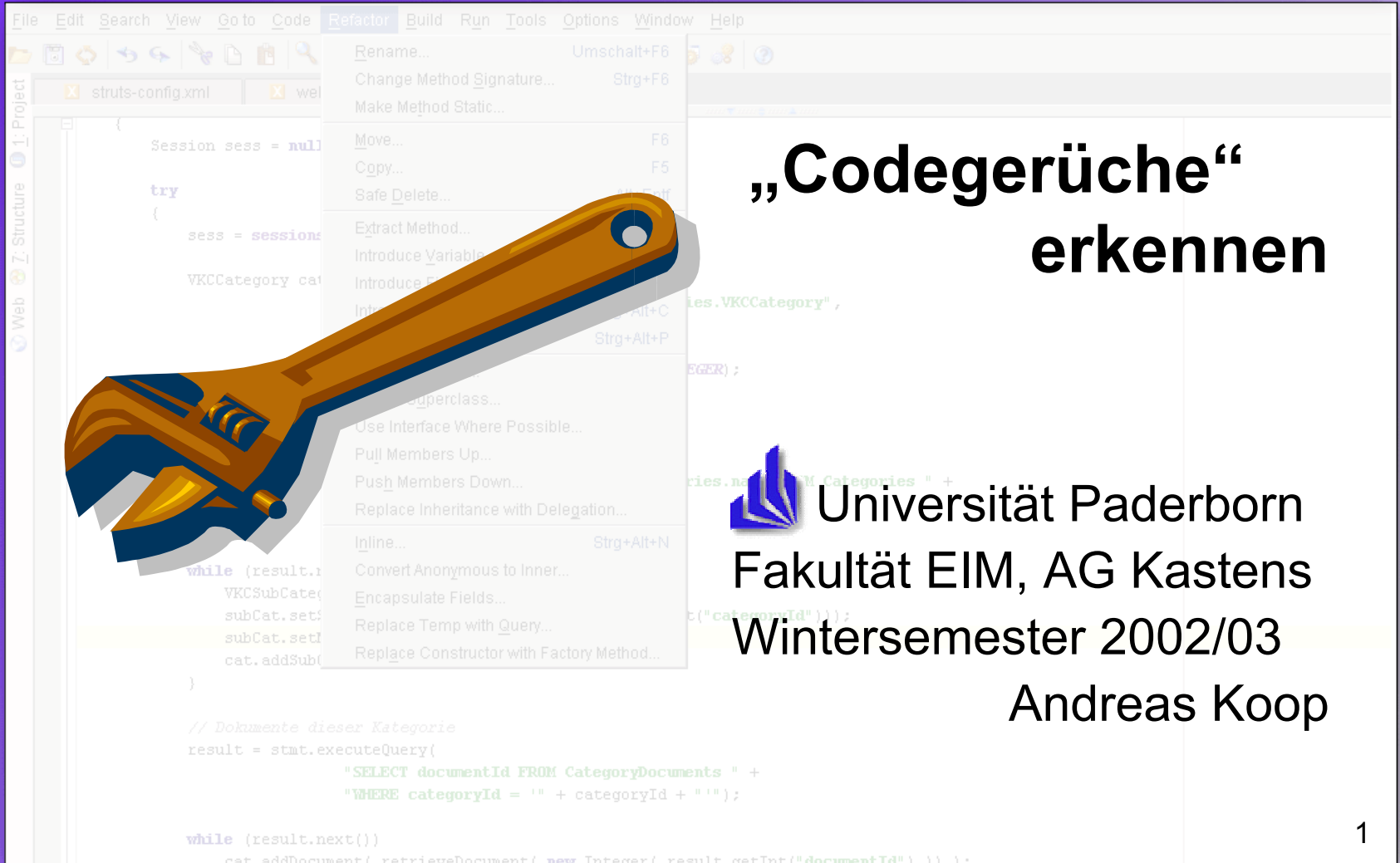



Refactoring in eXtreme Programming



The image shows a screenshot of an IDE with a refactoring menu open. The menu items include: Rename... (Umschalt+F6), Change Method Signature... (Strg+F6), Make Method Static..., Move... (F6), Copy... (F5), Safe Delete..., Extract Method..., Introduce Variable..., Introduce Field..., Introduce Parameter..., Introduce Local Variable..., Introduce Superclass..., Use Interface Where Possible..., Pull Members Up..., Push Members Down..., Replace Inheritance with Delegation..., Inline... (Strg+Alt+N), Convert Anonymous to Inner..., Encapsulate Fields..., Replace Temp with Query..., and Replace Constructor with Factory Method... A large wrench icon is overlaid on the menu. The background shows code in a Java file named 'struts-config.xml' with a 'try' block and a 'while' loop. The code includes session management and database queries for categories.

**„Codegerüche“
erkennen**

 Universität Paderborn
Fakultät EIM, AG Kastens
Wintersemester 2002/03
Andreas Koop

Motivation

```
File Edit Search View Goto Code Refactor Build Run Tools Options Window Help
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 ();
    }
}

// Dokumente dieser Kategorie
result = stmt.executeQuery(
    "SELECT documentId FROM CategoryDocuments " +
    "WHERE categoryId = " + categoryId + "");

while (result.next())
    cat.addDocument( retrieveDocument( new Integer( result.getInt("documentId") ) );

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. Anforderungen

1.3. „Codegerüche“ \leftrightarrow 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

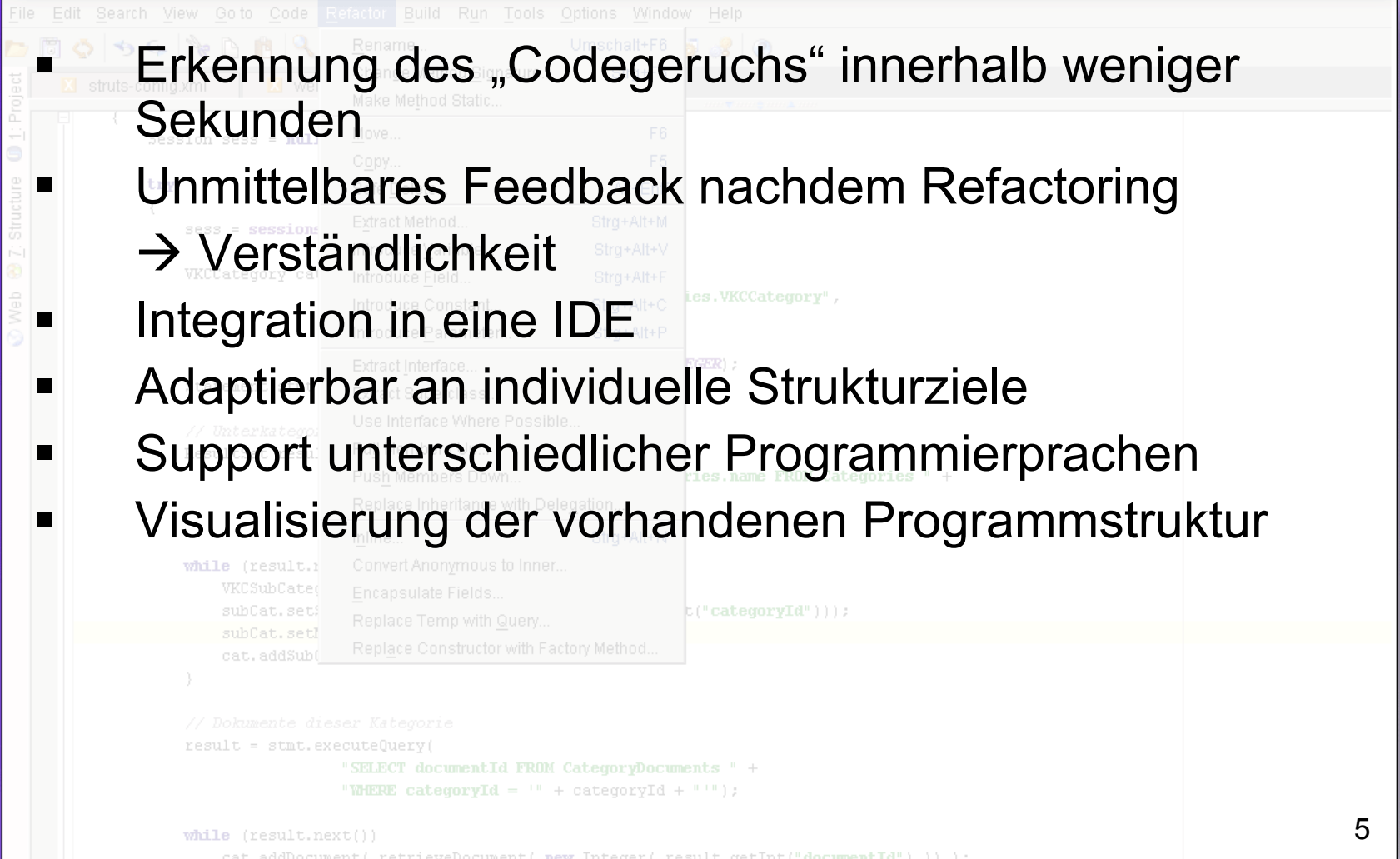
1.1 Problem

- 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 Anforderungen

- Erkennung des „Codegeruchs“ innerhalb weniger Sekunden
- Unmittelbares Feedback nachdem Refactoring
→ Verständlichkeit
- Integration in eine IDE
- Adaptierbar an individuelle Strukturziele
- Support unterschiedlicher Programmierprachen
- Visualisierung der vorhandenen Programmstruktur



1.3 Refactorings \leftrightarrow „Codegerüche“

- Konzentration auf 4 Refactorings mit den korrespondierenden motivierenden „Codegerüchen“
- **Move Method:** Eine Methode nutzt oder wird von mehr Features 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)

- **Kohärenzprinzip:** „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
- $p(x) := \{Eigenschaften \cdot p_i \in B \mid x \cdot besitzt \cdot p_i\}$
- B : Menge von Eigenschaften für einen konkreten Ähnlichkeitsgesichtspunkt, zum Beispiel:

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

```
File Edit Search View Goto Code Refactor Build Run Tools Options Window Help
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 ();
    }

    // Dokumente dieser Kategorie
    result = stmt.executeQuery(
        "SELECT documentId FROM CategoryDocuments " +
        "WHERE categoryId = " + categoryId + " ");

    while (result.next())
        cat.addDocument( retrieveDocument( new Integer( result.getInt("documentId") ) ) );
}
```

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

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

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

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

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

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

```
File Edit Search View Goto Code Refactor Build Run Tools Options Window Help
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 ();
    }
}

// Dokumente dieser Kategorie
result = stmt.executeQuery(
    "SELECT documentId FROM CategoryDocuments " +
    "WHERE categoryId = " + categoryId + " ");

while (result.next())
    cat.addDocument( retrieveDocument( new Integer( result.getInt("documentId") ) ) );

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 ();
    }
}
```

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

▪ Grundlegende Vorgehensweise:

- (1) Identifikation korrespondierender Eigenschaften (Menge B)
- (2) Berechnung der Distanzen (hier:am Einführungsbeispiel)

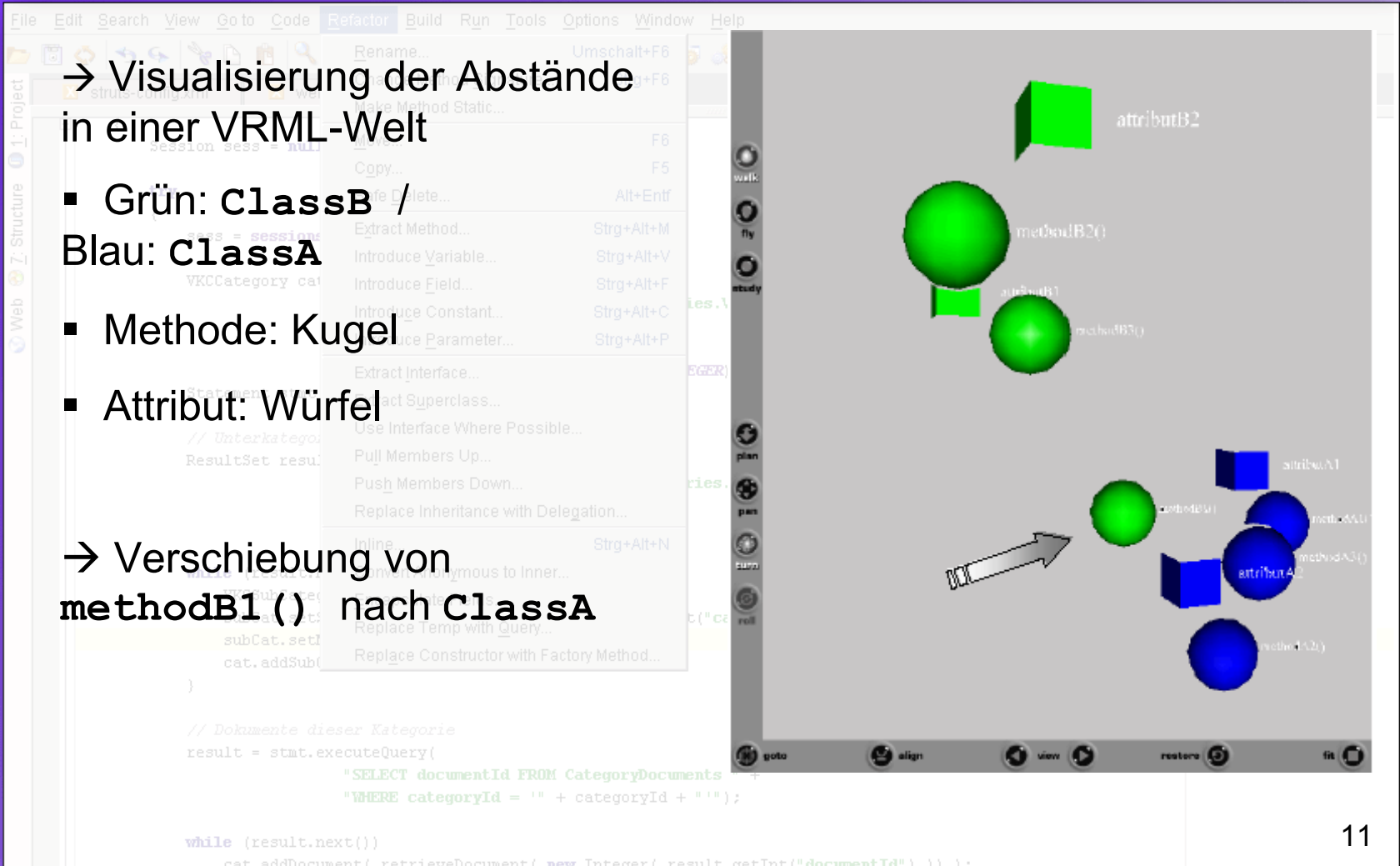
	mA1()	mA2()	mA3()	mB1()	mB2()	mB3()	aA1	aA2	aB1	aB2
mA1()	0									
mA2()	0.5	0								
mA3()	0.4	0.4	0							
mB1()	0.6	0.6	0.5	0						
mB2()	1	1	1	1	0					
mB3()	1	1	1	0.86	0.6	0				
aA1	0.5	0.67	0.33	0.5	1	0.88	0			
aA2	0.83	0.6	0.5	0.67	1	0.86	0.5	0		
aB1	1	1	1	1	0.5	0.25	1	1	0	
aB2	1	1	1	1	0.33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

2.2. Identifikation von *Move Method* Situationen (3)

→ 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)

```
File Edit Search View Goto Code Refactor Build Run Tools Options Window Help
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 ();
    }
}

// Dokumente dieser Kategorie
result = stmt.executeQuery(
    "SELECT documentId FROM CategoryDocuments
    WHERE categoryId = " + categoryId + " ");

while (result.next())
    cat.addDocument( retrieveDocument( new Integer( result.getInt("documentId") ) ) );

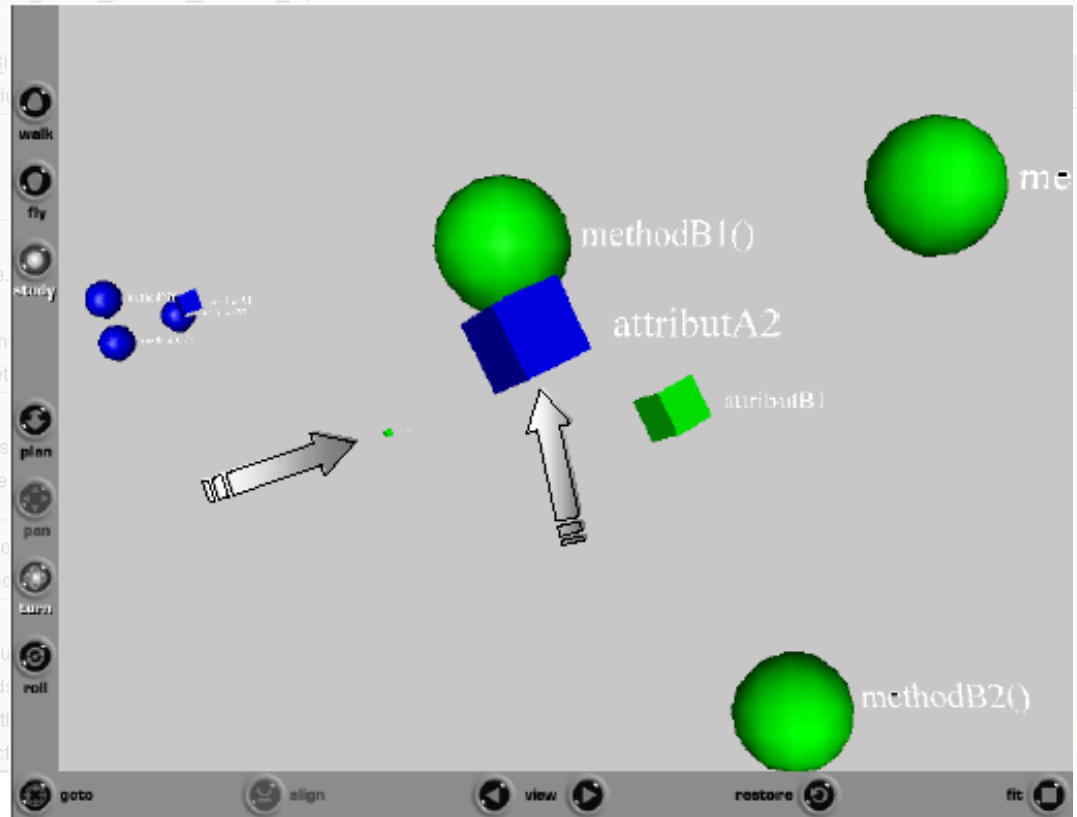
public class ClassB {
    public static int attributeB1;
    public static int attributeB2;
    public static void methodB1 () {
        ClassA.attributeA2 = 0;
        methodB2 ();
        methodB3 ();
    }
    public static void methodB2 () {
        ClassA.attributeA2 = 0;
        attributeB1 = 0;
    }
    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 **attributA2** nach **ClassB**

→ Löschung von **attributB2**



```
// Dokumente dieser Kategorie
result = stmt.executeQuery(
    "SELECT documentId FROM CategoryDocuments " +
    "WHERE categoryId = " + categoryId + "");

while (result.next())
    cat.addDocument( retrieveDocument( new Integer( result.getInt("documentId") ) );
```

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 m_i und 4 Attributen 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, \dots ; 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, \dots ; m_{20} nutzt m_{11}, a_3, a_4 ;

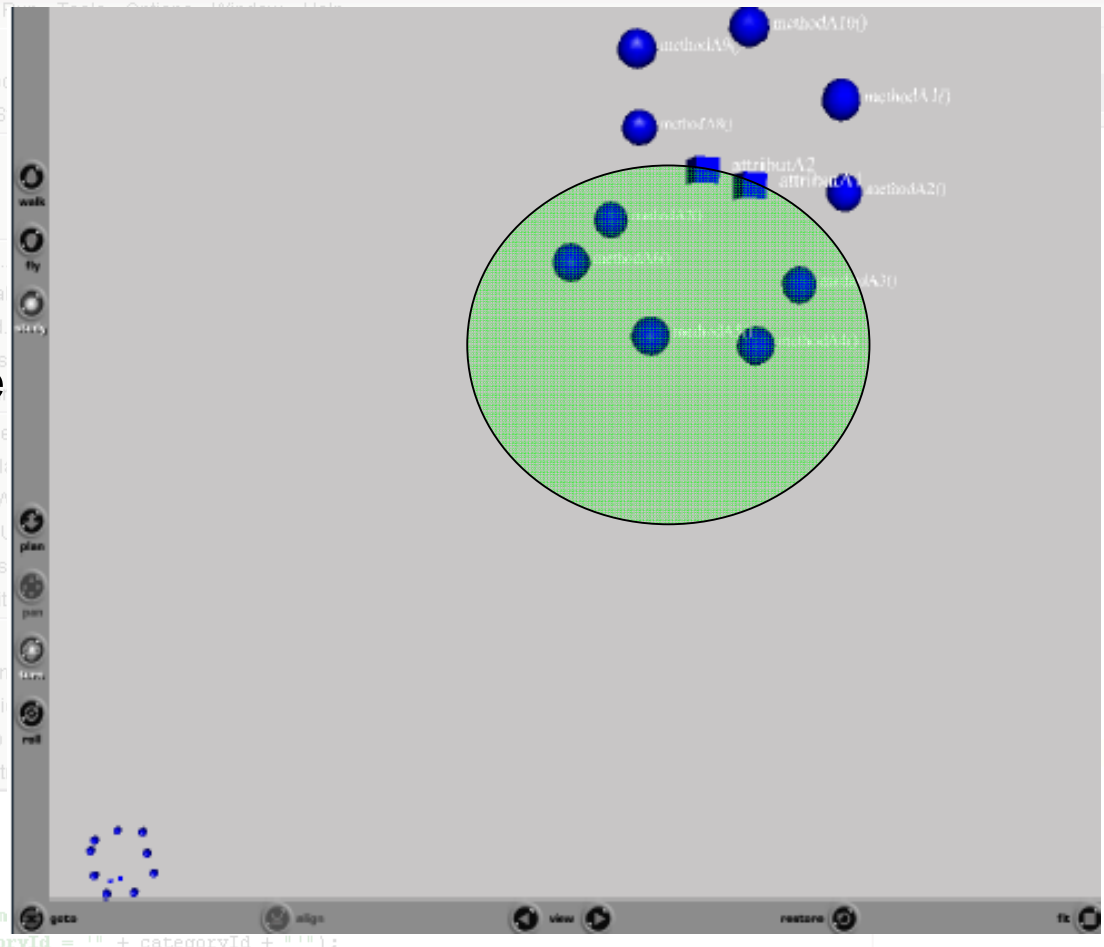
2.4. Identifikation von *Extract/Inline Class* Situationen (2)

- 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!



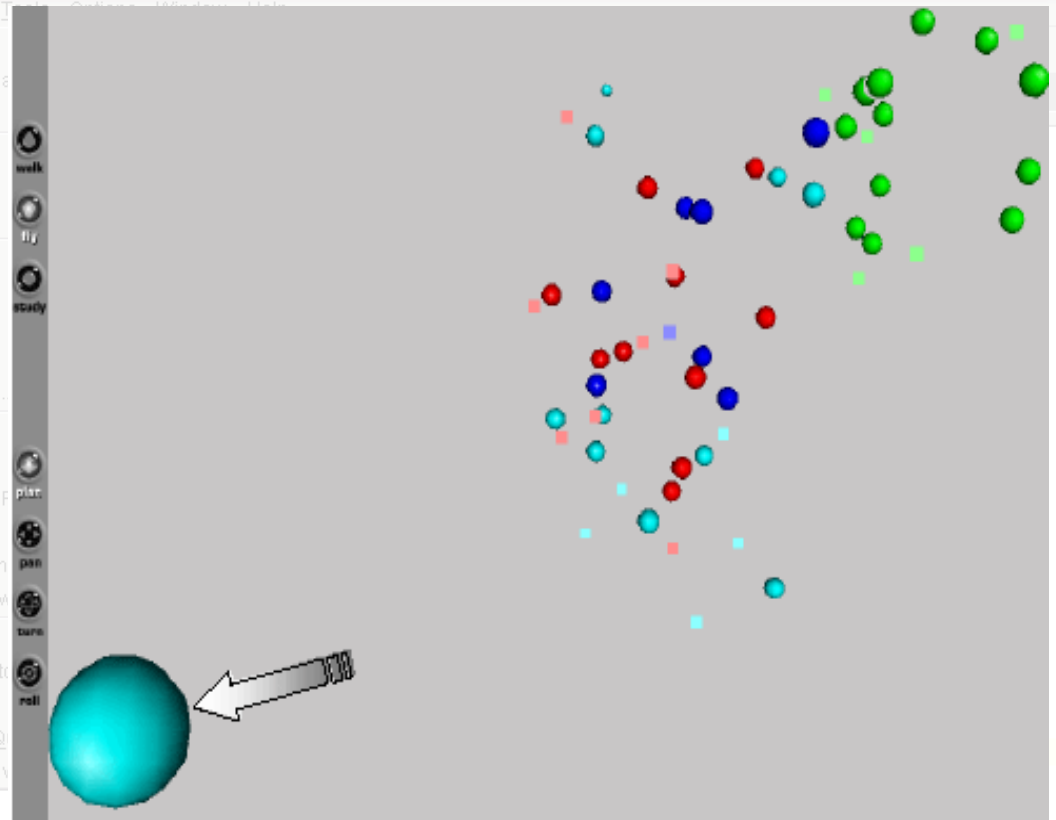
Visualisierung eines „real world“-Softwaresystems

- 4 Klassen, bestehend aus 40 Methoden, 18 Attributen

→ Unübersichtlich

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

→ Hilfreich, nützlich



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
- Klassen in Vererbungshierarchien
 - Bsp.: 2 Methoden werden als nicht kohärent erkannt, weil die benutzten Attribute aus der Oberklasse nicht berücksichtigt werden → Flattening (Abflachen)

Überblick

1. Einführung

1.1. Problem

1.2. Anforderungen

1.3. „Codegerüche“ \leftrightarrow 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)

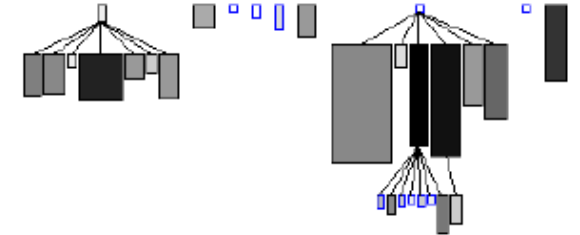
3.2. Code Inspection Tool (IntelliJ IDEA)

4. Zusammenfassung/ Ausblick

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



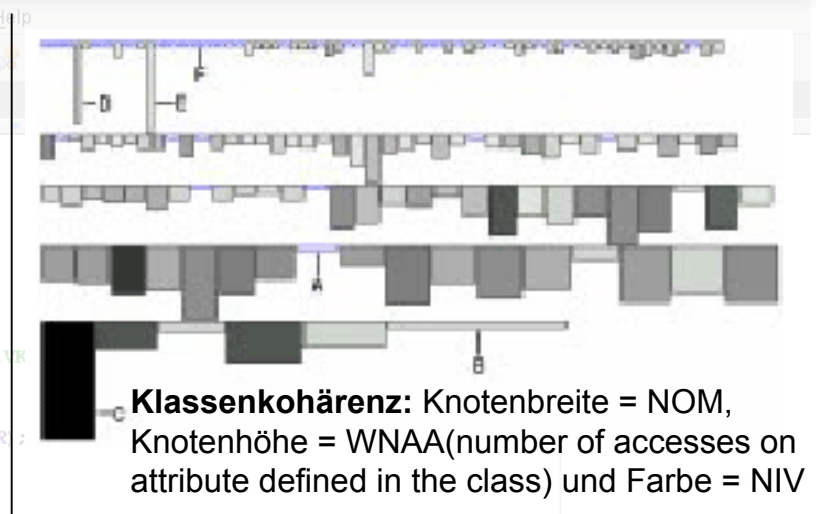
Vererbungsbaum: Knotenbreite = NIV,
Knotenhöhe = NOM und Farbe = NCV

Beispiel: Klassenkohärenz

- Graph zur Untersuchung der Kohärenz von Klassen eines Softwaresystems

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

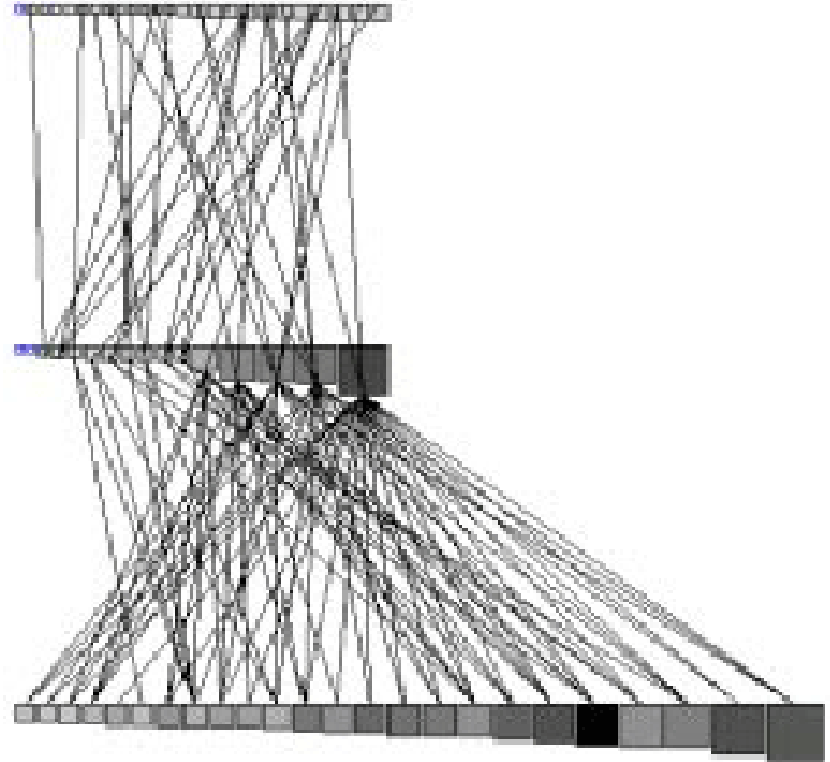


Bipartiter Konfrontationsgraph

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

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

The screenshot shows the IntelliJ IDEA interface with an 'Inspection - Unused declaration' window open. The window is divided into two main sections: a tree view on the left and a details pane on the right. The tree view shows a project structure with 'NewProject.ipr' containing a 'default package' with four items: 'MyClass.methodDemo()', 'MyClass.methodToGo()', 'MyClass.myInt', and 'MyClass.myMethod()'. The 'MyClass.methodDemo()' item is selected and highlighted. The details pane on the right shows the following information:

- Name:** public method void **methodDemo()**
- Location:** [MyClass](#) (default package)
- Problem synopsis:** Method is never used.

At the bottom of the window, there are several tabs for different inspection types: 'Unused declaration', 'Access modifiers', 'Can be static', 'Can be final', 'Same return value', 'Empty method', 'Local variable or parameter can be final', and 'JavaDoc'. The 'Unused declaration' tab is currently active.

3.2. Beispiele zum Code Inspection Tool

The screenshot shows an IDE interface with three highlighted code snippets and their corresponding inspection messages:

- Redundantes Type cast:** A code snippet shows a `while` loop with a `while (it.hasNext()) {` condition. A yellow highlight is under the `while` keyword. A tooltip message says "Type cast is redundant". A button labeled "Remove Redundant Cast" is visible.
- Unbenutzte Methode:** A tooltip message says "Private method isGroupAccountValid(java.sql.Connection, java.lang.Integer) is never used". A blue arrow points to the right side of the tooltip.
- Überflüssiger Parameter:** A code snippet shows a method signature: `mission(Connection c, VKCAcl acl, Integer userId, char permToken) {`. The parameter `c` is circled in blue. A tooltip message says "Parameter c is never used". A blue arrow points from the circled `c` to the tooltip.

```
while (it.hasNext()) {
    allCtxPerms.add((VKCPermission) it.next());
}

// ...

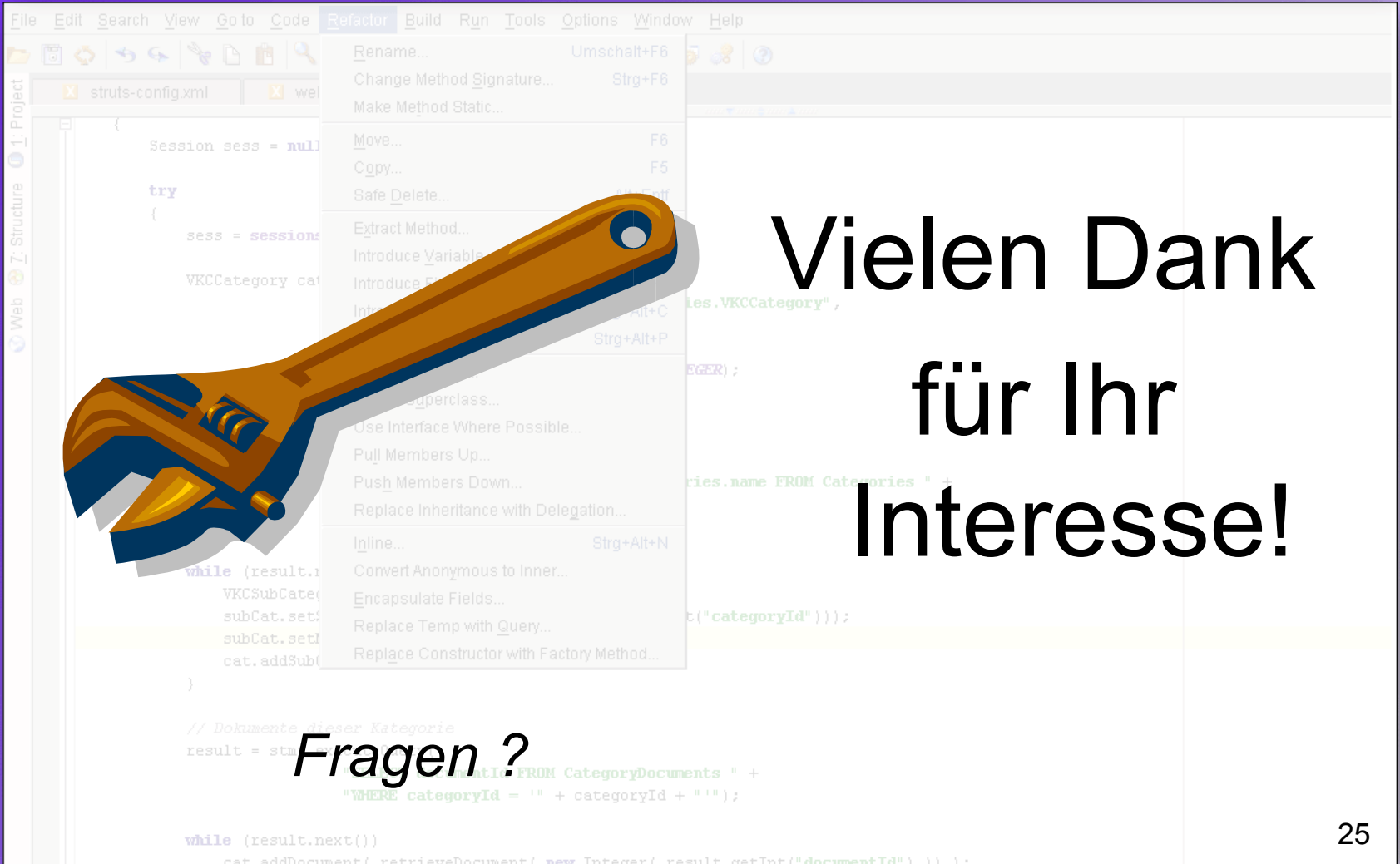
private boolean isGroupAccountValid(java.sql.Connection, java.lang.Integer) {
    // ...
}

mission(Connection c, VKCAcl acl, Integer userId, char permToken) {
    VKCCategoryAcl acl = new VKCCategoryAcl();
    // ...
}
```

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



The image shows a screenshot of an IDE with a refactor menu open. A large wrench icon is overlaid on the left side of the menu. The menu items include: Rename... (Umschalt+F6), Change Method Signature... (Strg+F6), Make Method Static..., Move... (F8), Copy... (F5), Safe Delete..., Extract Method..., Introduce Variable..., Introduce Field..., Introduce Parameter..., Use Interface Where Possible..., Pull Members Up..., Push Members Down..., Replace Inheritance with Delegation..., Inline... (Strg+Alt+N), Convert Anonymous to Inner..., Encapsulate Fields..., Replace Temp with Query..., and Replace Constructor with Factory Method... The background code shows a Java class with a try block and a while loop. The text 'Fragen?' is written in a large, stylized font over the bottom part of the code.

```
File Edit Search View Goto Code Refactor Build Run Tools Options Window Help
struts-config.xml
Session sess = null
try
(
sess = session
VKCCategory cat
while (result.next())
VKCSubCate
subCat.set
subCat.set
cat.addSub
)
// Dokumenten User Kategorie
result = sta
"WHERE categoryId = " + categoryId + " ";
while (result.next())
cat.addDocument( retrieveDocument( new Integer( result.getInt("documentId") ) ) );
```

**Vielen Dank
für Ihr
Interesse!**

Fragen?