Aspektorientierte Programmierung und AspectJ

Von Markus Wondrak

Agenda

- Was ist Aspektorientierte Programmierung?
  - Motivation
  - Komponenten und Aspekte
- Was ist AspectJ?
  - Konzepte
  - Weaving
  - Beispiel Anwendungen

Wozu ein neues Paradigma?

(Motivation)

Ein Beispiel: Konto

Anforderungen

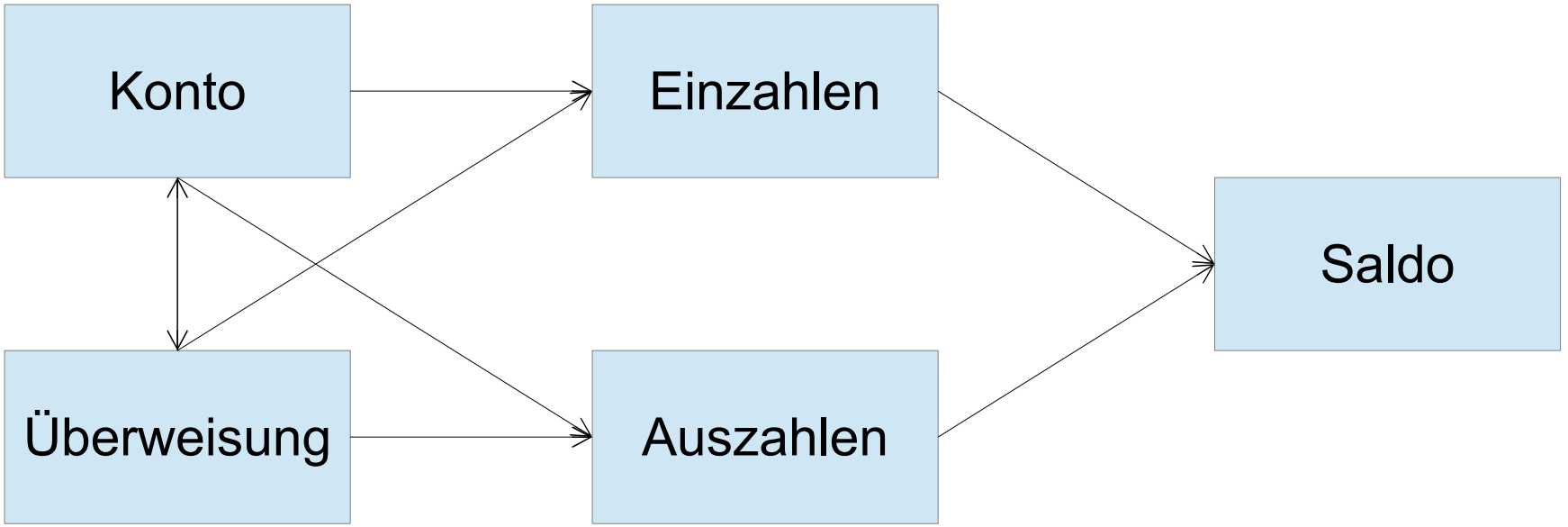
- Saldo
- Eindeutige Kennung
- Einzahlen
- Auszahlen
- Überweisen

Komponenten als

- Prozedur
- Klasse
- Funktion

Funktionale Dekomposition

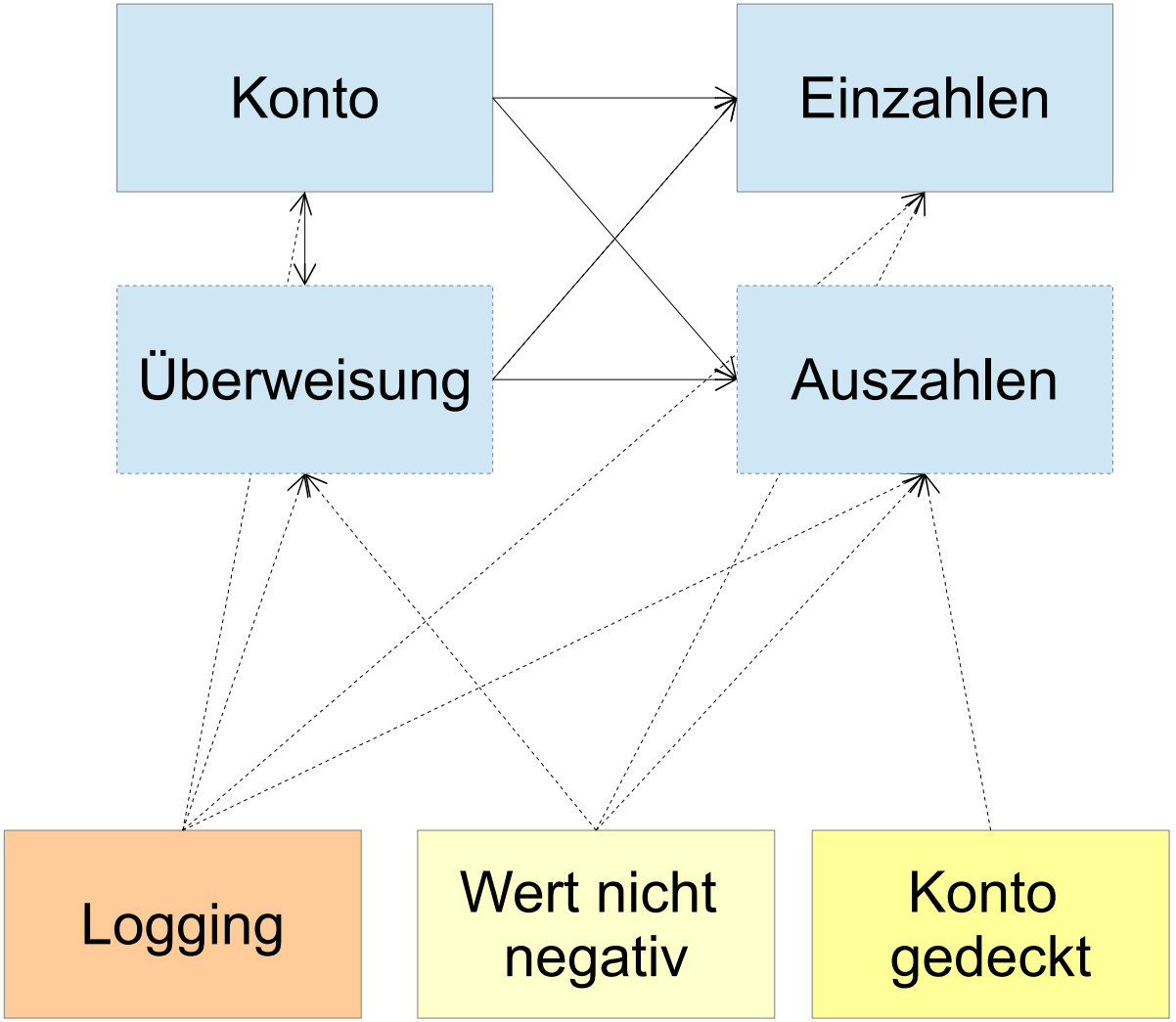
Komposition der Komponenten


Konto
Einzahlen
Saldo
Überweisung
Auszahlen

```java
public class Account {
  private double amount = 0.0;
  private int id;

  public Account(int id) {
    this.id = id;
  }

  public void deposit(double value) {
    amount += value;
  }

  public void withdraw(double value) {
    amount -= value;
  }

  public void transfer(Account other, double value) {
    withdraw(value);
    other.deposit(value);
  }

  Public double getAmount() {
    return amount;
  }

  public int getId() {
    return id;
  }
}
```

Zusätzliche Anforderungen:

- Logging (Transaktionen / Fehler)
- Validierung, dass der Wert nicht < 0
- Validierung, dass Konto gedeckt


Konto
Einzahlen
Überweisung
Auszahlen
Logging
Wert nicht negativ
Konto gedeckt

```java
public class Account {

  private static final Logger log =
      Logger.getLogger(Account.class);

  private static final String NOT_ENOUGH_MONEY =
      "not enough money";
  private static final String VALUE_IS_NEGATIVE =
      "value is negative";

  private double amount = 0.0;
  private int id;

  public Account(int id) {
    this.id = id;
  }

  public void deposit(double value) {
    if (value  < 0) {
      log.error(VALUE_IS_NEGATIVE);
      throw new
        IllegalArgumentException(VALUE_IS_NEGATIVE);
    }

    amount += value;

    log.info("deposit " + value);
  }

  ...
```

```java
  public void withdraw(double value) {
    if (value  < 0) {
      log.error(VALUE_IS_NEGATIVE);
      throw new
        IllegalArgumentException(VALUE_IS_NEGATIVE);
    }

    if (value > amount) {
      log.error(NOT_ENOUGH_MONEY);
      throw new
        IllegalArgumentException(NOT_ENOUGH_MONEY);
    }

    amount -= value;

    log.info("withdraw " + value + " from " + id);
  }

  public void transfer(Account other, double value) {
    if (value  < 0) {
      log.error(VALUE_IS_NEGATIVE);
      throw new
        IllegalArgumentException(VALUE_IS_NEGATIVE);
    }

    withdraw(value);
    other.deposit(value);

    log.info("transfer ");
  }

}
```

Was ist passiert?

Cross Cutting Concerns (CCCs)

„In general, whenever two properties being programmed must compose differently and yet be coordinated, we say that they cross cut each other“

aus [aop97]

```
public class Account {

  private static final Logger log =
      Logger.getLogger(Account.class);

  private static final String NOT_ENOUGH_MONEY =
      "not enough money";
  private static final String VALUE_IS_NEGATIVE =
      "value is negative";

  private double amount = 0.0;
  private int id;

  public Account(int id) {
    this.id = id;
  }

  public void deposit(double value) {
    if (value  < 0) {
      log.error(VALUE_IS_NEGATIVE);
      throw new
        IllegalArgumentException(VALUE_IS_NEGATIVE);
    }

    amount += value;

    log.info("deposit " + value);
  }

  ...
```

```
  public void withdraw(double value) {
    if (value  < 0) {
      log.error(VALUE_IS_NEGATIVE);
      throw new
        IllegalArgumentException(VALUE_IS_NEGATIVE);
    }

    if (value > amount) {
      log.error(NOT_ENOUGH_MONEY);
      throw new
        IllegalArgumentException(NOT_ENOUGH_MONEY);
    }

    amount -= value;

    log.info("withdraw " + value + " from " + id);
  }

  public void transfer(Account other, double value) {
    if (value  < 0) {
      log.error(VALUE_IS_NEGATIVE);
      throw new
        IllegalArgumentException(VALUE_IS_NEGATIVE);
    }

    withdraw(value);
    other.deposit(value);

    log.info("transfer ");
  }

}
```

Ein neuer Ansatz

(Komponenten und Aspekte)

Unterteilung in Komponenten und Aspekte

Komponenten

- Können abstrahiert werden
- Einheit der Dekomposition

Aspekte

- Können nicht abstrahiert werden
- Betreffen mehrere Komponenten

System	Komponenten	Aspekte
Bildbearbeitung	Filter	Schleifenfusion Ergebnisteilung
Webanwendung	Datenbankzugriff URL-Routing	Sicherheit Transaktionen Fehlerbehandlung
Matrix Algorithmen	Lineare Algera Operationen	Matrix Representation

Eigene Sprachen für …

- Aspekte
- Komponten

Und ein Mechanismus um aus beiden zusammen **ein** System zu erstellen

System in einer herkömlichen Sprache:

- Sprache
- Compiler
- Programm geschrieben in der Sprache

System in einer Sprache mit AOP:

- Sprache für...
  - Komponenten
  - Aspekte
- *Weaver* zum kombinieren
- Ein Programm...
  - Aspekte in Aspekt Sprache
  - Komponten in Komponenten Sprache

Aspekt Sprache für Java

(AspectJ)

AspectJ – Open Source unter eclipse Foundation

AOP Implementierung für JVM

Definition von Aspekten:

- Join Point
- Pointcut
- Advice
- (Intertype Declaration)

Was ist ein Join Point?

Komposition der Aspekte

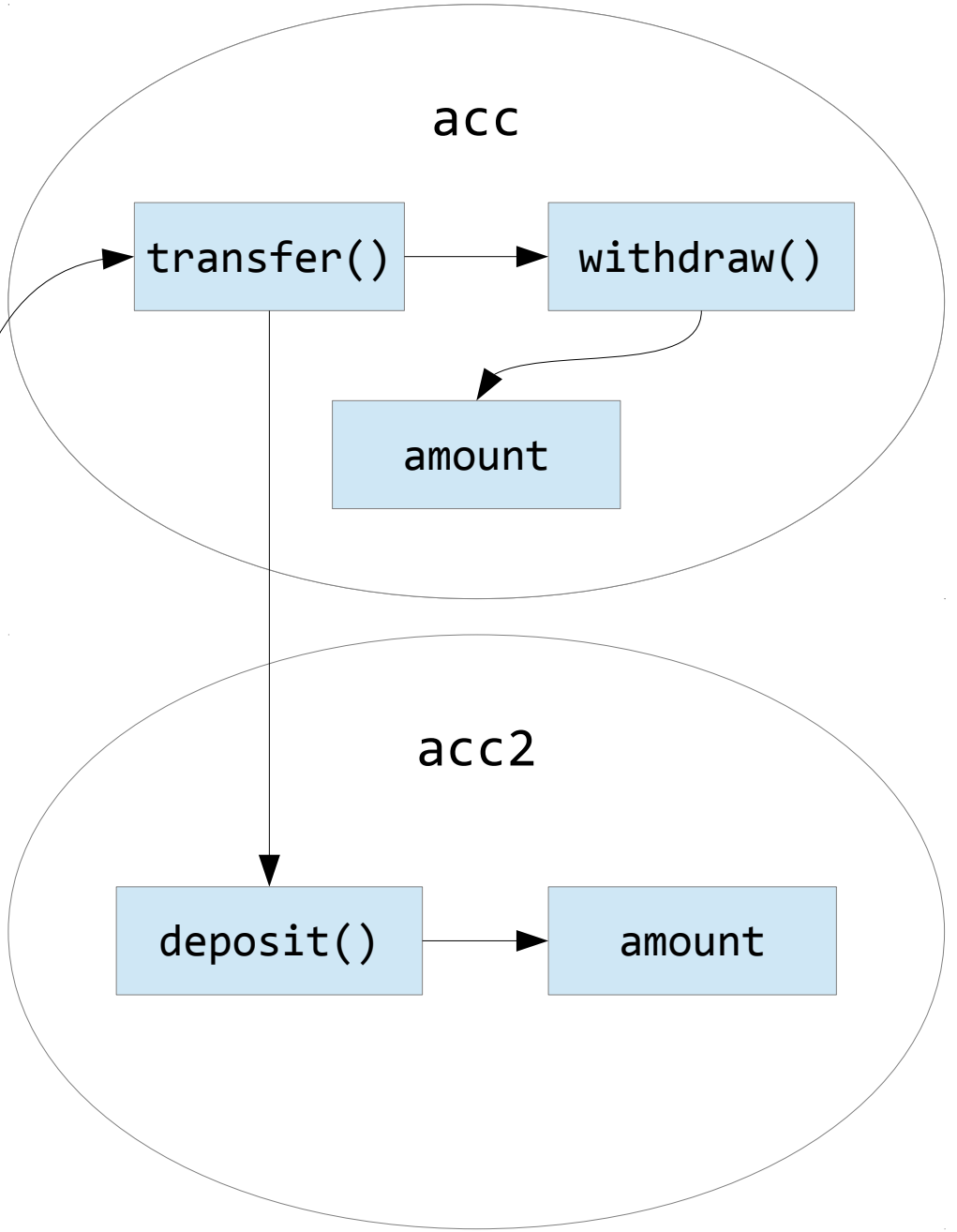
Join Point Model – Stellen für das weaving

Besteht aus dem Ausführungsgraphen

- Knoten: Nutzung einer Referenz, Methodenaufruf
- Kanten: Kontrolstrukturen

```java
public void deposit(double value) {
  amount += value;
}

public void withdraw(double value) {
  amount -= value;
}

public void transfer(
      Account other, double value) {
  withdraw(value);
  other.deposit(value);
}
```

```java
public static void main(
      String[] args) {

  Account acc = ...
  Account acc2 = ...

  acc.transfer(acc2, 10);
}
```



Die wichtigsten Join Points:

- `call()` / `execution()`

  Ausführung einer Methode / Konstruktor

- `get()` / `set()`

  Zugriff auf Referenz

- `handler()`

  Exception wird gefangen

Was ist ein Pointcut?

Definition einer Menge von *Join Points*

Kombinieren verschiedener Definitionen

Anreichern der Definition mit Kontext Informationen

```
pointcut accessAmount() :
  get(double com.bank.model.with.Account.amount) ||
  set(double com.bank.model.with.Account.amount);
```

```
pointcut everyAccountMethod() :
  execution(public * com.bank.model.with.Account.*(..));
```

```
pointcut everyStringMethod() :
  execution(* String ..*(..));
```

```
pointcut createAccount() :
  call(public com.bank.model.with.Account.new(..));
```

Zugriff auf den Kontext mit

- `this`

  Zugriff auf das Betreffende Objekt
- `target`

  Zugriff auf das Ziel Objekt
- `args`

  Zugriff auf die Parameter

```
pointcut amountMethod(double amount, Account account) :
  execution(public * com.bank.model.with.Account.*(.., double))
    && args(.., amount)
    && this(account);
```

```
pointcut callTranfer(Account account) :
  execution(public * com.bank.model.with.Account.transfer(..))
    && target(account);
```

Kontrollflußbasierter Bezeichner `cflow`

Alle Join Points innerhalb der Pointcut Definition

Nützlich für rekursive Funktionen

```
pointcut recursiveCall() :
  call(* * ..MyObject.myFunc()) &&
  !cflow(call(* * ..MyObject.myFunc()));
```

Und ein Advice?

Methoden-ähnliches Konstrukt

Setzt sich zusammen aus:

- Typ
- Parameter Liste
- Pointcut Definition
- Aspektcode („normaler“ Javacode)

Spezielle Kontext Objekte

- `thisJoinPoint`

  Informationen über den aktuellen Join Point

- `thisJoinPointStaticPart`

  Reduziert auf den rein statischen Teil

Typ definiert die Stelle am Join Point

- `before`
- `after` (`returning` / `throwing`)
- `around`

Before - Advice

Code wird vor dem Join Point ausgeführt

```
before() : allImportantMethods() {
  System.out.println("entering " + thisJoinPoint);
}
```

After – Advice

Code wird nach dem Join Point ausgeführt

Problem:

returned der Join Point oder fliegt Exception

Lösung:

Qualifizierung des *After*-Advices

```
after() returning (String result) : someDifficultOperation() {
  System.out.println("result is " + result);
}
```

```
after() throwing (IllegalStateException e) :
    someDifficultOperation() {
  System.out.println("operation failed with " + e.getMessage());
}
```

Around – Advice

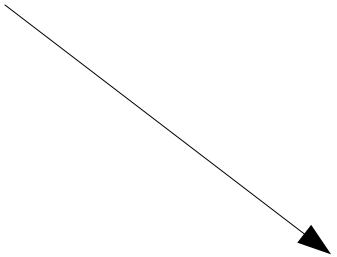
Besitzt die komplette Kontrolle über Join Point

`proceed()` führt den Join Point aus

Zugriff auf Parameter und Rückgabewert

```java
public String someMethod(String a, String b) {
  System.out.println("a: " + a + "; b: " + b);
  return a + b;
}
```

```java
SomeObject o = new SomeObject();
System.out.println(o.someMethod("Hallo", "aspectj"));
System.out.println(o.someMethod("hey", "ho"));
```



```
a: Hallo, ; b: aspectj
Hallo, aspectj
a: hey!; b: ho!
hey!ho!
```

```
pointcut someMethod(String a, String b) :
  execution(public String ...*.someMethod(..))
    && args(a, b);

String around(String a, String b) : someMethod(a, b) {
  if (b.equals("aspectj")) {
    b = "Welt!";
  }

  String result = proceed(a, b);

  if (a.equals("hey!")) {
    proceed(b, a);
  }

  return "'" + result + "'";
}
```

```
pointcut someMethod(String a, String b) :
  execution(public String ...*.someMethod(..))
    && args(a, b);

String around(String a, String b) : someMethod(a, b) {
  if (b.equals("aspectj")) {
    b = "Welt!";
  }

  String result = proceed(a, b);

  if (a.equals("hey!")) {
    proceed(b, a);
  }

  return "'" + result + "'";
}
```

```
a: Hallo, ; b: Welt!
'Hallo, Welt!'
a: hey!; b: ho!
a: ho!; b: hey!
'hey!ho!'
```

Und wie funktioniert das?

Aspekt- in Komponenten-Code einweben (*weaving*)

Statische Transformation des Java Bytecodes

AspectJ besteht aus Front- und Backend

Frontend

- Kompiliert AspektJ- und Java-Sourcecode
- Advices werden zu Methoden
- Zusätzlicher Parameter:
  - `thisJoinPoint`
  - Kontext Objekte durch `this` / `target` / `args`
  - *Around* Advice erfordert Closure Objekt

Backend

- Identifiziert Join Points zu Pointcut Definitionen
  - Statische
  - Dynamische
- Webt Methoden Aufruf ein
  - Stellt Parameter zur Verfügung
  - Aufruf der „Advice-Methode“

```
public void setSomeValue (Object o) {
  // ...
}
```

```
   L0
     LINENUMBER 18 L0
     RETURN
```

```
before (String s) : call (public void ...set*(..))
  && args ( s ) {
  // ...
}
```

```
   L0
    ALOAD 1: o
    ASTORE 2
   L1
    LINENUMBER 18 L1
    ALOAD 2
    INSTANCEOF String
    IFEQ L2
    INVOKESTATIC SomeAspect.aspectOf () : SomeAspect
    ALOAD 2
    CHECKCAST String
    INVOKEVIRTUAL SomeAspect.ajc$before$com_bank_aspects_SomeAspect$3$9befb9f7 (String)
   L2
   FRAME APPEND [Object]
    RETURN
```

```
public void setSomeValue ( Object o ) {
  if (o instanceof String) {
    SomeAspect a = SomeAspect.aspectOf();
    a.ajc$before$com_bank_aspects_SomeAspect$3$9befb9f7((String) o);
  }
}
```

Dynamische Tests (*Residues*)

Aufwendiger als statische

Optimierung um *Residues* zu vermeiden

```
public void setSomeInt(int i) {
  // ...
}
```

```
before (String s) : call (public void ...set*(..))
  && args ( s ) {
  // ...
}
```

Keine Transformation!

```
public void setSomeString(String str) {
  // ...
}
```

```
before (String s) : call (public void ...set*(..))
  && args ( s ) {
  // ...
}
```

Kein `instanceof` Test!

Before:

- Aufruf der Advice Methode vor dem Join Point

After returning:

- Nicht nur ein `return` Statement
- wenn # ≤ 1:

  an dem Rücksprung Punkt einweben
- sonst:

  neue `return` Anweisung mit Aufruf des Advices

  `GOTO` Statements anstatt aller anderen `returns`

After throwing:

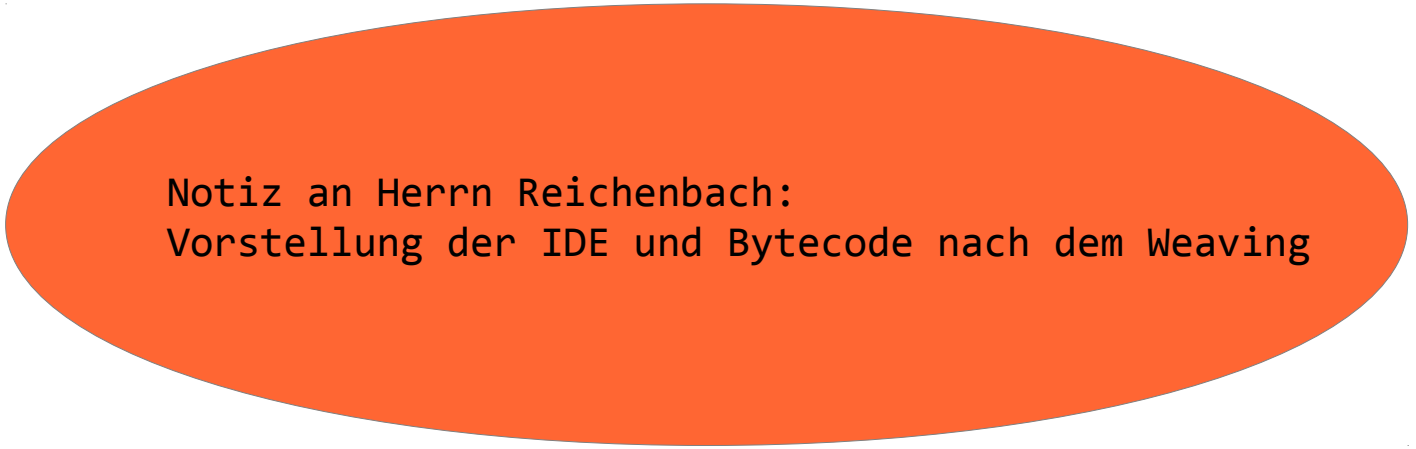
- Erstellen eines `try-catch`-Blocks
- Passend zu der definierten Exception
- Aufruf des Advices im `catch`-Block

Around:

- `AroundClosure` Objekt für `proceed()`
- Erwartet freie Variablen des Blocks und Parameter Liste des Advices
- `proceed()` führt die `run`-Methode aus
- Wird proceed nicht verwendet wird Join Point komplett ersetzt

Live Demo

(Wenn noch Zeit ist...)



Weitere Anwendungen:

- Patchen von kompilierten Anwendungen
- Spring Roo

Fragen

Vielen Dank für die

Aufmerksamkeit !!!!