

Seminar Refactoring in eXtreme Programming:  
"Codegerüche" erkennen  
(Wie Metriken helfen, Programmstellen zu erkennen, die transformiert werden  
sollen oder können)

Andreas Koop  
Universität Paderborn

28. Februar 2003

## Inhaltsverzeichnis

<b>1 Einführung</b>	<b>2</b>
1.1 Betrachtete Refactorings . . . . .	2
<b>2 Metrikbasiertes Refactoring</b>	<b>3</b>
2.1 Grundlagen . . . . .	3
2.2 Beispiel: Distanzmaß . . . . .	4
2.3 Vorgehensweise . . . . .	5
2.4 Identifikation von <i>Move Method</i> Situationen . . . . .	5
2.5 "Codegerüche" in "real-world" Systemen . . . . .	6
2.6 Stärken und Schwächen . . . . .	7
<b>3 Kombination von einfachen Graphen und Metriken</b>	<b>9</b>
3.1 Prinzip . . . . .	9
3.2 Beispiel: Klassenkohärenz . . . . .	10
<b>4 Ausblick</b>	<b>11</b>
<b>5 Literatur</b>	<b>12</b>

## 1 Einführung

Je länger objektorientierte Softwaresysteme in Gebrauch sind, desto wahrscheinlicher ist es, dass sie gewartet werden müssen, d. h. sie müssen optimiert, bezüglich bestehender Fehler korrigiert und nicht zuletzt an geänderte Anforderungsdefinitionen angepasst werden. Dies stellt in der Praxis jedoch eine große Herausforderung dar, weil die bestehende Software oft sehr schlecht implementiert ist. Doppelter Code, lange Methoden, große Klassen machen es dem Softwareentwickler schwer, die gewünschten Änderungen schnell und ohne Einführung neuer Fehler durchzuführen.

FOWLER[4] bezeichnet so etwas als *"Bad Smell"*, also übler Geruch, den der so implementierte Code verbreitet, und präsentiert viele verschiedene Arten von Refactorings, um die Verständlichkeit des Codes zu verbessern und ihn änderungsfreundlicher zu gestalten. Doch auch wenn man weiß, wie unterschiedliche Refactorings angewendet werden, so ist immer noch nicht sofort klar, an welcher Stelle. Wo fangen wir an und wo hören wir auf, den bestehenden Code zu restrukturieren? Diese Frage wird noch schwieriger, wenn man bedenkt, dass nach FOWLERS Aussagen Refactorings auf menschlicher Intuition basieren ([4], Seite 75), was den Eindruck vermittelt, dass Refactorings niemals automatisch durchgeführt werden könnten.

Letzten Endes bleibt die Entscheidung für die Anwendung eines Refactorings in der Hand des Programmierers. Nichtsdestotrotz kann eine automatisierte Erkennung von "Codegerüchen" die menschliche Intuition auf effizientem Wege unterstützen. In dieser Seminararbeit sollen einige Methoden und Techniken vorgestellt werden, die es dem Softwareentwickler ermöglichen, "Codegerüche" für einige konkrete Refactorings zu erkennen.

Im folgenden Abschnitt werden zunächst die betrachteten Refactorings den korrespondierenden "Codegerüchen" gegenübergestellt. Die Abschnitte 2 und 3 beschreiben jeweils einen Ansatz zur Erkennung von "Codegerüchen". Abschließend wird ein Fazit gezogen und ein kurzer Ausblick gegeben.

### 1.1 Betrachtete Refactorings

**Move Method** *Verschiebung einer Methode von der Klasse, in der sie definiert ist, in eine andere.* Der auslösende "Codegeruch" für dieses Refactoring ist, dass eine Methode mehr Eigenschaften einer anderen Klasse nutzt oder von mehr Eigenschaften benutzt wird, als in der sie definiert ist.

**Move Attribute** *Verschiebung eines Attributes von einer Klasse in eine andere.* Die dafür verantwortliche Anomalie ist, dass ein Attribut von einer anderen Klasse öfter benutzt wird, als in der es definiert ist.

**Extract Class** *Erstellung einer neuen Klasse und Verschiebung funktional zusammenhängender Attribute und Methoden von der alten in die*

*neue Klasse.* Motivation dafür ist eine Klasse, die zu viel Funktionalität enthält, d. h. Funktionalität, welche von mindestens zwei Klassen angeboten werden sollte.

**Inline Class** Das Pendant zu Extract Class. *Verschiebung aller Methoden und Attribute in eine andere Klasse und Löschung der alten.* Ein Indiz dafür ist, dass eine Klasse zu wenig oder kaum Funktionalität enthält.

## 2 Metrikbasiertes Refactoring

### 2.1 Grundlagen

Die Motivation für viele Refactorings rührt oft von der Verletzung des so genannten Kohärenzprinzips: *Packe zusammen, was funktional zusammen gehört.* Um nun festzustellen, wie stark Teile eines Softwaresystems zusammen gehören, gibt es zahlreiche Kohärenzmetriken. Tabelle 1 gibt einen kurzen Überblick beispielhafter Metriken.

Name	Beschreibung
HNL	<i>Hierarchy nesting level</i>
NOM	<i>Number of methods</i>
LOC	<i>Lines of code</i>
	⋮

Tabelle 1: Einige ausgewählte Metriken

Der Grad der Zusammengehörigkeit zweier Entitäten, also z. B. von *Klassen, Methoden, Attributen* kann mit Hilfe eines generischen Kohärenzmeßverfahrens berechnet werden. Dazu macht man sich die Tatsache zu nutze, dass die Ähnlichkeit zweier Entitäten proportional zu ihren gemeinsamen Eigenschaften ist, und definiert nach [2] folgendes Distanzmaß:

$$\text{dist}_{\mathbf{B}}(x, y) := 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|} \quad (1)$$

$$\text{mit } \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|} =: \text{sim}_{\mathbf{B}}(x, y) \quad (2)$$

Dabei ist  $\mathbf{B}$  eine vorab festgelegte Menge von Eigenschaften bezüglich eines konkreten Ähnlichkeitsvergleichspunktes, z. B. der in Tabelle 2 gewählten, und  $x, y$  sind die zwei betrachteten Entitäten (*Klassen, Methoden, Attribute*) mit  $p(x) := \{p_i \in \mathbf{B} | x \text{ besitzt Eigenschaft } p_i\}$ .

Das so definierte Distanzmaß macht die Kohärenz zweier Entitäten, also den Grad dessen, wie stark zwei Komponenten zusammen gehören, messbar, d. h.

Für eine Methode:	Für ein Attribut:
Die Methode selbst	Das Attribut selbst
Alle benutzten Methoden	Alle Methoden, die das Attribut nutzen
Alle benutzten Attribute	

Tabelle 2: Mögliche Wahl der Eigenschaften zweier Entitäten

Entitäten mit kleinen Distanzen sind stark kohärent, wohingegen Entitäten mit großen Distanzen weniger kohärent sind, wobei sich alle Distanzen im Intervall  $[0, 1]$  abspielen.

## 2.2 Beispiel: Distanzmaß

Wir wollen uns das Distanzprinzip nun an einem kurzen Beispiel veranschaulichen. Betrachten wir einmal folgende Klasse:

```
public class ClassA {
    public static int attributeA1;
    public static int attributeA2;

    public static void methodA1(){
        attributeA1 = 0;
        attributeA2 = 0;
        methodA2();
    }
    public static void methodA2(){
        attributeA2 = 0;
        attributeA1 = 0;
    }
}
```

Was lässt sich über die Kohärenz zwischen Methode `methodA1()` und `methodA2()` sagen? Anwendung der Distanzformel (Gleichung 1) mit den Eigenschaften aus Tabelle 2 für die Menge **B** ergibt:

$$\begin{aligned}
 p(mA1()) &= \{mA1(), aA1, aA2, mA2()\} \\
 p(mA2()) &= \{mA2(), aA2, aA1\}
 \end{aligned}$$

$$\begin{aligned}
 p(mA1()) \cap p(mA2()) &= \{aA1, aA2, mA2()\} \Rightarrow |\dots| = 3 \\
 p(mA1()) \cup p(mA2()) &= \{mA1(), aA1, mA2(), aA2\} \Rightarrow |\dots| = 4
 \end{aligned}$$

$$\Rightarrow dist(mA1(), mA2())_{\mathbf{B}} = 1 - \frac{3}{4} = 0,25$$

Auf Grund des relativ geringen Abstandes schließen wir, dass die beiden Methoden bezüglich gegenseitiger Nutzung kohärent und damit "geruchsfrei" sind.

### 2.3 Vorgehensweise

Um das Distanzkonzept in der Praxis für die Erkennung von "Codegerüchen" anzuwenden, ist die Menge **B** der betrachteten Eigenschaften mit größter Sorgfalt zu wählen. Denn je nach Wahl dieser Eigenschaftsmenge kann der Grad der Ähnlichkeit eines Entitätspaares von 'identisch' nach 'grundverschieden' permutieren. Doch nun zur prinzipiellen Vorgehensweise:

1. Da die Distanzmessung generisch definiert und prinzipiell auf viele Dinge anwendbar ist, gilt es zunächst die Entitäten festzulegen. In unseren Beispielen beschränken wir uns auf *Methoden* und *Attribute*, für welche die Distanzen gemäß Gleichung 1 berechnet werden müssen.
2. Als nächstes ist die Menge **B** der korrespondierenden Eigenschaften zu bestimmen, die Distanzen sind zu ermitteln.
3. Die so berechneten Distanzen zwischen Methoden und Attributen stehen nun in einer Dreiecksmatrix zur Verfügung und können graphisch als euklidische Abstände dargestellt werden. Mit Techniken wie z. B. mehrdimensionaler Skalierung ist es möglich, die Distanzen dreidimensional zu visualisieren. Im letzten und schwierigsten Schritt ist die so entstandene Graphik von Abständen zu interpretieren. Wir können damit inkohärente Programmstellen – was ja unserer Definition nach einem "Codegeruch" entspricht – visuell erfassen und dem Entwickler einen Vorschlag für ein Refactoring zur Beseitigung der aufgedeckten Anomalie machen.

### 2.4 Identifikation von *Move Method* Situationen

Als nächstes wollen wir zeigen, wie man mit Hilfe des Distanzkonzeptes tatsächlich "Gerüche im Code" ausfindig machen kann, die genug Anlass für die Anwendung des Refactorings *Move Method* geben. Gegeben seien folgende zwei Klassen:

```
public class ClassA {
    public static int attributeA1;
    public static int attributeA2;

    public static void methodA1(){
        attributeA1 = 0;
        methodA2();
    }
    public static void methodA2(){
        attributeA2 = 0;
        attributeA1 = 0;
    }
}

public class ClassB {
    public static int attributeB1;
    public static int attributeB2;

    public static void methodB1(){
        ClassA.attributeA1 = 0;
        ClassA.attributeA2 = 0;
        ClassA.methodA1();
    }
    public static void methodB2(){
        attributeB1 = 0;
    }
}
```

```

}
public static void methodA3(){
    attributeA1 = 0;
    attributeA2 = 0;
    methodA1();
    methodA2();
}
}

attributeB2 = 0;
}
public static void methodB3(){
    attributeB1 = 0;
    methodB1();
    methodB2();
}
}

```

Wie man leicht erkennen kann, scheint die Methode `methodB1()` einen üblen Geruch zu verbreiten. `methodB1()` nutzt ausschließlich Methoden und Attribute der Klasse `ClassA` und wird selbst in `ClassB` nur einmal referenziert, d. h. diese Methode wäre in Klasse A besser aufgehoben.

Soweit die Erwartung, doch können wir diese Anomalie auch mit dem soeben vorgestellten Konzept aufdecken? Dazu verfahren wir gemäß der Vorgehensweise von 2.3:

1. Da die betrachteten Entitäten in diesem Falle *Methoden* sind und diese letzten Endes intern auch nur auf Methoden oder Attribute zugreifen, bleibt es bei diesen 2 Entitäten.
2. Auf Grund der Wahl in 1) wählen wir als Menge **B** die bereits in Tabelle 2 definierten Eigenschaften. Nun werden die Distanzen nach Gleichung 1 berechnet und in einer VRML-Welt visualisiert. Ein Schnappschuss dessen ist in Abbildung 1 zu sehen<sup>1</sup>. Dabei stellen grüne Objekte Entitäten der Klasse `ClassB`, blaue Objekte Entitäten der Klasse `ClassA` dar, ein Würfel entspricht einem Attribut und eine Kugel einer Methode.
3. **Interpretation:** In der Abbildung sind deutlich zwei Gruppen von Objekten zu erkennen, d. h. die Entitäten dieser Gruppen sind stark kohärent. Desweiteren sieht man, dass `methodB1()` (durch den Pfeil angedeutet) zu Entitäten aus `ClassA` eine größere Kohärenz als zu Entitäten aus `ClassB` aufweist. Folglich liegt eine Verschiebung von `methodB1()` von der einen in die andere Klasse nahe, was dem Refactoring *Move Method* entspricht.

Da die Identifikation von *Move Attribute*, *Extract Class*, *Inline Class* auf ähnliche Weise (Distanzmessung) stattfindet und den Rahmen dieser Seminararbeit sprengen würde, sei an dieser Stelle auf [1] verwiesen.

## 2.5 "Codegerüche" in "real-world" Systemen

Nachdem wir uns die Idee des metrikbasierten Refactoring an einem konkreten Beispiel angesehen haben, ist jedoch noch zu prüfen, welche Probleme bei der Anwendung in großen Softwaresystemen auftreten können. Stellen

<sup>1</sup>Die Darstellung der VRML-Welt auf Papier ist sicherlich nicht zufriedenstellend, macht die wesentlichen Merkmale aber deutlich

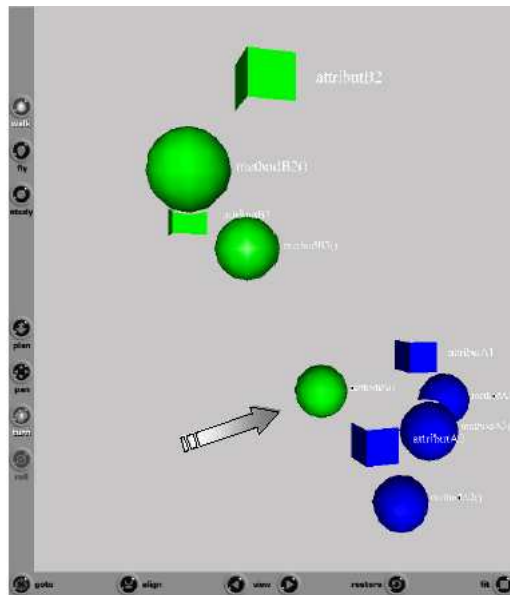


Abbildung 1: Motivation von *Move Method* - Quelle:[1]

wir uns einmal ein System mit über 1000 Klassen vor, so ist es sicherlich nicht möglich, alle paarweise berechneten Distanzen der einzelnen Entitäten übersichtlich in einer VRML-Welt darzustellen.

In diesem Fall müssen wir zunächst die für das System wichtige Klassen herausfiltern, also z. B. irgendwelche Startklassen, mit der `main()`-Methode kooperierende Klassen, etc. Problematisch ist desweiteren, dass diese Klassen oft in Vererbungshierarchien eingebunden sind, was zu erheblich verfälschten Distanzmaßen führen kann. Doch dazu mehr unter Punkt 2.6. Wie schwierig die Interpretation der Visualisierung ist, macht Abbildung 2 in anschaulicher Weise deutlich.

Auch wenn die Visualisierung anfangs unübersichtlich erscheint, so ist es mit ein wenig Erfahrung doch möglich, vorhandene Anomalien oder "Codegerüche" zu entdecken. Die mit dem Pfeil markierte Methode in Abbildung 2 hat eine auffallend große Distanz zu allen restlichen Entitäten, was vermuten lässt, dass diese Methode in keiner Interaktion mit anderen Entitäten des Softwaresystems steht und folglich gelöscht werden sollte.

## 2.6 Stärken und Schwächen

Schauen wir uns nun einmal die Stärken und vorhandenen Schwächen des metrik- bzw. distanzbasierten Refactorings an.

**Stärken:** Es ist offensichtlich, dass mit Unterstützung der eben vorgestellten Methode, dem Softwareentwickler ein mächtiges Werkzeug zur

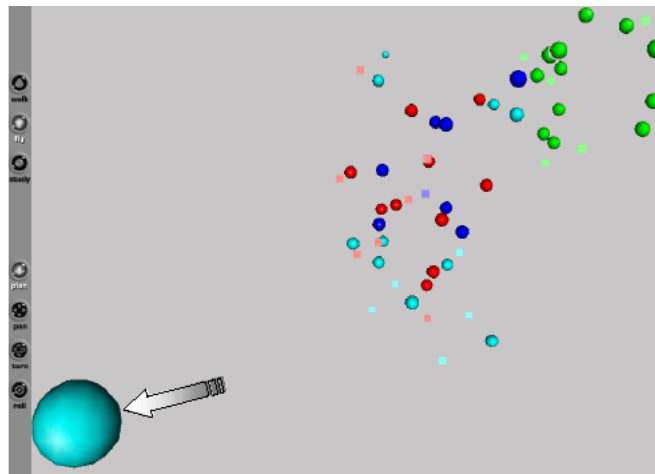


Abbildung 2: VRML-Welt mit 4 Klassen, bestehen aus 40 Methoden und 18 Attributen - Quelle:[1]

Verfügung steht, um effizient Codeanomalien aufzudecken. Die Durchführung des eigentlichen Refactorings ist anschließend nur noch Formsache.

Darüberhinaus können etwa 100 *Methoden/Attribute* mit der heutigen Rechenleistung in wenigen Sekunden analysiert werden, was also auch kein großes Hindernis darstellt.

**Schwächen:** Die Berechnung der paarweise vorhandenen Distanzen zwischen Entitäten von mehr als 1000 Klassen benötigt über 15 Minuten. Diese werden dann aber in einer Datenbank gespeichert, was die relative geringe Zeit bei der anschließenden Analyse erklärt. Problematisch dabei ist jedoch immer noch, dass bei geringen Änderungen im Code, die Distanzdatenbank aktualisiert werden muss, um die Visualisierung nicht zu verfälschen.

Ein weiteres Problem ist das bereits Erwähnte: Klassen in Vererbungshierarchien. Zum Beispiel können einige Methoden als nicht-kohärent identifiziert werden, weil gemeinsame, von der Oberklasse geerbte Attribute, bei der Distanzmessung nicht berücksichtigt worden sind.

### 3 Kombination von einfachen Graphen und Metriken

#### 3.1 Prinzip

Das Prinzip dieser Technik ist, einfache Graphen, die beispielsweise die Klassenstruktur eines Systems darstellen, mit Metrikinformationen anzureichern. In einem zweidimensionalen Graphen können damit bis zu 5 Metriken an einem Knoten visualisiert werden. Folgende Knoteneigenschaften werden hierfür verwendet:

**Größe** Mit der Breite und Höhe eines Knotens können genau zwei Metrikmaße dargestellt werden. Je breiter/ höher der Knoten, desto größer das Maß der Metrik.

**Position** Die Position(X,Y) des Knotens kann ebenfalls zwei Metrikmaße widerspiegeln. Voraussetzung dafür ist jedoch ein Koordinatensystem mit beliebigem, aber festgelegtem Ursprung.

**Farbe** Darüberhinaus kann die Farbe im Intervall von weiß nach schwarz ein weiteres Metrikmaß repräsentieren. Je höher der Wert der Metrik, desto dunkler die Farbe des Knotens.

Abbildung 3 zeigt einen solchen Graphen.

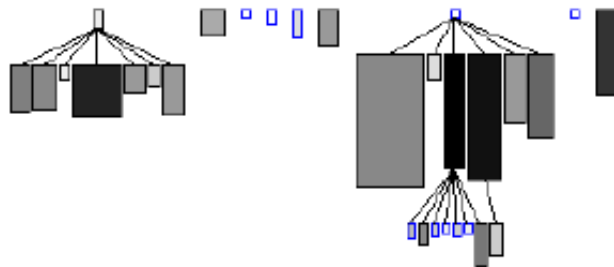


Abbildung 3: **Vererbungsbaum:** Knotenbreite = NIV (number of instance variables), Knotenhöhe = NOM (s. Tabelle 1) und Farbe = NCV (number of class variables) - Quelle:[3]

Aus diesem Graphen können wir nun viele Eigenschaften des dargestellten Systems auf einen Blick erfassen. Deutlich zu erkennen sind die Vererbungsbeziehungen und Klassen mit hoher Funktionalität, z.B. Klassen, die als relativ lange Rechtecke dargestellt sind, besitzen viele Methoden, was der Klasse eine wichtige Rolle beimisst.

### 3.2 Beispiel: Klassenkohärenz

An einem Beispiel wollen wir das vorgestellte Konzept nun nutzen, um einige Klassen eines Softwaresystems auf Kohärenz zu untersuchen. Dazu betrachten wir den Graphen in Abbildung 4 und geben folgende Interpretation:

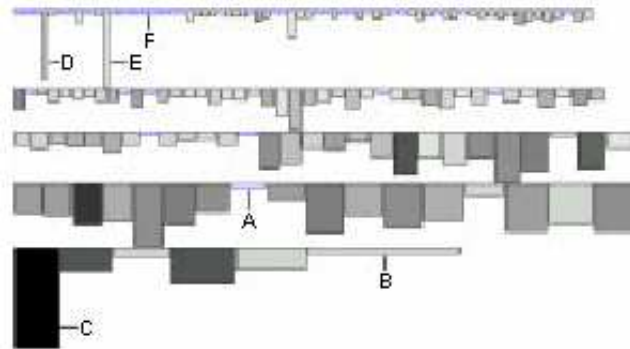


Abbildung 4: **Klassenkohärenz:** Knotenbreite = NOM (s. Tabelle 1), Knotenhöhe = WNAA (number of times all defined attributes are accessed) und Farbe = NIV (number of instance variables) - Quelle:[3]

1. Aus der Festlegung der Metriken erkennen wir sofort, dass weiße Knoten keine Instanzvariablen haben (A). Das bedeutet, dass auf ihnen dann natürlich auch kein Zugriff erfolgen kann, weswegen wir ein flaches Rechteck sehen.
2. Knoten D und E fallen auf Grund ihrer schmalen Form und hellen Farbe auf. Das heißt, dass sie nur wenige Methoden und Attribute, aber dafür sehr viele Zugriffe auf Instanzvariablen haben. Der Grund dafür könnte sein, dass die Zugriffe alle direkt von Unterklassen erfolgen. In diesem Fall wäre das Refactoring *Encapsulate Field/Attribute* angebracht.
3. Weiter auffallend ist die Klasse C. Sie hat die meisten Instanzvariablen (schwarz), weniger Methoden und viele Zugriffe auf Instanzvariablen außen. Diese Klasse könnte z. B. ein Kandidat für geringe Kohärenz sein. Offensichtlich enthält Klasse C sehr viel Funktionalität, was in diesem Fall das Refactoring *Extract Class* motivieren würde.

Mit dieser ersten Analyse haben wir also kritische Klassen ausfindig gemacht und können nun in einem weiteren Schritt diese auffälligen Klassen weiter analysieren. Dazu verwendet man einen bipartiten *Konfrontationsgraphen*: ein Graph, in dem eine Kante zwischen Instanzvariable und Methode einen

Zugriff von der Methode auf die Instanzvariable indiziert. Kanten zwischen zwei gleichen Entitäten gibt es nicht. Treten dabei ersichtliche Gruppen von Entitäten auf, so hätte man tatsächlich eine Klasse gefunden, die schwach kohärent ist, was genug Grund für das Refactoring *Extract Class* wäre. Sind jedoch alle Entitäten relativ gleichmäßig miteinander verknüpft, so hätten wir damit ein Indiz für recht hohe Kohärenz in der betrachteten Klasse. Von einer Änderung dieser Klasse wäre dann abzuraten.

Neben der Untersuchung auf Kohärenz gibt es noch weitere Anwendungen dieses Konzeptes, auf die wir in dieser Arbeit jedoch nicht eingehen wollen, die aber alle in [3] nachgelesen und studiert werden können.

## 4 Ausblick

In den vorangegangenen Abschnitten haben wir 2 Ansätze zur Erkennung von "Codegerüchen" kennen gelernt. Dabei wird in beiden Fällen eine Visualisierung der aktuellen Implementation des untersuchten Softwaresystems generiert, welche dem Softwareentwickler hilft, Kandidaten für ein Refactoring zu erkennen. Selbstverständlich bleibt die Entscheidung über die Durchführung eines Refactorings in der Hand des Programmierers, aber durch eine derart generierte Visualisierung gelingt es ihm, schnell und effizient "stinkende" Programmstellen zu identifizieren.

Die vorgestellten Techniken sind recht zufriedenstellend, beschränken sich bislang aber noch auf einige wenige Refactorings. Außerdem gibt es beim metrikbasierten Refactoring nach wie vor das Problem mit Klassen in Vererbungshierarchien. Jedoch gibt es dafür bereits Konzepte wie z. B. das *flattening* bzw. *Abflachen* (nachzulesen in [1]), bei welchem geerbte Attribute und Methoden aus Oberklassen mit berücksichtigt werden. Desweiteren ist im Zusammenhang mit Vererbung die Erkennung von "Codegerüchen" für Refactorings wie *Pull up/down Method/Attribute* erstrebenswert.

Zwar ist es immer noch sehr schwierig ein derartiges Konzept auf große Softwaresysteme anzuwenden, aber wir haben gesehen, dass es heute bereits mit geringem Aufwand möglich ist.

Es gibt daneben schon Tools in IDEs wie z. B. *IntelliJ IDEA*, *Code Inspection Tool*, mit denen unerreichbarer Code, also unbenutzte Klassen, Methoden, Attribute, zur Implementierungszeit erkannt werden und die dem Programmierer einen Hinweis für ein mögliches Refactoring geben.

Insgesamt gibt es also bereits viele Ansätze und Tools für die Erkennung von "Codegerüchen", was uns erwartungsvoll in die Zukunft blicken lässt.

## 5 Literatur

- [1] FRANK SIMON, FRANK STEINBRUCKNER, CLAUS LEWERENTZ: *Metrics Based Refactoring*. In *CSMR*, Seiten 30-38, 2001
- [2] FRANK SIMON, SILVIO LÖFFLER, CLAUS LEWERENTZ: *Distance based cohesion measuring*. In *FESMA '99*, Technologisch Instituut Amsterdam, 1999
- [3] SERGE DEMEYER, STÉPHANE DUCASSE, MICHELE LANZA: *A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation*. In *WCRE'99*, IEEE, 1999
- [4] MARTIN FOWLER: *Refactoring: Improving the design of existing code*. Addison-Wesley, Kapitel 3, 1999