

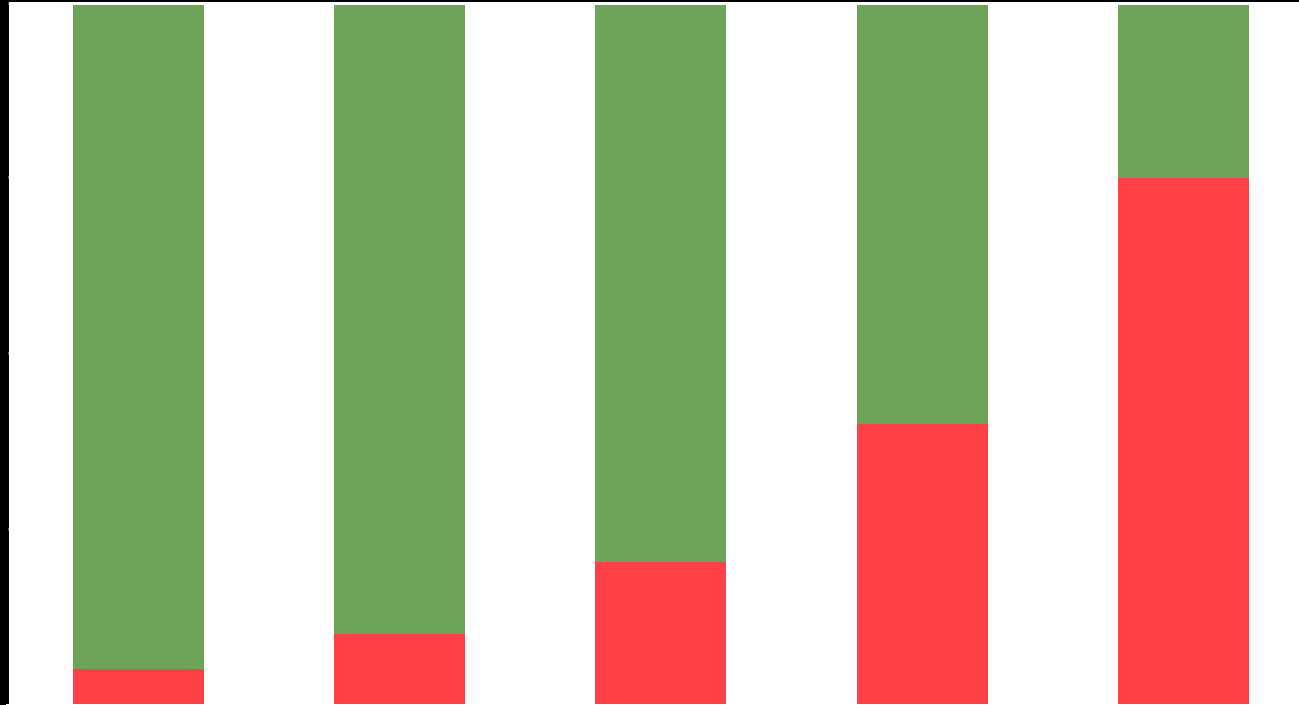
# Refactoring

Techniken des evolutionären Designs

# Softwarequalität

- Software besitzt Geschäftswert durch:
  - ▶ Funktionale Qualität:  
Funktionalität und Fehlerfreiheit für die einwandfreie Benutzung
  - ▶ Strukturelle Qualität:  
Design und Codestruktur für die nahtlose Weiterentwicklung
- Erfolgreiche Software muss beide Qualitäten besitzen

# Aufwandsverteilung bei wachsender Design-Schuld



Zeit für neue Features



Zeit für Defektbehebung

# Entscheidende Frage

Wie Sorge ich dafür, dass  
Veränderungen und  
Erweiterungen meiner Software  
**nicht zunehmend teurer** werden?

# Antwort

Indem ich mein Softwaredesign  
**ständig** aufräume und alle  
**unnötige Komplexität** entferne!

# REFACTORING

IMPROVING THE DESIGN  
OF EXISTING CODE

**MARTIN FOWLER**

With Contributions by **Kent Beck, John Brant,  
William Opdyke, and Don Roberts**

Foreword by **Erich Gamma**  
Object Technology International Inc.



*The Addison-Wesley Signature Series*

*"Any fool can write code that a computer can understand.  
Good programmers write code that humans can understand."*

—M. Fowler (1999)



# REFACTORING

Improving the Design of Existing Code

**Martin Fowler**

with contributions by  
**Kent Beck**



SECOND EDITION

# Refactoring?

„Eine Änderung an der internen Struktur eines Programms, um es leichter verständlich und besser modifizierbar zu machen, ohne dabei sein beobachtbares Verhalten zu ändern.“

[Fowler 99]

# Code Smells





# Code Smell

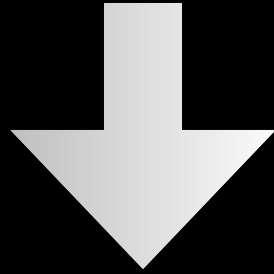
„A code smell is a **surface indication** that **usually** corresponds to a deeper problem in the system.“

Martin Fowler

<http://martinfowler.com/bliki/CodeSmell.html>

# Code Smell

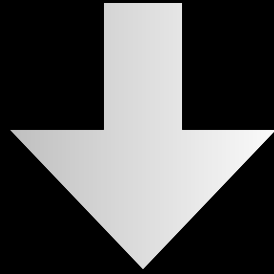
Indiz für notwendiges Refactoring



## Ein Refactoring

Muster um typische Codeveränderung sicher durchzuführen

# Long Method



# Extract Method

Bei Bedarf mehrfach

# Typische Code-Smells

- ▶ Lange Funktionen
- ▶ Unverständliche Namen
- ▶ Duplizierte Logik
- ▶ Kommentare
- ▶ Verschachtelte if-then-else
- ▶ Primitive Typen für fachliche Daten
- ▶ Code, der seine Intention nicht ausdrückt
- ▶ Datenklassen ohne wirkliches Verhalten

# Atomare Refactorings

- Rename
- Extract Method / Variable
- Inline Method / Variable
- Move Method
- Substitute Algorithm
- Encapsulate Field
- Add Parameter
- Remove Parameter / Variable / Method

# Schlechte Namen erzeugen Schall und Rauch

```
public class A {  
    private int b = 0;  
    public int gb() {  
        return b;  
    }  
    public void d(int a) {  
        c(a);  
        b += a;  
    }  
    private void c(int a) {  
        if (a <= 0.0) {  
            throw new IAE();  
        }  
    }  
}
```

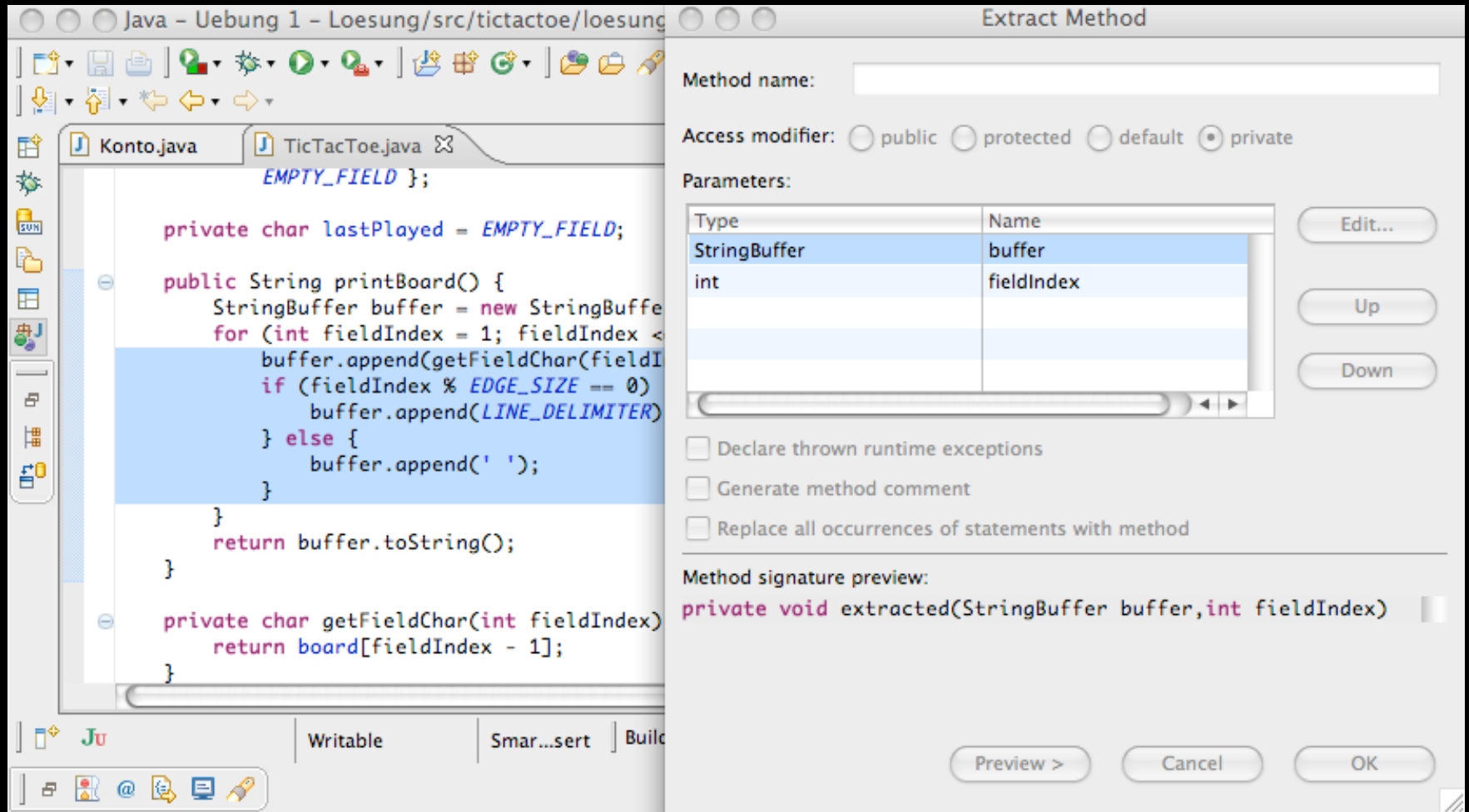
# Schlechte Namen erzeugen Schall und Rauch

```
public class Account {  
    private int balance = 0;  
    public int getBalance() {  
        return balance;  
    }  
    public void deposit(int amount) {  
        checkAmountIsNotNegative(amount);  
        balance += amount;  
    }  
    private void checkAmountIsNotNegative(int amount) {  
        if (amount <= 0.0) {  
            throw new IllegalArgumentException();  
        }  
    }  
}
```

*„Der Thesaurus ist mein  
wichtigstes Programmier-  
Werkzeug“*

Ward Cunningham

# Divide and Conquer: Extract Method



The screenshot shows an IDE window titled "Java - Übung 1 - Loesung/src/tictactoe/loesung" with two tabs: "Konto.java" and "TicTacToe.java". The "TicTacToe.java" tab is active, displaying the following code:

```
EMPTY_FIELD };
```

```
private char lastPlayed = EMPTY_FIELD;
```

```
public String printBoard() {  
    StringBuffer buffer = new StringBuffer(  
        for (int fieldIndex = 1; fieldIndex <  
            buffer.append(getFieldChar(fieldIndex));  
            if (fieldIndex % EDGE_SIZE == 0)  
                buffer.append(LINE_DELIMITER);  
            } else {  
                buffer.append(' ');  
            }  
        }  
    }  
    return buffer.toString();  
}
```

```
private char getFieldChar(int fieldIndex)  
    return board[fieldIndex - 1];  
}
```

The "Extract Method" dialog box is open, showing the following configuration:

- Method name:
- Access modifier: ☐ public ☐ protected ☐ default ☒ private
- Parameters:

Type	Name
StringBuffer	buffer
int	fieldIndex

Buttons: Edit... Up Down

- ☐ Declare thrown runtime exceptions
- ☐ Generate method comment
- ☐ Replace all occurrences of statements with method

Method signature preview:  
**private void** extracted(StringBuffer buffer,int fieldIndex)

Buttons: Preview > Cancel OK



# Wann ist es sinnvoll, zu extrahieren?

- Um die Lesbarkeit zu verbessern und Kommentare loszuwerden
- Um alle Aufrufe auf der gleichen Detailstufe zu haben
- Um redundante Abschnitte zusammenzufassen
- Zur Vorbereitung weiterer Umstellungen:
  - ▶ Um Codestücke zu extrahieren, die alle an ein (anderes) Objekt gehen
  - ▶ Um Teile an der API bereit zu stellen
  - ▶ Um abweichendes Verhalten in Subklassen überschreiben zu können

# Zusammengesetzte Refactorings

- Change Method Signature
  - Introduce Parameter Object
  - Replace Method with Method Object
  - Introduce Null Object
  - Extract Interface / Subclass / Superclass
  - Pull Up Field / Method / Constructor Body
  - Push Down Field / Method
  - **Replace Conditional with Polymorphism**
  - Replace Inheritance with Delegation
- ➔ <http://refactoring.com/catalog/index.html>

„If you want to refactor, the essential precondition is having solid tests.“

Martin Fowler

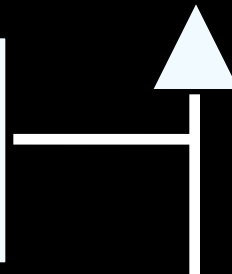
# Replace Conditional with Polymorphism

```
public class Konto...  
    private boolean istBetragGedeckt(double betrag) {  
        switch (kontotyp) {  
            case GIRO:  
                return betrag <= saldo + dispo;  
            case SPAR:  
                return betrag <= saldo;  
        }  
        throw new RuntimeException("Unbekannter Kontotyp");  
    }  
}
```

```
public abstract class Konto...  
    abstract protected boolean istBetragGedeckt(double betrag);
```

```
public class Sparkonto extends Konto...  
    protected boolean istBetragGedeckt(double betrag) {  
        return betrag <= saldo;  
    }  
}
```

```
public class Girokonto extends Konto...  
    protected boolean istBetragGedeckt(double betrag) {  
        return betrag <= saldo + dispo;  
    }  
}
```



# Refactoring-Gelegenheiten

- Ständig: Clean up the kitchen sink
- Regelmäßig: Clean up your garage
- Selten: Restructure the walls

# Refactoring-Rhythmus

1. Verifiziere, dass alle automatisierten Tests funktionieren
2. Entscheide, was du ändern möchtest
3. Führe einen Refactoring-Schritt vorsichtig durch
4. Lasse alle **automatisierten Tests** laufen, um sicherzustellen, dass die Änderungen nichts kaputt gemacht haben
  1. Wenn **OK**, dann versioniere/integriere aktuellen Stand
  2. Wenn **nicht OK**, dann gehe auf letzten lauffähigen Stand zurück
5. Zurück zu 3 bis das größere Refactoring fertiggestellt ist

# Demo:

# Replace Conditional with Polymorphism

# Technische Grenzen des automatischen Refactorings

- Es kann nur Code geändert werden, der auch geladen ist
- Nur wo der Compiler Abhängigkeiten erkennt, kann er konsistent ändern: Vorsicht mit Reflection!
- Es gibt nur wenige sprachübergreifenden Automatisierung, z.B. Java/SQL, Java/JavaScript, Java/XML
- Viele nennenswerte Refactorings sind nicht automatisierbar, weil sie Wissen über die Absicht des Codes verlangen



# Organisatorische Grenzen des Refactorings

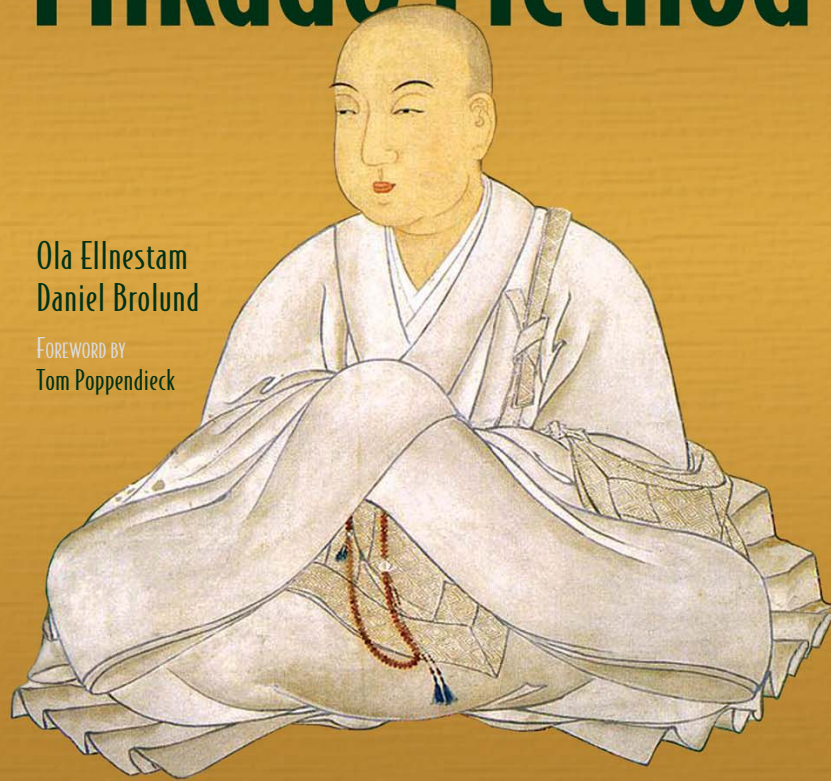
- Besitzdenken über Code verhindert oft notwendige Umbauten
- Verteilte Entwicklung verhindert notwendige Abstimmungen
- Refactoring braucht manchmal Mut – der muss auch in die Kultur passen
- Wenn „Produktivität“ zu fein gemessen wird, ist keine Zeit mehr für „unproduktives“ Refaktorisieren – und die Produktivität sinkt
- „Wenn Sie es gleich richtig gemacht hätten, müssten Sie jetzt keine Zeit für Umbauten verschwenden!“
- Conway's Law: „Architecture follows organization“

 MANNING

# The Mikado Method

Ola Ellnestam  
Daniel Brolund

FOREWORD BY  
Tom Poppendieck

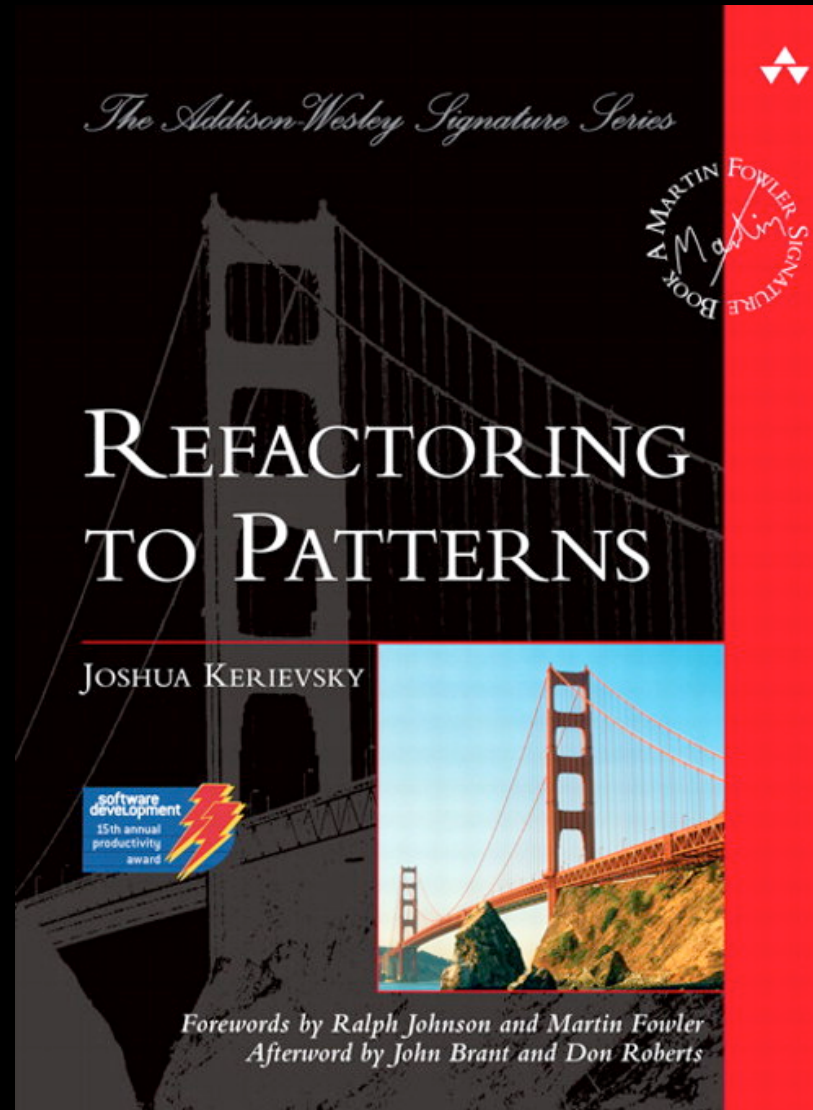


# Die Mikado-Methode

<http://mikadomethod.info/>

- Ein strukturierter Weg, um signifikante Änderungen an komplexem Code durchzuführen
- Grundlegende Konzepte des Vorgehens
  - ▶ Ziel bestimmen
  - ▶ Experiment durchführen
  - ▶ Ergebnis visualisieren
  - ▶ Änderungen rückgängig machen

# Refactoring to Patterns





Martin Lippert · Stefan Roock

# Refactorings in großen Softwareprojekten

Komplexe Restrukturierungen erfolgreich durchführen

Mit Beiträgen von Walter Bischofberger und Henning Wolf

dpunkt.verlag

*The Addison-Wesley Signature Series*



# REFACTORING DATABASES

EVOLUTIONARY  
DATABASE DESIGN

SCOTT W. AMBLER

PRAMOD J. SADALAGE



*Forewords by Martin Fowler, John Graham,  
Sachin Rekhi, and Dr. Paul Dorsey*