

**FernUnivesität in Hagen**

**Fachbereich Mathematik und Informatik**

**Lehrstuhl Programmiersysteme**

Abschlussarbeit im Studiengang Bachelor of Science im Fach Informatik

**Ein Eclipse-Plugin zum Aufspüren toter  
Codefragmente auf Basis von  
Sichtbarkeitsconstraints**

Autor: Anastasia Pasenkova

Betreuer: Prof. Dr. Friedrich Steimann, Dipl.-Inf. Andreas Thies

# Inhaltsverzeichnis

1	Einleitung und Motivation.....	1
1.1	Aufbau der Arbeit.....	1
2	Das Problem toten Codes.....	2
2.1	Arten von totem Code.....	2
2.2	Auswirkungen von totem Code.....	3
3	Sichtbarkeitsconstraints als Grundlage für die Erkennung toten Codes.....	8
3.1	Das Constraintsystem .....	8
3.1.1	Constraint-Grundlagen.....	9
3.1.2	Sichtbarkeitsconstraints.....	10
3.1.3	Constraints des CDCD-Plugin.....	14
3.1.4	Implementierung des Constraint-Systems.....	16
4	Aufbau und Funktionsweise des CDCD-Plugins.....	17
4.1	Anforderungen an den CDCD.....	17
4.2	Entfernen toten Codes als Refactoring.....	20
4.3	Implementierung des Plugins.....	20
4.3.1	Paketstruktur.....	20
4.3.2	Programmablauf.....	21
4.3.2.1	Start der Suche.....	21
4.3.2.2	Die Klasse org.intoj.cdcd.CDCDetector.....	22
4.3.2.3	Aufbau und Lösung des Constraintsystems.....	23
4.3.2.4	Analyse der Lösung.....	26
4.3.2.5	Darstellung der Ergebnisse.....	29
4.3.3	Entfernen toten Codes.....	33
4.3.3.1	QuickFixes.....	33
4.3.3.2	Refactoring in Eclipse.....	36
4.3.3.3	CDCD-Refactorings.....	37
4.3.3.4	Entfernen aller toten Deklarationselemente.....	45
4.3.4	Theoretische Aspekte .....	45
4.3.4.1	Eindeutigkeit der optimalen Lösung .....	45
4.3.4.2	Notwendigkeit der Änderung aller Sichtbarkeiten.....	47
5	Analyse des CDCD-Plugins.....	49
5.1	Anwendungsszenarien und Ergebnisse.....	49
5.2	Verifikation der Funktionsweise des CDCD-Plugin.....	51

5.3 Qualitative Analyse der Ergebnisse.....	52
5.3.1 Projekt codec.....	52
5.3.2 Projekt DeadCodeDetector.....	53
5.3.3 Projekt jaxen.....	53
5.3.4 Projekt jbook.....	54
5.3.5 Projekt Jester.....	54
5.3.6 Projekt JHotDraw.....	55
5.3.7 Projekt schemalizer.....	55
5.4 Laufzeitverhalten.....	56
5.5 Bewertung der Ergebnisse.....	56
6 Existierende Ansätze zum Aufspüren toten Codes in Java-Programmen.....	58
6.1 Analyse von Java-Bytecode.....	58
6.1.1 Aufbau eines Aufrufgraphen.....	58
6.1.2 Aufbau eines Abhängigkeitsgraphen.....	61
6.1.3 Analyse des abstrakten Syntaxbaums.....	61
6.1.4 Weitere Ansätze .....	62
6.2 Analyse von Java-Sourcecode.....	63
6.2.1 Eliminierung toten Codes als Compiler-Optimierung.....	63
6.2.2 Dead Code Detektoren als Hilfswerkzeuge für Softwareentwickler.....	64
6.3 Einschränkungen statischer Programmanalyse.....	65
7 Zusammenfassung und Ausblick.....	66
7.1 Ausblick.....	67
7.1.1 API-Elemente.....	67
7.1.2 Test-Klassen.....	67
7.1.3 Unbenutzte Interface-Methoden.....	68
7.1.4 Reflection.....	68
Abbildungsverzeichnis.....	69
Listingverzeichnis.....	69
Tabellenverzeichnis.....	70
Literaturverzeichnis.....	71
Anhang A Bedienungsanleitung für das CDCD Plugin.....	73
Anhang B Inhaltsverzeichnis der beiliegenden CD.....	80

# 1 Einleitung und Motivation

Die vorliegende Arbeit befasst sich mit dem Problem toten Codes in Java-Programmen, speziell mit deklarierten Typen, Methoden und Attributen, die in einem Programm an keiner Stelle verwendet werden. Die Existenz solcher Programmelemente kann verschiedene Ursachen haben. Meist handelt es sich um Programmteile, deren Funktionalität im Lauf der Programmentwicklung überflüssig geworden ist, weil sie von anderen Teilen übernommen wurde oder weil sie ursprünglich nur zu Testzwecken vorhanden war. In [Tip02] wird auch die Modularisierung von Programmen als mögliche Ursache genannt: wenn Komponenten einer Anwendung unabhängig voneinander von verschiedenen Entwicklern entworfen und implementiert werden, kann die von anderen Komponenten genutzte Funktionalität nicht immer genau vorhergesehen werden, was die Existenz überflüssiger Methoden in der Endkomponente nach sich ziehen kann.

Moderne IDEs wie Eclipse [EclURL] warnen zwar, wenn private Codeelemente nicht verwendet werden, die Erkennung unbenutzter Elemente anderer Sichtbarkeit bleibt jedoch dem Entwickler überlassen. Der in dieser Arbeit vorgestellte Ansatz zum Aufspüren solcher Codeelemente basiert ebenso auf statischer Analyse von Java-Sourcecode, bei der die zulässige Sichtbarkeit jedes Programmelementes durch Constraints bestimmt wird. Tote Elemente sind demnach diejenigen, für die durch kein Constraint eine minimale Sichtbarkeit gefordert wird. Praktisch umgesetzt wurde das Sichtbarkeits-Constraint-Modell in [Ste09]. In der vorliegenden Arbeit wurde auf dieser Grundlage ein Plugin für Eclipse entwickelt, das unter Benutzung der Sichtbarkeit-Constraint-Funktionalität tote Codeelemente eines Java-Projektes identifiziert, die einen Zugriffsmodifikator besitzen können – eine Bedingung, die aus dem Umstand resultiert, dass in Java die Sichtbarkeit von Elementen durch Zugriffsmodifikatoren, bzw. deren Abwesenheit, festgelegt wird. Dieses Plugin mit dem Namen Constraint based Dead Code Detector, kurz CDCD, verfolgt das Ziel, Entwickler durch die Möglichkeit automatischer Erkennung unbenutzter Funktionalität bei der Wartung des Programmcodes zu unterstützen und bietet die Möglichkeiten, tote Codeelemente auszukommentieren oder zu löschen.

## 1.1 Aufbau der Arbeit

Das anschließende Kapitel 2 verschafft einen Überblick über das Problem toten Codes, in Kapitel 3 werden die Sichtbarkeitsconstraints beschrieben, auf deren Grundlage das Aufspüren toter Codeelemente im CDCD-Plugin erfolgt. Die Implementierungsdetails, sowie die Funktionsweise des CDCD-Plugins werden in Kapitel 4 erläutert. Kapitel 5 befasst sich

mit der Leistungsanalyse des CDCD-Plugins bei Anwendung auf verschiedenen Java-Projekten. In Kapitel 6 werden die bereits existierenden Ansätze, toten Code in einem Programm ausfindig zu machen, beschrieben und ein Vergleich mit dem CDCD angestellt. Kapitel 7 schließt die Arbeit mit einer Zusammenfassung und dem Ausblick ab.

## 2 Das Problem toten Codes

### 2.1 Arten von totem Code

In der Literatur findet sich keine einheitliche Definition des Begriffs „toter Code“, es ist dabei aber immer von Code die Rede, der im Programm nicht benutzt wird und dessen Entfernung keine Auswirkungen auf die Programmfunktion hat, wohl aber zur Reduzierung der Größe und besserem Laufzeitverhaltens des Programms, sowie zur Verbesserung der Übersichtlichkeit und Verständlichkeit des Programmcodes, beitragen kann.

Toter Code kann unerreichbaren Code mit einschließen, d.h. Code, der nie ausgeführt werden kann, da kein Pfad im Kontrollfluss zu ihm führt [Deb00], vgl. Listing 1:

```
public void m() {  
    int i = 1;  
  
    if (i > 0) {  
        // ...  
    } else {  
        // wird nie erreicht  
    }  
}
```

*Listing 1: Unerreichbarer Code*

Innerhalb von Methoden werden Deklarationen nicht referenzierter lokaler Variablen als toter Code bezeichnet. Auch Anweisungen, die zwar ausgeführt werden, aber keinen Einfluss auf die Programmfunktion haben oder Berechnungen, deren Ergebnisse nicht weiter im Programm verwendet werden, können als toter Code betrachtet werden [Deb00]. Diese Arten von totem Code, dargestellt in Listing 2, werden bereits durch den Java Compiler [JavURL] erkannt und bei der Übersetzung des Source- in Bytecode verworfen.

```

public void m() {
    int i;

    // Unbenutzte lokale Variable
    int j = 0;

    // Unnötige Berechnung, n wird nicht mehr verwendet
    int n = 0;
    for (i=0; i< 10; i++){
        n++;
    }

    // Überflüssige Anweisung i = 1+1;
    i = 1+1;
    i = 0;
}

```

*Listing 2: Toter Code in Methoden*

Manche Compiler, z.B. der Eclipse Compiler [EclURL], können auch unbenutzten Code auf Klassenebene, wie private Attribute, Methoden und innere Klassen, erkennen. Das in dieser Arbeit vorgestellte CDCD-Plugin erweitert Eclipse um die Möglichkeit, tote Deklarationselemente mit beliebigen Zugriffsmodifikatoren aufzuspüren. Im Einzelnen handelt es sich dabei um Klassen, Interfaces, Enum-Typen, Methoden (inklusive Konstruktoren), sowie Attribute und Enum-Konstanten, die im umschließenden Projekt an keiner Stelle benutzt werden, was so viel bedeutet, dass sie weder referenziert werden, noch dass deren Existenz nötig ist, um die syntaktische und semantische Korrektheit des Programms zu gewährleisten.

## 2.2 Auswirkungen von totem Code

In erster Linie wirkt sich die Existenz von totem Code in einem Programm negativ auf die Wartbarkeit und Verständlichkeit des Sourcecode aus. Wie die Beispiele aus Abschnitt 2.1 zeigen, kann die Funktion einer Methode mit totem und unerreichbarem Code nicht so einfach nachvollzogen werden, was die Fehleranalyse erschwert.

Eine potentielle Fehlerquelle, was die Semantik des Programms angeht, ist in Listing 3 dargestellt, das einen Ausschnitt aus der Klasse `NullHandle` des Projekts<sup>1</sup> `JHotDraw`<sup>2</sup> zeigt.

<sup>1</sup> Die Versionen aller Projekte, auf die im Folgenden verwiesen wird, sind in Tabelle 1 (Kapitel 5.1) aufgeführt.

<sup>2</sup> <http://www.jhotdraw.org/>

```

public class NullHandle extends LocatorHandle {

    /**
     * The handle's locator.
     */
    protected Locator fLocator;

    public NullHandle(Figure owner, Locator locator) {
        super(owner, locator);
    }
    // ...
}

```

*Listing 3: Unbenutztes Attribut als potentielle Fehlerquelle*

Hier wird ein Attribut `fLocator` deklariert, obwohl in der Superklasse ein gleichnamiges Attribut existiert, das als `private` deklariert ist. Das neue Attribut wird nirgendwo initialisiert und an keiner Stelle im Programm benutzt, daher sollte es entfernt werden – Zugriffe von neuen Subklassen aus könnten zu unerwarteten Ergebnissen, im schlimmsten Fall einer `NullPointerException`, führen.

Das Projekt `JHotDraw` enthält weitere Beispiele von totem Code, der zur Fehleranfälligkeit eines Programms beiträgt und zudem Entwicklern die Einarbeitung in das Projekt erschwert.

So finden sich in der Klasse `CommandMenu` folgende zwei Methoden:

```

public synchronized void remove(Command command) {
    throw new JHotDrawRuntimeException("not implemented");
}

public synchronized void remove(MenuItem item) {
    throw new JHotDrawRuntimeException("not implemented");
}

```

*Listing 4: Unbenutzte Methoden der Klasse CommandMenu*

Hier scheint es sich um einen Fall zu handeln, in dem eine Funktionalität geplant war, jedoch später aufgegeben wurde, die angelegten Methoden aber erhalten geblieben sind.

Ein ähnlicher Fall findet sich in der Klasse `PaletteButton`: hier ist eine Methode `value()` deklariert, die lediglich `null` zurückgibt und nirgendwo benutzt wird. Womöglich war in der Klasse die Möglichkeit vorgesehen, einen bestimmten Wert zu speichern, der mittels dieser Methode erfragt werden kann, ist aber nicht weiter verfolgt worden.

Im Projekt `schemalizer`<sup>3</sup> ist ein weiteres Beispiel zu finden: in der Klasse `SimpleTypeInferer` hat die Methode `buildValidatorSubElements()` einen leeren Rumpf. Auch die Umstrukturierung von Code kann zur Existenz toter Codeelemente führen: die Klasse `StandardDrawingView` des Projekts `JHotDraw` enthält die Methoden `keyTyped(KeyEvent e)` und `keyReleased(KeyEvent e)`, beide mit leerem Rumpf. Dem

<sup>3</sup> <http://schemalizer.sourceforge.net/>

Namen der Methoden nach handelt es sich um Methoden des Interfaces `java.awt.event.KeyListener`, das die Klasse wohl einst implementiert hat. In der vorliegenden Version enthält `StandardDrawingView` eine innere Klasse, die das `KeyListener`-Interface implementiert. Dem Anschein nach wurde beim Löschen des `implements`-Statements das Entfernen dieser beiden Methoden vergessen. Weitere Beispiele für toten Code, der aus der Ersetzung einer Funktionalität durch eine andere resultiert, finden sich in den Listings 5 und 6:

```
/**
 * Handles a change of the current selection. Updates all
 * menu items that are selection sensitive.
 * @see DrawingEditor
 */
public void figureSelectionChanged(DrawingView view) {
    setupAttributes();
}

public void viewSelectionChanged(DrawingView oldView,
                                DrawingView newView) {
}
```

*Listing 5: Obsolete Methode der Klasse `DrawApplet`*

Hier sind zwei Listener-Methoden deklariert, von denen die zweite einen leeren Rumpf hat und nirgendwo benutzt wird, jedoch immer noch im Programmcode vorhanden ist.

```
/**
 * Open a new window for this application containing the
 * passed in drawing, or a new drawing if the passed
 * in drawing is null.
 */
public void newWindow(Drawing initialDrawing) {
    DrawApplication window = createApplication();
    if (initialDrawing == null) {
        window.open();
    } else {
        window.open(window.createDrawingView(initialDrawing));
    }
}

public final void newWindow() {
    newWindow(createDrawing());
}
```

*Listing 6: Obsolete Methode der Klasse `DrawApplication`*

Die Methode `newWindow()` hat zwar keinen leeren Rumpf, wie in den vorherigen Beispielen, wurde vom CDCD-Plugin aber als unbenutzt eingestuft. Die Existenz der überladenen Methode, sowie die Abwesenheit eines Javadoc-Kommentars lässt vermuten, dass `newWindow()` im Lauf der Entwicklung ersetzt, jedoch nicht gelöscht wurde.

Ähnliche Beispiele finden sich auch in anderen Projekten – Listing 7 zeigt einen Ausschnitt aus der Klasse State des Projekts jbook<sup>4</sup>. Die laut CDCD toten Attribute in den Zeilen 2 bis 7 werden nach dem Einführen des Attributs props (Zeile 8) nicht mehr benötigt, da der Zustand nun in den Properties und nicht in den Attributen gespeichert wird. Die Attribute wurden jedoch nicht aus dem Programm entfernt und es wird, da sie als paketlokal deklariert sind, keine Warnmeldung von Eclipse über unbenutzte Attribute gemeldet.

In Abb. 1 ist das unbenutzte Interface XMLFileChangeListener des Projekts schemalizer

```
1: public class State {
2:   Color colorBackground = Color.black;
3:   Color colorText = Color.yellow;
4:   int iX = -1;
5:   int iY = -1;
6:   int iWidth = -1;
7:   int iHeight = -1;
8:   Properties props = new Properties();
9:   // ...
10:  public void setWidth(int iWidth) {
11:    setIntProperty("width", iWidth);
12:  }
13:
14:  public int getWidth() {
15:    return getIntProperty("width", -1);
16:  }
17:  // ...
18: }
```

Listing 7: Obsolete Attribute im Projekt jbook

zu sehen. Im Package Explorer View von Eclipse ist das Interface XMLFileClosedListener markiert, das vermutlich als Ersatz des XMLFileChangeListener-Interfaces eingeführt wurde, da nur das Ereignis „Schließe XML-Datei“ sich für das Programm als relevant herausgestellt hat.

---

4 <http://jbook.sourceforge.net/>

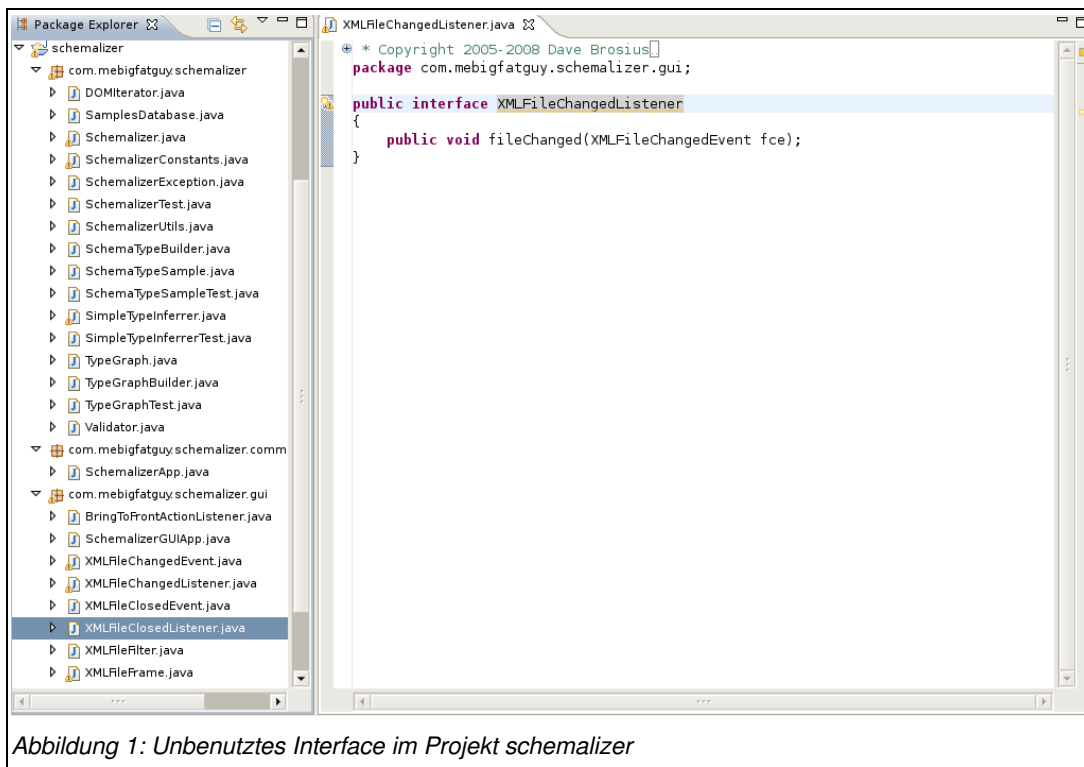


Abbildung 1: Unbenutztes Interface im Projekt schemalizer

All diese Beispiele zeigen, dass toter Code die Fehleranfälligkeit eines Programms erhöhen kann, wenn in der Weiterentwicklung des Programms von Komponenten Gebrauch gemacht wird, die keine sinnvolle Funktion mehr haben, es aber keine direkten Hinweise im Sourcecode auf diesen Umstand gibt.

Unbenutzter Code hat auch Auswirkungen auf die Größe des Programms, vor allem, wenn es sich um Code von externen Bibliotheken handelt, von denen nur ein geringer Teil im Programm verwendet wird. Die Entfernung unbenutzter Bibliotheksfunktionalität trägt dazu bei, dass die Größe von im Internet übertragbaren Programmen, wie Java Applets, und damit deren Übertragungszeit minimiert wird, indem nur so viel Code wie nötig mit übertragen wird [Tip03]. Auch bei eingebetteten Systemen spielt die Programmgröße eine wichtige Rolle, da hier Ressourcen wie Rechenleistung, sowie zur Verfügung stehender Arbeits- und Datenspeicher im Vergleich zu herkömmlichen PCs stark eingeschränkt sind [Tip99].

## 3 Sichtbarkeitsconstraints als Grundlage für die Erkennung toten Codes

Das im Rahmen dieser Arbeit entwickelte CDCD-Plugin stützt sich auf das am Lehrgebiet Programmiersysteme der FernUniversität Hagen entwickelte Sichtbarkeits-Constraint-Modell, mit dessen Hilfe die Erhaltung der syntaktischen und semantischen Korrektheit eines Programms bei Änderungen von Zugriffsmodifikatoren überwacht werden kann. Die semantische Korrektheit wird dabei mit dem Bestehen der JUnit-Tests des Programms gleichgesetzt.

Der restriktivste Zugriffsmodifikator in Java ist `private`, es folgen `package` (wobei paketlokaler Zugriff durch Abwesenheit eines Modifikators ausgedrückt wird), `protected` und `public`. Im Constraint-Modell wird ein weiterer Modifikator, den minimal mögliche, – `absent`, verwendet. Bekommt ein Deklarationselement den Modifikator `absent` zugewiesen, bedeutet dies, dass es nicht vorhanden sein braucht oder darf, so dass es entfernt werden kann bzw. muss. Die Definition und Verwendung des Modifikators `absent` stellen die Grundlage für das Auffinden toter Codefragmente dar.

Ein Anwendungsmöglichkeit dieses Constraint-Modells ist die Berechnung möglichst restriktiver Zugriffsmodifikatoren für die Deklarationselemente eines Java-Programms, mit dem Ziel, die Kapselung des Programms zu verbessern, wie es in [Bou08] mit Hilfe des Vorläufers des Constraint-Modells angestrebt wurde.

Eine weitere Anwendungsmöglichkeit ist die Optimierung von Refactoring-Tools, dessen Ausführung Änderungen von Zugriffsmodifikatoren nach sich ziehen kann, ein Beispiel solch eines Refactorings ist das Verschieben von Elementen. Dadurch dass für jedes Deklarationselement festgelegt wird, welche Zugriffsmodifikatoren zulässig sind, wird eine zusätzliche Vorbedingung dieser Refactorings definiert, und somit die Fehlerrate bei deren Ausführung verringert. [Ste09]

Der Constraint-basierte Ansatz zur Modellierung der Sichtbarkeit von Deklarationselementen aus [Ste09] wird im folgenden Abschnitt kurz vorgestellt.

### 3.1 Das Constraintsystem

Die Java-Sichtbarkeitsregeln werden in [Ste09] durch Constraints modelliert, in denen Zugriffsmodifikatoren Variablen sind. Die Grundlage dieser Constraints bildet die Java-Sprachspezifikation. Jedes Sichtbarkeits-Constraint grenzt den Wert einer oder mehrerer Variablen ein, indem Abhängigkeiten zwischen Variablen festgelegt oder Konstanten als

Werte zugewiesen werden. Der Wert einer Variablen kann dabei durch mehrere Constraints eingeschränkt werden. Das Constraintsystem stellt die Gesamtheit aller Constraints eines Programms dar, seine Lösung besteht aus Variablenzuweisungen, mit denen jedes Constraint erfüllt ist, wobei es es keine oder unterschiedlich viele Lösungen geben kann [Ste09]. Gibt es mehrere Lösungen, muss aus diesen die beste bestimmt werden. Beim Auffinden totter Deklarationselemente ist es offensichtlich diejenige, die die meisten absent-Zuweisungen enthält.

Bei der Constraintgenerierung werden den Variablen Anfangswerte zugewiesen, so dass das Constraintsystem zu Beginn den Ausgangszustand des Programms widerspiegelt. Ein aus einem (syntaktisch und semantisch) korrekten Programm generiertes Constraintsystem hat immer eine Lösung, da alle Constraints mit den Ausgangsbelegungen der Variablen erfüllt sein müssen, sofern die Constraint-Regeln konsistent sind. Im Umkehrschluss bedeutet dies, dass alle Zuweisungen, bei denen das Constraintsystem erfüllt ist, ein korrektes Programm repräsentieren, sofern die Constraint-Regeln vollständig sind [Ste09].

### 3.1.1 Constraint-Grundlagen

In den Sichtbarkeits-Constraints werden folgende in [Ste09] definierten Mengen verwendet:

$D$	Menge der Deklarationselemente $d_1, d_2, \dots$ usw.
$R$	Menge der Referenzen $r_1, r_2, \dots$ usw.
$C \subset D$	Menge aller Klassen
$I \subset D$	Menge aller Interfaces
$M \subset D$	Menge aller Methoden (inklusive Konstruktoren)
$F \subset D$	Menge aller Attribute (fields),
$S$	Menge aller als static deklarierten Elemente
$A$	Menge der Zugriffsmodifikatoren
$L$	Menge der Orte (locations) $l_1, l_2, \dots$ usw.

Die Menge  $A$  ist total geordnet und enthält die Elemente  $\{ public, protected, package, private, absent \}$ .

Dabei gilt  $public < protected < package < private < absent$ , wobei ' $<$ ' eine geringere Einschränkung der Sichtbarkeit und damit erweiterten Zugriff bedeutet.<sup>5</sup> [Ste09] Der Zugriffsmodifikator eines Deklarationselements  $d$  wird mit  $\langle d \rangle$  dargestellt.

Weiterhin werden folgende Hilfsfunktionen benötigt, die in [Ste09] beschrieben sind:

<sup>5</sup> In [Ste09] war die Menge in der umgekehrten Reihenfolge geordnet, bei der Implementierung des Constraintsystems wurde aus technischen Gründen die aktuelle Ordnung gewählt.

### **Bindung**

$\beta : R \rightarrow D$ , wobei  $\beta(r) = d$  bedeutet, dass die Referenz  $r$  an das Deklarationselement  $d$  gebunden wird.

### **Ort (location)**

$$\lambda : D \cup R \rightarrow L$$

Jedes  $d \in D$ , sowie jedes  $r \in R$  ist an einem Ort  $l \in L$  deklariert, bzw. zu finden. Beispielsweise ist der Ort einer deklarierten Klasse das die Klasse enthaltende Paket.

### **Sichtbarkeit**

$$\alpha : L \times L \rightarrow A$$

Das erste Argument ist hierbei der Ort der Referenz, das zweite der Ort des referenzierten Deklarationselements. Die Funktion  $\alpha(\lambda(r), \lambda(d))$  berechnet also den kleinsten Zugriffsmodifikator, der  $r$  den Zugriff auf  $d$  gewährt.

### **Empfänger (receiver)**

Für ein  $d \in M \cup F$  kann der Ort der Referenz auf  $d$  nicht ausreichend für die Bestimmung der Sichtbarkeit von  $d$  sein, da der statische Typ, durch den auf  $d$  zugegriffen wird, ebenfalls sichtbar sein muss. Dieser Umstand wird durch die Funktion  $\rho : R \rightarrow L$  modelliert, die den Ort entsprechen dem Rumpf des Empfänger-Typen bestimmt.

## **3.1.2 Sichtbarkeitsconstraints**

Im Constraint-System sind 13 Constraints implementiert, die in [Ste09] beschrieben sind. Diese Constraint-Regeln zur Einschränkung möglicher Zugriffsmodifikatoren im Zusammenhang mit der Erreichbarkeit deklarerter Elemente, Vererbung, Suptyping, dynamischer Bindung, sowie weitere spezielle Constraints aus [Ste09] werden im Folgenden kurz skizziert.

### **Erreichbarkeit**

Der Zugriffsmodifikator eines Deklarationselements  $d$  muss die Erreichbarkeit von einer Referenz  $r$  aus gewährleisten, was durch folgendes Constraint, Acc-1 (vom englischen accessibility), ausgedrückt wird:

$$\beta(r) = d \rightarrow \langle d \rangle \leq \alpha(\lambda(r), \lambda(d)) \quad \text{Acc-1}$$

Beim Zugriff auf als `protected` deklarierte Elemente eines Typen, muss zusätzlich sichergestellt werden, dass paketübergreifender Zugriff auf diese Elemente nur von seinen Subtypen aus möglich ist. Es wird also ein zweites Erreichbarkeits-Constraint benötigt, das restriktiver als `Acc-1` in diesem Spezialfall ist. Dieses Constraint ist `Acc-2`:

$$\beta(r) = d \wedge \alpha(\lambda(r), \lambda(d)) = \text{protected} \wedge d \notin S \wedge \rho(r) \notin \text{subclasses}(\lambda(r)) \cup \{\lambda(r)\}$$

$$\rightarrow \langle d \rangle = \text{public} \qquad \text{Acc-2}$$

Mit  $\text{subclasses}(\lambda(r))$  wird die Menge aller Orte, die den Rümpfen echter Subklassen der Klasse, dessen Rumpf  $\lambda(r)$  entspricht, bezeichnet.

### Vererbung

Laut den Sichtbarkeitsregeln von Java muss auf geerbte Elemente genauso wie auf entsprechende Elemente der Superklasse zugegriffen werden können. Das `Acc-1`-Constraint ist nicht immer ausreichend, wie folgendes Beispiel aus [Ste09] zeigt:

```
package a;
public class A {
    protected int i;

    protected void n() {
        (new b.B()).i = 1;
    }
}

package b;
public class B extends a.A {
}
```

Listing 8: Nötige Sichtbarkeit bei Vererbung

Hier muss `i` mindestens die Sichtbarkeit `protected` haben, damit es von `B` geerbt werden kann und der Zugriff auf `i` in `n()` möglich ist. Der minimale erforderliche Zugriffsmodifikator bei der Referenzierung von der Methode `A.n()` aus ist jedoch `private`. Damit in diesem Fall die Sichtbarkeit von `i` nicht reduziert wird und Vererbung bestehen bleibt, wird das Constraint, `Inh-1` (inheritance), das den statischen Typen des Empfängers berücksichtigt, benötigt:

$$\beta(r) = d \wedge \rho(r) \neq \lambda(d) \rightarrow \langle d \rangle \leq \alpha(\rho(r), \lambda(d)) \qquad \text{Inh-1}$$

Inh-1 ist also eine Verschärfung von Acc-1 für den Spezialfall, dass der Empfänger sich an einem anderen Ort als das referenzierte Deklarationselement befindet.

Ein weiteres Problem, das im Zusammenhang mit Vererbung auftreten kann, betrifft den Zugriff auf statische Attribute. Wird ein statisches Attribut  $s$  in einer Klasse  $A$  als `private` und in einem Interface  $I$  ein gleichnamiges Attribut als `public` deklariert, darf die Sichtbarkeit von  $s$  in  $A$  nicht erweitert werden, wenn eine Klasse  $B$  sowohl von  $A$ , als auch von  $I$  erbt und ein Zugriff auf  $s$  erfolgt: nach Erweiterung der Sichtbarkeit von  $A$ ,  $s$  kann der Zugriff nicht mehr eindeutig aufgelöst werden. Dieser Umstand wird durch das Constraint Inh-2 ausgedrückt:

$$\begin{aligned} \{d, d'\} \subseteq F \cap S \wedge \iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in \text{superclasses}(\rho(r)) \\ \rightarrow \langle d \rangle > \alpha(\lambda(r), \lambda(d')) \end{aligned} \quad \text{Inh-2}$$

Hier bezieht sich  $\iota(d)$  auf den unqualifizierten Bezeichner von  $d$ , und *superclasses* auf die Menge der Superklassen, analog zu *subclasses* im Constraint Acc-2.

### **Subtyping**

In Java darf die Sichtbarkeit überschriebener oder verdeckter Methoden nicht reduziert werden, was durch die Constraint-Regel Sub-1 ausgedrückt wird, wobei *overrides* für das Überschreiben und *hides* für das Verdecken steht:

$$\{d, d'\} \subseteq M \wedge \text{overrides}(d, d') \vee \text{hides}(d', d) \rightarrow \langle d' \rangle \leq \langle d \rangle \quad \text{Sub-1}$$

Eine weitere Regel, Sub-2, die sich aus Vererbung ergibt, besagt, dass von einer Klasse, die ein Interface implementiert, geerbte Methoden, die Sichtbarkeit `public` haben müssen, wenn es sich um Interface-Methoden handelt:

$$\begin{aligned} \{d, d'\} \subseteq M \wedge \text{subsignature}(d', d) \wedge \{c, c'\} \subseteq C \wedge i \in I \wedge \lambda(d) = i \wedge \lambda(d') = c' \\ \wedge \text{implements}(c, i) \wedge \text{inherits}(c, d', c') \rightarrow \langle d' \rangle = \text{public} \end{aligned} \quad \text{Sub-2}$$

Dabei ist *subsignature*( $d', d$ ) wahr, wenn  $d$  die gleiche Methodensignatur wie  $d'$  hat, oder wenn die Signatur von  $d$  den gleichen Namen wie  $d'$  hat und die Typen der Methodenparameter von  $d$  Type Erasures der Typen der Methodenparameter von  $d'$  sind, d.h. die Typen müssen ohne Berücksichtigung generischer Elemente übereinstimmen

[GosURL]. Mit  $implements(c, i)$  ist gemeint, dass die Klasse  $c$  das Interface  $i$  implementiert, mit  $inherits(c, d', c')$  – dass die Klasse  $c$  die Methode  $d'$  von der Klasse  $c'$  erbt.

### **Dynamische Bindung**

Damit dynamische Bindung möglich ist, muss eine überschriebene Methode der Superklasse in der Subklasse, in der sie überschrieben wird, sichtbar sein, was folgendes Beispiel aus [Ste09] demonstriert:

```

package a;
class A {
    void m() {}

    void n() {m(); }
}

package a;
class B extends A {
    void m() {}
}

```

*Listing 9: Nötige Sichtbarkeit bei dynamischer Bindung*

Die Sichtbarkeit von  $A.m()$  darf nicht auf `private` reduziert werden, da sonst die Semantik des Programm verletzt wird: der Aufruf von  $m()$  in  $B.n()$  wird dann auf  $A.m()$ , statt auf  $B.m()$ , aufgelöst. Dieser Fall wird durch das Constraint Dyn-1 verhindert:

$$overrides(d, d') \rightarrow \langle d \rangle \leq \alpha(\lambda(d'), \lambda(d)) \quad \text{Dyn-1}$$

In [Ste09] wird hierzu angemerkt, dass es prinzipiell unmöglich ist, durch statische Analyse zu entscheiden, ob semantische Fehler nach dem Verlust dynamischer Bindung tatsächlich auftreten, da der dynamische Typ des Empfängers erst zur Laufzeit bekannt ist. Dyn-1 ist daher ein konservatives Constraint, das den Verlust dynamischer Bindung in allen Fällen verbietet.

Auch versehentliches Hinzufügen dynamischer Bindung kann zu semantischen Fehlern im Programm führen. Wäre die Methode  $A.m()$  im oberen Beispiel als `private` deklariert, dürfte ihre Sichtbarkeit nicht erweitert werden, da sonst der Aufruf von  $m()$  in  $A.n()$  auf  $B.m()$  statt auf  $A.m()$  aufgelöst würde. In diesem Fall kommt das Constraint Dyn-2 zum Einsatz:

$$\lambda(d) \in superclasses(\lambda(d')) \wedge subsignature(d', d) \wedge \neg overrides(d, d') \rightarrow \langle d \rangle > \alpha(\lambda(d'), \lambda(d)) \quad \text{Dyn-2}$$

### ***Weitere Constraintregeln***

In [Ste09] finden sich noch sogenannte vorausschauende Constraints, die die nötigen Sichtbarkeiten beim Verschieben von Deklarationselementen festlegen: hier müssen besondere Vorkehrungen getroffen werden, z.B. wenn durch das Verschieben Deklarationselemente verdeckt werden. Beim Aufspüren toten Codes spielen die vorausschauenden Constraint jedoch keine Rolle, da nur die aktuellen Sichtbarkeiten relevant sind und kein Verschieben von Deklarationselementen stattfindet.

Weiterhin kommen noch Constraints hinzu, die sich direkt aus der Sprachspezifikation von Java ergeben:

- **Main-Constraint:** Main-Methoden werden nicht direkt referenziert, müssen aber die Sichtbarkeit `public` haben, da sie Einsprungspunkte in das Programm darstellen. Die Sichtbarkeit von Klassen, die eine main-Methode enthalten, darf ebenfalls nicht reduziert werden.
- Das **Same-Declaration-Constraint** besagt, dass alle Attribute, die innerhalb einer Deklaration stehen, die gleiche Sichtbarkeit haben müssen.
- **Api-Constraint:** Es kann auch andere Einsprungspunkte als die main-Methode, sowie Elemente, die in ihrer Sichtbarkeit nicht eingeschränkt werden dürfen, in einem Programm geben. Bei der Generierung von Constraints werden daher auch Annotationen betrachtet: mit `@API` können Einsprungspunkte festgelegt werden, für die, wie für main-Methoden, nur der Zugriffsmodifikator `public` erlaubt ist; für Elemente, die mit `@SIC` annotiert sind, fordert das API-Constraint, dass deren Sichtbarkeit nicht verändert werden darf. Elemente, die mit den JUnit-4-Annotationen `@Test`, `@Before` oder `@After` annotiert sind, sowie JUnit-Testklassen, die von `TestCase` erben, müssen die Sichtbarkeit `public` haben.

### **3.1.3 Constraints des CDCD-Plugin**

Das Acc-1-Constraint verhindert, dass tote Deklarationen als solche erkannt werden, wenn zyklische Abhängigkeiten zwischen ihnen bestehen, es aber keine Referenzen auf diese Elemente von außerhalb des Zyklus gibt [Ste09]. Aber auch bei einseitiger Abhängigkeit werden nicht alle toten Elemente bei Anwendung von Acc-1 und Inh-1 erkannt: wenn ein Deklarationselement  $d$  nur von toten Elementen referenziert wird, verlangen diese Constraints trotzdem, dass  $d$  eine minimale Sichtbarkeit hat. Für die Constraints Dyn-1 und Sub-2 gilt ebenfalls, dass eine minimale Sichtbarkeit für  $d$ , bzw.  $d'$  gefordert wird, selbst wenn  $d'$  bzw.  $d$  der Modifikator `absent` zugewiesen ist.

Aus diesem Grund wurde ein neuer Constraint-Typ, `AccessModifierRelationConstraint`, als Subtyp von `AccessModifierConstraint` implementiert, zu dem folgende Constraints gehören: `Acc-1-Relation`, `Inh-1-Relation`, `Dyn-1-Relation`, `Sub-2-Relation` und `Sub-1-Relation`. Diese Constraints, in denen auch die Sichtbarkeit des Deklarationselementes, das die Menge der zulässigen Zugriffsmodifikatoren eines anderen Elements einschränkt, berücksichtigt wird, ersetzen die entsprechenden bestehenden Constraints beim Aufspüren toten Codes. In einigen der neuen Constraints wird die Funktion `absent` benutzt, die wie folgt definiert ist:

$$absent : R \rightarrow \{ true, false \}$$

$absent(r)$  wird dabei zu `true` ausgewertet, wenn für ein  $r \in R$  und ein  $d \in D$  gilt:

$$\lambda(r) = \lambda(d) \wedge \langle d \rangle = absent$$

Die Constraints `Acc-1-Relation` und `Inh-1-Relation` sehen folgendermaßen aus:

$$\beta(r) = absent \wedge \neg absent(r) \rightarrow \langle d \rangle \leq \alpha(\lambda(r), \lambda(d)) \quad \textbf{Acc-1-Relation}$$

$$\beta(r) = d \wedge \rho(r) \neq \lambda(d) \wedge \neg absent(r) \rightarrow \langle d \rangle \leq \alpha(\rho(r), \lambda(d)) \quad \textbf{Inh-1-Relation}$$

Die Constraints sind erfüllt, wenn das Deklarationselement  $d'$ , in dem  $r$  zu finden ist, beliebige Sichtbarkeit<sup>6</sup> hat und  $\langle d \rangle \leq \alpha(\lambda(r), \lambda(d))$  bzw.  $\langle d \rangle \leq \alpha(\rho(r), \lambda(d))$  gilt. Ist  $d'$  der Zugriffsmodifikator `absent` zugewiesen (was mit  $absent(r)$  ausgedrückt wird), darf  $d$  beliebige Sichtbarkeit haben. Hier zu fordern, dass  $\langle d \rangle$  dabei ebenfalls `absent` ist, würde zu einem unlösbaren Constraintsystem führen, wenn andere Constraints eine minimale Sichtbarkeit für  $d$  fordern.

`Dyn-1-Relation` und `Sub-2-Relation` sehen dementsprechend wie folgt aus:

$$overrides(d', d) \wedge \langle d' \rangle \neq absent \rightarrow \langle d \rangle \leq \alpha(\lambda(d'), \lambda(d)) \quad \textbf{Dyn-1-Relation}$$

$$\{ d, d' \} \subseteq M \wedge signature(d', d) \wedge \{ c, c' \} \subseteq C \wedge i \in I \wedge \lambda(d) = i \wedge \lambda(d') = c'$$

$$\wedge implements(c, i) \wedge inherits(c, d', c')$$

$$\wedge \langle d \rangle \neq absent \rightarrow \langle d' \rangle = public \quad \textbf{Sub-2-Relation}$$

---

<sup>6</sup> wobei seine Sichtbarkeit natürlich durch andere Constraints eingeschränkt werden kann

Werden Methoden überschrieben, verlangt das Constraint Sub-1, dass die Sichtbarkeit dieser Methoden nicht reduziert werden darf. Das hat zur Folge, dass diese Methoden selbst dann nicht als unbenutzt gelten, wenn die Klasse, in der sie deklariert sind, den Modifikator *absent* besitzt, wodurch weitere tote Deklarationselemente unentdeckt bleiben können, sofern sie nur in diesen Methoden referenziert werden. Für das Aufspüren toten Codes muss Sub-1 also durch ein Constraint ersetzt werden, in dem die Sichtbarkeit solcher Methoden auch von der Sichtbarkeit der überschriebenen Methode abhängt. Ist diese Methode in einer Bibliotheksklasse deklariert, wird das obige Problem jedoch nicht gelöst, hier muss die Sichtbarkeit der Methode von der der umschließenden Klasse abhängen. Einen Sonderfall stellen dabei Methoden von anonymen Klassen dar: hier ist die Sichtbarkeit des Deklarationselements, in dem die anonyme Klasse deklariert ist, relevant. Das Constraint Sub-1-Relation, das im CDCD-Plugin anstelle von Sub-1 eingesetzt wird, lautet wie folgt:

$$\begin{aligned} & \{d, d'\} \subseteq M \wedge \text{overrides}(d', d) \vee \text{hides}(d', d) \wedge \\ & (\lambda(d') \in C \wedge \langle \lambda(d') \rangle \neq \text{absent} \vee \text{anonymous}(\lambda(d')) \wedge \lambda(\lambda(d')) \in M \cup C \wedge \\ & \langle \lambda(\lambda(d')) \rangle \neq \text{absent}) \rightarrow \langle d' \rangle \leq \langle d \rangle \end{aligned} \quad \text{Sub-1-Relation}$$

Der zweite neue Constraint-Typ ist *Acc2RelationConstraint* und ersetzt den bisher eingesetzten Typ *AccessModifierAcc2Constraint*, denn das Constraint Acc-2 muss für das Aufspüren toten Codes analog zu den vorherigen Constraints durch ein Constraint ersetzt werden, in dem die Sichtbarkeit von *d* nur dann *public* sein muss, wenn das Deklarationselement, von dem es referenziert ist, nicht den Modifikator *absent* besitzt:

$$\begin{aligned} & \beta(r) = d \wedge \alpha(\lambda(r), \lambda(d)) = \text{protected} \wedge d \notin S \wedge \rho(r) \notin \text{subclasses}(\lambda(r)) \cup \{\lambda(r)\} \wedge \\ & \neg \text{absent}(r) \rightarrow \langle d \rangle = \text{public} \end{aligned} \quad \text{Acc-2-Relation}$$

### 3.1.4 Implementierung des Constraint-Systems

Alle Constraints werden in einem Schritt, beim Traversieren des abstrakten Syntaxbaumes (AST), generiert. Alle in den Constraints vertretenen Hilfsfunktionen bis auf  $\alpha$  sind mit Hilfe der entsprechenden JD-T-Methoden implementiert. Für die Lösung des Constraintsystems wird der Constraint-Löser Cream [CreURL] verwendet. [Ste09]

## 4 Aufbau und Funktionsweise des CDCD-Plugins

### 4.1 Anforderungen an den CDCD

Ein Werkzeug zur Erkennung toten Codes lässt sich vor allem dann effektiv einsetzen, wenn es in eine IDE integriert ist. Aus diesem Grund wurde der CDCD als Plugin für Eclipse<sup>7</sup>, eines der meist genutzten IDEs<sup>8</sup>, entwickelt. Neben einer komfortablen Möglichkeit für die Suche nach unbenutzten Deklarationselementen standen dabei eine übersichtliche Darstellung der Ergebnisse, einfache Navigation über gefundene Elemente, sowie die automatische Entfernung toten Codes, im Vordergrund. Um diesen Anforderungen gerecht zu werden, wurde Eclipse um die hierfür nötigen Elemente unter Nutzung des Extension Point-Mechanismus<sup>9</sup> erweitert. So wurde unter anderem ein Kontextmenü-Eintrag, über den die Suche nach totem Code gestartet werden kann, hinzugefügt.

Um bestimmte Stellen im Quellcode hervorzuheben, z.B. um den Entwickler auf (potentielle) Fehler aufmerksam zu machen, werden in Eclipse Marker eingesetzt: Abb. 2 zeigt einen Marker, der eine unbenutzte Methode markiert.

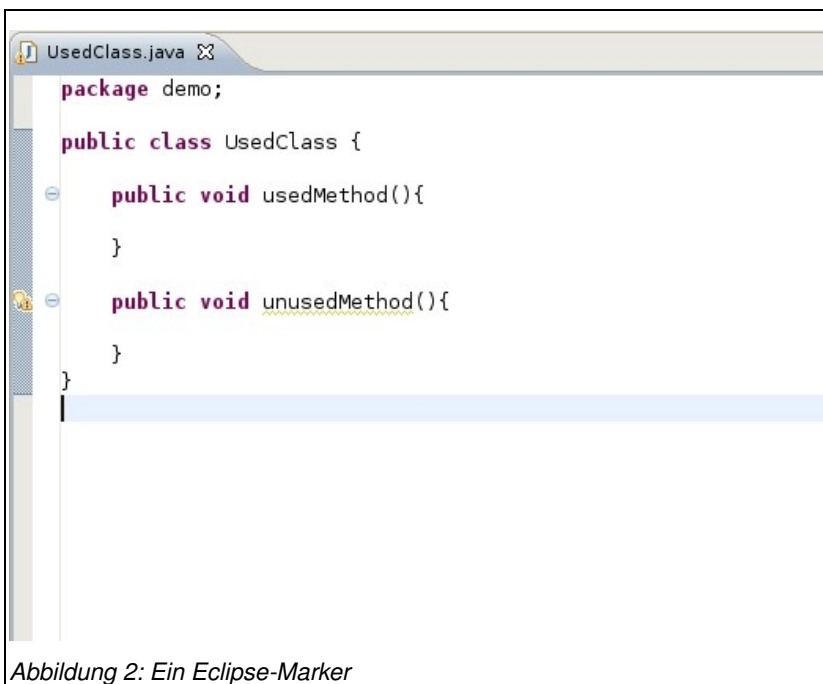


Abbildung 2: Ein Eclipse-Marker

<sup>7</sup> <http://www.eclipse.org/>

<sup>8</sup> <http://weblogs.java.net/blog/editor/archive/2009/10/09/poll-result-eclipse-and-netbeans-are-most-used-ides>

<sup>9</sup> Näheres dazu, sowie ein guter Überblick über die Grundlagen der Plugin-Entwicklung in Eclipse ist z.B. in [Bac07], Anhang B, zu finden.

Handelt es sich bei den Markern um Warnungen oder Fehlermeldungen, werden sie im Problem View von Eclipse angezeigt. Ein Eclipse-View ist ein Bereich, in dem bestimmte Informationen, z.B. hierarchische Strukturen oder Ergebnisse von Suchanfragen, dargestellt werden. Im Problem View (Abb. 3) werden alle Warn- und Fehlermeldungen zu Projekten im Workspace angezeigt, die vom Compiler und anderen Plugins gemeldet werden. Dieser View kann als eine Tabelle betrachtet werden, deren Zeileneinträge Informationen zum Problem und dem Marker, der es im Editor kennzeichnet, enthalten.

Beim Doppelklick auf eine Zeile im Problem View wird das markierte Dokument im Editor geöffnet und der Cursor auf den Marker gesetzt. Das CDCD-Plugin erweitert Eclipse um zwei eigene Views, einen Dead Elements View, in dem alle CDCD-Marker aufgeführt sind und somit ein Überblick über alle toten Elemente gegeben ist (Abb. 4) und den Dependencies View (Abb. 5), in dem Abhängigkeiten zwischen toten Elementen dargestellt werden.

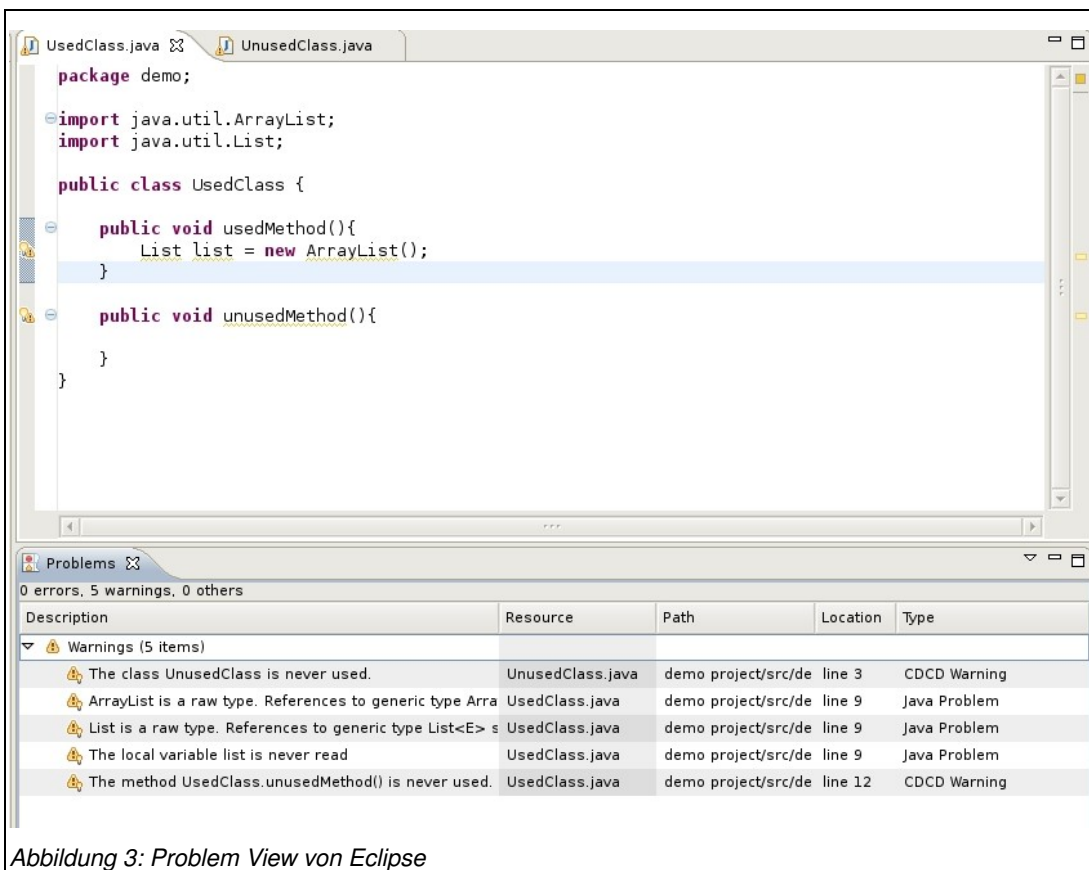


Abbildung 3: Problem View von Eclipse

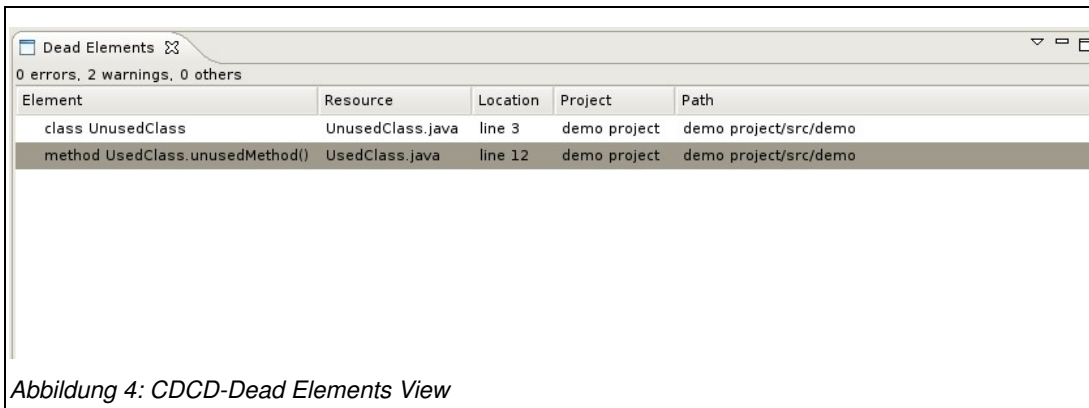


Abbildung 4: CDCD-Dead Elements View

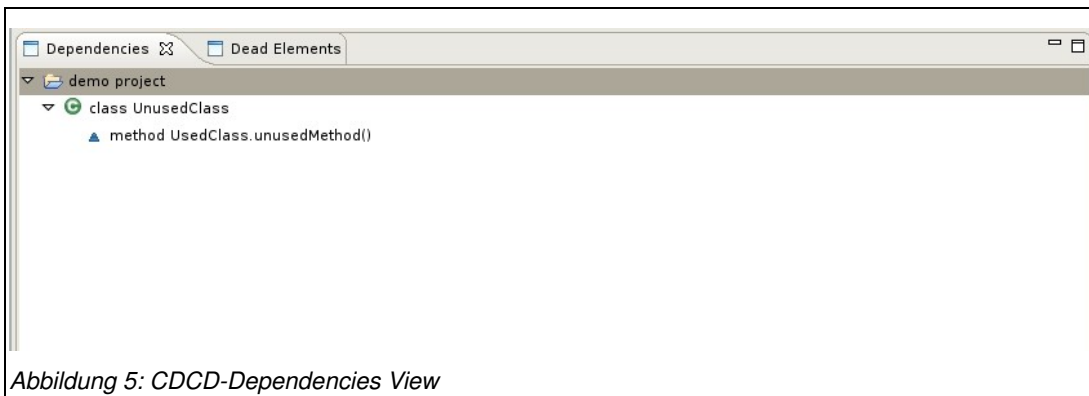


Abbildung 5: CDCD-Dependencies View

Zu Problem-Markern werden üblicherweise sogenannte Quick Fixes, automatische Lösungsvorschläge, angeboten. Die verfügbaren Quick Fixes zu einem Marker werden bei einem Mausklick auf das Markersymbol, bzw. einem Verweilen des Mauszeigers auf der Marker-Beschreibung, angezeigt, vgl. Abb. 6.

Im Fall toten Codes ist das Entfernen betroffener Codefragmente die nahe liegende Lösung. Ein weiterer vom CDCD bereitgestellter Lösungsvorschlag ist, den Code lediglich auszukommentieren, um die durch das Plugin vorgenommen Änderungen überschaubar zu halten. Auch die Möglichkeit, alle unbenutzten Deklarationselemente eines Projekts auf einmal zu entfernen, ist gegeben – dazu muss über das Dead Elements View Menü der entsprechende Eintrag ausgewählt werden. Selbstverständlich können alle Änderungen am Code durch eine Undo-Operation wieder rückgängig gemacht werden.

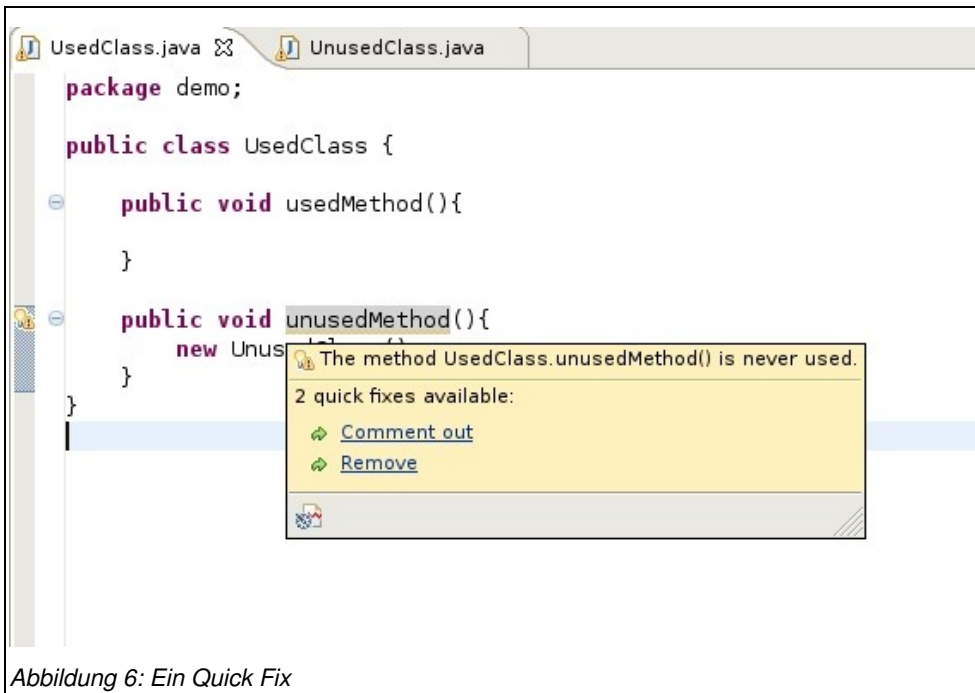


Abbildung 6: Ein Quick Fix

## 4.2 Entfernen toten Codes als Refactoring

Als Refactoring wird die manuelle oder automatisierte Strukturverbesserung von Programmcode unter Beibehaltung des beobachtbaren Programm-Verhaltens bezeichnet. Dabei sollen die Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden, mit dem Ziel, den jeweiligen Aufwand für Fehleranalyse und funktionale Erweiterungen deutlich zu senken [FOW00]. Das Entfernen toten Codes kann somit als Refactoring betrachtet werden, da dabei die Struktur eines Programms so verändert wird, dass mit den genannten positiven Auswirkungen zu rechnen ist, die Programmfunktion aber erhalten bleibt. Bei der Implementierung des CDCD-Plugins wurde daher die Eclipse Refactoring API für die Umsetzung der Änderungen am Code benutzt.

## 4.3 Implementierung des Plugins

Nachdem die Motivation, sowie die Anforderungen bekannt sind, befasst sich dieses Kapitel mit der eigentlichen Implementierung des CDCD-Plugins. Es wird das Zusammenspiel seiner Komponenten und die Vorgehensweise bei der Suche nach unbenutzten Elementen erläutert und anschließend einige theoretische Fragen geklärt.

### 4.3.1 Paketstruktur

Ein Übersicht über die Pakete des Plugins und die von ihren Klassen bereitgestellte Funktionalität bietet die folgende Tabelle:

<b>Paketname</b>	<b>Beschreibung</b>
org.intoj.cdcd	Enthält die Plugin-Hauptklasse, sowie die Controller-Klasse des Programms, CDCDetector.
org.intoj.cdcd.actions	Enthält Klassen, die für das Bereitstellen der Popup-Menüpunkte nötig sind.
org.intoj.cdcd.cream	Enthält die für das Lösen des Constraintsystems benötigte Klasse.
org.intoj.cdcd.markerresolution	Enthält Klassen für die Realisierung der vom Plugin angebotenen Quick Fixes.
org.intoj.cdcd.refactoring	Enthält die Refactoring-Klassen.
org.intoj.cdcd.refactoring.change	Enthält die Change-Klassen.
org.intoj.cdcd.refactoring.registry	Enthält Klassen für das Verwalten toter Deklarationselemente, die dem Model nach MVC-Muster entsprechen.
org.intoj.cdcd.refactoring.registry.event	Enthält die für die Umsetzung des Beobachter-Musters nötigen Typen, um Änderungen an toten Elementen zu registrieren.
org.intoj.cdcd.util	Enthält für die Ausführung der Code-Änderung benötigte Hilfsklasse
org.intoj.cdcd.refactoring.view.dependencies	Enthält Klassen für den View zum Anzeigen der Abhängigkeiten zwischen toten Deklarationselementen
org.intoj.cdcd.refactoring.view.elements	Enthält Klassen für den View zum Anzeigen aller toten Deklarationselemente

## 4.3.2 Programmablauf

### 4.3.2.1 Start der Suche

Um die Suche nach totem Code auf einem Projekt anzustoßen, muss der Menüpunkt „Detect Dead Code“ im Kontextmenü eines Projekts im Package Explorer View von Eclipse ausgewählt werden. Das Anzeigen dieses Menüpunkts wird durch die Implementierung des Erweiterungspunktes `org.eclipse.ui.popupMenus` ermöglicht. Listing 10 zeigt den diese Kontribution betreffenden Abschnitt der Datei `plugin.xml`. In den Zeilen 3 bis 13 wird das Element `ObjectContribution` definiert. Durch die Verwendung einer `ObjectContribution`

wird sichergestellt, dass der Menüpunkt nur bei Auswahl von Objekten eines bestimmten Typs, in diesem Fall `IJavaProject` (Eintrag in Zeile 5), sichtbar ist. In den Zeilen 6 bis 12 wird die zum Menüpunkt gehörende Action definiert. Das Attribut `enablesFor` in Zeile 8 gibt dabei an, dass der Menüpunkt nur aktiv ist, wenn genau ein Projekt selektiert wird und in Zeile 11 ist angegeben, dass es sich um einen gewöhnlichen Menüeintrag handelt. Das Attribut `class` in Zeile 7 verweist auf die Action-Klasse (`DetectDeadCodeAction`), die bei Auswahl des Menüpunkts aufgerufen wird. Da es sich um eine `ObjectContribution` handelt, muss diese Klasse `org.eclipse.ui.IObjectActionDelegate` implementieren.

```
1: <extension
2:     point="org.eclipse.ui.popupMenus">
3:     <objectContribution
4:         id="org.intoj.cdcd.iProjectContribution"
5:         objectClass="org.eclipse.jdt.core.IJavaProject">
6:         <action
7:             class="org.intoj.cdcd.actions.DetectDeadCodeAction"
8:             enablesFor="1"
9:             id="org.intoj.cdcd.deadCodeAction"
10:            label="Detect Dead Code"
11:            style="push">
12:         </action>
13:     </objectContribution>
14: </extension>
```

*Listing 10: Definition des Kontributors für den Kontextmenü-Eintrag*

In der Klasse `DetectDeadCodeAction` sind zwei Methoden von Bedeutung: `selectionChanged(IAction action, ISelection selection)` wird aufgerufen, wenn die Auswahl sich geändert hat (der neue Wert wird dabei in einem Attribut der Klasse `DetectDeadCodeAction` gespeichert), und die Methode `run(IAction action)` wird ausgeführt, wenn ein Menüpunkt ausgewählt wird. In der `run`-Methode wird eine Referenz auf die (einzige) `CDCDetector`-Instanz bezogen und die Methode `detectDeadCode(IJavaProject p)` aufgerufen. Der Ablauf dieser Methode ist im nächsten Abschnitt beschrieben.

#### 4.3.2.2 Die Klasse `org.intoj.cdcd.CDCDetector`

Die Klasse `org.intoj.cdcd.CDCDetector`, die als Singleton implementiert ist, enthält folgende Attribute zur Verwaltung gefundener toter Deklarationselemente:

- `private List<IProject> projects`: eine Liste mit allen Projekten, für die die Suche nach totem Code aufgerufen wurde.
- `private RegistryContainer registryContainer`: Ein Objekt, in dem zu jedem Projekt die zugehörige Registry aller toten Deklarationselemente gespeichert wird.

In der Methode `detectDeadCode()` wird dem `registryContainer` eine Registry für das ausgewählte Projekts hinzugefügt und die Suche gestartet. Wird die Suche für ein Projekt aufgerufen, für das Ergebnisse bereits vorliegen, wird in der Methode `reset()` die vorhandene Registry zu diesem Projekt durch eine neue ersetzt und alle vom CDCD-Plugin gesetzten Marker aus diesem Projekt entfernt.

Für die eigentliche Suche ist die private innere Klasse `org.intoj.cdcd.DeadCodeSearch`, eine Subklasse von `org.eclipse.core.runtime.jobs.Job`, zuständig. In ihrer `run()`-Methode wird das Constraintsystem aufgebaut und die optimale Lösung berechnet, aus der anschließend alle Elemente, die den Zugriffsmodifikator `absent` besitzen, ermittelt und dargestellt werden. Über die der `run()`-Methode übergebene Instanz von `org.eclipse.core.runtime.IProgressMonitor` wird der Benutzer über diese Vorgänge informiert und kann den Suchvorgang jederzeit abbrechen.

#### *4.3.2.3 Aufbau und Lösung des Constraintsystems*

Die Klasse `org.intoj.amm3.constraints.ConstraintSystem` stellt unter anderem die statische Methode `buildConstraintSystem(final IJavaProject project, ForesightSetup foresightSetup, IProgressMonitor monitor)` bereit, um ein Constraintsystem zu einem Projekt zu erzeugen. In der `run()`-Methode der Klasse `DeadCodeSearch` werden dieser Methode die Referenzen auf das selektierte Projekt, sowie die `IProgressMonitor`-Instanz übergeben. Der Parameter `foresightSetup` bezieht sich auf die für vorausschauende Constraints relevanten Programmänderungen durch mögliche Refactorings und ist hier nicht von Bedeutung. Es wird `null` übergeben – in diesem Fall wird beim Aufbau des Constraintsystems ein `ForesightSetup`-Objekt erzeugt.

Zum Lösen des Constraintsystems wird die Klasse `org.intoj.cdcd.cream.MinimizeSolver` eingesetzt. Eine Instanz dieser Klasse wird in der privaten `DeadCodeSearch`-Methode `getSolver(ConstraintSystem constraintSystem, IProgressMonitor monitor)` erzeugt, wie in Listing 11 zu sehen ist. Um die Lösungsmenge bereits vor dem Lösen eingrenzen zu können wird das Constraintsystem an dieser Stelle um ein zusätzliches Constraint erweitert. Dazu werden die Summe der Variablenbelegungen des Ausgangsprogramms (gespeichert in der Variablen `sum`) und die maximal mögliche Summe als Summe der oberen Schranken aller Variablen-Domains berechnet. Diese zwei Werte bilden die untere und obere Schranke der neuen Variable `objective` (Zeile 32), die dem `MinimizeSolver` als objektive Variable übergeben wird. Das neue Constraint fordert die gleichen Werte für `sum` und `objective`.

Dem `DefaultSolver`, und damit auch dem verwendeten `MinimizeSolver`, kann im

Konstruktor eine Option übergeben werden, die die Lösungsstrategie bezüglich der objektiven Variable festlegt. Im Konstruktor der Klasse `MinimizeSolver` wird die Strategie `Solver.MAXIMIZE` festgelegt, was bedeutet, dass eine Lösung angestrebt wird, in der die objektive Variable den größtmöglichen Wert hat. Bezogen auf die Zugriffsmodifikatoren bedeutet dies, dass diese so restriktiv wie möglich sind, d.h. möglichst viele Deklarationselemente den Modifikator `absent` besitzen. Durch die Festlegung der unteren Schranke der objektiven Variablen wird die Lösungssuche beschleunigt, da nur restriktivere Lösungen als die aktuelle betrachtet werden müssen.

Bei der Verwendung der Klasse `DefaultSolver` war die Lösungssuche bei großen Projekten (Constraintsystem mit ca. 3000 Variablen) sehr zeitintensiv, da ein `DefaultSolver` für jeden Wert der objektiven Variablen, ausgehend von der unteren Schranke, die passende Lösung berechnet und schließlich die letzte davon als Ergebnis liefert. Diese vielen Lösungsschritte werden im `MinimizeSolver` eingespart, indem von der oberen Schranke ausgegangen wird und die Lösung somit in nur einem Schritt ermittelt werden kann. Zu diesem Zweck wurde die Methode `solve()` der Klasse `DefaultSolver` entsprechend überschrieben. Die Lösung wird an das Constraintsystem über die Methode `setConstraintSystemValues(Solution solution)` übermittelt. Ist diese Lösung gültig, d.h. das Constraintsystem ist erfüllt, wird mit der Analyse der Lösung fortgefahren.

```

1: private Solver getSolver(ConstraintSystem constraintSystem,
2:     IProgressMonitor monitor) {
3:     Solver.resetIDCounter();
4:     Network network = constraintSystem.getNetwork();
5:
6:     // Summe der aktuellen Werte
7:     int actualValue = 0;
8:
9:     // Anzahl der Variablen
10:    int size = constraintSystem.getVariables().size();
11:
12:    // Maximal mögliche Summe der Werte aller Variablen
13:    int maxValue = 0;
14:
15:    // Summe der aktuellen Werte ermitteln
16:    for (AccessModifierVariable variable : constraintSystem.getVariables()){
17:        actualValue += variable.getAccessModifierValue().ordinal();
18:    }
19:
20:    // Maximal mögliche Summe ermitteln
21:    int[] values = new int[network.getVariables().size()];
22:    for (int i = 0; i < values.length; i++) {
23:        maxValue += ((IntDomain) network.getVariable(i).getDomain()).max();
24:    }
25:
26:    // Variable für die Summe aller Werte erzeugen
27:    IntVariable sum = (IntVariable) network.getVariable(0);
28:    for (int i = 1; i < size; i++) {
29:        sum = sum.add((IntVariable) network.getVariable(i));
30:    }
31:
32:    IntVariable objective = new IntVariable(network, actualValue,maxValue);
33:
34:    // Zusätzliches Constraint definieren
35:    objective.equals(sum);
36:
37:    network.setObjective(objective);
38:    Solver solver = new MinimizeSolver(network, monitor);
39:    return solver;
40: }

```

Listing 11: Methode `DeadCodeSearch.getSolver(ConstraintSystem, IProgressMonitor)`

#### 4.3.2.4 Analyse der Lösung

Nachdem das Constraintsystem gelöst ist, müssen alle Deklarationselemente, die den Zugriffsmodifikator `absent` besitzen, ermittelt werden. Ausnahmen stellen anonyme Klassen und `private` parameterlose Konstruktoren dar. Für erstere ist nur der Modifikator `absent` zugelassen, da sie keinen Zugriffsmodifikator besitzen können, bei der Constraintgenerierung jedoch berücksichtigt werden müssen. Da die Zuweisung des Modifikators `absent` an eine anonyme Klasse nicht bedeutet, dass diese Klasse entfernt werden kann, werden anonyme Klassen nicht in die Ergebnismenge aufgenommen. Bei privaten parameterlosen Konstruktoren wird davon ausgegangen, dass sie bewusst deklariert werden, um Instantiierungen der jeweiligen Klassen zu verhindern.

Um die Lösung des Constraintsystems analysieren zu können wird eine Referenz auf alle Constraint-Variablen benötigt, diese kann über die Methode `getVariables()` der Klasse `ConstraintSystem` erhalten werden. Jede Constraint-Variable ist ein Objekt vom Typ `org.intoj.amm3.constraints.AccessModifierVariable` und führt eine Referenz auf ein Objekt vom Typ `org.intoj.amm3.constraints.DeclarationElement`, das das entsprechende Deklarationselement repräsentiert. Als Pendant dazu gibt es im CDCD-Plugin die Klasse `org.intoj.cdcd.registry.DeadElement`, dessen Objekte toten Deklarationselementen entsprechen. Der Konstruktor dieser Klasse hat einen Parameter vom Typ `DeclarationElement`, ein totes Element kann also nur aus einem Deklarationselement erzeugt werden.

Beim Setzen der Warn-Marker gilt zu beachten, dass bei toten Typen keine Marker für deren Elemente gesetzt werden, sondern nur für die Typen selbst (vgl. nächsten Abschnitt). Bei der Analyse werden daher zunächst alle Variablen, also Zugriffsmodifikatoren  $\langle d \rangle$ , bestimmt, für die die Bedingung  $\lambda(d) = d' \wedge \langle d' \rangle = \text{absent}$  erfüllt ist.

Anschließend werden alle Constraints der Typen `Acc-1-Relation`, `Dyn-1-Relation` und `Sub-1-Relation` inspiziert. Die Referenz auf alle Constraints kann von der `ConstraintSystem`-Instanz über die Methode `getConstraints()` bezogen werden. Da in den genannten Constraints Abhängigkeiten zwischen Deklarationselementen bezüglich ihrer Sichtbarkeiten festgehalten werden, wird geprüft, ob die entsprechenden Constraint-Variablen beide den Wert `absent` haben und somit eine Abhängigkeit zwischen zwei toten Elementen vorliegt: wird ein unbenutztes Deklarationselement A von einem anderen unbenutzten Deklarationselement B referenziert, darf A nicht aus dem Programm entfernt werden, ohne dass B ebenfalls entfernt wird, A ist also abhängig von B. Die aus `Dyn-1-Relation` und `Sub-1-Relation` resultierenden Abhängigkeiten werden anhand des Codebeispiels aus Listing 12 erläutert:

```

1: interface I {
2:     void n();
3: }
4:
5: class SubclassB implements I {
6:     @Override
7:     public void n() {}
8: }
9:
10: class Superclass {
11:     void m() {}
12: }
13:
14: class SubclassA extends Superclass {
15:     @Override
16:     void m() {}
17: }

```

Listing 12: Abhängigkeiten zwischen toten Deklarationselementen

1. SubclassB.n() ist abhängig von I.n(): nach Entfernung der Methode n() aus SubclassB würde diese Klasse das Interface I nicht mehr implementieren, wenn I.n() erhalten bleibt.
2. I.n() ist abhängig von SubclassB.n(): wird die Interface-Methode entfernt, gibt es eine Fehlermeldung vom Compiler, wenn SubclassB.n(), wie im obigen Beispiel, mit @Override annotiert ist.<sup>10</sup>
3. Superclass.m() ist abhängig von SubclassA.m() aus dem gleichen Grund wie in Punkt 2.
4. SubclassA.m() ist abhängig von Superclass.m(). In diesem Fall gibt es zwar keinen Compilerfehler wie in Punkt 2 nach Entfernen von SubclassA.m(), die Methode der Superklasse sollte aber trotzdem entfernt werden, da die Vererbungsbeziehung sonst bestehen bleibt. Werden nach der Entfernung Änderungen im Programm vorgenommen, so dass die Methode Superclass.m() doch benutzt wird, kann es zu einer Änderung der Programmsemantik kommen, wenn diese Methode auf einer SubclassA-Instanz aufgerufen wird und dabei nicht berücksichtigt wird, dass die überschreibende Methode nicht mehr existiert. Wäre Superclass abstrakt, müsste SubclassA.m() wiederum zwingend entfernt werden, da SubclassA alle abstrakten Methoden der Superklasse implementieren muss und der Compiler einen Fehler meldet, wenn es nicht der Fall ist.

---

<sup>10</sup> Das CDCD-Plugin berücksichtigt zwar keine @Override-Annotationen, die Abhängigkeitsbeziehung, und damit das Entfernen von SubclassB.n(), resultiert hier aus dem Sub1-Relation-Constraint

Die einer Constraint-Variablen entsprechende DeadElement-Instanz wird aus der Map `DeadCodeSearch.elementsMap` ermittelt, in der die Zuordnung von `AccessModifierVariable`- zu `DeadElement`-Objekten gespeichert wird. Dabei wird ein neues `DeadElement`-Objekt erzeugt, falls in der `elementsMap` noch kein Eintrag für eine `Variable` existiert. Ohne diese Variablenzuordnungen könnten mehrere `DeadElement`-Objekte zu der gleichen Variablen erzeugt werden, da eine `Variable` in mehreren `Constraints` vorkommen kann. Abhängigkeiten zwischen Deklarationselementen werden in der Klasse `DeadElement` in den Attributen `private Set<DeadElement> dependentElements` (enthält alle von dem Element abhängigen Elemente), `private Set<DeadElement> dependsOn` (Set mit allen Elementen, von denen das Element abhängt), sowie `private Set<DeadElement> cycleMembers` verwaltet. Im letzten Set sind alle Elemente enthalten, zu denen eine zyklische, Abhängigkeit existiert (d.h. zwei Elemente sind voneinander abhängig).

Beim Hinzufügen von Abhängigkeitsbeziehungen gibt es einiges zu beachten:

- eine Klasse muss nicht von ihren Elementen abhängig sein, da diese mit dem Entfernen der Klasse automatisch entfernt werden<sup>11</sup>
- bei Konstruktoren von toten Klassen muss eine Abhängigkeitsbeziehung zu der umschließenden Klasse hinzugefügt werden, falls es sich um den einzigen Konstruktor dieser Klasse handelt.<sup>12</sup>

Da Elemente von toten Typen nicht separat markiert werden, und damit auch nicht einzeln vom CDCD entfernt werden können, werden diese Elemente auch nicht im `Dependencies View` dargestellt. Die Abhängigkeitsbeziehungen zwischen solchen Elementen müssen dennoch festgehalten werden, für den Fall dass eins der Elemente auch von einem markierten Element abhängig ist und dessen Entfernen korrekt erfolgt. Solche Abhängigkeiten werden in den Attributen `private Set<DeadElement> invisibleDependent` und `private Set<DeadElement> invisibleDependsOn` eines `DeadElement`-Objekts verwaltet.

Nachdem die `Constraint`-Variablen samt der Abhängigkeitsbeziehungen gespeichert sind, werden sie aus der Liste aller Variablen entfernt und aus den verbleibenden Variablen diejenigen mit dem Wert `absent` ermittelt und gespeichert. Anschließend wird geprüft, ob zyklische Abhängigkeiten existieren – in diesem Fall wird das Element aus dem Set `dependsOn` bzw. `invisibleDependsOn` entfernt und in das Set `cycleMembers` aufgenommen. Alle toten Deklarationselemente werden in die `DeadElementsRegistry` des Projekts aufgenommen.

---

<sup>11</sup> solche Abhängigkeitsbeziehungen können aber durchaus aus `Constraints` resultieren

<sup>12</sup> Der Konstruktor kann zwar nicht einzeln entfernt werden, sein Entfernen kann jedoch eine Voraussetzung für das Löschen eines anderen Elements sein

### 4.3.2.5 Darstellung der Ergebnisse

#### *Setzen der Marker*

Zu jedem toten Deklarationselement wird ein Marker erzeugt, es sei denn, es handelt sich um ein Element, das sich in einem anderen toten Deklarationselement befindet, z.B. eine Methode einer toten Klasse. In diesem Fall werden keine Marker gesetzt, zum einen wegen einer besseren Übersichtlichkeit, zum anderen aus dem Grund, dass ein CDCD Warn-Marker das automatische Entfernen des toten Codeelements ermöglicht (vgl. Abschnitt Quick Fixes), und das Entfernen einzelner Elemente einer toten Klasse in der Praxis als wenig sinnvoll erscheint.

Für das Setzen der Marker stellt das CDCD-Plugin eine Erweiterung am Erweiterungspunkt `org.eclipse.core.resources.markers` bereit. Da der CDCD-Warnmarker Subtyp von `problemmarker` ist, wird er auch im Eclipse-Problems View angezeigt. Um zu einem Deklarationselement einen Marker zu erzeugen, wird die Methode `createMarker(DeadElement element)` der Klasse `CDCDetector` aufgerufen. Listing 13 zeigt diese Methode. Die Konstante `CDCDetector.MARKER_WARNING` verweist dabei auf den Wert des Attributes `id` des Markers. In den Zeilen 4 bis 13 werden die Attribute des Markers gesetzt: in Zeile 5 die Beschreibung des Problems, auf das der Marker verweist; in Zeile 7 wird festgelegt, dass es sich um eine Warnung handelt; die folgenden drei Attribute beziehen sich auf die Position des Markers: die Zeilennummer, sowie den Bereich, der markiert wird. Das Attribut `IMarker.SOURCE_ID` in Zeile 11 wird benötigt, um zu einem Marker das zugehörige Deklarationselement auffinden zu können, wenn ein Quick Fix ausgewählt wird. Dabei wird ein Deklarationselement anhand seines Hash-Codes identifiziert. Das bei der Definition des Markers festgelegte Attribut `ELEMENT_NAME` verweist auf den Namen des Deklarationselements und ist für die Darstellung aller toten Elemente im Dead Elements View erforderlich.

```

1: private void createMarker(DeadElement element) {
2:     IMarker marker;
3:     try {
4:         marker = element.getResource().createMarker(CDCDetector.MARKER_WARNING);
5:         marker.setAttribute(IMarker.MESSAGE, "The "
6:             + element.toString() + " is never used.");
7:         marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_WARNING);
8:         marker.setAttribute(IMarker.LINE_NUMBER, element.getLineNumber());
9:         marker.setAttribute(IMarker.CHAR_START, element.getNameOffset());
10:        marker.setAttribute(IMarker.CHAR_END, element.getNameEnd());
11:        marker.setAttribute(IMarker.SOURCE_ID, String.valueOf(element
12:            .hashCode()));
13:        marker.setAttribute("ELEMENT_NAME", element.toString());
14:    } catch (CoreException e) {
15:        throw new RuntimeException(e);
16:    }
17: }

```

*Listing 13: Die Methode createMarker(DeadElement element)*

## Views

Das CDCD-Plugin erweitert Eclipse um zwei eigene Views: den Dead Elements View zum Anzeigen aller gefundenen toten Deklarationselemente und den Dependencies View zur Veranschaulichung der existierenden Abhängigkeiten zwischen toten Deklarationselementen. Listing 14 zeigt die Definition dieser Views entsprechender Erweiterungen am Erweiterungspunkt `org.eclipse.ui.views`. Der Wert des Attributes `allowMultiple` in Zeile 4 bzw. 11 gibt an, dass nur eine Instanz dieser Views erzeugt werden darf. Das Attribut `restorable` in Zeile 8 bzw. 15 hat ebenfalls den Wert `false`, was bedeutet, dass diese Views bei einem Neustart von Eclipse nicht automatisch angezeigt werden. Das Attribut `class` in Zeile 5 bzw. 12 verweist auf die Klasse, die die Funktionalität dieser Views realisiert, wobei diese Klasse das Interface `org.eclipse.ui.IViewPart` implementieren oder die Klasse `org.eclipse.ui.part.ViewPart` erweitern sollte. Die Klasse `DependenciesView` ist als Subklasse von `org.eclipse.ui.part.ViewPart` implementiert, die Klasse `ElementsView` wurde jedoch von der Klasse `org.eclipse.ui.views.markers.MarkerSupportView` abgeleitet, da es sich hierbei um einen View zum Anzeigen von Markern handelt. Auf diese Weise ist es ohne zusätzlichen Aufwand möglich, über alle gefundenen Elemente zu navigieren und die Anzeige stets konsistent mit den angezeigten Markern zu halten, selbst wenn tote Elemente entfernt werden, da diese Funktionalität von Eclipse-Markern bereitgestellt wird.

```

1: <extension
2:     point="org.eclipse.ui.views">
3:     <view
4:         allowMultiple="false"
5:         class="org.intoj.cdcd.view.elements.ElementsView"
6:         id="org.intoj.cdcd.elementsView"
7:         name="Dead Elements"
8:         restorable="false">
9:     </view>
10:    <view
11:        allowMultiple="false"
12:        class="org.intoj.cdcd.view.dependencies.DependenciesView"
13:        id="org.intoj.cdcd.dependenciesView"
14:        name="Dependencies"
15:        restorable="false">
16:    </view>
17: </extension>

```

*Listing 14: Definition der Kontributoren für die Views*

Um einen Marker-View zu erstellen, muss der Erweiterungspunkt `org.eclipse.ui.ide.markerSupport` implementiert werden, wobei die Elemente `markerField` und `markerContentGenerator` von Bedeutung sind. Listing 15 zeigt die Definition dieses Kontributors. Durch die Definition der `markerField`-Elemente werden dem View zusätzliche Spalten zugefügt – eine für den Namen des Elements (Zeilen 3-7) und eine für den Namen des Projekts, in dem dieses Element zu finden ist (Zeilen 8-12). Das Attribut `class` gibt in beiden Fällen die Klasse an, über die die Spalte instantiiert wird. Diese Klasse muss `org.eclipse.ui.views.markers.MarkerField` erweitern und die Methode `getValue(MarkerItem item)` überschreiben. Die Kind-Elemente des Elements `markerContentGenerator` in den Zeilen 16 bis 30 entsprechen den Spalten, die im View angezeigt werden. Das Element `markerTypeReference` gibt an, welche Marker im View dargestellt werden, in diesem Fall sind es Marker vom Typ `org.intoj.cdcd.warningMarker`.

```

1:  <extension
2:      point="org.eclipse.ui.ide.markerSupport">
3:      <markerField
4:          class="org.intoj.cdcd.view.elements.ElementMarkerField"
5:          id="org.intoj.cdcd.nameField"
6:          name="Element">
7:      </markerField>
8:      <markerField
9:          class="org.intoj.cdcd.view.elements.ProjectMarkerField"
10:         id="org.intoj.cdcd.projectField"
11:         name="Project">
12:      </markerField>
13:      <markerContentGenerator
14:         id="org.intoj.cdcd.MarkerContentGenerator"
15:         name="CDCD Marker Generator">
16:      <markerFieldReference
17:         id="org.intoj.cdcd.nameField">
18:      </markerFieldReference>
19:      <markerFieldReference
20:         id="org.eclipse.ui.ide.resourceField">
21:      </markerFieldReference>
22:      <markerFieldReference
23:         id="org.eclipse.ui.ide.locationField">
24:      </markerFieldReference>
25:      <markerFieldReference
26:         id="org.intoj.cdcd.projectField">
27:      </markerFieldReference>
28:      <markerFieldReference
29:         id="org.eclipse.ui.ide.pathField">
30:      </markerFieldReference>
31:      <markerTypeReference
32:         id="org.intoj.cdcd.warningMarker">
33:      </markerTypeReference>
34:      </markerContentGenerator>
35:  </extension>

```

*Listing 15: Definition des Kontributors für den Marker View*

Im Dependencies-View werden Abhängigkeiten zwischen Deklarationselementen dargestellt. Da ein Element von mehreren abhängen kann und es unter diesen auch solche geben kann, die selber von anderen Elementen abhängen, bietet es sich an die Klasse `org.eclipse.jface.viewers.TreeViewer` für die Visualisierung des Abhängigkeit-Baums zu verwenden. Eine Instanz dieser Klasse wird bei der Initialisierung des `DependenciesView`-Objekts erzeugt. Als Modell dienen Objekte vom Typ `org.intoj.cdcd.registry.DeadElementContainer`: das `RegistryContainer`-Objekt bildet dabei die Wurzel, deren Kind-Knoten `DeadElementsRegistry`-Objekte sind, die Projekte repräsentieren, und Kind-Knoten vom Typ `DeadElement` und `DeadElementCycle` enthalten. Letztere sind Objekte für die Darstellung von zyklischen Abhängigkeiten – jeder Zyklus-Knoten besitzt dabei zwei Kind-Knoten, die den voneinander abhängigen Elementen entsprechen.

Für die Verknüpfung von Modell und View werden zwei Klassen benötigt: eine, die `org.eclipse.jface.viewers.ITreeContentProvider` implementiert und deren Aufgabe es ist, die Beziehung zwischen Modell-Objekten an den `TreeViewer` weiter zu geben, und eine, die `org.eclipse.jface.viewers.ILabelProvider` implementiert und den `TreeViewer` mit einem Text, sowie einem Bild für jedes Modell-Objekt versorgt. Im CDCD-Plugin sind diese Klassen `DeadElementsContentProvider` und `DeadElementsLabelProvider`, die sich im Paket `org.intoj.cdcd.view.dependencies` befinden. Bei Änderungen des Modells, z.B. wenn tote Deklarationselemente entfernt werden, muss der View aktualisiert werden. Zu diesem Zweck implementiert die Klasse `DeadElementsContentProvider` das Interface `org.intoj.cdcd.registry.event.IElementsListener` und meldet sich bei `DeadElementContainer`-Instanzen als solcher an. Beim Hinzufügen und Entfernen von Elementen wird der `ContentProvider` benachrichtigt, worauf hin er den `Viewer` aktualisiert. Eine detaillierte Beschreibung des Zusammenspiels von Modell, `TreeViewer`, `ContentProvider` und `LabelProvider` findet sich im Eclipse-Artikel „How to use the JFace Tree Viewer“<sup>13</sup>.

### 4.3.3 Entfernen toten Codes

#### 4.3.3.1 QuickFixes

In Eclipse werden Marker nicht nur verwendet, um den Entwickler auf Fehler oder Designschwächen im Sourcecode hinzuweisen, die Marker enthalten in der Regel auch Lösungsvorschläge für diese Probleme, sogenannte Quick Fixes. Für die CDCD-Warn-Marker sind zwei Quick Fixes verfügbar – das Auskommentieren und das Entfernen des markierten Deklarationselementes. Um Quick Fixes mit einem Marker-Typ zu verknüpfen muss eine Erweiterung am Erweiterungspunkt `org.eclipse.ui.ide.markerResolution` vorgenommen werden, indem die Methode `getResolutions(IMarker marker)` des Interfaces `org.eclipse.ui.IMarkerResolutionGenerator` implementiert wird, die ein Array mit Instanzen von `org.eclipse.ui.IMarkerResolution` zurück liefert. Wie in Listing 16 zu sehen ist, liefert der `MarkerResolutionGenerator` ein Array mit zwei Einträgen zurück.

---

13 <http://www.eclipse.org/articles/Article-TreeViewer/TreeViewerArticle.htm>

```

1: public class CDCDMarkerResolutionGenerator implements
2:             ImarkerResolutionGenerator {
3:     @Override
4:     public IMarkerResolution[] getResolutions(IMarker marker) {
5:         IMarkerResolution[] resolutions = new IMarkerResolution[2];
6:         resolutions[0]= new CommentOutMarkerResolution();
7:         resolutions[1]= new RemoveMarkerResolution();
8:         return resolutions;
9:     }
10: }

```

Listing 16: Die Klasse *CDCDMarkerResolutionGenerator*

Wenn ein Quick Fix ausgewählt wird, wird die Methode `run(IMarker marker)` der entsprechenden `IMarkerResolution`-Instanz ausgeführt. Listing 17 zeigt, wie diese Methode in der Klasse `CommentOutMarkerResolution` implementiert ist. Zunächst muss ermittelt werden, für welches Deklarationselement der Quick Fix aufgerufen wurde. Jeder Marker ist an eine Resource gebunden, über die das Projekt ermittelt werden kann, in dem der Marker definiert ist (die Resource kann mit dem Projekt identisch sein). Zu diesem Projekt wird von der Methode `getRegistry(IProject project)` der Klasse `CDCDetector` die Registry des Projektes zurück geliefert, in der alle toten Deklarationselemente dieses Projekts verwaltet werden. Das gesuchte tote Element wird in der Registry anhand des Schlüssels erkannt, der im Marker-Attribut `SOURCE_ID` gespeichert ist. Wenn es vom Element Abhängigkeiten zu anderen toten Deklarationselementen gibt, kann nur fortgefahren werden, wenn der Benutzer der Entfernung all dieser Elemente zustimmt – die Anweisung in Zeile 9 veranlasst das Anzeigen eines entsprechenden Dialogs. Die Anzahl dieser Elemente kann recht groß sein, so dass das Auskommentieren ein relativ langwieriger Vorgang sein kann. In Zeile 15 wird daher das Auskommentieren, genauer gesagt die `run`-Methode der inneren Klasse `CommentOutRunnable` (Listing 18), mit einer Fortschrittsanzeige im UI Thread gestartet. Letzteres bedeutet, dass keine Interaktion mit der Benutzeroberfläche möglich ist, solange der Vorgang läuft. Da es sich um eine atomare Änderung am Workspace handelt, ist dies durchaus erwünscht.

```

1: @Override
2: public void run(IMarker marker) {
3:     DeadElementsRegistry registry = CDCDetector.getInstance().getRegistry(
4:         marker.getResource().getProject());
5:     DeadElement element = registry.getElement(marker.getAttribute(
6:         IMarker.SOURCE_ID, ""));
7:     QuickFixUtil util = new QuickFixUtil();
8:     if (element.isDependent()) {
9:         if (!util.showQuestionDialog(element))
10:            return;
11:     }
12:     List<DeadElement> elements = util.getDependencies(element);
13:     CommentOutRefactoring refactoring = new CommentOutRefactoring(elements);
14:     try {
15:         PerformRefactoringOperation op = new PerformRefactoringOperation(
16:             refactoring, CheckConditionsOperation.ALL_CONDITIONS);
17:         ResourcesPlugin.getWorkspace().run(op, new NullProgressMonitor());
18:     } catch (CoreException e) {
19:         throw new RuntimeException(e);
20:     }
21: }

```

*Listing 17: run-Methode der CommentOutMarkerResolution*

In Zeile 5 von Listing 18 liefert die Methode `getDependencies(DeadElement element)` der Hilfsklasse `org.intoj.cdcd.util.QuickFixUtil` eine Liste mit allen Elementen, von denen das zu entfernende Element abhängig ist, inklusive der Elemente, von denen diese Elemente selbst abhängen, zurück. In Zeile 7 wird das Refactoring-Objekt mit einer Referenz auf diese Liste erzeugt.

Die Implementierung der `run`-Methode in der Klasse `RemoveMarkerResolution` stimmt mit Listing 17 bis auf Zeile 16 überein: statt einem `CommentOutRunnable`-Objekt wird dort ein `RemoveRunnable`-Objekt (ebenfalls eine innere Klasse) erzeugt, dessen `run`-Methode wie in Listing 18 aussieht, mit dem Unterschied, dass in Zeile 7 ein `org.intoj.cdcd.refactoring.RemoveRefactoring`-Objekt erzeugt wird. Das Thema Refactoring in Eclipse ist Gegenstand des nächsten Abschnitts.

```

1: @Override
2: public void run(IProgressMonitor monitor)
3:     throws InvocationTargetException, InterruptedException {
4:     QuickFixUtil util = new QuickFixUtil();
5:     List<DeadElement> elements = util.getDependencies(element);
6:
7:     CommentOutRefactoring refactoring = new CommentOutRefactoring(
8:         elements);
9:
10:    monitor.beginTask("Comment out...", elements.size() * 2);
11:    try {
12:        PerformRefactoringOperation op = new PerformRefactoringOperation(
13:            refactoring, CheckConditionsOperation.ALL_CONDITIONS);
14:        ResourcesPlugin.getWorkspace().run(op, monitor);
15:    } catch (CoreException e) {
16:        e.printStackTrace();
17:        throw new RuntimeException(e);
18:    }
19: }

```

*Listing 18: run-Methode der Klasse CommentOutRunnable*

#### 4.3.3.2 Refactoring in Eclipse

Um ein neues Refactoring in Eclipse hinzuzufügen muss die abstrakte Klasse `org.eclipse.ltk.core.refactoring.Refactoring` erweitert werden. Folgende Methoden dieser Klasse müssen implementiert werden:

- `checkInitialConditions(IProgressMonitor pm)`: diese Methode wird aufgerufen, nachdem das Refactoring initialisiert wurde und prüft, ob die Vorbedingung für dieses Refactoring erfüllt ist. Das Ergebnis dieser Prüfung, ein Objekt vom Typ `org.eclipse.ltk.core.refactoring.RefactoringStatus`, ist vom dem Element abhängig, für das das Refactoring aufgerufen wurde. Hat der Refactoring Status den Grad FATAL, kann das Refactoring für dieses Element nicht ausgeführt werden.
- `checkFinalConditions(IProgressMonitor pm)`: nachdem die Vorbedingung geprüft ist, können unter Umständen noch Benutzereingaben benötigt werden, um das Refactoring durchführen zu können. Sind diese erfolgt, wird durch diese Methode nochmals geprüft, ob die Vorbedingung eventuell verletzt wurde.
- `createChange(IProgressMonitor pm)`: diese Methode wird aufgerufen, wenn alle Prüfungen erfolgreich verlaufen sind und erzeugt das Change-Objekt, das die aus dem Refactoring resultierenden Änderungen am Code enthält.

Eine einfache Möglichkeit, das Refactoring auszuführen, besteht in der Verwendung der Klasse `org.eclipse.ltk.core.refactoring.PerformRefactoringOperation`. Eine

Instanz dieser Klasse wird der Methode `IWorkspace.run(IWorkspaceRunnable, IProgressMonitor)` übergeben (vgl. Listing 18, Zeile 14), die die Operation ausführt. Dabei wird der beschriebene Refactoring-Zyklus durchlaufen und nach Ausführung der Änderungen das Change-Objekt für eine Zurücksetzung (Undo) gespeichert.

#### 4.3.3.3 CDCD-Refactorings

Im CDCD-Plugin sind zwei Refactorings implementiert: `CommentOutRefactoring` und `RemoveRefactoring`. Die Vorbedingung, die es bei beiden zu prüfen gibt, besagt lediglich, dass die Liste zu entfernender Elemente nicht null sein darf, und auch keine null-Elemente enthalten darf. In der Methode `checkFinalConditions` wird in beiden Klassen keine weitere Prüfung vorgenommen, sondern lediglich der Status OK zurückgegeben, da nach der ersten Prüfung keine Interaktion mit dem Benutzer erfolgt. Beide Refactorings erzeugen ein Change-Objekt vom Typ `org.intoj.cdcd.refactoring.change.CDCDChange`. Diese Klasse ist von `org.eclipse.ltk.core.refactoring.CompositeChange` abgeleitet, denn die von den Refactorings erzeugten Änderungen sind aus zwei Teil-Änderungen zusammengesetzt: neben den Änderungen am Sourcecode müssen noch die Warn-Marker entfernt, und beim Auskommentieren zusätzlich Info-Marker gesetzt werden.

Bei der Ausführung einer `CompositeChange` werden die Änderungen in der gleichen Reihenfolge ausgeführt, in der die entsprechenden Change-Objekte der `CompositeChange`-Instanz hinzugefügt wurden. Beim Auskommentieren toter Deklarationselemente werden erst alle Warn-Marker entfernt und anschließend die Elemente selbst, da das entfernte Element auch aus der Registry entfernt wird und der Marker nach der Entfernung dem Element nicht mehr zugeordnet werden kann. So wird in der `createChange`-Methode der Klasse `CommentOutRefactoring` für jedes Element zunächst `createMarkerChange(DeadElement element)` und anschließend `createElementChange(DeadElement element)` aufgerufen. Listing 19 zeigt diese Methoden.

```

1: private Change createMarkerChange(DeadElement element)
2:                                     throws CoreException {
3:     QuickFixUtil util = new QuickFixUtil();
4:     IMarker elementMarker = util.findMarker(element);
5:     return new MarkerChange(element, elementMarker, true);
6: }
7:
8: private Change createElementChange(DeadElement element) {
9:     IResource resource = element.getResource();
10:    IFile file = null;
11:    if (!(resource instanceof IFile)) {
12:        throw new RuntimeException("No file for " + element.getName()
13:                                   + " found");
14:    } else {
15:        file = (IFile) resource;
16:    }
17:    CommentOutChange change = new CommentOutChange(element, file);
18:    return change;
19: }

```

*Listing 19: Methoden für das Erzeugen von Change-Objekten im Comment Out Refactoring*

In den Zeilen 4 und 5 wird der Marker des Elements bestimmt und eine Instanz der Klasse `org.intoj.cdcd.refactoring.change.markers.MarkerChange` erzeugt. Diese Klasse ist eine Subklasse von `org.eclipse.ltk.core.refactoring.Change`, ihre `perform`-Methode ist in Listing 20 dargestellt. In Zeile 9 wird der im Konstruktor übergebene Marker, sofern einer vorhanden war, entfernt, nachdem seine Attribute für die Undo-Change gespeichert wurden. Da im Konstruktor `true` für `createInfoMarker` übergeben wurde (Listing 19, Zeile 5), wird ein Marker vom Typ `org.intoj.cdcd.infoMarker` gesetzt, der darauf hinweist, dass das Element vom CDCD-Plugin auskommentiert wurde. Dieser Marker ist ebenfalls Subtyp von `problemmarker`, definiert im Gegensatz zum Warn-Marker aber keine zusätzlichen Attribute. Dadurch dass der Wert des Attributs `IMarker.SEVERITY` auf `IMarker.SEVERITY_INFO` gesetzt wird, taucht dieser Marker nicht im Problem View auf. Die Definition eines speziellen Marker-Typs ist aus folgendem Grund nötig: ein Marker vom Typ `IMarker.TEXT` wird nicht im Editor angezeigt, ein Marker vom Typ `IMarker.PROBLEM` wird dagegen im Problem View angezeigt, selbst wenn das Attribut `IMarker.SEVERITY` den Wert `INFO` hat.

```

1: @Override
2: public Change perform(IProgressMonitor pm) throws CoreException {
3:     pm.subTask(element.getName());
4:
5:     // Warn-Marker entfernen
6:     Map<String, Object> attributes = null;
7:     if (warningMarker != null && warningMarker.exists()) {
8:         attributes = warningMarker.getAttributes();
9:         warningMarker.delete();
10:    }
11:
12:    // Info-Marker setzen
13:    IMarker infoMarker = null;
14:    if (createInfoMarker) {
15:        infoMarker = element.getResource().createMarker(
16:            CDCDetector.MARKER_INFO);
17:        infoMarker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_INFO);
18:        infoMarker.setAttribute(IMarker.LINE_NUMBER, element
19:            .getLineNumber());
20:        infoMarker.setAttribute(IMarker.TRANSIENT, true);
21:        infoMarker.setAttribute(IMarker.MESSAGE,
22:            "Code was commented out by Dead Code Detector");
23:    }
24:
25:    // Element aus der Registry entfernen
26:    DeadElementsRegistry registry = CDCDetector.getInstance().getRegistry(
27:        element.getResource().getProject());
28:    registry.removeElement(element);
29:
30:    pm.worked(1);
31:    return new MarkerUndoChange(element, attributes, infoMarker);
32: }

```

*Listing 20: perform-Methode der Klasse MarkerChange*

Im letzten Schritt wird das markierte Element aus der Registry, genauer gesagt aus dem Set unbenutzter Elemente für die Darstellung im Dependencies View, entfernt. Ganz aus der Registry darf das Element jedoch nicht entfernt werden, damit es, samt seiner Abhängigkeiten, bei einer Undo-Operation wieder aufgefunden werden kann. Die Methode liefert als Undo-Change ein Objekt vom Typ `org.intoj.cdcd.refactoring.change.markers.MarkerUndoChange` zurück, das mit Referenzen auf das Element, die Attribute des Warn-Markers und den Info-Marker instanziiert wird. In der perform-Methode dieser Klasse wird das Element in die Liste abhängiger Elemente der Projekt-Registry eingefügt und die Marker-Änderungen werden wieder rückgängig gemacht.

Für das eigentliche Auskommentieren toter Deklarationselemente ist die Klasse `org.intoj.cdcd.refactoring.change.CommentOutChange` zuständig. Da es in Eclipse nicht möglich ist, Kommentar-Knoten in den abstrakten Syntaxbaum einzufügen (es gibt zwar den Knoten-Typ `ASTNode.LINE_COMMENT`, Knoten diesen Typs können jedoch nur

verwendet werden, um Kommentare aus dem AST auszulesen), müssen die Änderungen direkt in der entsprechenden Sourcecode-Datei vorgenommen werden. Die Klasse `CommentOutChange` ist aus diesem Grund von `org.eclipse.ltk.core.refactoring.TextFileChange` abgeleitet. Eine weitere Besonderheit ist, dass die Positionen, an denen Kommentar-Zeichen eingefügt werden, nicht vor der Ausführung der `Change` bestimmt werden können: müssen mehrere Deklarationselemente gemeinsam entfernt werden, weil Abhängigkeiten zwischen ihnen bestehen, und befinden sich diese Elemente in der gleichen Sourcecode-Datei, kann das Einfügen von Kommentar-Zeichen an einer Stelle zur Verschiebung der Positionen anderer Elemente führen. Die Position jedes Deklarationselements wird daher in der `perform`-Methode der `CommentOutChange` bestimmt. Dadurch dass alle Änderungen in einer `CompositeChange` zusammengefasst sind, mit einem `Change`-Objekt pro Element, werden solche Positions-Änderungen automatisch berücksichtigt.

Änderungen an einer Datei werden durch ein `org.eclipse.text.edits.TextEdit`-Objekt repräsentiert. `TextEdit`-Instanzen werden einer `TextFileChange` über die Methode `addEdit(TextEdit edit)` hinzugefügt, dabei muss zunächst ein Haupt-Edit mittels Aufruf von `setEdit(TextEdit edit)` gesetzt werden. Alle Änderungen werden in der `perform`-Methode vorgenommen. Im Konstruktor der Klasse `CommentOutChange` wird als Haupt-Edit ein `MultiTextEdit`-Objekt gesetzt, das mehrere Edits zusammenfasst. Listing 21 zeigt einen Auszug aus der Klasse `CommentOutChange`. In der `perform`-Methode wird geprüft, ob ein Zeilen- oder Block-Kommentar eingefügt wird (Zeile 9), danach werden die entsprechenden Edits bestimmt und dem Haupt-Edit hinzugefügt. In den Zeilen 25 bis 27 wird die Änderung durchgeführt. Hierzu muss die betroffene `Compilation Unit` zunächst geöffnet werden, damit die Änderungen übernommen werden.

Die Methode `isLineCommentNeeded` (Zeile 31) durchsucht den auszukommentierenden Codeabschnitt nach einem öffnenden Block-Kommentar Zeichen („/\*“). Ist ein solches vorhanden, darf in diesen Abschnitt laut den Java-Syntaxregeln<sup>14</sup> kein weiterer Block-Kommentar eingefügt werden. Listing 22 zeigt die Methode `getLineCommentEdits`. In den Zeilen 20 bis 27 wird der Codeabschnitt Zeile für Zeile durchlaufen und dabei ein `InsertEdit` erzeugt, der das Kommentarzeichen „/\*“ am Anfang jeder Zeile einfügt. In Zeile 23 werden die Änderungen auch in das Dokument geschrieben, so dass die Positionen auch dort stets aktuell sind. Listing 23 zeigt die Methode `getBlockCommentEdits`. Im Normalfall werden hier zwei `InsertEdits` erzeugt, einer für das Setzen des öffnenden Block-Kommentar Zeichens „/\*“ am Anfang und des schließenden Block-Kommentar Zeichens

---

14 [http://java.sun.com/docs/books/jls/third\\_edition/html/lexical.html#3.7](http://java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.7)

„\*/“ am Ende des Codeabschnitts. Im Fall von durch Kommata getrennten Variablendeklarationen, wie es z.B. bei Enum-Deklarationen üblich ist, muss das führende Komma ebenfalls auskommentiert werden (vgl. Zeilen 12 bis 23).

```
1: @Override
2: public Change perform(IProgressMonitor pm) throws CoreException {
3:     if (getEdit().getChildrenSize() != 0)
4:         return super.perform(pm);
5:     String source;
6:     try {
7:         source = getSource(this.getFile());
8:         Document doc = new Document(source);
9:         if (!isLineCommentNeeded(doc)) {
10:            for (TextEdit edit : getBlockCommentEdits(doc))
11:                addEdit(edit);
12:        } else {
13:            for (TextEdit edit : getLineCommentEdits(doc))
14:                addEdit(edit);
15:        }
16:    } catch (JavaModelException e) {
17:        throw new RuntimeException(e);
18:    } catch (BadLocationException e) {
19:        MessageDialog.openError(PlatformUI.getWorkbench()
20:            .getActiveWorkbenchWindow().getShell(), "CDCD Error",
21:            "An error occured durin performing Comment Out");
22:
23:        return new NullChange();
24:    }
25:    element.getCompilationUnit().open(pm);
26:    Change undo = super.perform(pm);
27:    element.getCompilationUnit().close();
28:    return undo;
29: }
30:
31: private boolean isLineCommentNeeded(Document doc)
32:     throws BadLocationException {
33:     int offset = element.getOffset();
34:     int length = element.getLength();
35:     int n;
36:
37:     n = doc.search(offset, BLOCK_START, true, false, false);
38:     if (n > -1 && n <= offset + length) {
39:         return true;
40:     }
41:     return false;
42: }
```

Listing 21: Methoden perform und isLineCommentNeeded der Klasse CommentOutChange

```

1: private TextEdit[] getLineCommentEdits(Document doc)
2:     throws BadLocationException {
3:     int numberOflines = 0;
4:     TextEdit[] edits = null;
5:     int i = 0;
6:     int selectionLength = 0;
7:
8:     int offset = element.getOffset();
9:     int length = element.getLength();
10:
11:     if (offset == -1 || length == -1) {
12:         return new TextEdit[0];
13:     }
14:
15:     numberOflines = doc.getNumberOfLines(offset, length);
16:     i = doc.getLineOfOffset(offset);
17:     edits = new TextEdit[numberOflines];
18:     int pos = 0;
19:
20:     for (int n = i; n < i + numberOflines; n++) {
21:         selectionLength += doc.getLineLength(n);
22:         int lineoffset = doc.getLineOffset(n);
23:         doc.replace(lineoffset, doc.getLineLength(n), LINE_COMMENT
24:             + doc.get(lineoffset, doc.getLineLength(n)));
25:         edits[pos] = new InsertEdit(lineoffset - 2 * pos, LINE_COMMENT);
26:         pos++;
27:     }
28:     return edits;
29: }

```

*Listing 22: Methode für das Hinzufügen der Zeilen-Edits in der Klasse CommentOutChange*

```

1: private TextEdit[] getBlockCommentEdits(Document doc)
2:     throws BadLocationException {
3:     int offset = element.getOffset();
4:     int length = element.getLength();
5:
6:     if (offset == -1 || length == -1) {
7:         return new TextEdit[0];
8:     }
9:
10:    TextEdit[] edits = null;
11:
12:    if (element.isVariable()) {
13:        int n = doc.search(offset, ",", false, false, false);
14:        if (n > -1 && n < offset) {
15:            if (doc.get(n + 1, offset - n - 1).trim().isEmpty()) {
16:                edits = new TextEdit[2];
17:                edits[0] = new InsertEdit(n, BLOCK_START);
18:                edits[1] = new InsertEdit(n + (offset - n) + length - 1,
19:                    BLOCK_END);
20:                return edits;
21:            }
22:        }
23:    }
24:    edits = new TextEdit[2];
25:    edits[0] = new InsertEdit(offset, BLOCK_START);
26:    edits[1] = new InsertEdit(offset + length, BLOCK_END);
27:
28:    return edits;
29: }

```

*Listing 23: Methoden für das Hinzufügen der Block-Edits in der Klasse CommentOutChange*

Der Ablauf beim Erzeugen eines Change-Objekts in der Klasse RemoveRefactoring stimmt prinzipiell mit dem beschriebenen überein, es gibt jedoch einige Abweichungen, vgl. Listing 24.

```
1: @Override
2: public Change createChange(IProgressMonitor pm) throws CoreException,
3:     OperationCanceledException {
4:     CDCDChange change = new CDCDChange(getName());
5:     QuickFixUtil util = new QuickFixUtil();
6:
7:     for (DeadElement element : elements) {
8:         IMarker marker = util.findMarker(element);
9:         change.add(new MarkerChange(element, marker, false));
10:        change.add(createChangeForElement(element));
11:    }
12:    return change;
13: }
14:
15: private Change createElementChange(DeadElement element) {
16:     if (element.isType() && element.isSingleType())
17:         return new DeleteResourceChange(
18:             element.getResource().getFullPath(), true);
19:     return new RemoveChange(element, (IFile) element.getResource());
20: }
```

Listing 24: Erzeugen einer Change für das Remove Refactoring

Bei dem Entfernen toter Deklarationselemente werden keine Info-Marker gesetzt, der Parameter `createInfoMarker` im Konstruktoraufzuruf der `MarkerChange` hat daher den Wert `false`. In der Methode `createElementChange` wird zusätzlich geprüft, ob es sich bei dem Deklarationselement um den einzigen in einer Compilation Unit deklarierten Typen handelt (Zeile 16). In diesem Fall wird eine `org.eclipse.ltk.core.refactoring.resource.DeleteResourceChange` zurückgegeben, bei deren Ausführung die Sourcecode-Datei vom Dateisystem gelöscht wird. In allen anderen Fällen kommt die `org.intoj.cdcd.refactoring.change.RemoveChange` zum Einsatz, die, wie die `CommentOutChange`, von `TextFileChange` abgeleitet ist. Listing 25 zeigt die Implementierung der `perform`-Methode dieser Klasse. Das Entfernen von Deklarationselementen wird hier über das Entfernen des entsprechenden Knoten aus dem AST realisiert.

```

1: public Change perform(IProgressMonitor pm) throws CoreException {
2:     CompilationUnit cu = element.getAstRoot();
3:     ASTRewrite rewriter = ASTRewrite.create(cu.getAST());
4:     String bindingKey = element.getBindingKey();
5:     ASTNode declarationNode = cu.findDeclaringNode(bindingKey);
6:     if (declarationNode != null
7:         && declarationNode.getNodeType() ==
8:         ASTNode.VARIABLE_DECLARATION_FRAGMENT) {
9:         declarationNode = declarationNode.getParent();
10:    }
11:    rewriter.remove(declarationNode, null);
12:    TextEdit edits = rewriter.rewriteAST();
13:    addEdit(edits);
14:
15:    return super.perform(pm);
16: }

```

*Listing 25: perform-Methode der Klasse RemoveChange*

#### 4.3.3.4 Entfernen aller toten Deklarationselemente

Das Auskommentieren oder Entfernen aller toten Deklarationselemente kann über das Menü des Dead Elements View vorgenommen werden. Hierfür wurde der Erweiterungspunkt `org.eclipse.ui.viewActions` genutzt: in der Datei `plugin.xml` wurde ein entsprechender Eintrag hinzugefügt, in dem angegeben ist, in welchem View das Menü erscheinen und welche Einträge es haben soll. Zu jedem Menüeintrag muss eine Action-Klasse angegeben werden, die bei der Auswahl des Menüpunkts ausgeführt wird. Für das Auskommentieren ist die Action `CommentAllOutViewActionDelegate`, für das Entfernen die `RemoveAllViewActionDelegate` zuständig, beide sind von `IViewActionDelegate` abgeleitet und liegen im Paket `org.intoj.cdcd.actions`.

Die run-Methoden dieser Klassen entsprechen den der Marker Resolution-Klassen, mit dem Unterschied, dass hier alle toten Elemente, die über den Aufruf von `CDCDetector.getInstance().getAllDeadElements()` erhalten werden, an das entsprechende Refactoring übergeben werden.

### 4.3.4 Theoretische Aspekte

#### 4.3.4.1 Eindeutigkeit der optimalen Lösung

Beim Lösen des Constraintsystems wird nicht explizit eine Lösung gesucht, in der so viele Zugriffsmodifikatoren wie möglich den Wert *absent* haben, sondern als optimale Lösung diejenige betrachtet, in der die Summe der Werte aller Zugriffsmodifikatoren am Größten ist. Der Beweis dafür, dass es dabei keine Lösung geben kann, die weniger unbenutzte

Deklarationselemente, als eine Lösung mit dem gleichen Wert enthält, stützt sich auf folgende Gegebenheiten:

- 1) Die Funktion  $\alpha$  berechnet den minimalen notwendigen Modifikator, um den Zugriff auf eine Deklaration zu gewährleisten. Das Ergebnis dieser Berechnung kann also nie den Wert *absent* haben, da in diesem Fall die Deklaration nicht mehr vorhanden ist und somit auch kein Zugriff erfolgen kann.
- 2) In jedem Constraint wird der Wert eines Modifikators  $\langle d \rangle$  eingeschränkt, indem eine Bedingung in Form einer Ungleichung formuliert wird, wobei insgesamt folgende Fälle möglich sind (für alle gilt dabei  $d, d', d^* \in D$ ):
  - (1)  $\langle d \rangle$  muss größer als ein  $\alpha$ -Wert, d.h. eine Konstante aus der Menge  $A = \{ public, package, protected, private \}$  sein (Inh-2-Relation, Dyn-2-Relation)
  - (2) Es gibt ein  $\langle d' \rangle$ , so dass  $\langle d \rangle$  kleiner oder gleich einem  $\alpha$ -Wert sein muss, falls  $\langle d' \rangle < absent$  gilt und  $\langle d \rangle$  einen beliebigen Wert annehmen kann, falls  $\langle d' \rangle = absent$  gilt (Acc-1-Relation, Inh-1-Relation, Dyn-1-Relation)
  - (3) Es gibt ein  $\langle d' \rangle$ , so dass  $\langle d \rangle$  den Wert *public* haben muss, falls  $\langle d' \rangle < absent$  gilt und  $\langle d \rangle$  einen beliebigen Wert annehmen kann, falls  $\langle d' \rangle = absent$  gilt (Acc-2-Relation, Sub-2-Relation)
  - (4) Es gibt ein  $\langle d^* \rangle$ , so dass  $\langle d \rangle$  kleiner oder gleich dem Wert eines  $\langle d' \rangle$  sein muss, falls  $\langle d^* \rangle < absent$  gilt und  $\langle d \rangle$  einen beliebigen Wert annehmen kann, falls  $\langle d^* \rangle = absent$  gilt (Sub-1-Relation)
  - (5) Gibt es ein  $\langle d \rangle$ , dessen Wert von keinem Constraint eingeschränkt wird (d.h.  $\langle d \rangle$  taucht in keinem Constraint auf der linken Seite der Ungleichung auf), kann  $\langle d \rangle$  einen beliebigen Wert annehmen, was bedeutet, dass das Constraintsystem erfüllt ist, wenn alle Constraints erfüllt sind und  $public \leq \langle d \rangle \leq absent$  gilt.

- 3) Die Menge  $D$  der Deklarationselemente ist endlich und enthält  $n$  Elemente.

Seien nun  $X$  und  $Y$  Lösungen des Constraintsystems, d.h. Mengen mit Elementen aus  $A$ , deren Summe den größten Wert hat, bei dem das Constraintsystem erfüllt ist.

$Absent_X, Absent_Y$  seien wie folgt definiert:

$$Absent_X \subseteq X, \forall \langle d_x \rangle \in Absent_X : \langle d_x \rangle = absent,$$

$$Absent_Y \subseteq Y, \forall \langle d_Y \rangle \in Absent_Y: \langle d_Y \rangle = absent$$

und seien  $S_X, S_Y$  die Summen aller Werte in  $X$  bzw.  $Y$

Wenn nun  $S_X = S_Y \wedge |Absent_X| < |Absent_Y|$  gilt, muss es ein  $d_i \in D$  geben, so dass die Bedingung  $\langle d_i \rangle \in X < absent \wedge \langle d_i \rangle \in Y = absent$  erfüllt ist. Damit wäre aber  $S_X < S_Y$ , folglich muss es ein  $d_{i+1} \in D$ , so dass  $\langle d_{i+1} \rangle \in Y < \langle d_{i+1} \rangle \in X < absent$  gilt.

Aus  $\langle d_{i+1} \rangle \in X < absent$  und (5) folgt, dass es ein Constraint geben muss, das den Wert von  $\langle d_{i+1} \rangle$  einschränkt. Dabei kann es sich nicht um ein Constraint der Typen (1)-(3) handeln, da  $\langle d_{i+1} \rangle$  in jedem dieser Fälle von einem konstanten Wert beschränkt wird und damit in beiden Lösungen den gleichen, maximal möglichen Wert hat.

Es muss also, laut (4), ein weiteres Element  $d_{i+2} \in D$  geben und ein Constraint, das  $\langle d_{i+1} \rangle \leq \langle d_{i+2} \rangle$  fordert. Da die maximale Summe gesucht wird, wird  $\langle d_{i+1} \rangle$  in der Lösung den gleichen Wert wie  $\langle d_{i+2} \rangle$  haben. Für das Ausgangsproblem bedeutet es, dass auch  $\langle d_{i+2} \rangle \in Y < \langle d_{i+2} \rangle \in X < absent$  gelten muss und damit ein weiteres Element existieren muss, das den Wert von  $\langle d_{i+2} \rangle$  einschränkt.

Aus (6) folgt damit, dass schließlich  $\langle d_n \rangle \in Y < \langle d_n \rangle \in X < absent$  gelten und es ein Constraint vom Typ (1)-(3) geben muss, in dem  $\langle d_n \rangle$  auf der linken Seite vorkommt. Das bedeutet wiederum, dass der Wert von  $\langle d_n \rangle$  von einem Konstanten Wert beschränkt wird und  $\langle d_n \rangle$  damit in beiden Lösungen den gleichen Wert haben muss, wenn alle Modifikatoren mit ihrem maximal möglichen Wert in die maximale Summe eingehen.

Für einen Wert  $\langle d_{n-1} \rangle$  gilt damit  $\langle d_{n-1} \rangle \leq \langle d_n \rangle \rightarrow \langle d_{n-1} \rangle = \langle d_n \rangle$  und somit auch  $\langle d_{i+1} \rangle = \langle d_{i+2} \rangle$ , was bedeutet, dass  $\langle d_{i+1} \rangle$  in beiden Lösungen den gleichen Wert hat, sofern es sich um maximale Lösungen handelt, also auch  $S_X < S_Y$  gilt.

#### 4.3.4.2 Notwendigkeit der Änderung aller Sichtbarkeiten

In der Lösung des Constraintsystems sind alle Zugriffsmodifikatoren minimal, für das Aufspüren toten Codes sind die Modifikatoren von im Programm benötigten Elementen jedoch nicht relevant, und werden beim Entfernen toten Codes auch nicht verändert. Es könnte allerdings die Frage aufkommen, ob das Programm nach dem Entfernen toter Codeelemente tatsächlich in einen korrekten Zustand überführt wird, denn es wird nur eine Teilmenge der Constraint-Variablen verändert. Da hier aber nur Elemente betroffen sind, dessen Existenz keinen Einfluss auf die Korrektheit des Programms hat, und nur unbenutzte

Elemente von ihnen abhängen können, liegt nahe, dass nach dem Entfernen aller toten Elemente das Constraintsystem immer noch lösbar sein muss, woraus sich wiederum die Korrektheit des Programms erschließt.

## 5 Analyse des CDCD-Plugins

Dieses Kapitel befasst sich mit den praktischen Anwendungsmöglichkeiten des CDCD-Plugins und seinem Verhalten unter realen Bedingungen. Um diese analysieren zu können, wurden mehrere Java-Projekte, bei denen es sich um Opensource-Anwendungen handelt, auf das Vorhandensein toten Codes mit dem CDCD untersucht. Die gefundenen toten Codeelemente wurden anschließend vom CDCD entfernt und das Programm auf seine syntaktische und semantische Korrektheit geprüft. Der Prüfung auf semantische Korrektheit erfolgte dabei über die Ausführung der JUnit-Tests des Programms. Alle Tests erfolgten auf einem Notebook mit Betriebssystem Linux Ubuntu, einem Core2Duo-Prozessor mit 1.83 GHz-Taktung und 1 GB Hauptspeicher.

### 5.1 Anwendungsszenarien und Ergebnisse

Für die Test wurden folgende Projekte eingesetzt

- [apache.commons.codec](http://commons.apache.org/codec/)<sup>15</sup> - eine Klassenbibliothek für das Kodieren von Daten
- [Dead Code Detector](https://dcd.dev.java.net/)<sup>16</sup> - ein Werkzeug zum Aufspüren toten Codes in Java-Bytecode
- [jaxen](http://jaxen.codehaus.org/)<sup>17</sup> - eine XPath Klassenbibliothek
- [jbook](http://jbook.sourceforge.net/)<sup>18</sup> - ein Programm zum Herunterladen und Lesen elektronischer Texte
- [Jester](http://jester.sourceforge.net/)<sup>19</sup> - ein Programm zum Aufspüren von Code, der nicht durch JUnit-Tests abgedeckt ist
- [JHotDraw](http://www.jhotdraw.org/)<sup>20</sup> - ein Framework zur Erstellung von graphischen Editoren
- [schemalizer](http://sourceforge.net/projects/schemalizer/)<sup>21</sup> - ein Werkzeug zur Erstellung von XML-Schemata basierend auf XML-Dateien

Bei Projekten mit JUnit-Tests wurden jeweils zwei Tests durchgeführt, mit und ohne Betrachtung der JUnit-Test-Klassen bei der Suche nach totem Code<sup>22</sup>, woraus sich insgesamt zwölf Testläufe ergaben. Tabelle 1 zeigt die die Anzahl der Deklarationselemente in den verwendeten Projekten.

---

15 <http://commons.apache.org/codec/>

16 <https://dcd.dev.java.net/>

17 <http://jaxen.codehaus.org/>

18 <http://jbook.sourceforge.net/>

19 <http://jester.sourceforge.net/>

20 <http://www.jhotdraw.org/>

21 <http://sourceforge.net/projects/schemalizer/>

22 Zu diesem Zweck wurden die JUnit-Testklassen aus dem Projekt entfernt

<b>Projekt</b>	<b>Anzahl Typen</b>	<b>Anzahl Methoden</b>	<b>Anzahl Attribute</b>
codec 1.3 mit Tests	43	441	88
codec 1.3 ohne Tests	25	213	64
DeadCodeDetector 1.1	31	302	133
jaxen 1.1.1 mit Tests	306	2043	416
jaxen 1.1.1 ohne Tests	205	1331	315
jbook 1.4	16	126	56
Jester 1.3.7b mit Tests	78	434	119
Jester 1.3.7b ohne Tests	45	210	78
JHotDraw 6.0.1b mit Tests	543	5145	864
JHotDraw 6.0.1b ohne Tests	349	3250	693
schemalizer 0.40 mit Tests	35	120	115
schemalizer 0.40 ohne Tests	32	112	114

*Tabelle 1: Testprojekte*

Die Anzahl von toten Deklarationselemente, die vom CDCD-Plugin im jeweiligen Projekt gefunden wurden, ist in Tabelle 2 zusammengefasst.

<b>Projekt</b>	<b>Tote Typen</b>		<b>Tote Methoden</b>		<b>Tote Attribute</b>	
	<b>Anzahl</b>	<b>Prozent</b>	<b>Anzahl</b>	<b>Prozent</b>	<b>Anzahl</b>	<b>Prozent</b>
codec mit Tests	0	0%	1	0%	2	2%
codec ohne Tests	25	100%	213	100%	64	100%
DeadCodeDetector	0	0%	2	1%	0	0%
jaxen mit Tests	4	1%	47	2%	19	5%
jaxen ohne Tests	205	100%	1331	100%	315	100%
jbook	0	0%	5	4%	9	16%
Jester mit Tests	2	3%	22	5%	0	0%
Jester ohne Tests	0	0%	4	2%	0	0%
JHotDraw mit Tests	11	2%	227	4%	62	7%
JHotDraw ohne Tests	33	9%	285	9%	75	11%
schemalizer mit Tests	2	6%	6	4%	6	5%
schemalizer ohne Tests	2	6%	6	5%	6	5%

*Tabelle 2: Toter Code in Testprojekten*

## 5.2 Verifikation der Funktionsweise des CDCD-Plugin

Nach dem Entfernen toten Codes wurde für jedes Programm geprüft, ob es frei von Syntaxfehlern ist und ob seine JUnit-Tests, sofern vorhanden, fehlerfrei verlaufen. Das Ergebnis dieser Prüfung ist in Tabelle 3 dargestellt, der man entnehmen kann, dass das CDCD-Plugin nicht in jedem Fall fehlerfrei arbeitet.

<b>Projekt</b>	<b>Syntaxfehler</b>	<b>JUnit-Tests fehlerfrei</b>
codec mit Tests	0	ja
codec ohne Tests	0	-
DeadCodeDetector	0	-
jaxen mit Tests	0	nein
jaxen ohne Tests	0	-
jbook	0	-
Jester mit Tests	0	ja
Jester ohne Tests	0	-
JHotDraw mit Tests	1	-
JHotDraw ohne Tests	1	-
schemalizer mit Tests	0	ja
schemalizer ohne Tests	0	-

*Tabelle 3: Fehler nach Entfernen toten Codes*

Im Projekt jaxen schlug ein JUnit-Test fehl, da eine Methode, auf die nur mittels Java Reflection API zugegriffen wurde, vom CDCD entfernt wurde. Hier stößt der CDCD, wie jedes Werkzeug, das auf statischer Codeanalyse basiert, an seine Grenzen, da Methodenaufrufe per Reflection durch solch eine Analyse nicht erkannt werden können, vgl. Abschnitt 6.3.

Der Syntaxfehler im Projekt JHotDraw (Abb. 7), resultierte daraus, dass der Konstruktor der Klasse GraphNode vom CDCD entfernt wurde und dadurch das als final deklarierte Attribut node nicht initialisiert wird. Der Grund hierfür ist dabei aber nicht unbedingt in der Funktionsweise des CDCD zu suchen: der genannte Konstruktor wurde in einer API-Methode referenziert, auf die es keine anderen Referenzen gibt, und der CDCD sie daher als tot betrachtet.

```
class GraphNode {
    double x=0.0, y=0.0;
    /*double dx=0.0;*/
    /*double dy=0.0;*/
    final Figure node;

    /*GraphNode(Figure newNode) {
        node = newNode;
        update();
    }*/

    void update() {
        Point p = node.center();
        if (Math.abs(p.x - Math.round(x))>1 ||
            Math.abs(p.y - Math.round(y))>1) {
            x = p.x;
            y = p.y;
            //System.out.println(this+" has new coords: "+x+", "+y+"\n");
        }
    }
}
```

Abbildung 7: Syntaxfehler im Projekt JHotDraw

Im Fall einer eigenständigen Anwendungen kann solch ein Fehler allerdings nicht auftreten, da bei einer Referenz auf die Methode update (die eine minimale Sichtbarkeit erfordert) es auch eine Referenz auf den Konstruktor der Klasse geben müsste. Im Projekt mit JUnit Tests ist der Fehler auf mangelnde Abdeckung mit Testfällen zurückzuführen und kann kaum durch eine Änderung am Constraintsystem behoben werden: da final kein Zugriffsmodifikator ist, kann die Abhängigkeit des Attributs zum Konstruktor im Constraintsystem nicht modelliert werden. Ein Constraint, dass die Sichtbarkeit eines Konstruktors von der Sichtbarkeit der umschließenden Klasse abhängig macht, würde zwar Abhilfe schaffen, aber zugleich das Aufspüren unbenutzter Konstruktoren verhindern.

### 5.3 Qualitative Analyse der Ergebnisse

Nachdem die korrekte Arbeitsweise des CDCD-Plugin zum Großteil bestätigt wurde verschafft dieser Abschnitt einen Überblick darüber, welche Funde sich jeweils hinter den Zahlen aus Tabelle 5.2 verbergen.

#### 5.3.1 Projekt codec

Im Projekt codec ohne Tests wurde der gesamte Code vom CDCD als unbenutzt eingestuft, was als Folge fehlender Einsprungpunkte zu deuten ist. Eine weitere Schlussfolgerung aus dieser Tatsache ist, dass im Constraintsystem alle Deklarationselemente berücksichtigt wurden.

Im Projekt mit JUnit Tests stellen die Testklassen Einsprungpunkte dar – das API-Constraint fordert für sie eine minimale Sichtbarkeit. Die geringe Anzahl toter Deklarationselemente

lässt dabei nicht zwingend auf die Abwesenheit toten Codes schließen, da nur solche Deklarationselemente als unbenutzt eingestuft werden, die nicht mit JUnit-Tests abgedeckt sind. Bei den Funden handelt es sich im Einzelnen um

- eine Konstante mit Javadoc-Annotation
- eine öffentliche Methode, ebenfalls mit Javadoc-Annotation
- ein als `protected` deklariertes Attribut ohne Javadoc-Annotation

In den ersten zwei Fällen kann nicht mit Sicherheit bestimmt werden, ob die Deklarationselemente tatsächlich ohne Auswirkungen auf die von Clients der Bibliothek benötigte Programmfunktion entfernt werden können, oder ob lediglich Tests für diese Elemente fehlen. In jedem Fall liefern die CDCD-Warmarker aber Hinweise auf Verbesserungsmöglichkeiten des Quellcodes. Beim letzten Fund scheint es sich tatsächlich um ein totes Attribut zu handeln, vor allem in Anbetracht der Abwesenheit von Accessor-Methoden.

### 5.3.2 Projekt DeadCodeDetector

In diesem Projekt wurden vom CDCD nur zwei tote Konstruktoren gefunden. Da es sich bei dem Dead Code Detector (DCD) ebenfalls um ein Werkzeug zum Erkennen toten Codes handelt, lässt sich vermuten, dass das Projekt einer Prüfung mit dem DCD selbst unterzogen wurde. Bei der Anwendung des DCD auf sich selbst wurde für die beiden Konstruktoren keine Warnung gemeldet, dafür aber für unbenutzte Interface-Methoden in mehreren Klassen. Auf diesen Umstand wird in Abschnitt 5.5 näher eingegangen.

### 5.3.3 Projekt jaxen

Wie beim Projekt codec, wurden in jaxen alle Deklarationselemente als unbenutzt markiert. Für die Analyse sind daher nur die Ergebnisse aus dem Testlauf mit Berücksichtigung der JUnit-Testklassen interessant. Zwei vom CDCD als tot eingestuften Klassen sind Subklassen einer Klasse mit Namen `NodeTest`. Trotz Namensgebung hat diese Klasse keinen Bezug zu den JUnit-Tests des Projekts, sondern wird intern von jaxen verwendet. Diese Klassen enthalten zusammen 2 der gefundenen Attribute und 12 der gefundenen Methoden. Bei den anderen beiden Typen handelt es sich jeweils um mit `@deprecated` annotierte Klassen, die zusammen 16 der gefundenen Methoden und 5 der gefundenen Attribute enthalten. Bei den restlichen Attributen handelt es sich um benannte Konstanten. Die `@deprecated`-Annotation lässt die fehlenden Testfälle für diese Klassen erklären. Allerdings ist in der Liste toter Deklarationselemente eine Methode enthalten, die in einer mit `@deprecated` annotierten

Klasse mit Kommentar „this class will become non-public in the future; use the interface instead“ deklariert ist (für die Klasse selbst sind Tests vorhanden). Hier stellt sich wieder die Frage, ob es sich tatsächlich um eine unbenutzte Methode oder bloß mangelnde Testabdeckung handelt. Eine Unterscheidung zwischen ungetesteten Methoden und totem Code in den CDCD-Funden von jaxen ist auch bei den restlichen Funden schwer möglich, im Einzelnen handelt es sich um:

- Methoden mit Javadoc-Kommentaren, die in öffentlichen Klassen oder Interfaces, ebenfalls mit Javadoc, deklariert sind
- Eine Methode, die in einer Klasse ohne Javadoc-Kommentare deklariert ist, für deren restliche Methoden jedoch Tests vorhanden sind
- Methoden ohne Javadoc, aus Klassen, die Javadoc-Kommentare, sowie Methoden mit Javadoc-Annotationen enthalten
- Methoden einer Klasse, die nicht zum öffentlichen Interface gehören, das die Klasse implementiert. Hier scheint nur die API-Funktionalität getestet worden zu sein.
- Einen parameterlosen Konstruktor mit leerem Rumpf.
- Accessor-Methoden

#### **5.3.4 Projekt jbook**

Zu jbook wurden zwar wenige Funde vom CDCD gemeldet, jedoch handelt es sich bei ihnen mit großer Wahrscheinlichkeit um tote Deklarationselemente:

- vier als paketlokal deklarierte Attribute, die nirgendwo referenziert werden und zu denen keine Accessor-Methoden vorhanden sind.
- zwei Methoden (paketlokaler und privater Sichtbarkeit) und zwei Konstanten, die zusammen eine Funktion realisieren, die offensichtlich nicht benötigt wird – jbook ist als eigenständiges Programm und nicht als Framework entwickelt worden, so dass es sich nicht um API-Methoden handeln kann.

#### **5.3.5 Projekt Jester**

Jester scheint zu 98 Prozent frei von totem Code zu sein – im Projekt ohne Testklassen fand der CDCD lediglich zwei unbenutzte Konstruktoren von Exception-Klassen, sowie zwei unbenutzte Methoden. Bei den Methoden handelt es sich um eine unbenutzte Interface-Methode, die ihrem Namen, `testRunningCommand`, nach zu Test-Zwecken vorhanden ist, sowie ihre Implementierung aus einer Subklasse. Der größere Anteil toten Codes im Projekt mit JUnit-Tests ist auf die Existenz von Mock-Klassen, sowie zwei Test-Klasse mit den

Namen `Untested` und `NotTested` zurückzuführen, die nicht durch JUnit-Tests abgedeckt, aber auch nicht Teil des Programms sind. Die beiden Methoden `testRunningCommand` tauchen im Projekt mit Tests, im Gegensatz zu den Exception-Konstruktoren, nicht auf, da Testfälle für diese Methoden vorhanden sind.

### 5.3.6 Projekt JHotDraw

Auch in diesem Projekt ist die Unterscheidung zwischen totem und ungetestetem API-Code nicht immer einfach, da in allen Klassen, für die der CDCD Warnungen meldete, Javadoc-Kommentare vorhanden waren. Für folgende Deklarationselemente (vgl. auch Beispiele aus Abschnitt 2.2) lässt sich hingegen sagen, dass es sich tatsächlich um toten Code handelt:

- Vier private Attribute, die auch von Eclipse erkannt wurden.
- Zwei paketlokale Attribute ohne Accessor-Methoden und Referenzen
- Ein als `protected` deklariertes Attribut, das ein gleichnamiges privates Attribut der Superklasse verdeckt und nirgendwo (auch nicht in Subklassen) benutzt wird
- Acht private Methoden
- Zwei Methoden, die lediglich eine Exception mit der Meldung „not implemented“ werfen
- Fünf Methoden mit leeren Rümpfen, von denen in drei der Rumpf auskommentiert wurde.
- Eine Methode mit dem Namen `value()`, die lediglich null zurückgibt

### 5.3.7 Projekt schemalizer

Dieses Projekt weist einen geringen Anteil an totem Code auf. Bei den vom CDCD als unbenutzt eingestuft Deklarationselementen handelt es sich um benannte Konstanten, ein Listener-Interface mit zugehöriger `EventObject`-Klasse, sowie eine leere, als `public` deklarierte Methode. Bei dem unbenutzten Interface `XMLFileChangeListener` lässt sich vermuten, dass es im Laufe der Entwicklung durch das Interface `XMLFileClosedListener` ersetzt wurde, da nur diese Teilfunktionalität tatsächlich im Programm benötigt wurde, vgl. auch Abschnitt 2.2.

## 5.4 Laufzeitverhalten

Tabelle 4 zeigt den Zeitbedarf für den Aufbau und das Lösen des Constraintsystems, sowie für die Analyse der Lösung. Insgesamt lässt sich sagen, dass der CDCD eine recht gute Performance, selbst bei großen Projekten wie JHotDraw, aufweist, durch die er sich durchaus als in der Praxis einsetzbar erweist.

Projekt	Zeit in Sekunden		
	Aufbau	Lösen	Analyse
codec mit Tests	3.28	2.11	1.77
codec ohne Tests	0.53	0.17	4.16
DeadCodeDetector	1.24	0.93	0.32
jaxen mit Tests	5.78	16.01	7.53
jaxen ohne Tests	5.85	6.61	8.77
jbook	0.48	0.2	0.47
Jester mit Tests	1.73	0.8	0.33
Jester ohne Tests	1.83	0.34	0.28
JHotDraw mit Tests	12.12	80.03	30.01
JHotDraw ohne Tests	10.42	40.5	28.45
schemalizer mit Tests	1.01	0.58	0.79
schemalizer ohne Tests	0.7	0.41	0.54

Tabelle 4: Laufzeitverhalten des CDCD Plugins

## 5.5 Bewertung der Ergebnisse

Aus den Testergebnissen wird deutlich, dass toter Code in ausgelieferten Programmen durchaus vorhanden sein kann, auch wenn nicht alle getesteten Projekte in gleichem Maß davon betroffen sind, und dass bestimmte Formen toten Codes mit dem CDCD-Plugin erkannt werden können. Der CDCD erkannte unbenutzte Typen, Methoden und Attribute mit der Sichtbarkeit von private bis public. Die Funde vom CDCD können jedoch nicht immer definitiv als toter Code betrachtet werden. Das größte Problem liegt derzeit in der Differenzierung von API- und unbenutzter Funktionalität eines Programms. Die Abdeckung mit JUnit-Tests kann hier zwar eine Richtlinie sein, aber auch nur das, denn es kann einerseits Fälle geben, in denen keine JUnit-Tests zu API-Methoden vorhanden sind, und andererseits kann das Projekt eine gute Abdeckung mit Testfällen aufweisen, allerdings so, dass auch unbenutzte Funktionalität getestet wird und damit für den CDCD unerkant bleibt.

Während es bei Frameworks und Klassenbibliotheken zur Zeit keine für den CDCD erkennbaren Einsprungspunkte außer JUnit-Tests gibt, kann beim Aufspüren toten Codes in eigenständigen Programmen auf die Berücksichtigung von JUnit-Tests verzichtet werden. Beim Entfernen toten Codes vor der Auslieferung hätte es den zusätzlichen Vorteil, dass alle Tests automatisch entfernt würden. Bei Klassenbibliotheken sollte eine vollständige Testabdeckung allerdings gewährleistet sein, da sonst, wie im Projekt JHotDraw, ungetesteten Elemente als unbenutzt eingestuft und entfernt werden – mit einem möglichen Syntaxfehler als Ergebnis.

Bei Einsatz von Reflection ist ebenfalls Vorsicht geboten, denn die korrekte Funktionsweise des CDCD ist nicht mehr gewährleistet, wenn Zugriffe auf Deklarationselemente nur mittels Reflection erfolgen – eine statische Analyse kann solche Zugriffe nicht erkennen und die Elemente werden als tot eingestuft.

Eine weitere Frage, die es zu klären gibt, ist, ob die durch den CDCD bestimmte Menge toter Deklarationselemente zu einem Projekt auch wirklich alle vorkommenden toten Elemente enthält. Zwar wurde im Test gezeigt, dass bei fehlenden Einsprungspunkten jedes Deklarationselement als tot eingestuft wird, also dass alle Deklarationselemente im Constraintsystem vertreten sind, einen Schwachpunkt hat der Sichtbarkeitsconstraint-basierte Ansatz dennoch: Methoden einer Klasse, die von einem Interface geerbt, aber nicht benutzt werden, bleiben wegen des Sub-1-Relation-Constraints unerkannt<sup>23</sup> und damit auch tote Codeelemente, die nur in diesen Methoden referenziert werden.

---

23 Es sei denn, das Interface selbst wird an keiner Stelle im Programm benutzt.

## 6 Existierende Ansätze zum Aufspüren toten Codes in Java-Programmen

In diesem Abschnitt werden in der Praxis eingesetzte Verfahren zum Aufspüren toten Codes vorgestellt und die zu diesem Zweck eingesetzten Techniken statischer Programmanalyse diskutiert. Abschnitt 6.1 befasst sich mit der Analyse von Bytecode, in Abschnitt 6.2 wird die Analyse von Sourcecode betrachtet. Abschnitt 6.3 geht auf die Schwächen statischer Programmanalyse ein.

### 6.1 Analyse von Java-Bytecode

Unbenutzte Codefragmente können in Bytecode-Dateien über einen Aufrufgraphen, einen Abhängigkeitsgraphen oder durch die Analyse der Programmstruktur erkannt werden. Diese Verfahren, sowie auf ihnen basierende Werkzeuge, werden in diesem Abschnitt vorgestellt.

#### 6.1.1 Aufbau eines Aufrufgraphen

Mit Hilfe eines Aufrufgraphen kann die Menge aller Methoden bestimmt werden, die bei der Ausführung eines Programms aufgerufen werden könnten. Dabei werden ausgehend von Einsprungspunkten wie main-Methoden alle vorkommenden Methodenrümpfe nach weiteren Methodenaufrufen durchsucht. Zu beachten hierbei ist, dass auch alle Methoden, die aufgrund dynamischer Bindung aufgerufen werden können, ermittelt werden müssen [Tip03].

Es gibt zahlreiche Algorithmen für die Konstruktion eines Aufrufgraphen. Im Folgenden werden nur diejenigen betrachtet, die für das Aufspüren unbenutzten Codes in der Praxis verwendet werden: RTA, XTA und VTA.

##### ***Rapid Type Analysis (RTA)***

RTA [Bac97] basiert auf der Analyse der Klassenhierarchie (Class Hierarchy Analysis, CHA) [Dea95], bei der der Aufrufgraph unter Berücksichtigung der Vererbungsbeziehungen aufgebaut wird. Für jeden gefundenen Methodenaufruf wird geprüft, ob die Methode in Subtypen des statischen Typs des Empfängers überschrieben wurde. Ist dies der Fall, werden die entsprechenden Subtyp-Methoden in die Menge aller potentiell aufrufbaren Methoden aufgenommen. Da jedoch nicht bestimmt werden kann, welche dieser Methode zur Laufzeit tatsächlich aufgerufen wird, enthält diese Menge unter Umständen Methoden, die nie aufgerufen werden. In RTA wird diese Menge aufgrund von Informationen über im

Programm instantiierte Klassen reduziert. Dazu werden zusätzlich zu Methodenaufrufen im Programm vorkommende Konstruktoraufrufe betrachtet und daraus die Menge instantiiertes Typen,  $S$ , bestimmt. In die Ergebnismenge werden nur Methoden der Typen aus  $S$  aufgenommen.

Bei diesem Verfahren wird jedoch nicht berücksichtigt, dass es Typen in  $S$  geben kann, die aufgrund fehlender Sichtbarkeit für den Aufrufer nicht erreichbar sind, die Ergebnismenge kann daher immer noch unbenutzte Methoden enthalten. Die Erkennung solcher Methoden ist in XTA, dem Nachfolger von RTA, realisiert.

### **XTA**

XTA [Tip00] erweitert den Algorithmus von RTA, indem statt der Menge  $S$  für jede Methode eine Menge  $S_M$  von Typen, die in innerhalb dieser Methode erreichbar sind und für jedes Attribut eine Menge von Typen, dessen Objekte es referenzieren kann, bestimmt werden [Tip00] [Tip02]. Im Gegensatz zu der Vorgehensweise des CDCD-Plugins, wird die Erreichbarkeit nicht aus der Analyse von Sichtbarkeiten, sondern aus Objektfluss-bezogenen Constraints ermittelt. In die Menge  $S_M$  einer Methode  $m$  werden alle Typen, die innerhalb von  $m$  instantiiert werden, aufgenommen. Der Objektfluss wird bei Methodenaufrufen der Form  $a.n()$  innerhalb von  $m$  wie folgt modelliert:

Methoden  $m'$  der Subtypen des statischen Typen  $A$  von  $a$ , die in  $S_M$  enthalten sind, werden in die Ergebnismenge aufgenommen. Subtypen der Parametertypen von  $m'$ , die in  $S_M$  enthalten sind, sowie  $A$ , werden in  $S_M$  aufgenommen. Die Subtypen des Rückgabetypen von  $m'$ , die in  $S_M$  enthalten sind, werden in  $S_M$  aufgenommen. Analog dazu wird zu jedem Attribut  $x$  die Menge  $S_x$  berechnet [Tip00].

Laut der in [Tip02] beschriebenen Untersuchung sind RTA und XTA wesentlich effektiver als CHA: bei Anwendung von CHA wurden 47.1% der Methoden eines Programms als unbenutzt erkannt, mit RTA und XTA waren es beim gleichen Programm entsprechend 59.0% und 59.2%. Laut [Tip00] enthält der mit XTA konstruierte Graph durchschnittlich 1.6% weniger Methoden als der mit RTA. Obwohl RTA und XTA fast die gleichen Ergebnisse liefern, hat XTA einige Vorteile. Der Aufrufgraph enthält weniger redundante Kanten und es können deutlich mehr dynamisch gebundener Methoden einer einzigen Zielmethode zugeordnet werden [Tip00] [Tip02].

### **Analyse der Typen von Variablen**

Bei der Analyse der Typen von Variablen (Variable Types Analysis, VTA) [Sun00] wird die durch CHA ermittelte Menge potentiell aufrufbarer Methoden, wie bei XTA, unter Berücksichtigung des Objektflusses reduziert, jedoch wird statt der Festlegung von

Constraints ein weiterer Graph, der Typ-Ausbreitung-Graph (type propagation graph) gebildet. Dazu wird zunächst für jedes Attribut einer Klasse, deren Methoden im Aufrufgraphen enthalten sind, sowie für jeden formalen Parameter, jede lokale Variable, den impliziten Parameter `this` und den Rückgabewert jeder der im Aufrufgraphen enthaltenen Methode ein Knoten erzeugt. Im zweiten Schritt werden die Kanten hinzugefügt: für jede Zuweisung der Form `a = b` von `b` nach `a` und für jeden Methodenaufruf der Form `a = b.m(p1, p2, p3)` von `b` zum entsprechenden `this`-Objekt, vom Rückgabewert (sofern vorhanden) zu `a`, und von jedem Argument `pi` zum entsprechenden Parameter von `m`. Anhand dieses Graphen kann schließlich zu jeder Variable die Menge der erreichbaren Typen bestimmt werden. Bei der Auflösung dynamischer Methodenaufrufe kommen dann nur Methoden von Typen in Betracht, die der Empfänger referenzieren kann.

Laut [Sun00] können mit VTA bis zu 6% mehr tote Methoden, als mit RTA, erkannt werden. Ein Vergleich mit XTA wurde nicht durchgeführt, die Autoren von [Sun00] gehen jedoch davon aus, dass der Aufbau des Typ-Ausbreitung-Graphen in nur zwei Schritten zusätzlichen Vorteil bei der Analyse großer Programme gegenüber der iterativen Vorgehensweise von RTA bietet.

### ***Praktische Anwendung von Aufrufgraphen zum Aufspüren toten Codes***

Das von IBM entwickelte Extraktionswerkzeug für Java-Applikationen Jax [Tip99] [Tip02] [Tip03] benutzt XTA, um die für ein Programm notwendigen Teile verwendeter Klassenbibliotheken aus dem Bytecode zu extrahieren. In einer früheren Version wurde für den Aufbau des Aufrufgraphen der RTA-Algorithmus benutzt. Jax ist demnach in der Lage, wie auch das CDCD-Plugin, unbenutzte Typen, Methoden und Attribute in einem Programm ausfindig zu machen. Darüber hinaus können unbenutzte von einem Interface geerbte Methoden als solche erkannt werden. Das Entfernen dieser Methoden ist zwar nicht möglich, da die umschließende Klasse danach nicht mehr das Interface implementiert, Jax löscht in diesem Fall jedoch die Rümpfe der Interface-Methoden [Tip99]. Laut [Tip03] stuft Jax alle Attribute, auf die nur von unbenutzten Methoden aus zugegriffen wird, und alle Klassen, die nur unbenutzte Methoden enthalten und im Programm nicht instantiiert werden, als tot ein. Es wird allerdings nicht darauf eingegangen, ob bei dieser Vorgehensweise Attribute, die nicht in Methoden verwendet, jedoch in Attributdeklarationen referenziert werden, fälschlicherweise entfernt werden könnten. Das gleiche gilt für Klassen, die zwar nur tote Methoden, aber im Programm im genannten Sinn verwendete Attribute enthalten. Durch Sichtbarkeits-Constraints, die die Grundlage des CDCD-Plugins bilden, wird hingegen das Löschen solcher Elemente verhindert.

Jax verfolgt ein anderes Ziel, als das CDCD-Plugin, nämlich die Größe von Programmen,

und damit deren Übertragungszeit, zu minimieren. Neben dem Entfernen unbenutzter Methoden werden zusätzliche Optimierungen wie Namenskompression und Verschmelzen von in der Klassenhierarchie benachbarten Klassen vorgenommen [Tip03]. Da einige der durchgeführten Transformationen und Optimierungen den Code verändern, so dass er weniger verständlich wird [Tip99], lässt Jax sich als Hilfswerkzeug für Programmierer kaum nutzen, da aus dem erhaltenen Bytecode der ursprüngliche Sourcecode nicht mehr genau bestimmt werden kann.

Im Soot Bytecode Optimization Framework [Val99] sind verschiedene Bytecode-Optimierungen implementiert, die auf einem Aufrufgraphen, zu dessen Aufbau CHA, RTA oder VTA eingesetzt werden können, basieren, und zur besseren Performance von Java-Programmen beitragen. Zu diesen Optimierungen gehört auch die Eliminierung von toten Zuweisungen, unerreichbarem Code sowie unbenutzten lokalen Variablen. Die Erkennung solcher Arten toten Codes geht über die Funktionalität des CDCD-Plugins hinaus, da dieser nur Codeelemente, die einen Zugriffsmodifikator besitzen, erkennen kann. In [Val99] wird keine Aussage darüber gemacht, ob Soot ebenfalls in der Lage ist, derartige Elemente zu erkennen, bei der Verwendung von VTA sollte dies aber prinzipiell möglich sein.

### **6.1.2 Aufbau eines Abhängigkeitsgraphen**

Der in [Ray99] beschriebene Ansatz für die Extraktion von Bytecode für eingebettete Systeme basiert auf einem Entity-Relationship-Abhängigkeitsgraphen, mit dessen Hilfe die Menge für eine Applikation notwendigen Entities einer Bibliothek bestimmt wird. Dabei sind Entities Komponenten, wie ClassOrInterface, Method oder Field und als Beziehungen werden die Definitions- und Benutzungsbeziehung betrachtet. Nach diesem Modell wird ein Abhängigkeitsgraph gebildet und alle enthaltenen Entities aus dem Bytecode extrahiert. In [Tip02] wird auf eine Schwäche dieser Vorgehensweise hingewiesen: es werden nur Programmkonstrukte entfernt, die nicht referenziert werden. Code, der von totem Code referenziert wird, wird auf diese Weise, im Gegensatz zum Constraint-basiertem Ansatz, nicht erkannt.

### **6.1.3 Analyse des abstrakten Syntaxbaums**

Mit Jamit [JamURL] können Sichtbarkeiten von Java-Codeelementen reduziert werden. Die minimale Sichtbarkeit ist dabei dead, was bedeutet, dass das entsprechende Element ohne Auswirkungen auf die Programmfunktion entfernt werden kann. Die Vorgehensweise zur Bestimmung toter Codeelemente entspricht also der der Sichtbarkeiten-Constraints: der

abstrakte Syntaxbaum wird analysiert und die minimalen Sichtbarkeiten aufgrund von Constraints bestimmt. Da Jamit als Eingabe Bytecode-Dateien verwendet, werden mit den in Jamit verwendeten Constraints nur die für die Java Virtual Machine relevanten Aspekte der Zugriffsmodifikatoren modelliert [Ste09]. Jamit findet tote Codeelemente nicht nur, sie werden auch automatisch gelöscht und als Ergebnis die optimierten Bytecode-Dateien erstellt. Es gibt auch keine Möglichkeit, für Codeelemente anzugeben, dass deren Sichtbarkeit erhalten werden soll, was dazu führen kann, dass Elemente fälschlicherweise als tot eingestuft werden, z.B. wenn es keine Einsprungspunkte gibt. Die einzige Möglichkeit, dies zu verhindern, ist, ausführliche Testszenarios, die all diese Methoden abdecken, zum Projekt zu erstellen und Jamit auf der main-Methode der Testsuite auszuführen [JamURL]. In dieser Hinsicht ist das CDCD-Plugin deutlich flexibler und nicht so radikal in der Vorgehensweise.

#### **6.1.4 Weitere Ansätze**

Der Dead Code Detector (DCD) [DcdURL] benutzt die ASM-Bibliothek [AsmURL], um toten Java-Code aufzuspüren und zu entfernen. Mit Hilfe der ASM-Bibliothek können kompilierte Java-Klassen (Bytecode) generiert, transformiert und analysiert werden [Bru07]. DCD erkennt unbenutzte Klassen, Interfaces, Methoden (inklusive von Interfaces geerbten Methoden) und Attribute, sowie lokale Variablen. Obwohl der DCD in dieser Hinsicht mehr als das CDCD-Plugin, leistet ist er schwieriger in der Handhabung, da es ein eigenständiges Programm ist und als Ergebnis eine Liste mit allen unbenutzten Elementen liefert. Das CDCD-Plugin ist in Eclipse integriert und erlaubt es dadurch, einfach durch gefundene tote Elemente im Sourcecode zu navigieren.

FindBugs [FbgURL] analysiert Bytecode-Dateien auf sogenannte Bug Patterns – „Muster von fehlerhaftem Programmverhalten in Zusammenhang mit Programmierfehlern“ [All01]. Es können ca. 300 Bug Patterns erkannt werden, darunter auch Patterns, die toten Code betreffen, wie „unbenutztes Attribut“, „unbenutzte private Methode“ und „unbenutzte lokale Variable“. Dank der Plugin-Architektur können jederzeit Detektoren für neue Bug Patterns hinzugefügt werden [Aye07]. Für die Analyse werden die Bibliotheken BCEL [BceURL] und ASM verwendet. Manche Detektoren nutzen darüber hinaus Informationen aus dem Kontrollfluss-Graphen und der Datenfluss-Analyse [Aye07]. Das Aufspüren toten Codes ist nicht das Hauptanliegen von FindBugs und seine Verwendung mit diesem Ziel ist eher fraglich, da alle Arten toten Codes, die FindBugs erkennen kann, auch von Eclipse erkannt werden.

## 6.2 Analyse von Java-Sourcecode

In diesem Abschnitt werden zwei verschiedene Ansätze zum Auffinden toter Codeelemente durch Sourcecode-Analyse diskutiert. Zunächst werden die Techniken beschrieben, von denen Compiler Gebrauch machen, um toten Code zu entfernen, anschließend werden Werkzeuge beschrieben, die das gleiche Ziel, wie das CDCD-Plugin verfolgen – Unterstützung der Softwareentwickler.

### 6.2.1 Eliminierung toten Codes als Compiler-Optimierung

Beim Übersetzen des Sourcecodes werden durch Compiler diverse Optimierungen vorgenommen, die auf eine Verbesserung des Laufzeitverhaltens, sowie eine Minimierung des Speicherplatzbedarfs des Programms, abzielen. Eine davon ist die Eliminierung toten Codes (Dead Code Elimination), zu dessen Identifizierung vor allem die Techniken Lebendigkeits-Analyse (Live Variable Analysis) und statische Einmalzuweisung (Static Single Assignment form, SSA) von Bedeutung sind.

Die Lebendigkeits-Analyse ist eine statische rückwärtige Datenfluss-Analyse, mit der für jeden Programmpunkt alle Variablen, die beim Ausgang aus diesem Punkt lebendig sind, bestimmt werden können. Dabei wird eine Variable als lebendig angesehen, wenn sie im weiteren Programmverlauf verwendet und vor der Verwendung nicht neu definiert wird [Nie98].

Die statische Einmalzuweisung ist eine Zwischendarstellung, in der zu jeder Variablen genau einmal eine Zuweisung erfolgt. Dazu wird eine Variable  $x$  des ursprünglichen Programms in Versionen  $x_1$ ,  $x_2$ , usw. aufgeteilt. In einem Programm in SSA-Form können unbenutzte Zuweisungen leicht erkannt und somit auch Variablen, auf die im Programm nicht zugegriffen wird, entfernt werden [Sri03].

Unerreichbarer Code kann durch Kontrollfluss-Analyse aufgedeckt werden. Das Verfahren ähnelt der Analyse des Aufrufgraphen – ausgehen vom Einsprungspunkt wird der Kontrollfluss-Graph traversiert und dadurch die erreichbaren Blöcke ermittelt [Deb00].

Alle diese Optimierungen haben gemeinsam, dass tote Elemente nur innerhalb einzelner abgeschlossener Codefragmente, wie Methoden oder Anweisungsblöcken, gefunden werden können. Im Gegensatz zum Sichtbarkeitsconstraint-basierten Ansatz wird nicht das Programm als Ganzes, sondern einzelne Programmteile analysiert, die Erreichbarkeit ist also nur so viel von Bedeutung, als dass nur die im jeweiligen Fragment deklarierten und sichtbaren Elemente betrachtet werden. Konkret für Java bedeutet es, dass unbenutzte Elemente, die eine höhere Sichtbarkeit als private besitzen, auf diese Weise nicht aufgedeckt

werden können: wird z.B. eine öffentliche tote Methode m innerhalb einer toten privaten Methode n aufgerufen, lässt sich nicht feststellen, dass m ebenfalls entfernt werden kann.

Das CDCD-Plugin kann also als eine Ergänzung zu Compiler-Optimierungen beim Aufspüren toten Codes, und keinesfalls als eine Alternative dazu, betrachtet werden, da es all die Arten toten Codes identifizieren kann, bei denen die beschriebenen Ansätze machtlos sind, und umgekehrt.

## **6.2.2 Dead Code Detektoren als Hilfswerkzeuge für Softwareentwickler**

Bei den in diesem Abschnitt betrachteten Anwendungen zum Aufspüren toten (Source-) Codes steht die Unterstützung des Entwicklers bei der Code-Wartung im Vordergrund. Ist das Analysewerkzeug in eine IDE wie Eclipse integriert, kann toter Code auf einfache Weise entfernt oder korrigiert werden.

Das Eclipse-Plugin Unused Code Detector (UCD) [UcdURL] durchsucht den Sourcecode nach als public, protected oder paketlokal deklarierten Typen, Methoden und Attributen, die nicht referenziert werden. Für die Suche wird das Eclipse-Paket org.eclipse.jdt.core.search verwendet. Darüber hinaus können zyklische Abhängigkeiten zwischen unbenutzten Elementen erkannt werden. Ähnlich wie Jamit und das CDCD-Plugin, prüft UCD, ob Sichtbarkeiten reduziert werden können. Da diese Entscheidung wiederum nur auf Referenzen basiert, sind Fehler nach Änderung der Sichtbarkeit nicht ausgeschlossen. So wird z.B. für als protected deklarierte Attribute und Methoden, die nicht referenziert werden, aber in Subklassen überschrieben werden, vorgeschlagen, diese als private zu deklarieren, oder gar zu entfernen, selbst wenn es Referenzen auf die überschreibenden Methoden gibt.

PMD [PmdURL] prüft den Code auf bestimmte Regeln, ähnlich den Bug Patterns von FindBugs. Dabei wird der abstrakte Syntaxbaum des Programms traversiert und auf Verstöße gegen die Regeln untersucht. Anschließend wird ein Report, z.B. als XML-Datei ausgegeben. Das Hauptziel von PMD ist die Verbesserung der Codequalität, die Suche nach totem Code steht dabei, wie bei FindBugs, nicht im Vordergrund: es können ebenfalls nur tote private Methoden und Attribute, sowie lokale Variablen, erkannt werden, was PMD wenig geeignet zum Aufspüren toten Codes macht.

Bei allen bisher betrachteten Werkzeugen handelt es sich um Open-Source-Projekte, es gibt aber auch eine Reihe kommerzieller Anwendungen, die Entwickler bei der Optimierung des Sourcecode unterstützen. Der AppPerfect Java Code Analyzer [AppURL] wendet 750 Programmierregeln an und findet tote Codefragmente beliebiger Sichtbarkeit. Parasoft Jtest [ParURL] führt Pattern- und Fluß-basierte Analyse von Sourcecode durch und findet neben

zahlreichen Fehlern und Verletzungen von Programmierstandards auch unbenutzten Code.

### 6.3 Einschränkungen statischer Programmanalyse

Selbst die besten statischen Analyseverfahren sind in einigen Fällen machtlos. So ist das Aufspüren toten dynamisch geladenen Codes, oder Codes, auf den mittels Reflection zugegriffen wird, durch statische Analyse generell nicht möglich. In Jax kann der Benutzer jedoch auf diese Art erfolgte Aufrufe explizit angeben, sie werden dann, wie auch Konstruktoren von dynamisch geladenen Klassen, als Einsprungspunkte behandelt [Tip99].

In Jamit kann die Verwendung von Reflection ebenfalls spezifiziert werden [JamURL].

Ein weiteres Problem kann auftreten, wenn im Programm Klassen von Typen einer externen Bibliothek abgeleitet werden, dessen Code nicht zur Verfügung steht, sondern lediglich die Methodensignaturen bekannt sind. Werden dabei Methoden überschrieben, können sie aufgrund dynamischer Bindung von anderen Bibliotheksmethoden aufgerufen werden: wenn die Anwendungsklasse A Subtyp eines Bibliotheks-Interfaces I ist, und A die Methode m für I implementiert, könnte im Programm einer Methode n einer Bibliotheks-Klasse B, die einen Parameter vom Typ I hat, das A-Objekt übergeben bekommen. Wird im Rumpf von n B.m aufgerufen, bleibt der Aufruf von A.m für die Analyse unbemerkbar und die Methode wird entfernt, sofern sie nicht anderweitig im Programm verwendet wird. In Jax wird daher angenommen, dass jede überschriebene bzw. implementierte Bibliotheksmethode erreichbar ist [Tip99].

## 7 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Möglichkeit des Einsatzes von Sichtbarkeitsconstraints zum Aufspüren toten Codes untersucht. Dabei wurde der Begriff „toter Code“ auf unbenutzte Deklarationselemente von Java-Programmen, die einen Zugriffsmodifikator besitzen und ohne Auswirkungen auf die Programmfunktion entfernt werden können, eingeschränkt. In den eingesetzten Sichtbarkeitsconstraints aus [Ste09] wird für jedes Deklarationselement die nötige Sichtbarkeit aufgrund von Referenzen auf das Element, sowie der Existenz von Vererbung, Subtyping und dynamischer Bindung abhängig gemacht. Falls kein Constraint eine minimale Sichtbarkeit für ein Element fordert, wird das Element als tot betrachtet, was durch die Zuweisung des Modifikators `absent` ausgedrückt wird, und kann aus dem Sourcecode entfernt werden. Für die Erkennung toten Codes wurden die Constraints aus [Ste09] so modifiziert, dass ein Deklarationselement nur dann eine minimale Sichtbarkeit einen anderen Elements erzwingen kann, wenn es selbst nicht den Modifikator `absent` besitzt. Dadurch können auch zyklische Abhängigkeiten zwischen toten Elementen erkannt werden. Auf dieser Grundlage wurde der Constraint based Dead Code Detector (CDCD), ein Plugin für das IDE Eclipse, entwickelt. Der CDCD baut für die Bestimmung toter Deklarationselemente eines Java-Projekts das Constraintsystem beim Traversieren des abstrakten Syntaxbaums auf und berechnet eine Lösung, die die restriktivsten Zugriffsmodifikatoren enthält. Anschließend werden alle Modifikatoren mit dem Wert `absent` ermittelt und in die entsprechenden Deklarationselemente in einem Eclipse View dargestellt, sowie mit einem Marker versehen, der zwei Quick Fixes anbietet – das Auskommentieren und das Löschen der betroffenen Elemente. Bei der Entfernung eines Deklarationselements werden alle Elemente, von denen es referenziert wird oder anderweitig abhängig ist automatisch entfernt.

Die Funktionsweise des CDCD wurde in zwölf Testszenerien unter Einsatz auf Opensource-Projekten geprüft. Die Ergebnisse dieser Testläufe haben gezeigt, dass bei der Generierung des Constraintsystems alle Deklarationselemente eines Programms berücksichtigt werden, sowie das Entfernen der gefundenen toten Elemente das Programm in den meisten Fällen in einen syntaktisch und semantisch korrekten Zustand überführt. Die semantische Korrektheit wurde dabei mit dem Bestehen der JUnit-Tests des Programms gleichgesetzt. Bei den fehlgeschlagenen Testfällen handelte es sich zum Einen um Deklarationselemente, die fälschlicherweise als tot eingestuft wurden, da Zugriffe auf diese Element im Programm nur über den Reflection-Mechanismus erfolgten und solche Zugriffe durch eine statische Analyse nicht erkannt werden können. Zum anderen gab es false positives, die auf fehlende

Einsprungspunkte in API-Klassen, sowie auf eine mangelnde Abdeckung von API-Methoden mit Testfällen zurückzuführen waren.

Generell lässt sich sagen, dass das Aufspüren toter Deklarationselemente in API-Code nur bedingt erfolgreich sein kann, da keine Einsprungspunkte vorhanden sind und JUnit Tests nur als Anhaltspunkte für mögliche Einsprungspunkte betrachtet werden können. Eine vollständige Testabdeckung führt dabei zu unentdecktem totem Code, eine mangelnde Abdeckung kann false positives zur Folge haben. Hier kann der CDCD jedoch Hinweise auf fehlende Testfälle liefern. Bei eigenständigen Anwendungen kann der CDCD hingegen effektiv eingesetzt und auch dazu genutzt werden, alle Testklassen vor der Auslieferung des Programms zu entfernen.

## **7.1 Ausblick**

Momentan gibt es hinsichtlich der Schwächen des CDCD folgende Überlegungen.

### **7.1.1 API-Elemente**

Für die Analyse von API-Methoden können neben JUnit-Tests auch Javadoc-Kommentare betrachtet und dessen Abwesenheit in öffentlichen Deklarationen als Hinweise auf unbenutzten Code verstanden werden. Bei dieser Vorgehensweise muss jedoch wieder mit false positives gerechnet werden, allerdings werden so Warnungen für nicht dokumentierte Methoden generiert, was neben totem Code als eine Designschwäche betrachtet werden kann. Die Existenz von Javadoc-Kommentaren und JUnit-Tests könnte auch zusammen das Kriterium für eine API-Methode bilden. Im Idealfall sollten diese Möglichkeiten vom Benutzer einstellbar sein.

### **7.1.2 Test-Klassen**

Auch die Berücksichtigung von Testklassen bei der Ermittlung toter Deklarationselemente sollte eine vom Benutzer einstellbare Option sein. Nur so kann auch das Entfernen von Test-Code realisiert in eigenständigen Anwendungen realisiert werden. Die Möglichkeit der Angabe von Paketen, dessen Elemente auf die Existenz toten Codes geprüft werden sollen, kann in diesem Fall ebenfalls hilfreich sein. Außerdem wird dadurch die Suche nach totem Code nur in Programm-internen Paketen ermöglicht. Im Constraintsystem müssen dabei aber weiterhin alle Elemente des Programms berücksichtigt werden, hier wäre also ein zusätzliches Constraint nötig, in dem der Name eines von der Suche ausgenommenen Pakets den Einsprungspunkt ersetzt.

### **7.1.3 Unbenutzte Interface-Methoden**

Das Aufspüren unbenutzter Interface-Methoden in den implementierenden Klassen ist mit dem Constraintbasiert Ansatz ist nicht so einfach umsetzbar. Das Sub-1-Relation-Constraint müsste dafür so modifiziert werden, dass keine minimale Sichtbarkeit für eine Interface-Methode gefordert wird, sondern der Umstand festgehalten wird, dass diese Methode nicht benutzt wird, aber auch nicht entfernt werden darf, solange die Super-Methode vorhanden ist. Eine Möglichkeit wäre die Einführung eines neuen Zugriffsmodifikators, bei dessen Zuweisung nicht die Methode, sondern nur ihr Rumpf gelöscht, bzw. auskommentiert wird. Auf diese Weise könnte beim Einsatz des CDCD-Plugins eine weitere Designschwäche im Code aufgedeckt werden – wenn eine Klasse nur einen geringen Anteil der Methoden eines Interfaces, das sie implementiert, verwendet.

### **7.1.4 Reflection**

Wie bereits erwähnt, kann der Einsatz des Reflection-Mechanismus durch eine statische Analyse nicht automatisch berücksichtigt werden. Eine einfache Möglichkeit wäre, eine neue Annotation für die Constraint-Generierung einzuführen, die Elemente markiert, auf die mittels Reflection zugegriffen wird, so wie es z.B. bei Jax der Fall ist. In diesem Fall wäre das Kennzeichnen solcher Elemente jedoch dem Entwickler überlassen und damit ein nicht immer in Kauf nehmbarer Aufwand, sowie eine potentielle Fehlerquelle.

## Abbildungsverzeichnis

Abbildung 1: Unbenutztes Interface im Projekt schemalizer.....	7
Abbildung 2: Ein Eclipse-Marker.....	17
Abbildung 3: Problem View von Eclipse.....	18
Abbildung 4: CDCD-Dead Elements View.....	19
Abbildung 5: CDCD-Dependencies View.....	19
Abbildung 6: Ein Quick Fix.....	20
Abbildung 7: Syntaxfehler im Projekt JHotDraw .....	52

## Listingverzeichnis

Listing 1: Unerreichbarer Code.....	2
Listing 2: Toter Code in Methoden.....	3
Listing 3: Unbenutztes Attribut als potentielle Fehlerquelle.....	4
Listing 4: Unbenutzte Methoden der Klasse CommandMenu.....	4
Listing 5: Obsolete Methode der Klasse DrawApplet.....	5
Listing 6: Obsolete Methode der Klasse DrawApplication.....	5
Listing 7: Obsolete Attribute im Projekt jbook.....	6
Listing 8: Nötige Sichtbarkeit bei Vererbung.....	11
Listing 9: Nötige Sichtbarkeit bei dynamischer Bindung.....	13
Listing 10: Definition des Kontributors für den Kontextmenü-Eintrag.....	22
Listing 11: Methode DeadCodeSearch.getSolver(ConstraintSystem, IProgressMonitor).....	25
Listing 12: Abhängigkeiten zwischen toten Deklarationselementen.....	27
Listing 13: Die Methode createMarker(DeadElement element).....	30
Listing 14: Definition der Kontributoren für die Views.....	31
Listing 15: Definition des Kontributors für den Marker View.....	32
Listing 16: Die Klasse CDCDMarkerResolutionGenerator.....	34
Listing 17: run-Methode der CommentOutMarkerResolution.....	35
Listing 18: run-Methode der Klasse CommentOutRunnable.....	36
Listing 19: Methoden für das Erzeugen von Change-Objekten im Comment Out Refactoring.....	38
Listing 20: perform-Methode der Klasse MarkerChange.....	39
Listing 21: Methoden perform und isLineCommentNeeded der Klasse CommentOutChange.....	41

Listing 22: Methode für das Hinzufügen der Zeilen-Edits in der Klasse CommentOutChange.....	42
Listing 23: Methoden für das Hinzufügen der Block-Edits in der Klasse CommentOutChange.....	43
Listing 24: Erzeugen einer Change für das Remove Refactoring.....	44
Listing 25: perform-Methode der Klasse RemoveChange.....	45

## Tabellenverzeichnis

Tabelle 1: Testprojekte.....	50
Tabelle 2: Toter Code in Testprojekten.....	50
Tabelle 3: Fehler nach Entfernen toten Codes.....	51
Tabelle 4: Laufzeitverhalten des CDCD Plugins.....	56

## Literaturverzeichnis

- [All01] Allen Eric; Bug patterns: An introduction. Diagnosing and correcting recurring bug types in your Java programs;  
URL: <http://www.ibm.com/developerworks/library/j-diag1.html>; 2001
- [AppURL] <http://www.appperfect.com/products/java-code-test.html#keyfeatures>
- [AsmURL] <http://asm.ow2.org/>
- [Aye07] Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y; Evaluating static analysis defect warnings on production software; Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (San Diego, California, USA, June 13 - 14, 2007)
- [Bac07] Bach, Makus; Design und Implementierung eines Eclipse-Plugins zur Anzeige von möglichen Typgeneralisierungen im Quelltext; Masterarbeit FernuUniversität in Hagen; 2007
- [Bac97] Bacon, D.F.; Fast and Effective Optimization of Statically Typed Object-Oriented Language. Doctoral Dissertation, Computer Science Division, University of California, Berkeley; 1997.
- [BceURL] <http://jakarta.apache.org/bcel/>
- [Bou08] Bouillon P., Großkinsky E., Steimann F; Controlling accessibility in agile projects with the Access Modifier Modifier; Proc. of TOOLS 46; 2008
- [Bru07] Bruneton E; ASM 3.0 A Java bytecode engineering library; URL: <http://download.forge.objectweb.org/asm/asm-guide.pdf>; 2007
- [CreURL] Tamura N; Cream: Class Library for Constraint Programming in Java  
URL: <http://bach.istc.kobe-u.ac.jp/cream/>
- [DcdURL] <https://dcd.dev.java.net/>
- [Dea95] Dean, J., Grove, D., Chambers, C.; Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Proceedings of the 9th European Conference on Object-Oriented Programming; 1995
- [Deb00] Debray, S. K., Evans, W., Muth, R., and De Sutter, B; Compiler techniques for code compaction; ACM Trans. Program. Lang; 2000
- [EclURL] <http://www.eclipse.org/>
- [FbgURL] <http://findbugs.sourceforge.net/>
- [FOW00] Fowler M.; Refactoring. Wie Sie das Design vorhandener Software verbessern; Addison-Wesley Verlag; 2000

- [GosURL] Gosling J, Joy B, Steele G, Bracha G; The Java Language Specification;  
URL: <http://java.sun.com/docs/books/jls/>
- [JamURL] <http://grothoff.org/christian/xtc/jamit/>
- [JavURL] <http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html>
- [Nie98] Nielson F, Nielson H.R., Hankin C.; Principles of Program Analysis; Springer-Verlag; 1998
- [ParURL] <http://www.parasoft.com/jsp/products/home.jsp?product=Jtest>
- [PmdURL] <http://pmd.sourceforge.net/>
- [Ray99] Rayside, D., Kontogiannis, K.; Extracting Java Library Subsets for Deployment on Embedded Systems. In Proceedings of the Third European Conference on Software Maintenance and Reengineering; 1999.
- [Sri03] Srikant, Y. N., Shankar, P.; The compiler design handbook: optimizations and machine code generation; CRC Press; 2003
- [Ste09] Steimann, F., Thies, A.; From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In Proceedings of the 23rd European Conference on ECOOP 2009 --- Object-Oriented Programming; 2009.
- [Sun00] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.; Practical virtual method call resolution for Java. In Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications; 2000
- [Tip00] Tip, F., Palsberg, J.; Scalable propagation-based call graph construction algorithms. In Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications; 2000
- [Tip02] Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., Streeter, D.; Practical extraction techniques for Java. ACM Trans. Program. Lang. Syst. 24, 6; 2002
- [Tip03] Tip, F., Sweeney, P. F., Laffra, C. Extracting library-based Java applications. Commun. ACM 46, 8; 2003
- [Tip99] Tip, F., Laffra, C., Sweeney, P. F., Streeter, D.; Practical experience with an application extractor for Java. SIGPLAN Not. 34, 10; 1999
- [UcdURL] <http://www.ucdetector.org/index.html>
- [Val99] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V. Soot - a Java bytecode optimization framework. In Proceedings of the 1999 Conference of the Centre For Advanced Studies on Collaborative Research; 1999

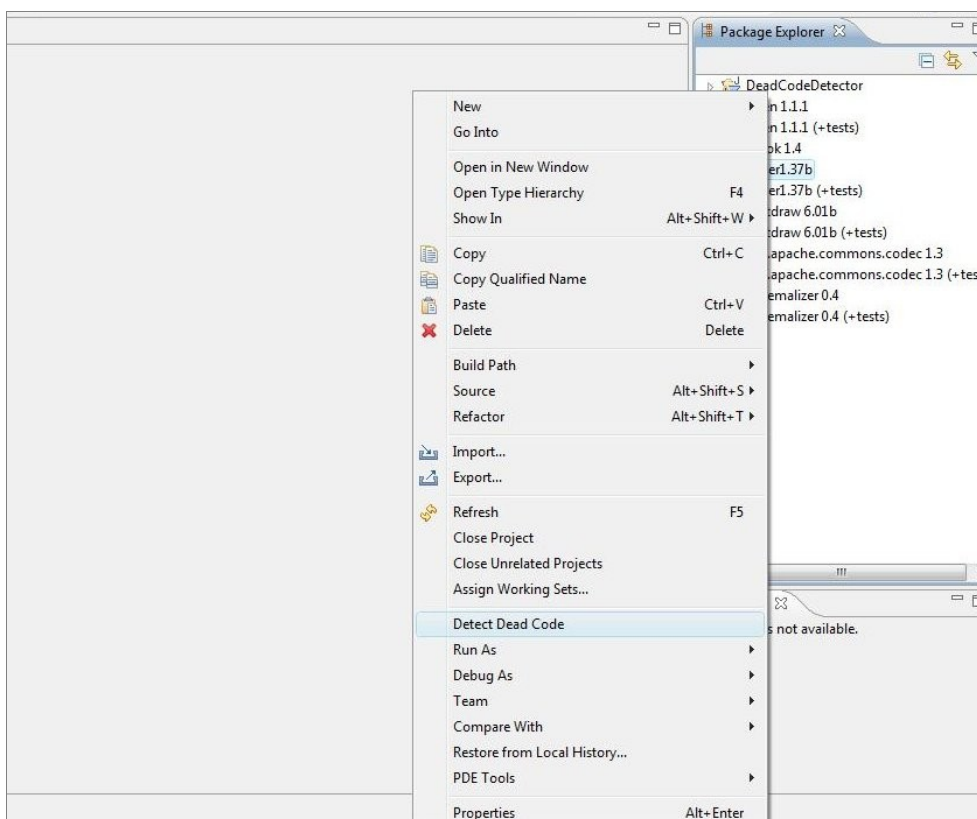
# Anhang A Bedienungsanleitung für das CDCD Plugin

## Installation

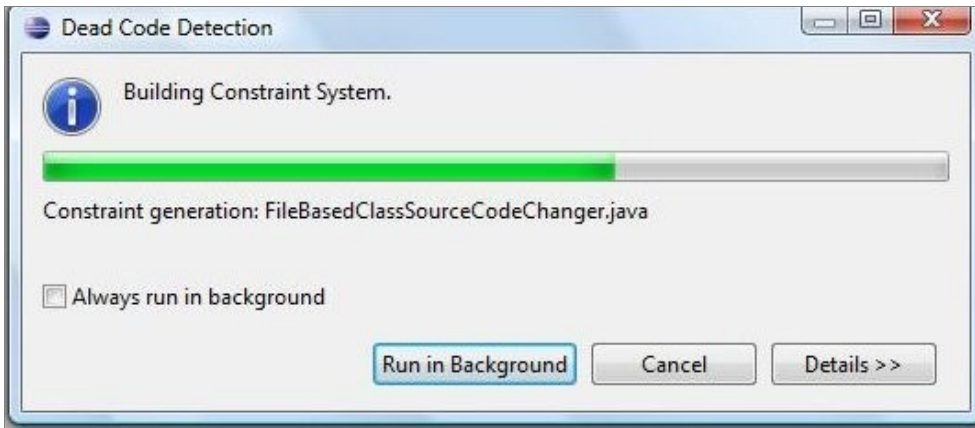
Für die Installation des CDCD Plugins in Eclipse müssen die Dateien `org.intoj.cdcd_1.0.0.jar` und `org.intoj.amm3_1.0.0.jar` in das Verzeichnis `plugins`, das sich im Eclipse-Verzeichnis befindet, kopiert werden. In der beiliegenden Eclipse-Version ist das CDCD Plugin bereits enthalten.

## Bedienung

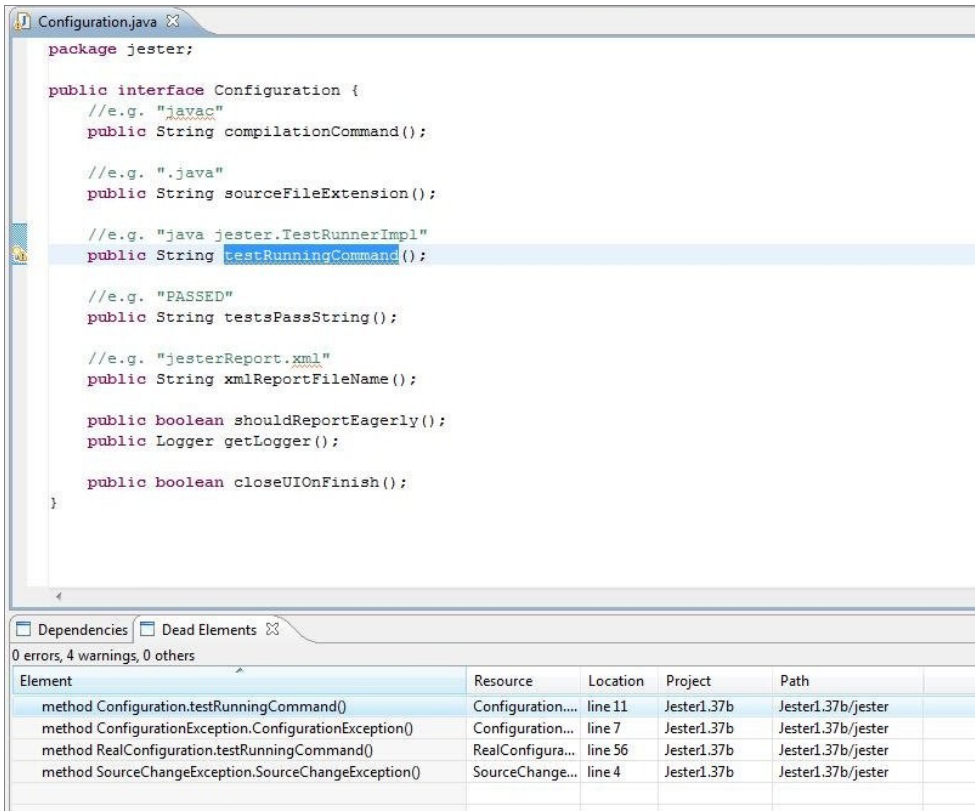
Zum Start der Suche wird der Menüpunkt „Detect Dead Code“ im Kontextmenü eines Projekts ausgewählt:



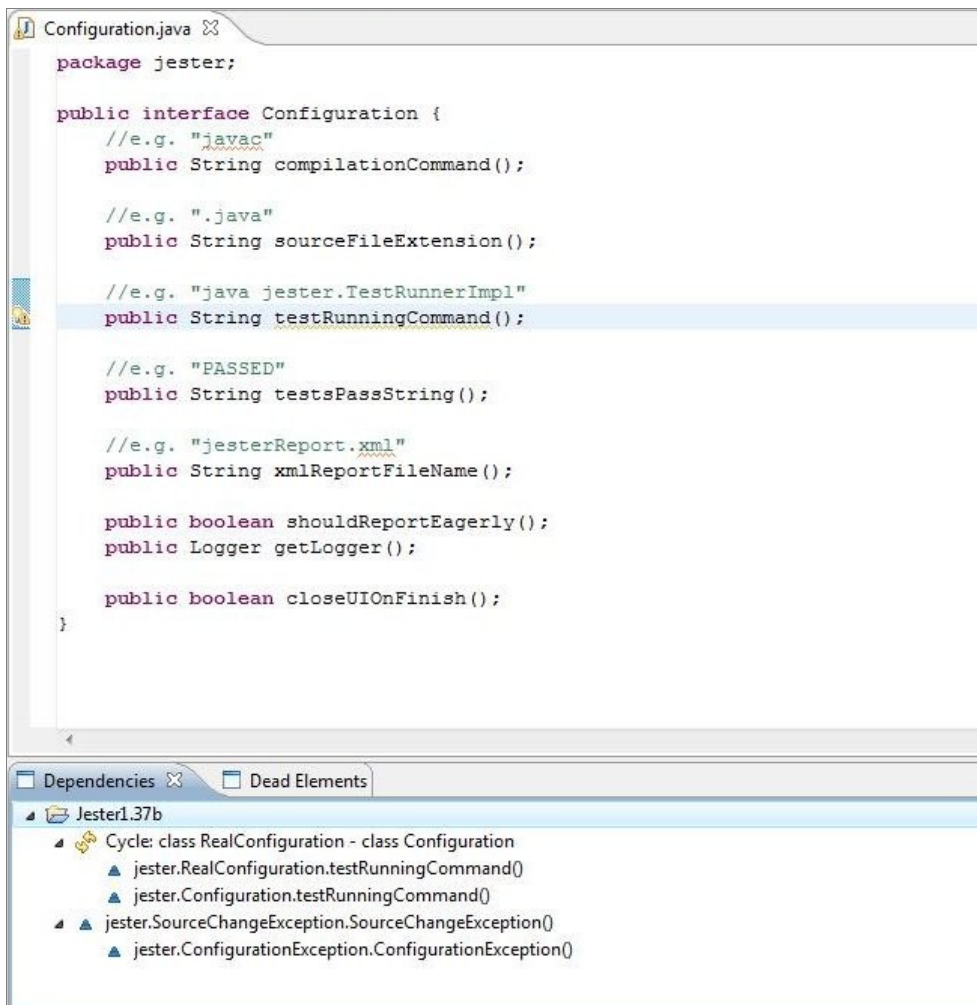
Während der Suche erscheint die Fortschrittsanzeige, über die der Suchlauf abgebrochen werden kann:



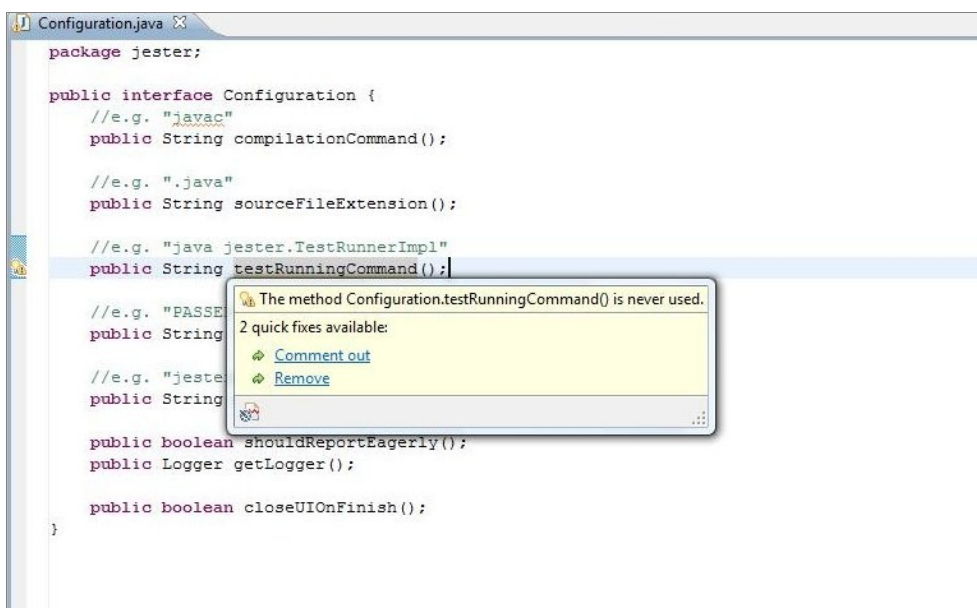
Die gefundenen toten Deklarationselemente werden im Dead Elements View angezeigt.  
 Beim Mausklick auf ein Element wird es im Editor geöffnet:



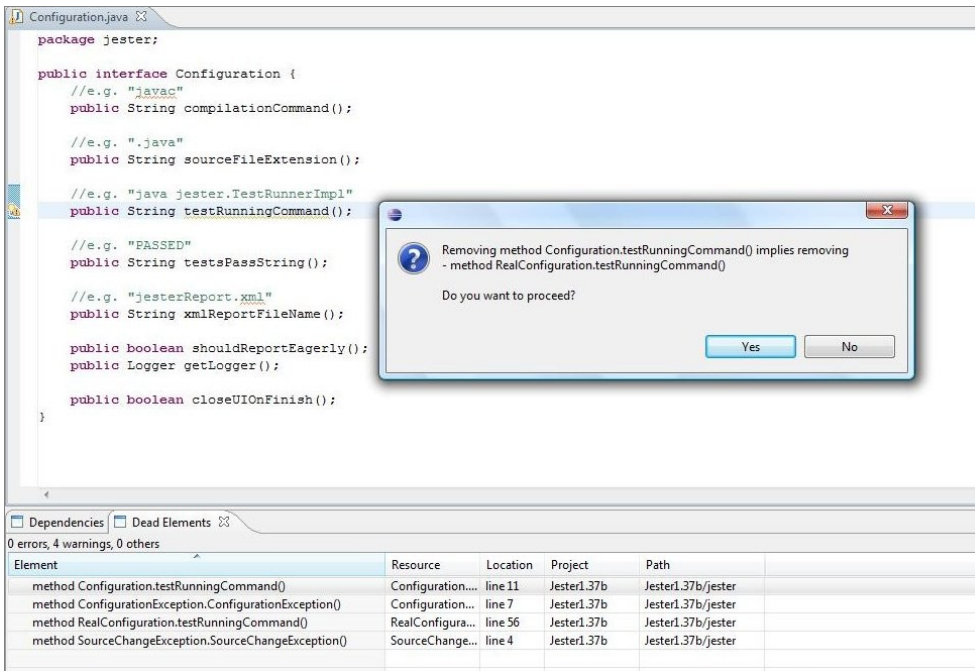
Abhängigkeiten zwischen toten Deklarationselementen werden im Dependencies View angezeigt:



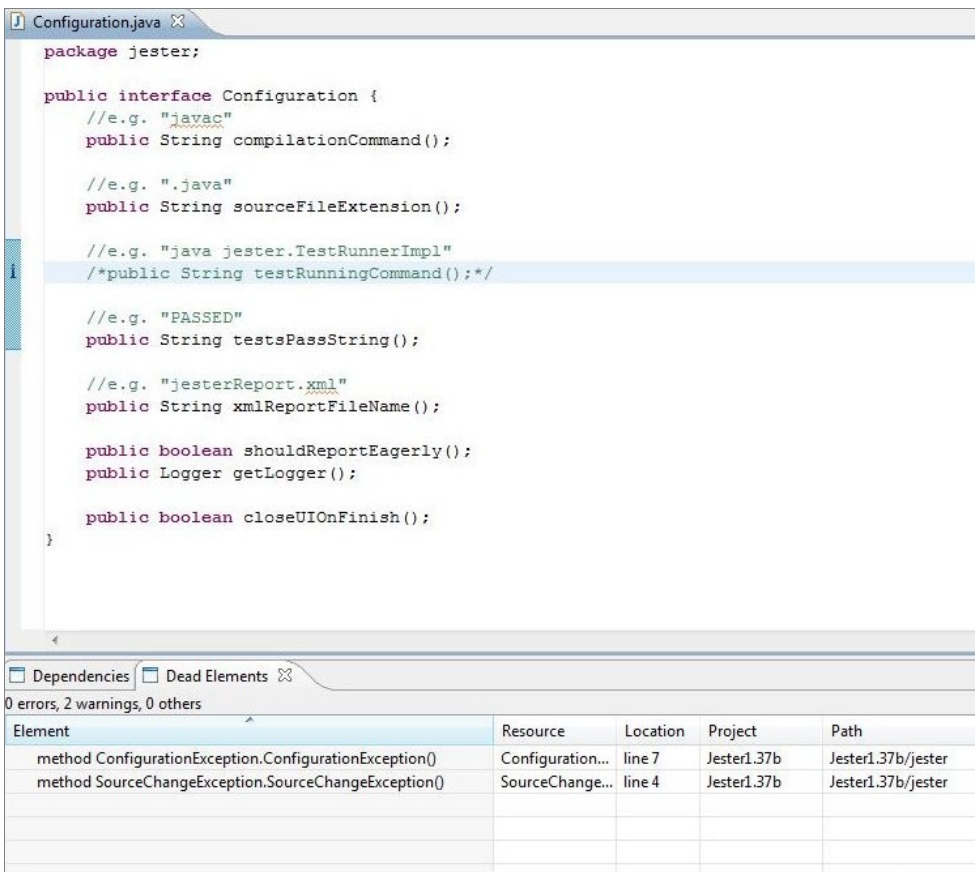
Ein totes Deklarationselement ist mit einem Warn-Marker versehen, zu dem zwei Quick Fixes verfügbar sind – Auskommentieren und Löschen des Elements:



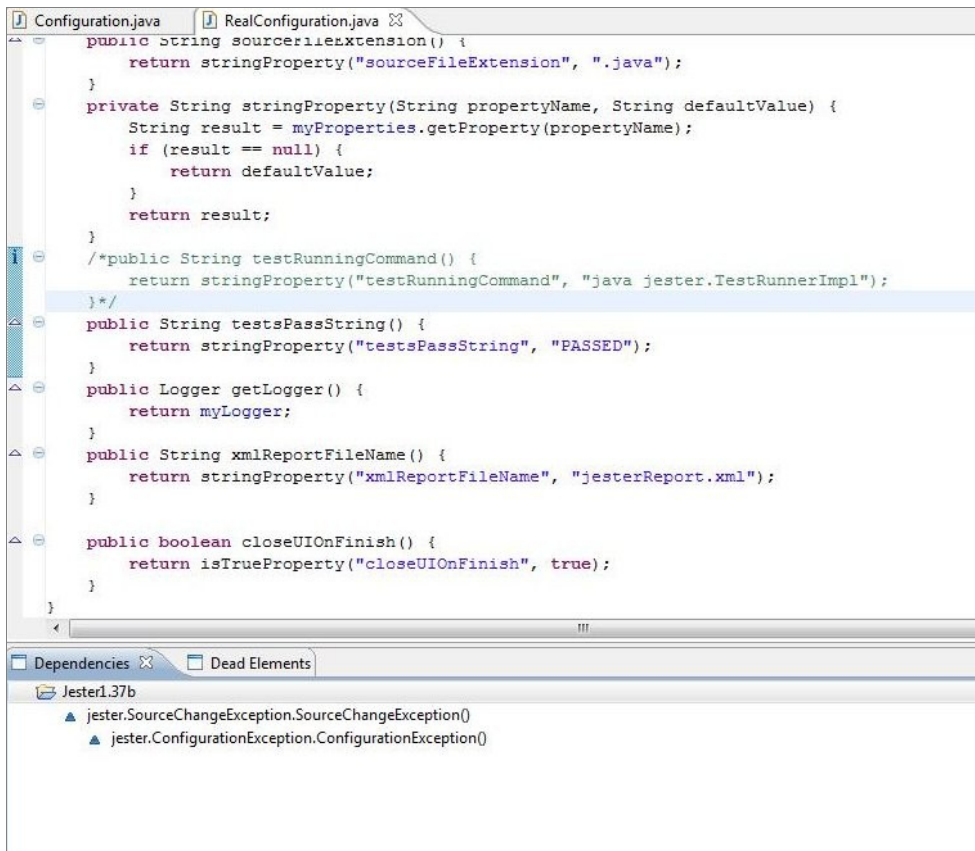
Ist das Element von anderen abhängig, d.h. das Auskommentieren ist nur beim gleichzeitigen Auskommentieren dieser Elemente möglich, wird ein Fragedialog angezeigt:



Bei Auswahl von OK wird fortgefahren und die Elemente auskommentiert:



Beide Elemente verschwinden aus dem Dead Elements View, auch die Abhängigkeitsbeziehung zwischen ihnen ist nicht mehr im Dependencies View zu sehen.



Die auskommentierten Elemente werden mit einem Info-Marker versehen, der anzeigt, dass die Kommentarzeichen vom CDCD Plugin eingefügt wurden:

```

Configuration.java
package jester;

public interface Configuration {
    //e.g. "javac"
    public String compilationCommand();

    //e.g. ".java"
    public String sourceFileExtension();

    //e.g. "java jester.TestRunnerImpl"
    Code was commented out by Dead Code Detector
    public String testRunningCommand();

    //e.g. "PASSED"
    public String testsPassString();

    //e.g. "jesterReport.xml"
    public String xmlReportFileName();

    public boolean shouldReportEagerly();
    public Logger getLogger();

    public boolean closeUIOnFinish();
}

```

Das Auskommentieren kann auch wieder rückgängig gemacht werden, z.B. über das Kontextmenü oder die Tastenkombination Strg-Z:

```

Configuration.java
package jester;

public interface Configuration {
    //e.g. "javac"
    public String compilationCommand();

    //e.g. ".java"
    public String sourceFileExtension();

    //e.g. "java jester.TestRunnerImpl"
    /*public String testRunningCommand();*/

    //e.g. "PASSED"
    public String testsPassString();

    //e.g. "jesterReport.xml"
    public String xmlReportFileName();

    public boolean shouldReportEagerly();
    public Logger getLogger();

    public boolean closeUIOnFinish();
}

```

Undo Comment Out Dead Code	Ctrl+Z
Revert File	
Save	Ctrl+S
Open Declaration	F3
Open Type Hierarchy	F4
Open Call Hierarchy	Ctrl+Alt+H
Show in Breadcrumb	Alt+Shift+B
Quick Outline	Ctrl+O
Quick Type Hierarchy	Ctrl+T
Show In	Alt+Shift+W
Cut	Ctrl+X
Copy	Ctrl+C
Copy Qualified Name	
Paste	Ctrl+V
Quick Fix	Ctrl+1
Source	Alt+Shift+S

Bei der Auswahl des Quick Fixes „Remove“ wird analog vorgegangen, das tote Deklarationselement wird samt seinen Abhängigkeiten entfernt.

Das Auskommentieren, bzw. Entfernen aller toten Elemente kann im Menü des Dead Elements View ausgewählt werden:

Element	Resource	Location	Project	Path
method Configuration.testRunningCommand()	Configuration...	line 11	Jester1.37b	Jester1.37b/jester
method ConfigurationException.ConfigurationException()	Configuration...	line 7	Jester1.37b	Jester1.37b/jester
method RealConfiguration.testRunningCommand()	RealConfigura...	line 56	Jester1.37b	Jester1.37b/jester
method SourceChangeException.SourceChangeException()	SourceChange...	line 4	Jester1.37b	Jester1.37b/jester

Element	Resource	Location	Project	Path
method Configuration.testRunningCommand()	Configuration...	line 11	Jester1.37b	Jester1.37b/jester
method ConfigurationException.ConfigurationException()	Configuration...	line 7	Jester1.37b	Jester1.37b/jester
method RealConfiguration.testRunningCommand()	RealConfigura...	line 56	Jester1.37b	Jester1.37b/jester
method SourceChangeException.SourceChangeException()	SourceChange...	line 4	Jester1.37b	Jester1.37b/jester

Das Undo für das Entfernen aller toten Deklarationselemente ist ebenfalls möglich.

## Anhang B Inhaltsverzeichnis der beiliegenden CD

Bachelorarbeit_Pasenkova.pdf	Die vorliegende Bachelorarbeit als pdf-Dokument
eclipse-SDK-3.4.2-win32_cdcd.zip	Eclipse Version 3.4.2 mit CDCD Plugin
Testprojekte.zip	Archiv mit Testprojekten (vgl. Abschnitt 5.1)
/doc/cdcd_doc.zip	Javadoc des CDCD-Plugins
/plugins/org.intoj.cdcd_1.0.0.jar	Das CDCD-Plugin
/plugins/org.intoj.amm3_1.0.0.jar	Constraintsystem-Plugin mit CDCD-Constraints
/src/org.intoj.cdcd_src.zip	Sourcecode des CDCD-Plugins
/src/org.intoj.amm3_src.zip	Sourcecode des Constraintsystem-Plugins

## Erklärung

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

Mannheim, den 21. Mai 2010