

# *JMangler*

Frithjof Kurtz

# JMangler

## Vortragsgliederung

- Motivation
- Java Grundlagen
- JMangler
  - ◆ Grundlagen
  - ◆ Transformationen
  - ◆ Algorithmen
- Anwendungsbeispiel CC4J

# Motivation

- Ständiger Wandel der Software
  - ◆ Vorhersehbare Änderung? → Änderung der Konfiguration
  - ◆ Änderung vor Kompilierung? → Quelltextmodifikation
  - ◆ Was aber tun, wenn...
    - Quelltext nicht zur Verfügung steht?  
→ Programmtransformation des Binärcodes
  - ◆ Besonders geeignet: Java
    - ② Genügend strukturelle und symbolische Informationen
    - ② Sehr weitgehende Rekonstruktion des Quelltextes möglich

# Transformationen in Java

- Ziel:
  - ◆ Alle Klassen einer Anwendung sollen abgedeckt werden
  - ◆ Problem: Nicht alle Klassen sind vor dem Start bekannt
    - Transformation während des Ladens der einzelnen Klassen
  - ◆ Wieder ein Problem: Java kennt keine Ladezeittransformation
    - JMangler benutzen

# Java Grundlagen

- Klassen können unterschiedlichen Ursprung haben
  - ◆ Lokal (Festplatte)
  - ◆ Netzwerk
  - ◆ Datenbanken
  - ◆ Dynamisch erzeugt werden
- Aufgabe des...

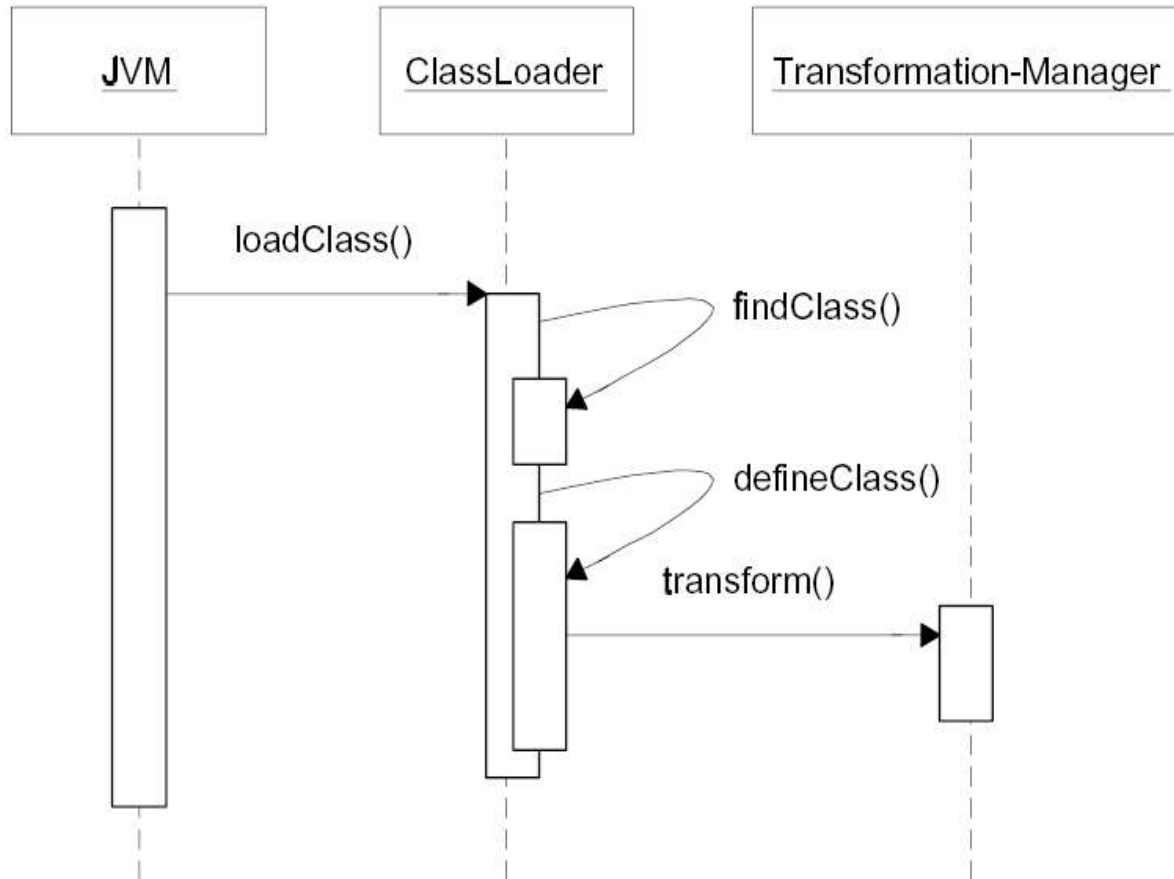
# Class Loader System

- Class Loader
  - ◆ Sind Java Objekte (mit einer Ausnahme)
  - ◆ Können auf jede von Java erlaubte Art Klassen laden
  - ◆ Werden von der JVM aufgerufen
- Bootstrap Class Loader
  - ◆ Teil der JVM
- Eigene Klassen über `findClass()` in das System laden

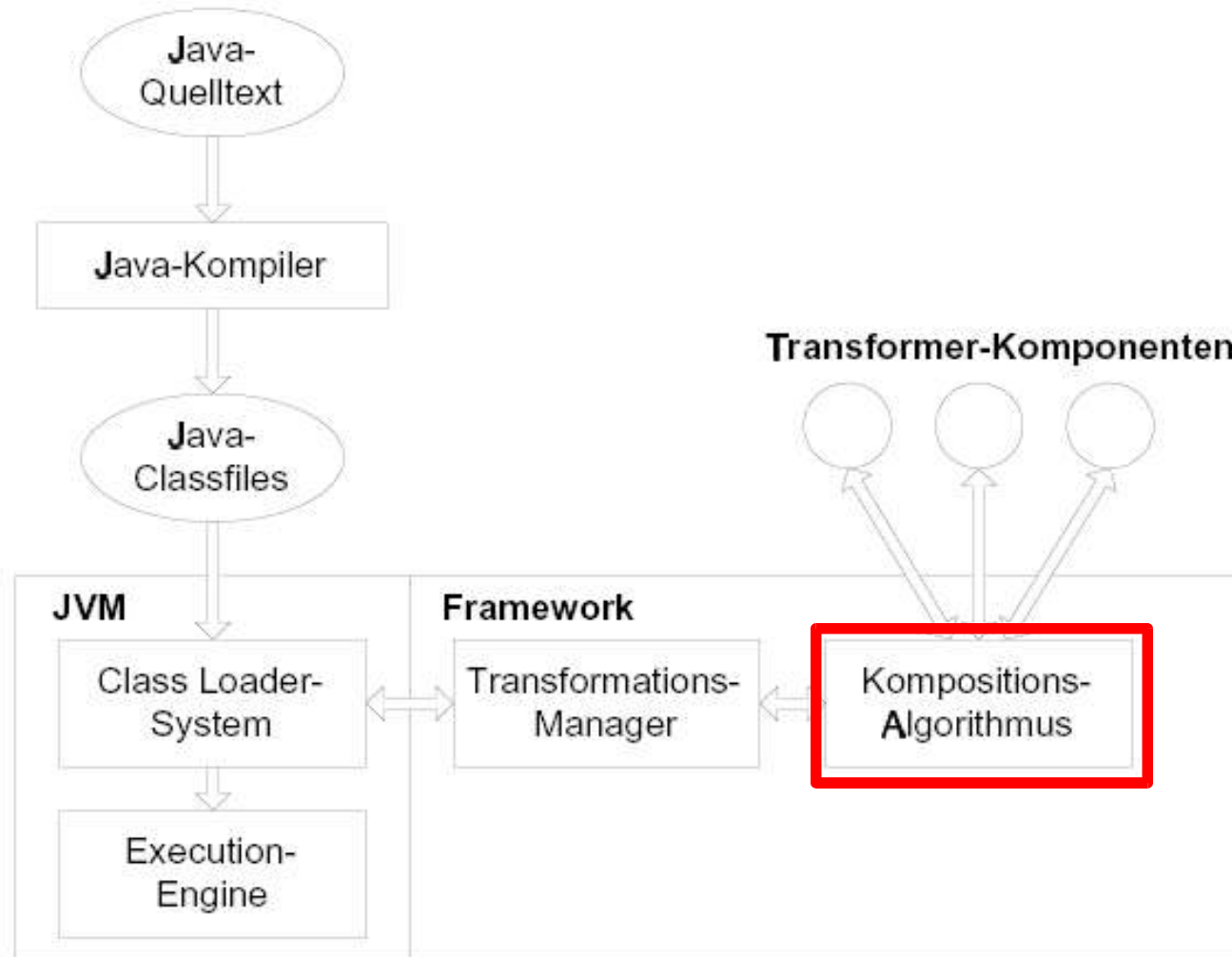
# Frameworkintegration in Java

- Klassen werden vom Class Loader geladen
- Klasse wird geladen und im Classfile Format auf dem Heap abgelegt. Danach Übergabe einer Referenz auf das Bytearray an die JVM durch `defineClass()`.
- `defineClass()` ist die einzige Möglichkeit Klassen in das System zu laden. Ansatzpunkt für Jmangler, da `defineClass()` final ist, daher keine Umgehung möglich.

# Integration des Transformationsaufrufs



# JMangler Architektur



# Anforderungen an Transformationsframework

- Unabhängig von JVM
- Verschiedene Transformer
- Unabhängige Transformer
- Transformer sollen auf Mengen von Klassen operieren können
- Bei Transformation Abhängigkeiten von Klassen beachten

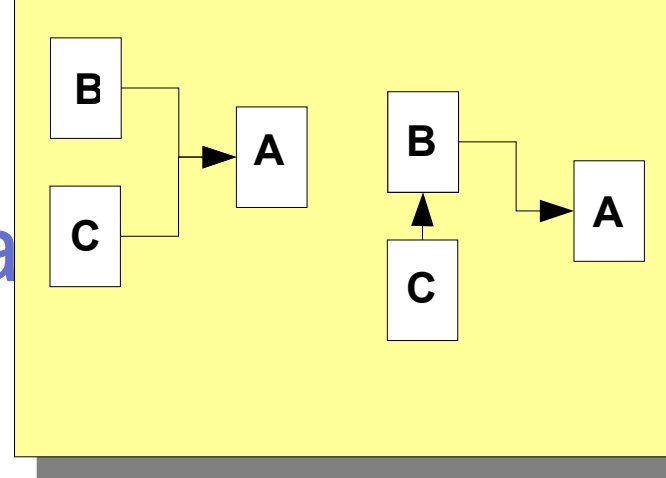
# Transformationen in Java (1)

- Potentielle Transformationen
  - ◆ Hinzufügen einer Klasse, Methode, Feld
    - ② Das Hinzufügen ist eine **legale Transformation**
  - ◆ Entfernen einer Klasse, Methode, Feld
    - ② Nur paketweit sichtbare Klassen, keine Referenzen auf die Klassen.
    - ② Unmöglich, zukünftige Referenzierungen festzustellen
    - ② Felder und Methode werden evtl. von der Reflection API genutzt
    - ② **Illegale Transformation**
  - ◆ Umbenennen einer Klasse, Methode oder Feldes
    - ② Werden über Name und Signatur referenziert
    - ② Umbenennung entspricht einer Entfernung mit Wiedereinfügung
    - ② **Illegale Transformation**

# Transformationen in Java (2)

- ◆ Änderung einer Methodensignatur
  - ② Es gilt das gleiche wie bei der Umbenennung von Methoden
  - ② **Illegale Transformation**
- ◆ Änderung einer Methoden-Throws-Klausel
  - ② Korrektheit der throws-Deklaration werden nur während der Kompilierung geprüft
  - ② **Legale Transformation**
- ◆ Änderung eines Methodenrückgabetyps
  - ② Ist verboten, weil dem Entfernen einer Methode und Einfügen einer geänderten Methode entsprechend
  - ② **Illegale Transformation**

# Transformationen in Java



- ◆ Änderung eines Feldtyps
  - ② Wird behandelt wie Änderung des Methodenrückgabetyps
  - ② **Illegale Transformation**
- ◆ Änderung der direkten Oberklasse einer Klasse
  - ② Die vollständige Menge der Oberklassen darf kein Element verlieren
  - ② **Legale Transformation**
- ◆ Änderung einer implements-Klausel
  - ② Es gilt das gleiche wie bei Änderung der Oberklasse
  - ② **Legale Transformation**
- ◆ Hinzufügen/Entfernen von Annotationen einer Klasse, Methode, eines Feldes
  - ② Ist dann erlaubt, solange die Sichtbarkeit nicht eingeschränkt wird.
  - ② **Legale Transformation**

# Legale Transformationen in Java

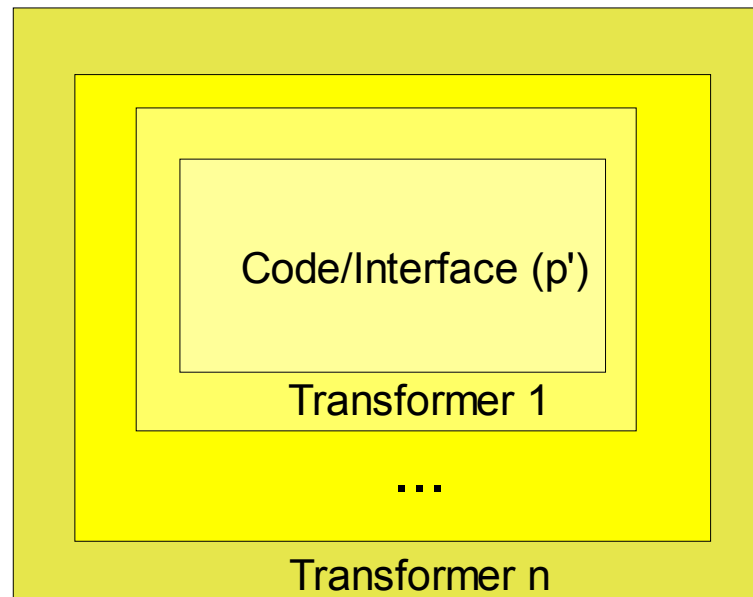
- Legale Transformationen
  - ◆ Änderung der direkten Oberklasse einer Klasse
  - ◆ Änderung einer implements-Klausel
  - ◆ Hinzufügen/Entfernen von Annotationen einer Klasse, Methode, eines Feldes
  - ◆ Änderung einer Methoden-Throws-Klausel

# Kompositionsalgorithmus NAIV

```
1. do
2.   p' = p;
3.   p =  $\kappa_n \circ \kappa_{n-1} \circ \dots \circ \kappa_1 (p')$ ;
4. until p' = p;
5. return p;
```

$b \circ a := b(a(x))$

# Kompositionsalgorithmus NAIV



Transformiertes Programm p

# NAIVes Beispiel (1)

## Ursprungsprogramm

```
public class C {  
    private B b = new B();  
  
    public void manipulateB() {  
        b.doSomething();  
    }  
}
```

## Transformer-Komponenten

ACCESS ○ COUNTER

```
public class C {  
    private B b = new B();  
    private int b_counter = 0; (1)  
  
    private void setB(B b) { (2)  
        this.b = b; (2)  
    } (2)  
  
    private B getB(B b) { (2)  
        b_counter++; (3)  
        return b; (2)  
    }  
  
    public void manipulateB() {  
        b_counter++; (1)  
        getB().doSomething(); (2)  
    }  
}
```

# NAIVes Beispiel (2)

## Ursprungsprogramm

```
public class C {  
    private B b = new B();  
  
    public void manipulateB() {  
        b.doSomething();  
    }  
}
```

## Transformer-Komponenten

**COUNTER** ○ **ACCESS**

```
public class C {  
    private B b = new B();  
    private int b_counter = 0; (2)  
  
    private void setB(B b) { (1)  
        this.b = b; (1)  
    } (1)  
  
    private B getB(B b) { (1)  
        b_counter++; (2)  
        return b; (1)  
    }  
  
    public void manipulateB() { (1)  
        getB().doSomething(); (1)  
    }  
}
```

# Schlußfolgerungen

- Transformierung des Interface Codes ist unabhängig von einer bestimmten Reihenfolge.
- Transformierung des Methoden Codes ist abhängig von einer bestimmten Reihenfolge!
- Lösung:
  - ◆ Aufteilen der Transformation in 2 Phasen
  - ◆ 1. Phase: Mengenmodifizierung (Interfaces)
  - ◆ 2. Phase: Listenmodifizierung (Code)
- Nur für Code Transformation Reihenfolge vorgeben.

# Weiterentwicklung von NAIV (1)

- Interface-Transformationen
  - ◆ Hinzufügen von: Klasse, Methode oder Feld
  - ◆ Methoden-Throws Klausel ändern
  - ◆ Direkte Oberklasse ändern
  - ◆ Implements-Klausel ändern
  - ◆ Annotationen einer Klasse, Methode oder eines Feldes hinzufügen oder entfernen
- Interface-Transformationen sind Erweiterungen einer Menge
  - ◆ Daher: Reihenfolge unerheblich
  - ◆ Transformationen von Komponenten können weitere Transformationen triggern → iterative Transformation

# Weiterentwicklung von NAIV (2)

- Code-Transformationen
  - ◆ Entspricht einer Listenmanipulation
  - ◆ Daher: Reihenfolge entscheidend, nicht willkürlich wählbar
  - ◆ Durch Festlegen einer Reihenfolge wird gegenseitiges Triggern ausgeschlossen
- Code-Transformation wird nicht iterativ ausgeführt
- Ergebnis ist...

# Kompositionsalgorithmus

## PROGRESS

1. do
2.  $p' = p;$
3.  $p = \kappa_n \circ \kappa_{n-1} \circ \dots \circ \kappa_1 (p');$
4. until  $p' = p;$
5.  $p = \xi_m \circ \dots \circ \xi_2 \circ \xi_1 (p);$
5. return  $p;$

# Schlußfolgerungen

- Interface-Transformationen noch problematisch
  - ◆ Einige Annotationen dürfen sowohl entfernt als auch hinzugefügt werden!
  - ◆ Reihenfolge ist dann wieder relevant
  - ◆ Interface-Transformatoren können sich gegenseitig stören, z.B. Transformer1 entfernt Annotation1 während Transformer2 Annotation 1 wieder hinzufügt!
- Lösung: Transformationsrichtung einführen
  - ◆ Definieren einer Halbordnung:  $\mathbf{p}_{\text{vor}} \subseteq \mathbf{p}_{\text{nach}}$
  - ◆ Beispiel: **private**  $\leq$  **protected**  $\leq$  **public**

# Weiterentwicklung PROGRESS

- Bisher keine Konflikterkennung
- Konflikterkennung für Interface-Transformationen ausreichend
  - ◆ Konflikte bei Code-Transformation können durch entsprechende Kompositionsreihenfolge verhindert werden
- Konflikterkennung für Interface-Transformer
  - ◆ Transformer analysieren Programm
  - ◆ Übergeben Kompositionsalgorithmus gewünschte Transformationen
  - ◆ Kompositionsalgorithmus prüft Anforderungen auf Konflikte, löst sie (falls möglich), transformiert schließlich Programm

# Mögliche Konflikte

- **Feldkonflikt:**
  - ◆ Transformer haben keine Wissen über ihre gegenseitige Existenz oder die Handlungen
  - ◆ Ein identisches Feld könnte zu Konflikten während des Programmlaufs führen
- **Methodenkonflikt**
  - ◆ Implementierungen zweier Methoden können nicht gemischt werden
- **Klassenkonflikt**
  - ◆ Gleichlautende Klassen werden mit unterschiedliche Zielen von unterschiedlichen Transformern eingesetzt
- **Oberklassenkonflikt**
  - ◆ Kann nur gelöst werden, wenn: 2 unterschiedliche Oberklassen in gegenseitiger Beziehung stehen, da keine multiple Vererbung möglich

# Kompositionsalgorithmus TAU

```
1. do
2.   for all  $\kappa_i$  do  $\delta_i = \kappa_i(p)$ ;
3.   if conflict( $\delta_1, \dots, \delta_n$ ) return  $\varepsilon$ ;
4.    $\delta_{ges} = \text{merge}(\delta_1, \dots, \delta_n)$ ;
5.    $p' = p$ ;
6.    $p = \text{apply}(\delta_{ges}, p')$ ;
7.   if  $\neg(p' \subseteq p)$  return  $\varepsilon$ ;
8. until  $p' = p$ ;
9.  $p = \xi_m \circ \dots \circ \xi_2 \circ \xi_1(p)$ ;
10. return  $p$ ;
```

# TAU Beispiel

## Ursprungsprogramm

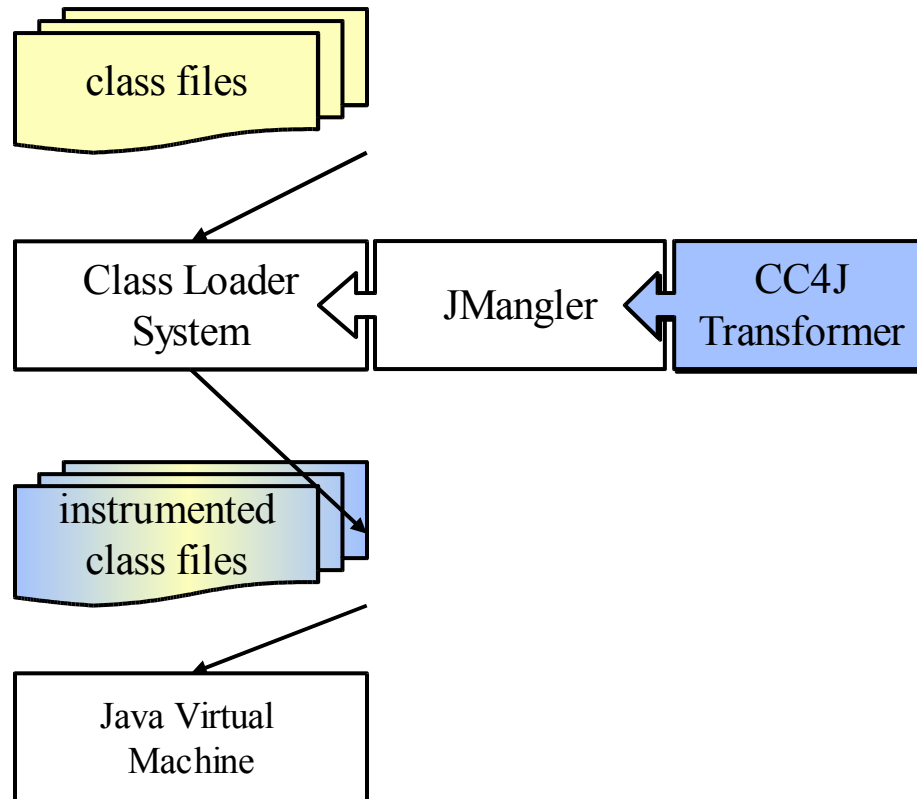
```
public class C {  
    private B b = new B();  
  
    public void manipulateB() {  
        b.doSomething();  
    }  
}
```

## Transformer-Komponenten

**COUNTER** ○ **ACCESS**

```
public class C {  
    private B b = new B();  
    private int b_counter = 0; (1)  
  
    private void setB(B b) { (1)  
        this.b = b; (1)  
    } (1)  
  
    private B getB(B b) { (1)  
        b_counter++; (3)  
        return b; (1)  
    }  
  
    public void manipulateB() {  
        getB().doSomething(); (2)  
    }  
}
```

# Beispiel CC4J (Code Coverage Tool)



Fertig :)

**Ihr habt doch bestimmt keine Fragen?**