Refactoring in eXtreme Programming

„Codegerüche“ erkennen

Universität Paderborn

Wintersemester 2002/03

Andreas Koop

Motivation

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

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

Überblick

1. Einführung
   - 1.1. Problem
   - 1.2. „Codegerüche“ ←→ Refactorings
2. Metrikbasiertes Refactoring
   - 2.1. Grundlagen: Ähnlichkeitsprinzip/ Distanzmaß
   - 2.2. Identifikation von *Move Method* Situationen
   - 2.3. Identifikation von *Move Attribute* Situationen
   - 2.4. Identifikation von *Extract / Inline Class* Situationen
   - 2.5. Stärken / Schwächen
3. Alternative Ansätze / Tools
   - 3.1. Programmvisualisierung (Komb. von Metriken und Graphen)
   - 3.2. Code Inspection Tool (IntelliJ IDEA)
4. Zusammenfassung/ Ausblick

1.1 Problem

- Schlechter Entwurf von objektorientierten Softwaresystemen führt bei der Implementation oft zu vielen „Codeverschmutzungen“ oder sog. „bad smells“
- An welcher Stelle sollte ein Refactoring durchgeführt werden ?
- Wir wissen zwar, wie Refactoring funktioniert, aber nicht unbedingt, wann und wo es eingesetzt werden sollte, weil wir nicht auf Anhieb diejenigen Stellen erkennen, die transformiert werden müssen

→ Bedarf an Methoden und Werkzeugen

1.2 Refactorings ←→ „Codegerüche“

- Konzentration auf 4 Refactorings mit den korrespondierenden motivierenden „Codegerüchen“
- ***Move Method:*** Eine Methode nutzt oder wird von mehr Featuren einer anderen Klasse, als in der sie definiert ist, benutzt.
- ***Move Attribute:*** Ein Attribut wird von einer anderen Klasse öfter benutzt, als in der es definiert ist.
- ***Extract class:*** Eine Klasse enthält zu viel Funktionalität
- ***Inline class:*** Eine Klasse enthält kaum Funktionalität

2. Metrikbasiertes Refactoring (Grundlagen)

- Viele Refactorings basieren auf der Verletzung des **Kohärenzprinzips:** „Packe zusammen, was funktional zusammen gehört!“
- **Generisches Distanzmaß:** $dist_B(x,y) := 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|}$
  - x,y: 2 unterschiedliche Entitäten, z.B. Methode, Attribut
  - $\boldsymbol{p}(x) := \{Eigenschaften \cdot \cdot p_i \in B | x \cdot besitzt \cdot p_i\}$
  - B: Menge von Eigenschaften für einen konkreten Ähnlichkeitsgesichtspunkt, zum Beipsiel:

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

Bsp. Zur Messung der Distanz

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

$$p(mA1()) = \{mA1(), aA1, mA2()\}$$
$$p(mA2()) = \{mA2(), aA2, aA1\}$$

$$p(mA1()) \cap p(mA2()) = \{aA1, mA2()\}$$
$$p(mA1()) \cup p(mA2()) = \{mA1(), aA1, mA2(), aA2\}$$

$$\rightarrow dist(mA1(), mA2()) = 1 - \frac{2}{4} = 0.5$$

2.2. Identifikation von *Move Method* Situationen(1)

- **Grundlegende Vorgehensweise:**

  (1) Identifikation korrespondierender Eigenschaften (Menge B)

  (2) Berechnung der Distanzen (hier:am Einführungsbeispiel)

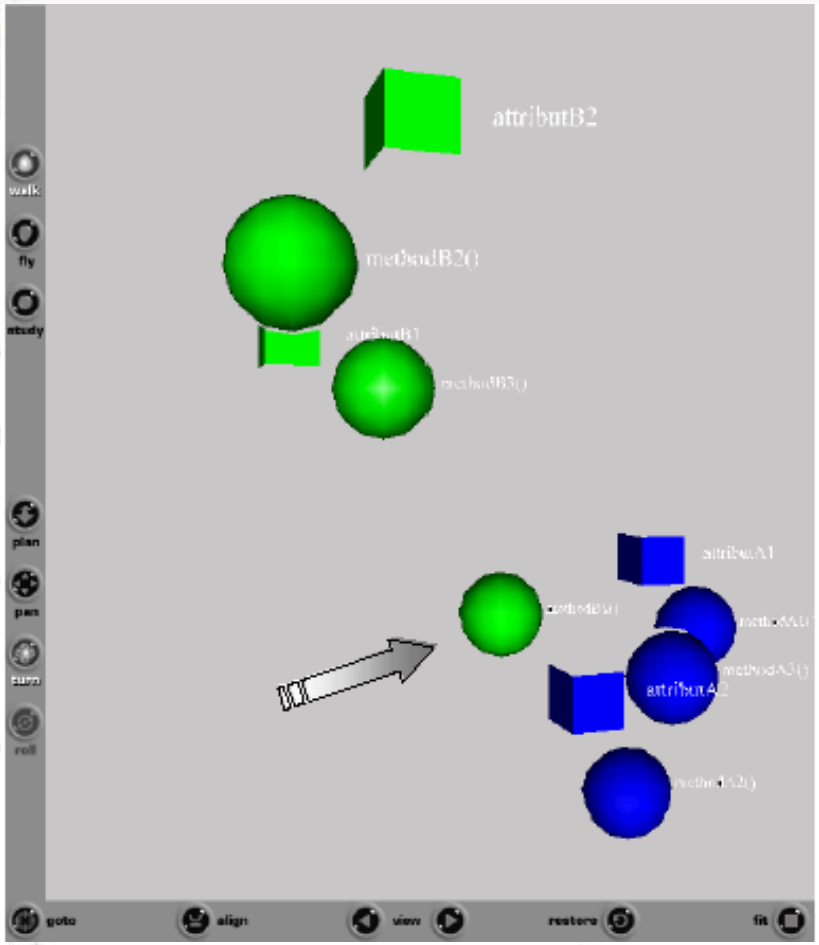
	mA1()	mA2()	mA3()	mB1()	mB2()	mB3()	aA1	aA2	aB1	aB2
mA1()	0									
mA2()	0.5	0								
mA3()	0.4	0.4	0							
mB1()	0.6	0.6	0.5	0						
mB2()	1	1	1	1	0					
mB3()	1	1	1	0.86	0.6	0				
aA1	0.5	0.67	0.33	0.5	1	0.88	0			
aA2	0.83	0.6	0.5	0.67	1	0.86	0.5	0		
aB1	1	1	1	1	0.5	0.25	1	1	0	
aB2	1	1	1	1	0.33	0.8	1	1	0.75	0

2.2. Identifikation von *Move Method* Situationen (2)

→Visualisierung der Abstände in einer VRML-Welt

- Grün: **`ClassB`** / Blau: **`ClassA`**
- Methode: Kugel
- Attribut: Würfel

→ Verschiebung von **`methodB1()`** nach **`ClassA`**



2.3. Identifikation von *Move Attribute* Situationen (1)

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

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

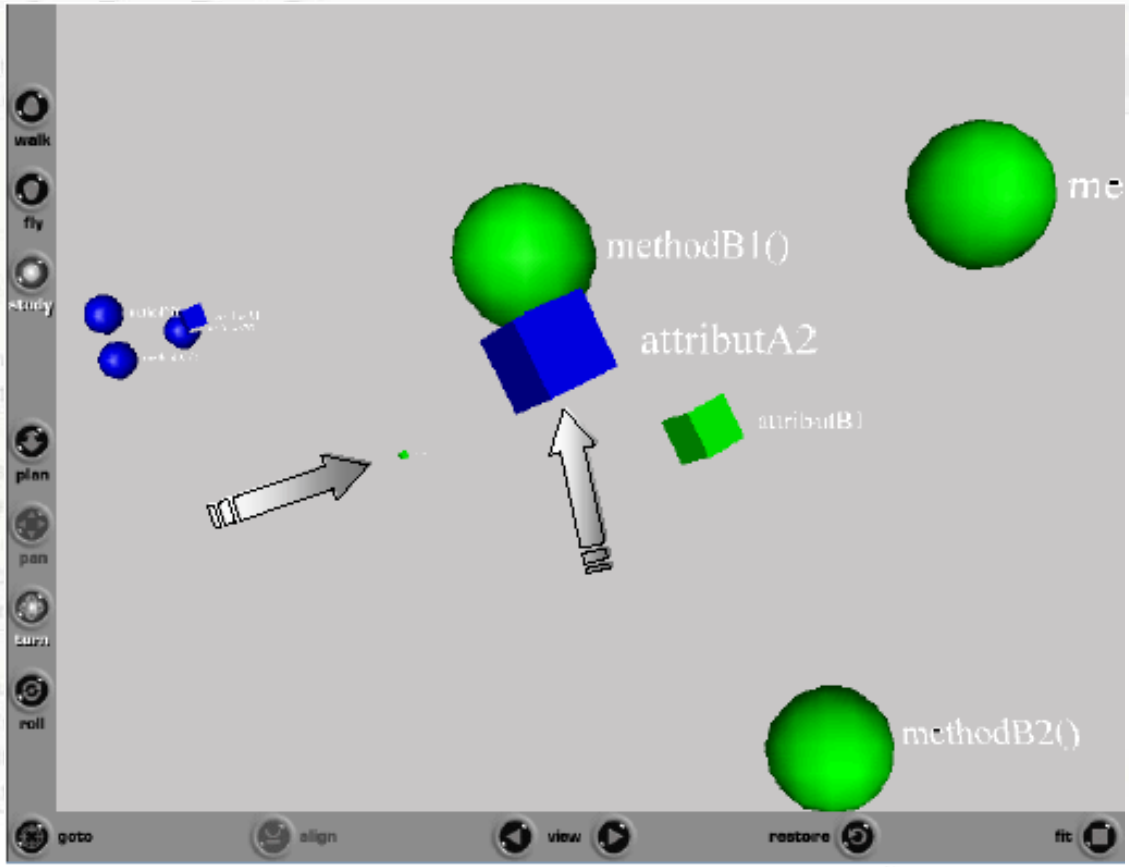
```
public class ClassB {
    public static int attributeB1;
    public static int attributeB2;

    public static void methodB1(){
          ClassA.attributeA2 = 0;
          methodB2();
          methodB3();
    }
    public static void methodB2(){
          ClassA.attributeA2 = 0;
          attributeB1 = 0;
    }
    public static void methodB3(){
          ClassA.attributeA2 = 0;
          attributeB1 = 0;
          methodB1();
    }
}
```

2.3. Identifikation von *Move Attribute* Situationen (2)

- Grün: **`ClassB`**
- Blau: **`ClassA`**
- Methode: Kugel
- Attribut: Würfel

→Verschiebung von **`attributeA2`** nach **`ClassB`**

→Löschung von **`attributeB2`**



2.4. Identifikation von *Extract/Inline Class* Situationen (1)

- *Extract Class* ist die Umkehrung von *Inline Class*

  →Ziel beider Refactorings: kohärente Klassen!

- Als Beispiel: Eine Klasse mit 20 Methoden $\mathbf{m_i}$ und 4 Attributen $\mathbf{a_i}$ mit den folgenden Eigenschaften:

  - Für die ersten 10 Methoden gilt:

    m_1 *nutzt* m_2, a_1, a_2; m_2 *nutzt* m_3, a_1, a_2;....; m_{10} *nutzt* m_1, a_1, a_2;

  - Für die anderen 10 Methoden:

    m_{11} *nutzt* m_{12}, a_3, a_4; m_{12} *nutzt* m_{13}, a_3, a_4;....; m_{20} *nutzt* m_{11}, a_3, a_4;

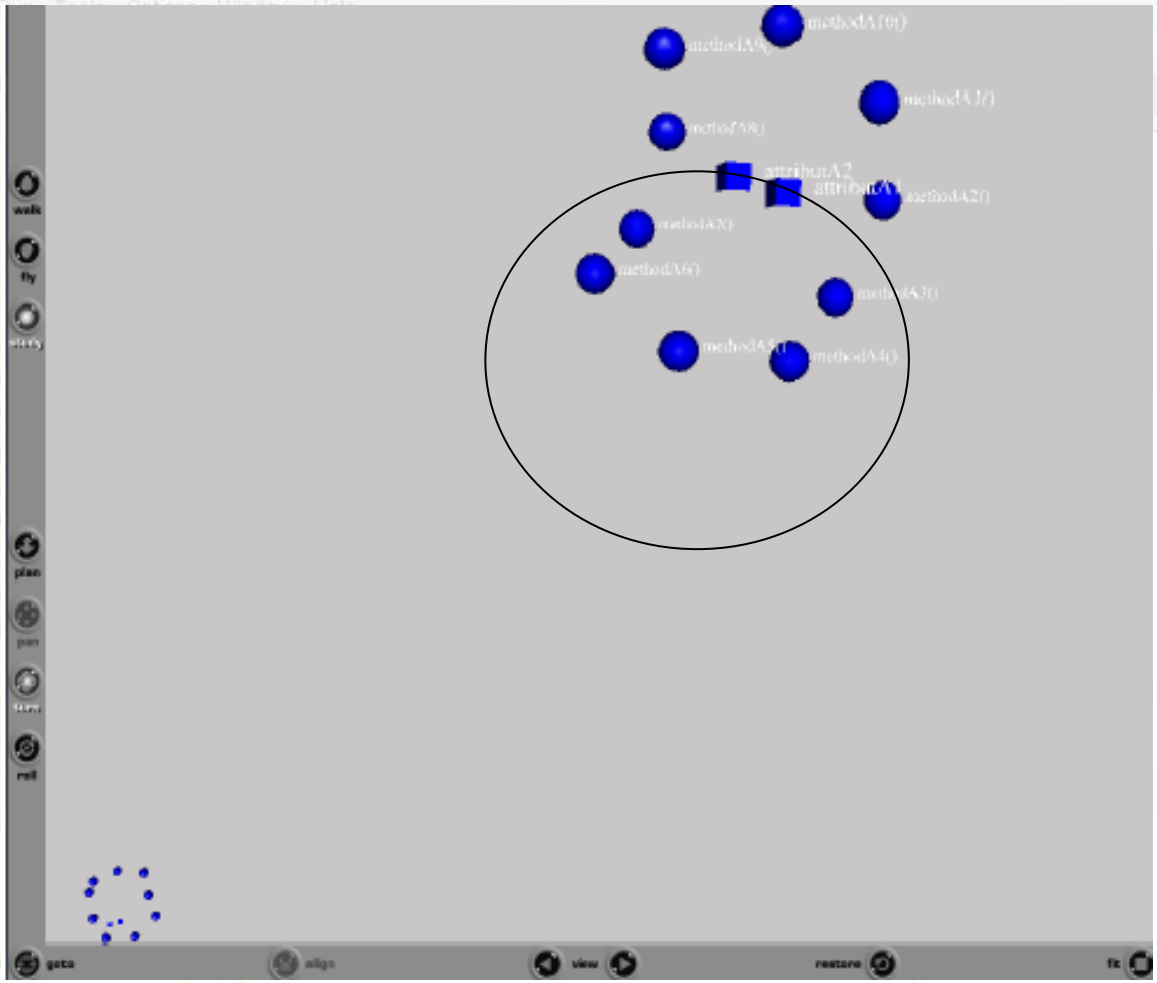
2.4. Identifikation von *Extract/Inline Class* Situationen (2)

• Motivation von *Extract Class*

•Gleiche Art der Visualisierung könnte auch das Refactoring *Inline Class* empfehlen:

Bsp: m1..m5 und a1 seien aus einer separaten Klasse

→ Mischung beider Klassen!



2.5 Stärken / Schwächen

- **Stärken / Vorteile:**
  - Entscheidungshilfe für die Anwendung von Refactorings
  - 100 Methoden/Attribute werden in wenigen Sekunden analysiert

- **Schwächen / Nachteile:**
  - Große OO Systeme mit vielen Klassen
    - Berechnung der Distanzen bei 1500 Klassen benötigt 15 min
    - Interpretation der Visualisierung schwierig →Folie15
  - Klassen in Vererbungshierarchien
    - Bsp.: 2 Methoden werden als nicht kohärent erkannt, weil die benutzten Attribute aus der Oberklasse nicht berücksichtigt werden

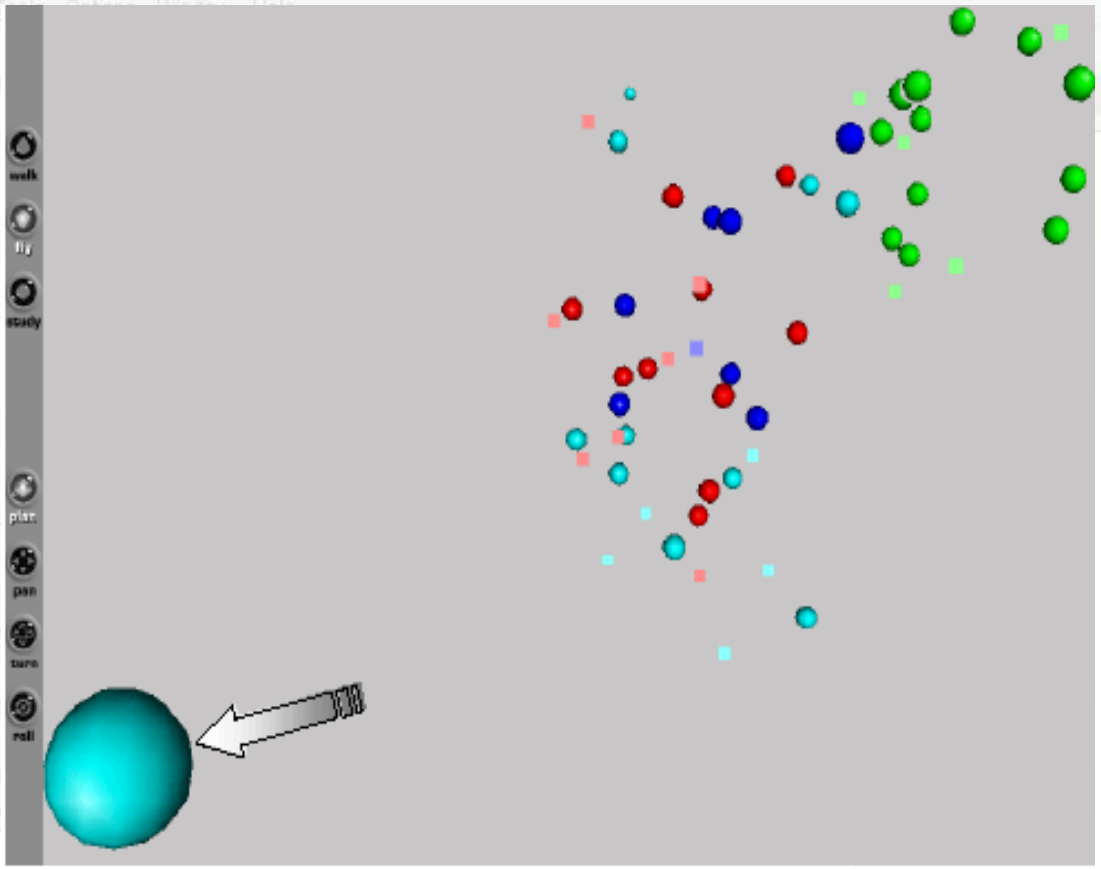
2.4. Visualisierung eines „real world“-Softwaresystems

- 4 Klassen, bestehend aus 40 Methoden, 18 Attributen

→Unübersichtlich

- Mit ein wenig Erfahrung kann man ein Gefühl dafür bekommen, wie „Codegeruch“ aussieht

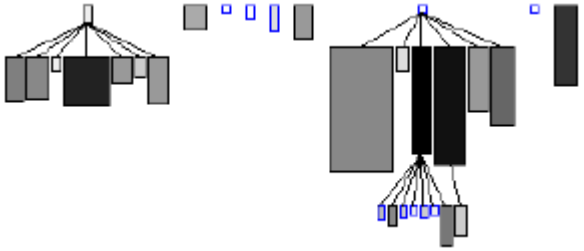
→ Hilfreich, nützlich



3.1. Komb. von Metriken und Graphen(1)

Prinzip: Bereicherung eines einfachen Graphen mit Metrikinformationen *(NIV, NOM, LOC, etc...)*

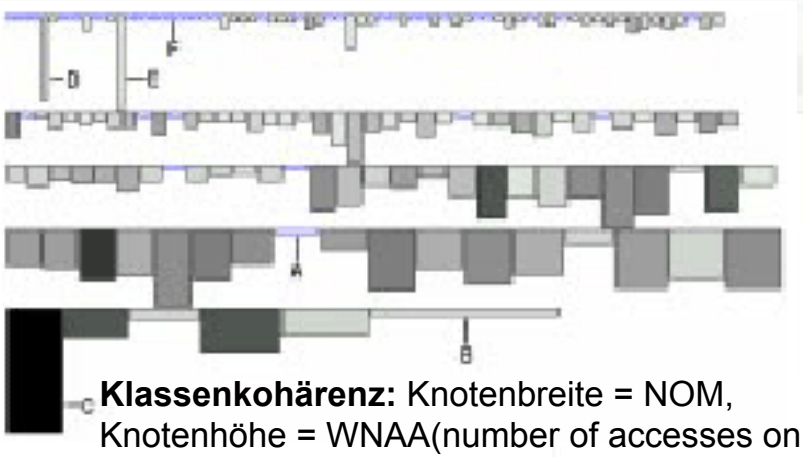
1. **Knotengröße:** Breite und Höhe eines Knotens kann 2 Metrikmaße darstellen. Je breiter/ höher der Knoten, desto größer das Maß
2. **Knotenposition:** X und Y Koordinaten stellen ebenfalls 2 Metrikmaße dar
3. **Knotenfarbe:** Je dunkler die Farbe, desto höher das Maß der Metrik



Vererbungsbaum: Knotenbreite = NIV, Knotenhöhe = NOM und Farbe = NCV

Beispiel: Klassenkohärenz

- Graph zur Untersuchung der Kohärenz von Klassen eines Softwaresystems



Klassenkohärenz: Knotenbreite = NOM, Knotenhöhe = WNAA(number of accesses on attribute defined in the class) und Farbe = NIV

Interpretation:

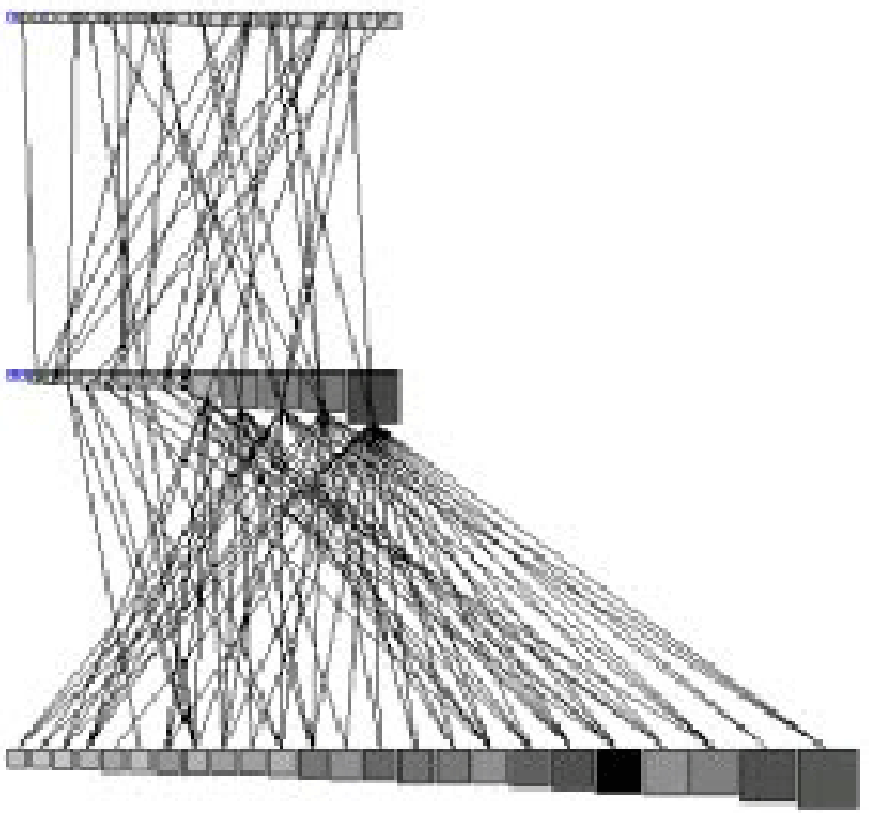
1. Weiße Knoten haben keine Instanzvariablen → kein Zugriff → flaches Rechteck
2. Knoten D und E haben wenige Methoden und Attribute, aber viele Zugriffe auf Instanzvariablen
3. Klasse B hat sehr viele Methoden im Vergleich zu Instanzvariablen
4. Klasse C hat die meisten Instanzvariablen, weniger Methoden und viele Zugriffe auf Instanzvariablen von außen

Bipartiter Konfrontationsgraph

- Mitte: Instanzvariablen
- Oben/unten: Methoden
- Kanten: Zugriff von Methoden auf Instanz-variablen

Interpretation:

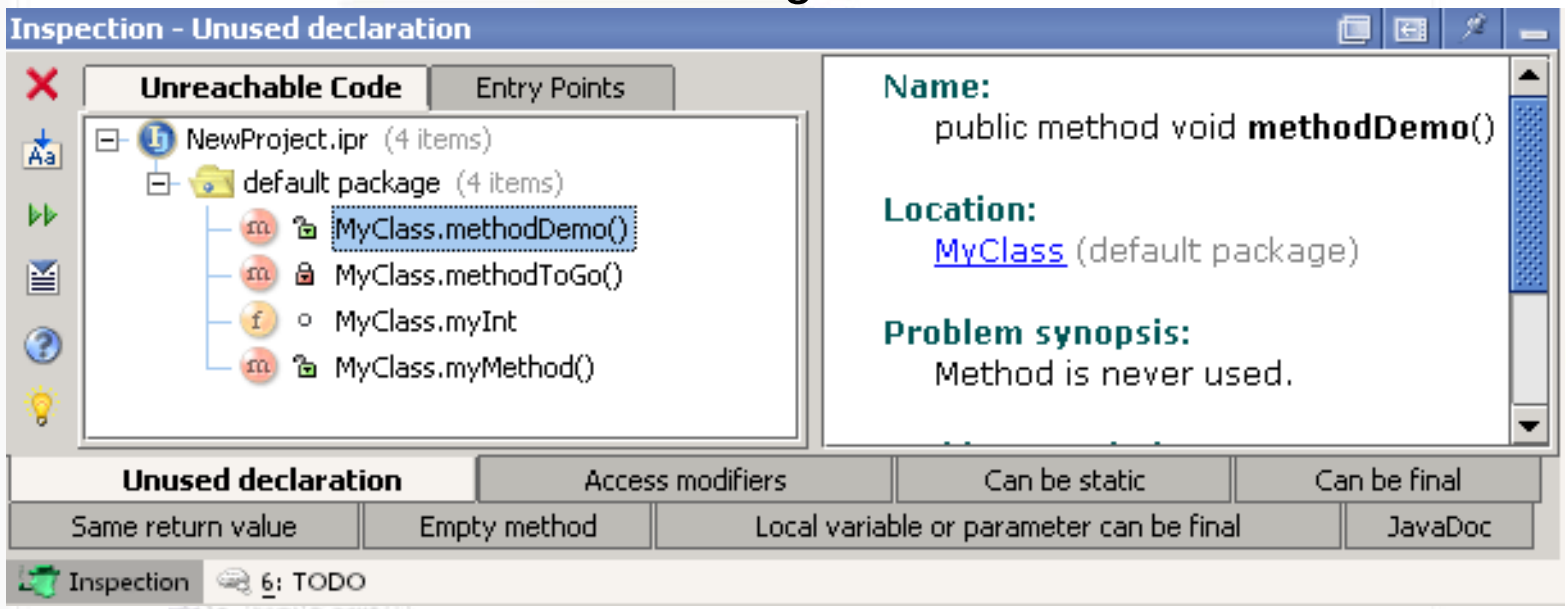
- Keine ersichtlichen Gruppen
  → kohärente Klasse
  → Refactoring *Extract Class* nicht empfohlen!



Klassenkohärenz: Methodenbreite/-höhe = NOS, Farbe = LOC; Attributbreite/ -höhe und Farbe = NAA (number of times accessed)

3.2. Code Inspection mit IntelliJ IDEA (1)

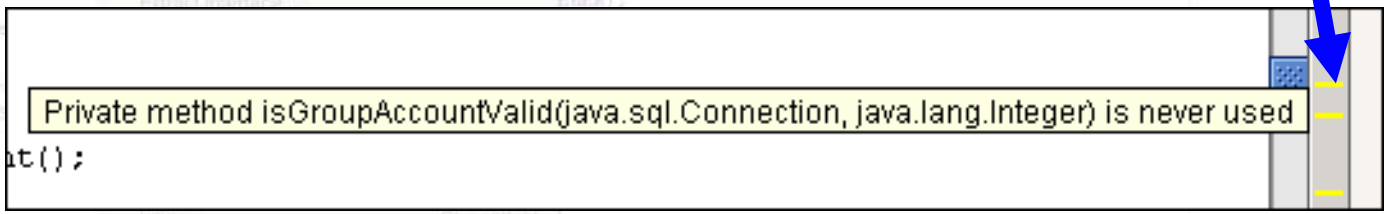
1. Erkennnung von unbenutzten Klassen, Methoden, Attributen
2. Überprüfung redundanter Deklarationen: Modifier, Sichtbarkeit, Typecast
3. Untersuchung auf mögliche `NullPointerExceptions`, unerreichbare `if`-Anweisungen



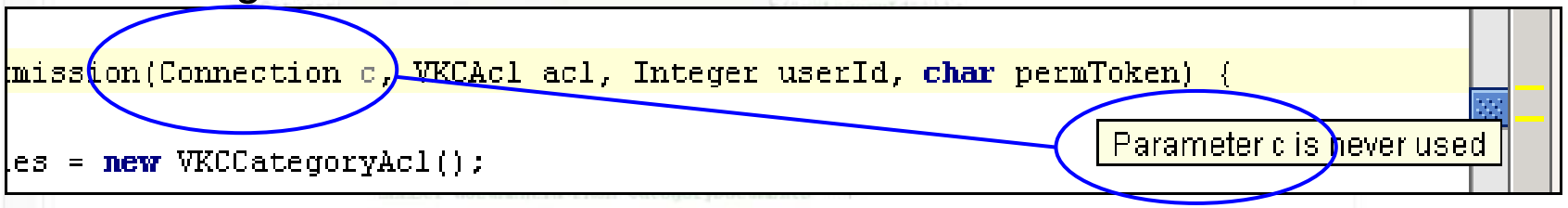
3.2. Beispiele zum Code Inspection Tool

Redundantes Type cast

Unbenutzte Methode



Überflüssiger Parameter



4. Zusammenfassung / Ausblick

- 2 Ansätze zur Visualisierung, die es dem Entwickler ermöglicht, Kandidaten für ein Refactoring *(Move Method, Move Attribute, Extract/ Inline Class)* zu erkennen.
- Interpretation der Visualisierung bei „real-world“ Softwaresystemen
- „Geruchshinweise“ mit dem Code Inspection Tool von IntelliJ IDEA
- Forschung: Erkennung von „Codegerüchen“ für weitere Refactorings, insbesondere *pull up/ push down method/ attribute*

„Codegerüche“ erkennen

Vielen Dank für Ihr Interesse!

Fragen ?