff.) angezeigt werden sollen, kann über die Schaltfläche „Indirect“ eingestellt werden.
- Uses-Abhängigkeiten, die durch verkettete Feldzugriffe oder Methodenaufrufe entstehen (siehe Kapitel 3.1, Seite 12), werden je nach Status des Buttons „Chained“ angezeigt oder nicht.
- In einem Paket deklarierte Interfaces, die Provided oder Required Interface (siehe Kapitel 3.3, Seite 21) eines in einem anderen Paket deklarierten Typen sind, werden bei aktiviertem „Lollipop“-Button angezeigt. Ihre Darstellung erfolgt jeweils links des Paketes, in dem sie deklariert sind oder, wenn dieses ausgeblendet ist, links neben dem in der Pakethierarchie ersten sichtbaren Oberpaket. Sie sind der Lollipop-Notation von UML (siehe [UML07], Seite 86 ff.) entsprechend mit jenen Paketen verbunden, in denen sie Typ eines Deklarationselements sind bzw. in denen sie implementiert werden. Zu Required Interfaces eines Paketes führen dabei durchgezogene Linien mit einem Halbkreis als Pfeilspitze (sogenannte „Sockets“), zu den Provided Interfaces durchgezogene Linien mit Kreis als Pfeilspitze („Lollipops“). Abbildung 9 zeigt ein Beispiel.

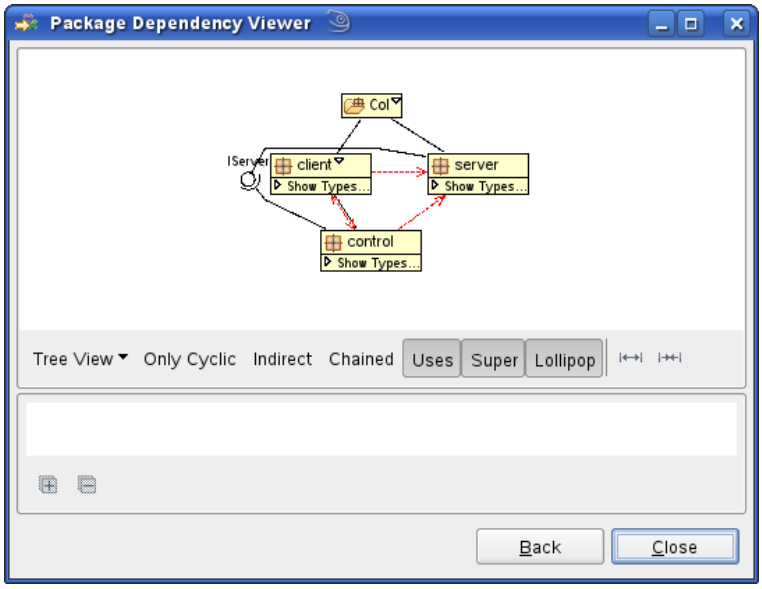

Abbildung 9: `IServer` ist Provided Interface des Paketes `server` und Required Interface von `control`.

Wie erwähnt, kann die Anzeige von Unterpaketen ein- und ausgeblendet werden. Sei M die Menge aller eingeblendeten Paketrepräsenentanten (also Baumknoten oder Kästen) einer View. $P(x)$ bezeichne das von $x \in M$ repräsentierte Paket. Sind die Kinder von x ausgeblendet, dann bezeichne $U(x)$ die Menge aller ausgeblendeten Kinder (sonst $U(x) = \emptyset$). Für disjunkte Elemente p und q aus M gilt:

Ein Pfeil von p nach q existiert genau dann, wenn

- eine Abhängigkeit von $P(p)$ zu $P(q)$ gefunden wurde oder
- eine Abhängigkeit von $P(p)$ zu einem $P(q')$ mit $q' \in U(q)$ gefunden wurde oder
- eine Abhängigkeit von einem $P(p')$ mit $p' \in U(p)$ zu $P(q)$ gefunden wurde oder
- eine Abhängigkeit von einem $P(p')$ mit $p' \in U(p)$ zu einem $P(q')$ mit $q' \in U(q)$ gefunden wurde.

Nach dem Anklicken eines Pfeiles werden Informationen zu den repräsentierten Abhängigkeiten im unteren Teil des Dialoges angezeit, dem sogenannten „List Panel“ (siehe Abbildung 10).

Die Listeneinträge enthalten zunächst eine Beschreibung der Paketabhängigkeiten. Genauere Informationen können dann eingeblendet werden; die Details zu Uses- und Super-Abhängigkeiten sind nach folgendem Schema aufgebaut:

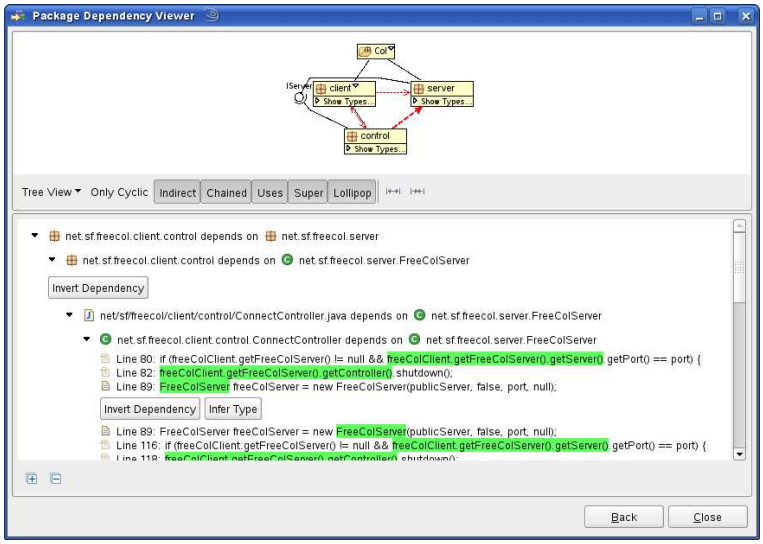

Abbildung 10: Details zu den Abhängigkeiten zwischen den Paketen `server` und `control`.

- Abhängigkeiten von einem Paket `a` zu einem Paket `b`
- Abhängigkeiten von `a` zu einem in `b` deklarierten Typ `B`
- Abhängigkeiten von einer Compilation Unit `CU.java` aus `a` zu `B`
- Abhängigkeiten von einem in `CU.java` deklarierten Typ `A` zu `B`
- Quelltextzeile(n), in der eine Abhängigkeit zwischen `A` und `B` entsteht

Die genaue Position im Quelltext, an der eine Abhängigkeit verursacht wurde, ist farblich hervorgehoben. Die Details zu Sockets und Lollipops beinhalten:

- Abhängigkeiten von einem Paket `a` zu einem Required/Provided Interface `I`
- Abhängigkeiten von einem in `a` deklarierten Typ zu `I`

Klickt der Benutzer auf eine Paketfigur, werden auf gleiche Weise all jene Abhängigkeiten im List Panel angezeigt, die Quelle oder Ziel dieses Paketes sind.

Für zahlreiche Uses-Abhängigkeiten `a` → `B` zwischen einem Paket `a` und einem Typ `B` kann, wie in Abschnitt 3.3 beschrieben, die Invert-Dependency-Methode angewendet werden, wenn in `a` mindestens ein Deklarationselement vom Typ `B` existiert. Die Detailanzeige enthält deswegen nach Möglichkeit einen Button „Invert Dependency“; wird er betätigt, wird Infer Type aufgerufen und mit allen nötigen Informationen versorgt, um einen innerhalb von `a` maximal verallgemeinerten Typ für Deklarationselemente vom Typ `B` zu bestimmen und einzuführen. Allerdings stellt Infer Type nicht die Möglichkeit bereit, für

generische Typen Interfaces einzuführen, die für alle innerhalb eines Paketes vorkommenden Deklarationselemente dieses Typs verwendet werden können (siehe [KEG07], Seite 52). Aus diesem Grund bietet Padev die Option Invert Dependency nicht an, wenn das Ziel ein gernerischer Typ ist. Durch Anwenden von Infer Type auf einzelne Deklarationselemente ist ein Entkoppeln des Typs unter Umständen aber dennoch möglich:

Unter den Quelltextzeilen stehen dort, wo die Anwendung von Infer Type auf ein Deklarationselement möglich ist, die beiden Buttons „Infer Type“ und „Invert Dependency“ zur Verfügung. Betätigen des erstgenannten startet Infer Type und ermöglicht das Einführen eines maximal verallgemeinerten Typs für das Deklarationselement auf die selbe Weise, wie wenn Infer Type im Editor gestartet werden würde. Details zum Vorgehen von Infer Type finden sich im fünften Kapitel von [KEG07].

„Invert Dependency“ ermöglicht das Umkehren von Abhängigkeiten zwischen Typen: Der inferierte Typ (der unter Umständen genau die Methoden deklariert, die der ursprüngliche Typ enthält) wird in das Paket eingefügt, das Quelle der Abhängigkeit ist. Im Gegensatz zum oben beschriebenen Invert-Dependency-Refactoring für Abhängigkeiten zwischen Paketen und Typen wird ein für das Deklarationselement maximal verallgemeinerter Typ berechnet, der nicht für alle Deklarationselemente des ursprünglichen Typs geeignet sein muss, die im Quellpaket vorhanden sind.

Nach dem erfolgreichen Refactoringschritt wird eine neue Suche nach Abhängigkeiten gestartet, um die Anzeige von Padev zu aktualisieren. Dabei muss im Allgemeinen der ursprüngliche Suchraum nicht komplett betrachtet werden, es genügen jene Pakete, zu oder von denen durch das Refactoring neue Abhängigkeiten entstanden sein könnten (siehe folgendes Kapitel, Seite 34).

4.2 Paketstruktur

Padev besteht aus 64 Klassen und Interfaces in zwölf Paketen und bedient sich des Model-View-Controller-Musters: Das Datenmodell repräsentiert Projekte mit ihren Paketen und Abhängigkeiten zwischen ihnen. Diese werden in der Tree oder Box View als Kästen und Pfeile zwischen ihnen dargestellt. Für Model-Objekte existieren Controller, die u. a. regeln, ob ein Kasten in der View ein- oder ausgeblendet wird. Die folgenden Abschnitte enthalten eine genauere Beschreibung der Pakete und deren Typen und erklären einige verwendete Konzepte. Die Namen der Pakete von Padev werden der besseren Lesbarkeit wegen gekürzt: Das Präfix `org.intoJ.padev` wird jeweils weggelassen; ein Paket, das den Namen `org.intoJ.padev.model.dependency` trägt, wird als `model.dependency` bezeichnet (Ausnahme: das Paket `org.intoJ.padev` wird nicht abgekürzt).

4.2.1 Die Pakete des Datenmodells

Die Typen des Datenmodells sind in den Paketen `model` und `model.dependency` abgelegt. Die Klasse `model.Padev` fungiert als die Wurzel des Modells: Sie repräsentiert ein Primärprojekt und eventuell weitere, vom Benutzer ausgewählte Projekte. Um auf die Projekte und deren Pakete und Typen zugreifen zu können, stehen Methoden bereit. Einige dieser Methoden, wie beispielsweise `getPackage(String)`, erwarten als Parameter einen String, der ein Java-Element (also z. B. das gewünschte Paket) eindeutig beschreibt. Dieser String wird von der Methode `getCanonicalName(IJavaElement)` erzeugt. Die Methode `startSearch(IProgressMonitor)` veranlasst die Suche nach allen Abhängigkeiten in den Paketen der ausgewählten Suchräume.

Alle nötigen Informationen über Java-Projekte stellt die Klasse `model.ProjectWrapper` zur Verfügung; sie erzeugt `model.PackageWrapper`-Objekte für alle im Projekt enthaltenen Pakete. Jedes `PackageWrapper`-Objekt unterhält Listen mit den Abhängigkeiten, deren Quelle oder Ziel es ist. Um alle von einem Paket abhängigen Pakete zu finden, wird die Methode `searchForDependencies(IProgressMonitor, Set<String>)` aufgerufen, die für jeden Typ ein `DependencySearcher`-Objekt erzeugt und ihn auf Abhängigkeiten zu anderen, im Zielsuchraum deklarierten Typen untersucht (siehe unten).

Die folgenden, Abhängigkeiten repräsentierenden Klassen implementieren das Interface `model.dependency.DependencyDescriber`: Ist ein Paket `a` von einem anderen `b` abhängig, wird ein Objekt des Typs `model.dependency.Dependency` erzeugt. Es enthält Objekte vom Typ `model.dependency.PackageToTypeDependency`, die Abhängigkeiten zwischen `a` und Typen aus `b` repräsentieren und unter anderem alle nötigen Informationen bereitstellen, die für das Invert-Dependency-Refactoring benötigt werden; aus diesem Grund implementiert `PackageToTypeDependency` das Interface `model.dependency.TypeDecouplingProvider`. Entsprechend der Gliederung der Details zu Abhängigkeiten (siehe Seite 27) enthält ein `PackageToTypeDependency`-Objekt Abhängigkeiten zwischen Compilation Units aus `a` und Typen aus `b` beschreibende `model.dependency.CUToTypeDependency`-Objekte. Objekte vom Typ `model.dependency.TypeToTypeDependency` entsprechen den Abhängigkeiten zwischen Typen und enthalten für jede Abhängigkeit, die bei der Quelltextanalyse gefunden wird, ein `model.dependency.DependencyInformation`-Objekt. Auf Basis dieser Objekte wird ein `model.dependency.SelectedDependency`-Objekt erzeugt, das die Infer-Type-Klasse `org.intoJ.inferType3.internal.constraintModel.util.SelectedElement` erweitert und Infer Type übergeben wird, sobald der Benutzer eines der erwähnten Refactorings anstößt.

Für die Provided und Required Interfaces werden `model.dependency.Lollipop-` bzw. `model.dependency.Socket`-Objekte erzeugt, die diese Abhängigkeiten näher beschreiben. `Socket` stellt als `TypeDecouplingProvider` die nötigen Informationen bereit, um das Invert-Dependency-Refactoring anzuwenden.

Die Suche nach Abhängigkeiten übernimmt `model.dependency.DependencySearcher`, eine Unterklasse von `org.eclipse.jdt.core.dom.ASTVisitor`. Ein `DependencySearcher`, der die Abhängigkeiten eines bestimmten Typs `a.A` von in anderen Paketen deklarierten Typen innerhalb des Zielsuchraums findet, besucht die Knoten des abstrakten Syntaxbaums („Abstract Syntax Tree“, kurz „AST“) der Compilation Unit von `A`. Wird in `A` eine Abhängigkeit von einem Typ `B` gefunden, wird überprüft, ob `B` in einem von `a` verschiedenen Paket `b` deklariert und `b` Teil des Zielsuchraums ist. Ist das der Fall, wird in den `a` und `b` repräsentierenden `PackageWrapper`-Objekten die Abhängigkeit $\texttt{a} \rightarrow \texttt{b}$ in Form eines `DependencyInformation`-Objektes in die erwähnten Listen eingetragen. Beim Besuchen der AST-Knoten wird je nach dessen Typ `T` eine Methode `visit(T)` aufgerufen. Einige dieser in `ASTVisitor` deklarierten und standardmäßig leeren Methoden sind in `DependencySearcher` überschrieben und untersuchen den jeweiligen Knoten auf die in Kapitel 3.1 beschriebenen Abhängigkeiten:

- **Referenzabhängigkeit:** Ob ein in einem anderen Paket des Zielsuchraums deklarierter Typ z. B. in einer Variablendeklaration referenziert wird, analysieren die Methoden `visit(SimpleName)` und `visit(QualifiedName)` analysieren mit Hilfe von `checkNameNode(Name, ITypeBinding)`.

- **(Indirekte) Abhängigkeit durch Aufrufen nichtstatischer Methoden:** Die Methode `visit(MethodInvocation)` wird immer dann aufgerufen, wenn ein AST-Knoten besucht wird, der einen Methodenaufruf der Form `Ausdruck.methodenname(Parameter1, ..., ParameterN)` repräsentiert.[14] Der Ausdruck kann fehlen, wenn eine Objektmethode in einem Typ `B` deklariert ist und in einem Typ `A` aufgerufen wird und entweder `A` gleich `B` ist, `A` ein innerer Typ von `B` ist oder `B` ein Supertyp von `A` ist. Im Kontext von Padev ist nur der letztgenannte Fall von Interesse, da nur dann eine Abhängigkeit zwischen Typen aus verschiedenen Paketen vorliegen kann. Ist also `B` ein Supertyp von `A`, sind `A` und `B` in verschiedenen Paketen deklariert und wird in `A` eine in `B` deklarierte Methode aufgerufen, wird ein `DependencyInformation`-Objekt erzeugt, das eine indirekte Abhängigkeit $\texttt{A} \rightarrow_i \texttt{B}$ repräsentiert.

  Ist ein Ausdruck vorhanden, dann ist dessen Typ Ziel einer Abhängigkeit – wie immer vorausgesetzt, er ist in einem anderen Paket im Zielsuchraum deklariert. Padev ignoriert sie allerdings, wenn der Ausdruck einem Methodenparameter, eine lokalen Variable oder einem Constructoraufruf entspricht. Warum, sei an folgendem Quelltextausschnitt erklärt:

[14] Ausnahme: Ist der Ausdruck das Schlüsselwort `super` (z. B. `super.foo()`), dann erkennt die Methode `visit(SuperMethodInvocation)` eine dadurch entstehende Abhängigkeit zum die Methode deklarierenden Supertyp.

```
1 public void foo(A someA) {
2   someA.foo();
3   B someB = new B();
4   someB.bar();
5   new C().someMethod();
6 }
```

Dass die Abhängigkeiten mit den Zielen `A`, `B` und `C` gefunden werden, ist bereits durch die Referenzen in den Zeilen 1, 3 und 5 sichergestellt. Im Rahmen von Padev ist es nicht von Bedeutung, dass durch die Methodenaufrufe in 2, 4 und 5 weitere Abhängigkeiten mit den Zielen `A`, `B` und `C` entstehen. Daher wird nur dann eine Abhängigkeiten zum Typ des Objektes registriert, dessen Methode aufgerufen wird, wenn das Objekt ein Feld ist oder von einer anderen Methode zurückgegeben wurde. In jedem Fall wird außerdem geprüft, ob eine Methode in einem Supertyp des Objekts deklariert ist oder in einem Subtyp überschrieben wird und dann ggf. indirekte Abhängigkeiten zu den Super-/Subtypen registriert.

- **Abhängigkeit durch Aufrufen statischer Methoden:** Wenn eine statische Methode unter Angabe der Klasse, in der sie deklariert ist, aufgerufen wird (z. B. `SomeType.foo()`), dann wird die Abhängigkeit zu dieser bereits durch die Klassenreferenz gefunden; Padev registriert dann keine neue Abhängigkeit. Anders, wenn die Referenz fehlt: das ist dann der Fall, wenn die Methode in der selben Klasse oder einer ihrer Oberklassen deklariert ist, in der sie aufgerufen wird, oder wenn sie statisch importiert wurde. Wiederum übernehmen `visit(MethodInvocation)` und `visit(SuperMethodInvocation)` die nötigen Prüfungen und finden eventuell vorliegende Abhängigkeiten.

- **(Indirekte) Abhängigkeit durch Zugriff auf nichtstatische Felder:** Durch Zugriff auf Objektvariablen entstehende Abhängigkeiten werden von der Methode `visit(SimpleName)` gefunden. Ähnlich wie bei Methodenaufrufen wird dabei überprüft, ob auf das Feld mit oder ohne vorangestellten Ausdruck zugegriffen wird und ob Abhängigkeiten zum das Feld deklarierenden Typ oder dem des Objekts, auf dessen Feld zugegriffen wird, bereits vorliegen (z. B. weil auf das Feld einer lokalen Variable zugegriffen wird; neue Abhängigkeiten werden dann wie bei den nichtstatischen Methodenaufrufen nicht registriert).

- **Abhängigkeit durch Zugriff auf statische Felder:** Wie bei statischen Methoden gilt: Feldzugriff sind nur dann von Interesse, wenn die deklarierende Klasse nicht explizit angegeben ist. Ist das der Fall, überprüft wiederum `visit(SimpleName)`, ob eine Abhängigkeit zur deklarierenden Klasse vorliegt.

- **Indirekte Zuweisungs- und Methodenargumentabhängigkeit:** Aus den oben genannten Gründen registriert Padev auch hier nur dann Abhängigkeiten, wenn ein Feld oder ein von einer Methode zurückgegebenes Objekt zugewiesen wird bzw. einer

Methode übergeben wird; beispielsweise wird die Zuweisung einer lokalen Variable wiederum ignoriert. Andernfalls überprüfen die Methoden `visit(SimpleName)` für Felder sowie `visit(MethodInvocation)` und `visit(SuperMethodInvocation)` für Methoden durch Aufrufen von `isAssignedOrArgument(ASTNode)`, ob neue Abhängigkeiten zwischen Typen aus verschiedenen Paketen vorliegen.

In den `DependencyInformation`-Objekten, die für die gefundenen Abhängigkeiten erzeugt werden, wird jeweils die Art der Abhängigkeit genauer festgehalten, indem der Methode `setType(int)` eine oder mehrere (durch den `|`-Operator verknüpften) der entsprechenden Konstanten übergeben wird, die in `Dependency` deklariert sind. Insbesondere können so Abhängigkeiten als verkettet (die Überprüfung auf Ketten erfolgt in der Methode `isChain(ASTNode)` in `DependencySearcher`) oder indirekt ausgezeichnet werden.

Für das Anwenden eines der möglichen Refactorings benötigt Infer Type die Position im Quelltext, an der der Typ eines Deklarationselementes angegeben ist. Bei Abhängigkeiten, die durch Referenzen entstehen, findet `checkNameNode(Name, ITypeBinding)` sie in der Compilation Unit, in der die Abhängigkeit entsteht. Anders bei Methodenaufrufen und Feldzugriffen: Gesucht ist hier die Position innerhalb jener Compilation Unit, in der Methode oder das Feld deklariert ist; diese enthält aber im Allgemeinen nicht den Typ, dessen AST beim Auffinden einer Abhängigkeit gerade betrachtet wird. Daher ist es nötig, den AST des deklarierenden Typs zu untersuchen. Es wäre jedoch ineffizient, dies sofort zu machen, sobald eine Abhängigkeit durch einen Methodenaufruf oder Feldzugriff gefunden wird, weil z. B. eine Methode innerhalb des Quellsuchraums mehrmals aufgerufen wird und dann die relativ zeitintensive Positionsbestimmung jedes Mal durchgeführt werden müßte. Aus diesem Grund wird vor jedem Start einer Abhängigkeitensuche ein `model.dependency.StartPositionSearcher`-Objekt erzeugt, in dem beim Auffinden von Abhängigkeiten die aufgerufene Methode bzw. das zugegriffene Feld (genauer: sie repräsentierende `org.eclipse.jdt.core.dom.IMethodBinding`- bzw. `org.eclipse.jdt.core.dom.IVariableBinding`-Objekte) sowie die zugehörigen `DependencyInformation`-Objekte gespeichert werden. Nachdem alle ASTs des Quellsuchraums auf Abhängigkeiten untersucht wurden, wird dann für jede dieser Methoden bzw. Felder die Startposition innerhalb der deklarierenden Klasse nur einmal ermittelt und in alle entsprechenden `DependencyInformation`-Objekte eingetragen.

Alternativ zu dem vorgestellten Verfahren zum Auffinden von Abhängigkeiten hätte auch der von Infer Type benutzte Algorithmus zum Bestimmen von Typconstraints (also Mindestanforderungen an den Typ von Deklarationselementen; Details dazu siehe [KEG07], Kapitel 4) wiederverwendet werden können. Er untersucht Typen `A` und deren Supertypen auf Verwendung in anderen Compilation Units. Statt also wie oben beschrieben alle Typen des Quellsuchraums auf Abhängigkeiten zu im Zielsuchraum deklarierten Typen zu analysieren, könnten alle Typen des Zielsuchraums mit Hilfe des Infer-Type-Algorithmus darauf überprüft werden, an welchen Stellen im Quellsuchraum sie benötigt werden. Das hätte den Vorteil, dass so nicht nur die Abhängigkeiten gefunden werden würden, son-

dern auch die für die Anwendung von Infer Type nötigen Constraints. Wenn z. B. das Invert-Dependency-Refactoring gestartet werden würde, müßten sie nicht neu berechnet werden, sondern könnten an Infer Type übergeben werden, was das Refactoring beschleunigen würde. Im Extremfall, wenn ein Projekt aus n Paketen mit jeweils einem Typen bestehen würde und in jedem dieser Typen alle anderen verwendet werden würden, müßten allerdings insgesamt $n*(n-1)$ Syntaxbäume untersucht werden, sofern sich alle vorhandenen Paketen im Quell- und Zielsuchraum befinden würden. Dieser Aufwand lohnt sich im Allgemeinen nicht, zumal alle Constraints außerdem neu berechnet werden müßten, sobald ein Refactoring durchgeführt werden würde. Im Rahmen von Padev interessieren außerdem primär Abhängigkeiten zwischen Paketen und nicht, auf welche Methoden und Felder einzelner Deklarationselemente zugegriffen wird. Deswegen ist das beschriebene Verfahren geeigneter für die Suche nach Abhängigkeiten als das von Infer Type verwendete.

Ein Invert-Dependency- oder Infer-Type-Refactoring verursacht Änderungen innerhalb des Primärprojektes. Dabei können neue Pakete und Compilation Units angelegt oder bestehende geändert werden. Während der Laufzeit von Padev können durch den Benutzer oder andere Eclipse-Plugins außerdem Compilation Units, Pakete oder Projekte gelöscht und Projekte angelegt werden. Um die Anzeige der Pakete und Abhängigkeiten immer aktuell zu halten, ist es nötig, auf diese Änderungen im Eclipse-Workspace zu reagieren. Die Klasse `Padev` implementiert aus diesem Grund `resourceChanged(IResourceChangeEvent)` und wird als `org.eclipse.core.resources.IResourceChangeListener` informiert, sobald eine der beschriebenen Änderungen eintritt. In den folgenden Fällen müssen Maßnahmen getroffen werden:

- Das Einfügen einer neuen, Ändern oder Löschen einer bestehenden Compilation Unit eines Paketes `p`, das im ursprünglichen Quell-Zielraum liegt, verursacht eine Abhängigkeitssuche von `p` zu allen anderen Paketen des ursprünglichen Ziel-Suchraums, wenn `p` ein Paket des Primärprojektes ist und zu allen Paketen des ursprünglichen Ziel-Suchraums, die im Primärpaket liegen, wenn `p` ein Paket eines externen Projekts ist.

- Wird eine Compilation Unit aus `p` geändert oder entfernt, werden zusätzlich all jene Pakete neu auf Abhängigkeiten zu `p` untersucht, die vor dem Ändern/Löschen abhängig von `p` waren.

- Wird ein Paket gelöscht oder ein neues angelegt, werden die Paketinformationen des zugehörigen Projekts und die Pakethierarchie aktualisiert.

- Das Löschen des Primärprojekts führt dazu, dass Padev beendet wird.

Alle Pakete, die nach Änderungen als neu auf Abhängigkeiten untersucht werden müssen, werden in das Feld `needUpdate` zusammen mit den möglichen Zielen eingetragen. Die Methode `model.Padev.updateDependencies(IProgressMonitor)` veranlasst dann eine neue Suche, um das Model auf den aktuellen Stand zu bringen. In den drei erstgenannten Fällen wird abschließend die Anzeige in der View dem Model angepasst.

4.2.2 Die Controller-Pakete

Als Schnittstelle zwischen Datenmodell und den GUI-Komponenten dienen verschiedene Klassen im Paket `ui.editparts`, die das Interface `org.eclipse.gef.EditPart` implementieren. Einige von ihnen werden den Bedürfnissen der beiden verfügbaren Views angepasst und von Klassen in den Paketen `ui.treeview.editparts` und `ui.boxview.editparts` überschrieben. Für welches Model-Objekt welcher `EditPart` erzeugt werden soll, regelt eine `PartFactory`-Instanz. Ihr wird die Wurzel des Datenmodells, also ein `Padev`-Objekt übergeben, für das sie ein `DiagramPart`-Objekt erzeugt. Jeder `EditPart` ist an genau ein Model-Objekt gebunden (die Umkehrung gilt im Allgemeinen nicht, z. B. werden für Pakete, die nicht Quelle oder Ziel von Abhängigkeiten sind, `PackageWrapper`-Objekte erzeugt, jedoch keine `EditPart`s) und enthält die Methode `getModelChildren()`, die eine Liste mit eventuell vorhandenen Kindern dieses Objekts erzeugt. Was `getModelChildren()` für `DiagramPart`s zurückgibt, hängt von der verwendeten View ab. Für die Tree View gilt: Ein vorhandenes `PackageWrapper`-Objekt ist genau dann in der Liste vorhanden, wenn eine Abhängigkeit gefunden wurde, deren Quelle oder Ziel dessen Paket ist oder wenn es ein Interface enthält, das als Provided oder Required Interface eines anderen Paketes erkannt wurde oder wenn es eine Wurzel in der Pakethierarchie ist. Bei der Box View sind die letztgenannten `PackageWrapper`-Objekte die einzigen, die als `Padev`-Kinder ausgezeichnet werden. Zusätzlich werden in beiden Views all jene `InterfaceWrapper`-Objekte in die Liste eingefügt, die Ziel eines Sockets oder Lollipops sind, sofern der Benutzer die Anzeige der Lollipop-Notation ausgewählt hat.

Für jedes von `getModelChildren()` zurückgegebene Element erzeugt die `PartFactory` dann ein `PackagePart`- bzw. ein `InterfacePart`-Objekt, deren gemeinsame Oberklasse `NodePart` ist. Nur die `PackageParts` der Box View können weitere Kinder enthalten: die `getModelChildren()`-Methode eines `BoxPackageParts` liefert all jene `PackageWrapper`-Objekte, die als Unterpakete innerhalb dessen Figur dargestellt werden sollen; für sie werden wiederum `PackagePart`-Objekte erzeugt.

Auf ähnliche Weise werden die Verbindungen in Form von `ConnectionEditParts` zwischen den `NodeParts` erzeugt: die beiden Methoden `getModelSourceConnections()` und `getModelTargetConnections()` liefern jeweils Listen mit den `Connection`-Objekten, deren Quelle bzw. Ziel das `PackageWrapper`- oder `InterfaceWrapper`-Objekt ist, das von dem `NodePart` repräsentiert wird. Wiederum erzeugt dann die `PartFactory` die entsprechenden Controller-Objekte.

4.2.3 Die View-Pakete

Die für die Benutzeroberfläche benötigten Klassen und Interfaces sind im Paket `ui` und einigen Unterpaketen abgelegt. In `ui` befindet sich die Klasse `PreferencesDialog`, die für die Darstellung des Dialogs zur Bestimmung der Suchräume verantwortlich ist. Sie sorgt für die Anzeige der verfügbaren Pakete und Projekte und teilt dem `Padev`-Objekt die vom Be-

nutzer ausgewählten mit. Als `org.eclipse.jface.dialogs.TitleAreaDialog` verfügt sie über eine Titelleiste, in der eine Warnung eingeblendet wird, wenn eines der ausgewählten Pakete Fehler enthält: die Abhängigkeitensuche ist unter Umständen dann nicht korrekt.

Die Darstellung des Dialogs mit der ausgewählten View und dem List Panel übernimmt eine Instanz von `GEFDependencyViewerPanel`, die nach der Abhängigkeitensuche vom `PreferencesDialog` erzeugt wird. Sie enthält außerdem die Buttons, die u. a. die Art der angezeigten Abhängigkeiten festlegen lassen.

Für das List Panel ist eine Instanz von `DependencyListPanel` verantwortlich: sie sorgt dafür, dass die Details zu ausgewählten Abhängigkeiten angezeigt werden. Die dazu erzeugten Listenelemente sind vom Typ `DependencyListItem`.

Die für die View nötigen Grafikelemente befinden sich im Paket `ui.figures` und erweitern verschiedene in `org.eclipse.draw2d` deklarierte Typen. Für die Darstellung von Paketen werden Objekte vom Typ `PackageFigure` verwendet; `DependencyConnectionFigure`-, `SocketFigure`- und `LollipopFigure`-Objekte übernehmen die Visualisierung der Abhängigkeiten. Sowohl Tree als auch Box View verwenden Instanzen dieser Typen, ordnen sie jedoch unterschiedlich an: Die für das Layout zuständigen Klassen befinden sich in den Paketen `ui.boxview.layout` und `ui.treeview.layout` und erweitern jeweils die beiden abstrakten Klassen `LayoutManager` und `DiagramLayout` aus dem Paket `ui.layout`.

Immer wenn die Anordnung der Grafikelemente neu berechnet werden muss (z. B. nachdem Subpackages ausgeblendet wurden oder nach einem Refactoring) wird die Methode `layout(IFigure)` des `LayoutManager`s aufgerufen, die mit Hilfe eines `DiagramLayout`s die Bestimmung der Positionen der Figuren und das Routing der Verbindungen zwischen ihnen vornimmt; für letzteres wird ein `org.eclipse.draw2d.graph.ShortestPathRouter` verwendet. Dazu muss festgestellt werden, welche Figuren überhaupt angezeigt werden sollen: Für Pakete und Interfaces liefert ein Aufruf von `isHidden()` des entsprechenden `NodePart`s die gewünschte Information. Anschließend werden die von den `NodePart`s ausgehenden `ConnectionPart`s untersucht, die entweder Uses-Abhängigkeiten, Super-Abhängigkeiten, Lollipops, Sockets oder Subpackage-Beziehungen (nur Tree View) repräsentieren. Die Figur einer Subpackage-Kante ist genau dann sichtbar, wenn das `NodePart`-Ziel sichtbar ist. Für die anderen Arten von `ConnectionPart`s gilt: sie entsprechen einer eine Abhängigkeit darstellenden Kante zwischen zwei Paketen oder einem Paket und einem Interface. Zunächst werden von all jenen Quellen und Zielen von `ConnectionPart`s, die einem Paket entsprechen (also vom Typ `PackagePart` sind), die in der Hierarchie ersten sichtbaren Pakete (siehe Seite 27) durch Aufrufen der in `DiagramLayout` deklarierten Methode `getVisiblePackage(PackagePart)` ermittelt. Sind jene von Quelle und Ziel identisch, wird die Figur des `ConnectionPart`s ausgeblendet. Andernfalls stehen nun zwei `NodePart`s fest, zwischen deren Figuren eine Verbindung angezeigt werden soll. Möglicherweise existiert bereits eine Kante (dargestellt mittels eines `DependencyConnectionFigure`-Objekts) zwischen ihnen, die eine Verbindung derselben Art (z. B. eine Super-Abhängigkeit) wie der

des aktuell betrachteten `ConnectionPart`s repräsentiert. Ist das der Fall, wird die Kante wiederverwendet: jenes `ConnectionPart`-Objekt, das die Kante erzeugt hat, unterhält ein Feld, in das andere `ConnectionPart`s eingetragen werden, deren Model-Objekte mittels dieser `DependencyConnectionFigure` dargestellt werden. Dadurch „weiß“ jede Kante, welche Abhängigkeiten sie darstellt.

4.2.4 Sonstige Pakete

Das Paket `util` stellt drei Hilfsklassen zur Verfügung: `LineInfo` erzeugt Quelltextausschnitte aus Compilation Units, in denen jene Textstellen farblich hervorgehoben sind, die eine Abhängigkeit verursachen. Sie werden im List Panel angezeigt. Um der Benutzeroberfläche Bilder hinzuzufügen, wird die Methode `getImage(String)` der Klasse `ImageCache` aufgerufen. Sie erzeugt ein `org.eclipse.swt.graphics.Image`-Objekt zur gewünschten Bilddatei und fügt es zusätzlich in ein internes Feld ein. Wird das selbe Bild ein zweites Mal angefordert, wird das zuvor gespeicherte `Image`-Objekt zurückgegeben. Das verhindert das Erzeugen redundanter Objekte. Um die Suche nach Abhängigkeiten zu beschleunigen, werden die benötigten Syntaxbäume (siehe Seite 31) ebenfalls zwischengespeichert. Die Methode `getAST(ICompilationUnit)` aus der Klasse `ASTCache` erzeugt sie nur dann neu und fügt sie in ein internes Feld ein, wenn sie erstmalig angefordert werden. Das kann die Suche nach Abhängigkeiten erheblich beschleunigen, führt aber dazu, dass mehr Arbeitsspeicher benötigt wird.

Im Paket `org.intoJ.padev` befinden sich drei Klassen: `Activator` zeichnet Padev als ein in die Eclipse-Benutzeroberfläche integrierbares Plugin aus. Dazu erweitert sie die Klasse `org.eclipse.ui.plugin.AbstractUIPlugin`. Die in `InvocationAction` deklarierte `run(IAction)`-Methode wird immer dann aufgerufen, wenn der Benutzer Padev durch Klicken auf den Menüeintrag „Package Dependencies...“ startet. Sie erzeugt ein `Padev`-Objekt und veranlasst die Darstellung des Einstellungsdialogs. Die vom Benutzer im Einstellungsdialog ausgewählten Optionen werden während der Laufzeit des Plugins in einem `Preferences`-Objekt gespeichert.

5 Evaluation

Ob ein Infer-Type- oder Invert-Dependency-Refactoring das gewünschte Resultat liefert, hängt von der Korrektheit von Infer Type ab. Die richtige Darstellung der gefundenen Abhängigkeiten wurde bei zahlreichen Anwendungen auf verschiedene Projekte und Suchräume getestet. Zu Überprüfen bleibt, ob die Abhängigkeitensuche korrekt ist, ob also genau die Abhängigkeiten gefunden werden, die in Kapitel 3.1 definiert wurden. Einerseits wurden diverse Projekte mit Padev analysiert und unsystematisch Abhängigkeiten zwischen Paketen aufgelöst, um die Robustheit auf die Probe zu stellen. Andererseits bedarf es systematischer Tests, wofür Projekte konstruiert wurden, in denen die unterschiedlichen Arten von Abhängigkeiten auftreten. Im Anschluss wurde überprüft, ob Padev genau diese findet und darstellt. Die Testprojekte finden sich auf der beiliegenden CD und sind nach folgendem Schema aufgebaut: Sie enthalten zwei Pakete `a` und `b`. In `a` befindet sich jeweils eine Klasse `Quelle`, die Abhängigkeiten zu den Klassen aus `b` verursacht. Im Kommentar hinter verschiedenen Ausdrücken ist jeweils angegeben, ob er eine Abhängigkeit der untersuchten Art verursachen soll. Zusammenfassend seien noch einmal die verschiedenen Arten genannt:

- Referenzabhängigkeiten treten immer dann auf, wenn ein Typ im Quelltext explizit erwähnt wird. Das geschieht dadurch, dass der Simple oder Qualified Name eines anderen Typs angegeben wird, der in einem anderen Paket deklariert ist.
- Abhängigkeiten durch Aufrufe nichtstatischer Methoden werden in den auf Seite 31 genannten Fällen ignoriert. Das Aufrufen einer nichtstatischen Methode `m()`, die in einem Typ `T` deklariert ist, muss damit genau dann als Abhängigkeit erkannt werden (vorausgesetzt, die Methode wird in einem anderen Paket als jenem von `T` aufgerufen), wenn
  - die Methode eines Feld aufgerufen wird, z. B. `someFieldOfTypeT.m()` (dies führt zu einer verketteten Abhängigkeit) oder
  - die Methode des Rückgabeobjekts einer anderen Methode aufgerufen wird, z. B. `getT().m()` (verkettete Abhängigkeit) oder
  - der aufrufende Typ ein Subtyp von `T` ist und explizit die Methode des Supertyps aufgerufen wird, z. B. `super.m()`.
- Wodurch indirekte Abhängigkeiten durch Aufrufe nichtstatischer Methoden entstehen können, zeigt folgendes Beispiel:

```
1  package a;
2  public class Quelle extends b.Ziel2 {
3    void foo(b.Ziel z) {
4      z.m();
5      n();
6    }
7  }
8
9  package b;
10 public class Ziel {
11   public void m() {}
12   public void n() {}
13 }
14
15 public class Ziel2 extends Ziel {
16   public void m() {}
17 }
```

In Zeile 4 wird eine Methode eines `Ziel`-Objektes aufgerufen, die in der Unterklasse `Ziel2` überschrieben wurde; dadurch ist `Quelle` indirekt abhängig von `Ziel2`. Weil `n()` in `Ziel` deklariert ist und `Quelle` eine Unterklasse von `Ziel` ist, die `n()` nicht überschreibt, entsteht in Zeile 5 eine Abhängigkeit zwischen `Quelle` und `Ziel`.

- Der Aufruf einer statischen Methode, die in einem anderen Paket deklariert ist, führt zu einer Abhängigkeit zu der die Methode deklarierenden Klasse – unabhängig davon, ob sie explizit angegeben ist (z. B. bei `SomeClass.someStaticMethod()`) oder nicht (`someStaticMethod()`). Im ersten Fall wird die Abhängigkeit bereits durch die Referenz auf `SomeClass` gefunden.

- Durch Zugriff auf ein nichtstatisches Feld entstehende Abhängigkeiten können wieder in einigen Fällen ignoriert werden, wie auf Seite 32 erklärt. Als Abhängigkeiten dieser Art erkannt werden Ausdrücke der Form `expression.someField`, wobei `expression` ein weiterer Feldzugriff oder ein Methodenaufruf sein kann und somit die Abhängigkeit verkettet ist.

- Analog zu dem Aufruf statischer Methoden gilt beim Zugriff auf statische Felder: Während die durch `SomeClass.someStaticField` verursachte Abhängigkeit als Referenzabhängigkeit erkannt wird, führt der Zugriff auf ein Feld ohne vorangestellte Typangabe zu einer Abhängigkeit zum deklarierenden Typ, sofern er sich in einem anderen Paket befindet.

- Eine indirekte Abhängigkeit durch Zugriff auf ein nichtstatisches Feld entsteht in folgendem Beispiel in Zeile 6, in Zeile 5 jedoch nicht:

```
1  package a;
2  public class Quelle {
3    void foo() {
4      Object o;
5      o = new b.Ziel().f;
6      o = new b.Ziel().f2;
7    }
8  }
9
10 package b;
11
12 public class Ziel extends Ziel2 {
13   public String f;
14 }
15
16 public class Ziel2 {
17   public String f2;
18 }
```

- Eine Zuweisung der Form `someVariable = someDeclarationElement` verursacht eine indirekte Abhängigkeit zum Typ des Deklarationselements auf der rechten Seite, wenn es ein Feld oder eine Methode ist. Lokale Variablen werden wiederum ignoriert.
- Analoges gilt für indirekte Methodenargumentabhängigkeiten.

Die Tests haben gezeigt: alle Abhängigkeiten, die gefunden werden sollen, werden auch gefunden. Keine Abhängigkeit, die nicht gefunden werden soll, wird gefunden.

6 Diskussion

Padev findet zuverlässig die in Kapitel 3 vorgestellten Abhängigkeiten zwischen Paketen, wie zahlreiche Tests gezeigt haben. Wann immer es möglich ist, kann deren Richtung mit Hilfe des Invert-Dependency-Refactorings umgekehrt werden. Nicht nur der Vergleich mit anderen Programmen, die ähnlichen Zwecken wie Padev dienen und von denen einige im folgenden Abschnitt vorgestellt werden, offenbart allerdings auch kleinere Mängel:

Verbesserungsbedarf besteht beim Layout der Anzeige: Gerade bei größeren Projekten und Suchräumen kann der Abhängigkeitsgraph unübersichtlich werden, vor allem durch das oft nicht optimale Rooting der Abhängigkeiten darstellenden Kanten und die Platzierung von Interface-Knoten bei der Lollipop-/Socket-Anzeige.

Im Rahmen dieser Arbeit interessieren weniger die Details, an welchen Stellen genau eine Abhängigkeit im Quelltext zwischen zwei Typen auftritt; zum Finden aller Abhängigkeiten zwischen Paketen und zum Umkehren von Abhängigkeiten zwischen einem Paket `a` und einem Typ `b.B` würde es reichen, in `a` ein Deklarationselement vom Typ `B` zu finden. Der gewählte Ansatz hat den Vorteil, dass auch für einzelne Deklarationselemente Infer Type aufgerufen und deren Typ verallgemeinert werden kann. Allerdings könnten, sobald ein zum Invertieren geeignetes Deklarationselement vom Typ `b.B` gefunden wurde, alle weiteren Abhängigkeiten zwischen `a` und `b.B` ignoriert werden, was die Suche beschleunigen und die Anzahl der erzeugten Projekte verringern würde – letzteres kann bei großen Projekten und wenig verfügbarem Arbeitsspeicher zu Problemen führen.

6.1 Verwandte Arbeiten

Zur Analyse von Abhängigkeiten zwischen Java-Paketen wurden bereits einige Werkzeuge veröffentlicht, von denen solche mit interessanten Ansätzen, die sie von Padev unterscheiden, im Folgenden vorgestellt werden. Sie haben gemein, dass sie Abhängigkeiten auf verschiedene Weise darstellen, jedoch im Gegensatz zu Padev keine Automatismen zu deren Auflösung anbieten.

JDepend[15] untersucht nicht wie Padev Java-Quelltext, sondern die .class-Dateien eines Java-Projektes auf Abhängigkeiten. Das hat den Vorteil, dass auch Abhängigkeiten zu solchen Paketen gefunden werden können, deren Quelltext nicht vorliegt (z. B. zu Java-Archiven im JAR-Format) und dass die Suche von vergleichsweise kurzer Dauer ist. Die gefundenen Abhängigkeiten präsentiert JDepend wahlweise als XML-Datei, in Textform oder mit Hilfe einer grafischen Benutzeroberfläche (siehe Abbildung 11). Mittels verschiedener Metriken, die in [MAR94] definiert sind, werden die Abhängigkeiten analysiert, um z. B. einen Eindruck von der Stabilität eines Paketes zu gewinnen; auch ob ein Paket Teil eines Abhängigkeitenzyklus ist, wird angezeigt.

[15] Version 2.9 online verfügbar unter http://www.clarkware.com/software/JDepend.html

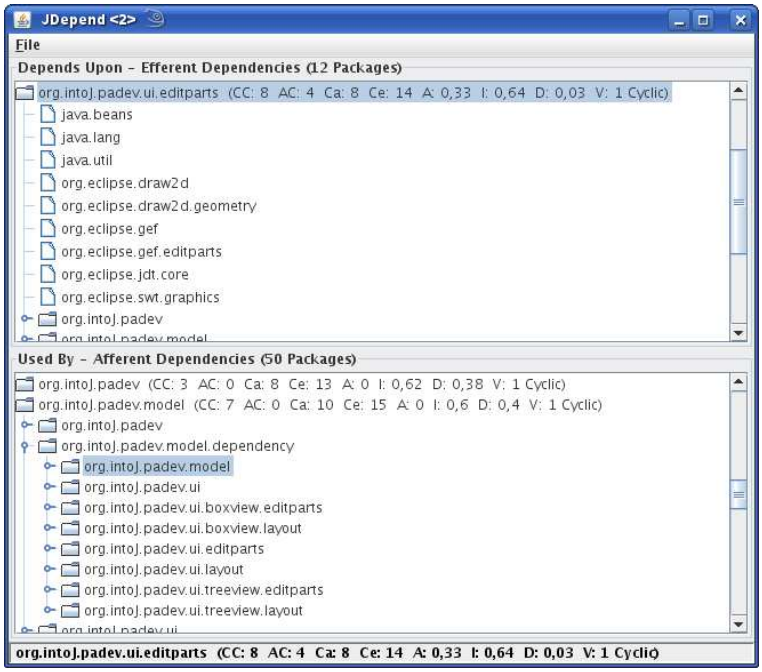

Abbildung 11: Der Grafikmodus von JDepend. Der obere Teil enthält jene Abhängigkeiten, deren Quelle die Pakete eines Projektes sind, der untere die Ziele dieser Abhängigkeiten.

JDepend findet zwar auch verkettete und einige der oben definierten Formen von indirekten Abhängigkeiten (z. B. jene, die beim Aufrufen einer in einem Supertyp deklarierten Methode entstehen, nicht jedoch indirekte Abhängigkeiten zu eine Methode überschreibenden Subtypen), hat aber Probleme mit Konstanten: Der folgende Quelltext enthält Klassen `A` und `B` aus verschiedenen Paketen; in `A` wird auf die in `B` deklarierte statische Konstante `BAR` zugegriffen, wodurch `a` abhängig von `b` ist.

```
1  package a;
2  public class A {
3    void m() {
4      String s = b.B.BAR;
5    }
6  }
7
8  package b;
9  public class B {
10   public final static String BAR = "";
11 }
```

Padev erkennt hier eine Referenzabhängigkeit, die JDepend nicht findet: Beim Erzeugen von A.class fügt der Compiler nicht eine Referenz auf die Konstante `BAR`, sondern deren Wert ein.

Metrics[16] ist wie Padev ein Plugin für Eclipse. Nach Auswahl eines Paketes stellt es alle ein- und ausgehenden Abhängigkeiten dar, wie Abbildung 12 zeigt. In einer weiteren Darstellung werden auch die Abhängigkeiten zwischen den Typen verschiedener Pakete angezeigt. Typen, die Ziel von besonders vielen Abhängigkeiten aus anderen Paketen sind, werden dabei hervorgehoben; dies dient der Hilfestellung bei der Entscheidung, ob ein Typ in ein anderes Paket verschoben werden soll.

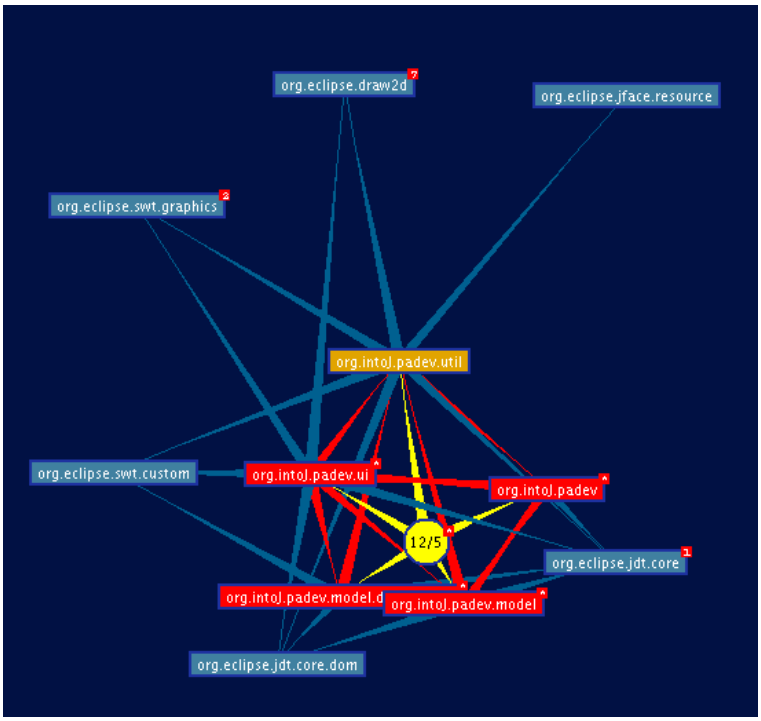

Abbildung 12: Analyse der Abhängigkeiten von und zu `org.intoJ.padev.util` mit Metrics. Rote Kanten zeigen Zyklen im Graph an.

Japan – Java Package Analyser[17] verwendet zum Finden von Abhängigkeiten einen Algorithmus, der Paketreferenzen im Quelltext sucht. Damit werden beispielsweise verkettete Abhängigkeiten nicht gefunden, solche zu Paketen, die im Quelltext in einem Kommentar erwähnt werden, jedoch fälschlicherweise schon.

Japan lässt sich in die Entwicklungsumgebung IntelliJ und in Apache Ant integrieren: Nachdem der Benutzer festlegt, zwischen welchen Paketen Abhängigkeiten erlaubt sind, überprüfen Ant und IntelliJ immer dann, wenn ein Projekt kompiliert wird, ob unerlaubte Abhängigkeiten vorhanden sind und melden dies gegebenenfalls. Dieser Ansatz eignet

[16] Version 1.3.6 online verfügbar unter http://metrics.sourceforge.net
[17] Version 0.2.4 online verfügbar unter http://japan.sourceforge.net

sich besonders als Hilfsmittel während der Entwicklung eines Programms, weniger bei der Wartung weit fortgeschrittener Projekte.

Dependency Finder[18] analysiert wie JDepend .class-Dateien und findet deswegen ebenfalls Abhängigkeiten nicht, die durch Zugriff auf Konstanten entstehen. Ähnlich wie bei Padev können auch Details zur Entstehung von Abhängigkeiten angezeigt werden; in Abbildung 13 ist beispielsweise zu erkennen, dass die Klasse `CircleAnchor` aus dem Paket `org.intoJ.padev.ui.editparts` von der Klasse `org.eclipse.draw2d.ChopboxAnchor` abhängt, weil deren Methode `getBox()` aufgerufen wird.

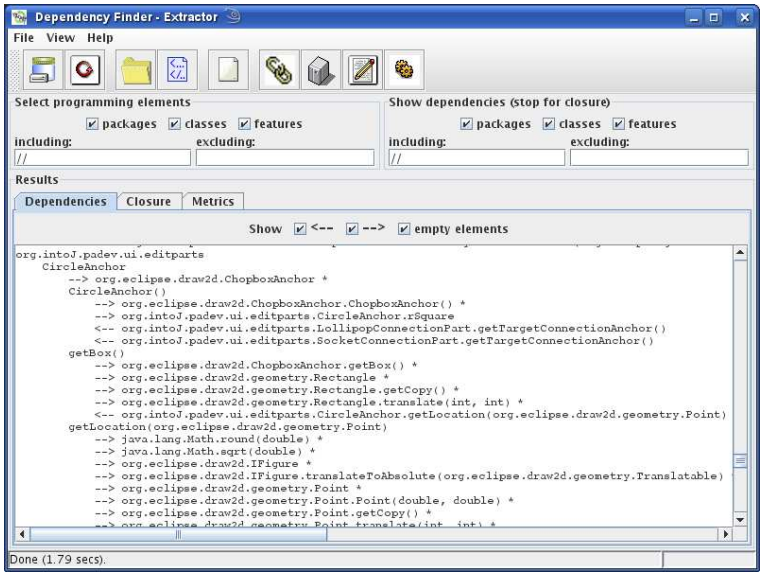

Abbildung 13: Darstellung einiger vom Paket `org.intoJ.padev.ui.editparts` ausgehender Abhängigkeiten mit Dependency Finder.

[18] Version 1.2.0 online verfügbar unter http://depfind.sourceforge.net

7 Schlussbetrachtungen

7.1 Zusammenfassung

Padev ist nicht nur wegen seiner Integration in die weitverbreitete Entwicklungsumgebung Eclipse ein alltagstaugliches Mittel, um Abhängigkeiten zwischen Java-Paketen darzustellen. Das Programm findet zuverlässig die in dieser Arbeit definierten Abhängigkeiten und kann sie in vielen Fällen im Gegensatz zu ähnlichen Programmen mit Hilfe von Infer Type auflösen. Dies ist zwar nicht immer vollständig möglich, durch weitere, in Padev zu integrierende Refactoringmechanismen, könnte deren Zahl aber noch erhöht werden (siehe folgender Abschnitt).

In der Darstellung des Abhängigkeitsgraphen unterscheidet sich Padev von ähnlichen Werkzeugen durch die Unterscheidung zwischen Super-Abhängigkeiten und Uses-Abhängigkeiten, für die weiteres Refactoring sinnvoll sein kann. Auch die Darstellung der Required und Provided Interfaces mittels der UML-Notationen Lollipop und Socket ist in verwandten Programmen nicht vorhanden.

7.2 Ausblick

Dadurch, dass Padev zum Invertieren von Abhängigkeiten Infer Type nutzt, „erbt“ es auch dessen Einschränkungen (siehe auch [KEG07], Kapitel 8.2). Beispielsweise kann Infer Type in der aktuellen Version keine innerhalb eines Paketes maximal verallgemeinerten Typen für generische Typen einführen, was Padev an der Möglichkeit hindert, das Invert-Dependency-Refactoring anzuwenden.

Das gewählte Verfahren zur Bestimmung von Abhängigkeiten untersucht den Quelltext von Java-Projekten. In der Realität ist dieser allerdings oftmals nicht vorhanden, wenn z. B. externe Projekte verwendet werden, von denen nur die kompilierten Dateien vorhanden sind. Sinnvoll ist daher eine Erweiterung von Padev um die Möglichkeit, auch Abhängigkeiten von und zu .class- oder .jar-Dateien zu finden.

In Kapitel 3.2 wurden einige Verfahren vorgestellt, wie Abhängigkeiten aufgelöst werden können. Deren Integration in Padev würde die Anzahl von Abhängigkeiten erhöhen, für die Refactorings zur Verfügung stehen. Zur Anwendung des Inject-Dependency-Refactorings wurde im Rahmen von [MON08] ebenfalls unter Zuhilfenahme von Infer Type ein Plugin für Eclipse entwickelt; dessen Einbindung in Padev wäre eine wertvolle Ergänzung. Auch für die durch Typenvergleiche mit `instanceof` oder Casts entstehenden Abhängigkeiten ist die Entwicklung geeigneter Werkzeuge sinnvoll.

8 Literaturverzeichniss

[FOW04] M. Fowler „Inversion of Control Containers and the Dependency Injection pattern“, 2004, online verfügbar unter http://martinfowler.com/articles/injection.html

[GOS05] J. Gosling, B. Joy, G. Steele, G. Bracha „The Java Language Specification, Third Edition“, 2005, online verfügbar unter http://java.sun.com/docs/books/jls/

[KEG07] H. Kegel „Constraint-basierte Typinferenz für Java 5“, Diplomarbeit, online verfügbar unter http://www.fernuni-hagen.de/ps/docs/Diplomarbeit-Kegel.pdf

[LIE89] K. J. Lieberherr, I. Holland „Assuring Good Style for Object-Oriented Programs“ in „IEEE Software, vol. 6, no. 5“, 1989, Seiten 38-48, online verfügbar unter http://citeseer.ist.psu.edu/lieberherr89assuring.html

[MAR94] R. C. Martin „OO Design Quality Metrics“, 1994, online verfügbar unter http://www.objectmentor.com/resources/articles/oodmetric.pdf

[MAR96] R. C. Martin „The Dependency Inversion Principle“, 1996, online verfügbar unter http://www.objectmentor.com/resources/articles/dip.pdf

[MAR05] R. C. Martin „Design Principles and Design Patterns“, 2000, online verfügbar unter http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[MON08] M. Mondl, „Ein Refaktorisierungswerkzeug für die Injektion minimierter Abhängigkeiten“, Master-Arbeit, online verfügbar unter http://www.fernuni-hagen.de/ps/arbeiten/master_mondl.html

[STE06] F. Steimann, P. Mayer, A. Meißner „Decoupling classes with inferred interfaces“ in „Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)“, 2006, Seiten 1404-1408, online verfügbar unter http://www.fernuni-hagen.de/ps/pubs/SAC2006.pdf

[STE07] F. Steimann „The Infer Type Refactoring and its Use for Interface-Based Programming“, in „Journal of Object Technology, vol. 6, no. 2, Special Issue OOPS Track at SAC 2006“, 2007, Seiten 99-120, online verfügbar unter http://www.jot.fm/issues/issue_2007_02/article5

[UML07] verschiedene „OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2“, 2007, online verfügbar unter http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/

A Inhalt der beiliegenden CD

Datei/Verzeichnis	Beschreibung
javadoc	Die Quelltextdokumentation von Padev
org.intoJ.padev	Der Padev-Quelltext als Eclipse-Projekt
Testprojekte	Die in Kapitel 5 zur Evaluation verwendeten Projekte
ausarbeitung.pdf	Dieses Dokument im PDF-Format
epl-v10.html	Der Text der Eclipse Public License - v 1.0
org.intoJ.inferType3_1.0.8.jar	Das Infer-Type-Plugin als JAR-Datei
org.intoJ.padev_1.1.0.jar	Das Padev-Plugin als JAR-Datei

B Erklärung

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

Scheyern, 25.09.2008, Sebastian Hauf