



Aspekt-orientierte Programmierung

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken





Separation der Interessen

Die Separation der Interessen (*separation of concerns*) ist ein Grundprinzip der Softwaretechnik.

Seit Beginn der Programmierung wurden Konzepte entwickelt, um Teile des Systems *isoliert* betrachten zu können.

Separation der Interessen: der *heilige Gral* der Softwaretechnik





Separation der Interessen (2)

Die Konzepte zur Separation der Interessen definieren ganze *Paradigmen* der Programmierung:

Unterprogramme und Funktionen in der imperativen Programmierung:
Jede Funktion realisiert eine bestimmte – Funktion.





Separation der Interessen (2) _____

Die Konzepte zur Separation der Interessen definieren ganze *Paradigmen* der Programmierung:

Unterprogramme und Funktionen in der imperativen Programmierung:

Jede Funktion realisiert eine bestimmte – Funktion.

Module und abstrakte Datentypen in der modularen Programmierung:

Ein Modul faßt Daten und Zugriffsfunktionen zusammen





Separation der Interessen (2)

Die Konzepte zur Separation der Interessen definieren ganze *Paradigmen* der Programmierung:

Unterprogramme und Funktionen in der imperativen Programmierung:

Jede Funktion realisiert eine bestimmte – Funktion.

Module und abstrakte Datentypen in der modularen Programmierung:

Ein Modul faßt Daten und Zugriffsfunktionen zusammen

Objekte und Klassen in der objektorientierten Programmierung

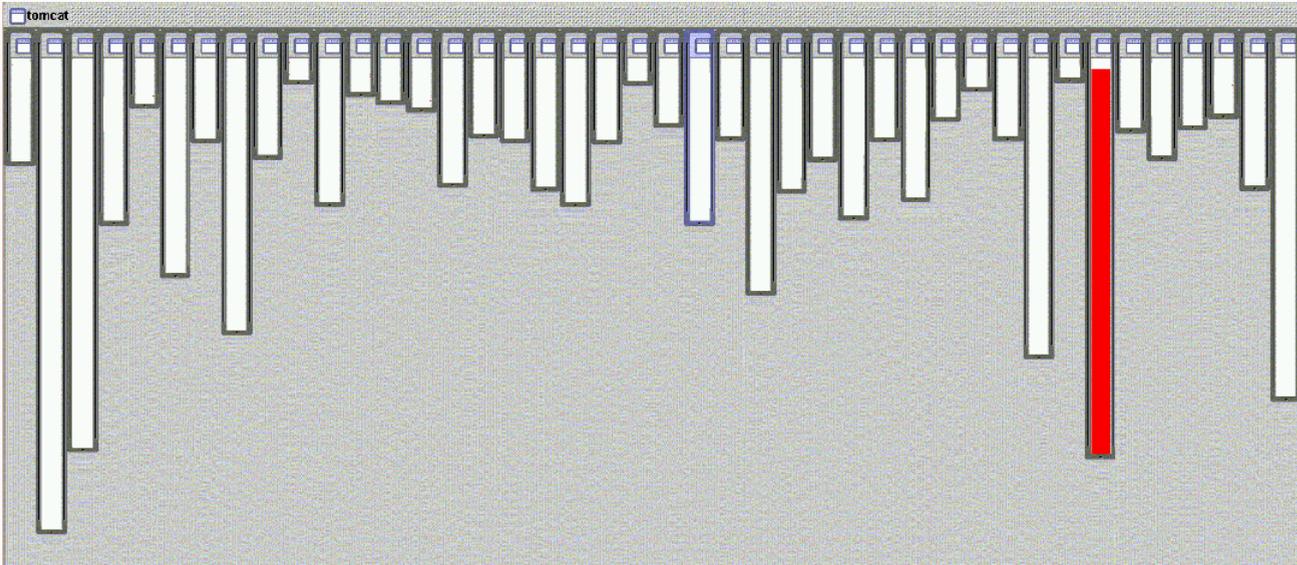
Oberklassen fassen Gemeinsamkeiten zusammen





Aspekte in Apache

Wir betrachten den Quellcode des *Apache*-Servers:



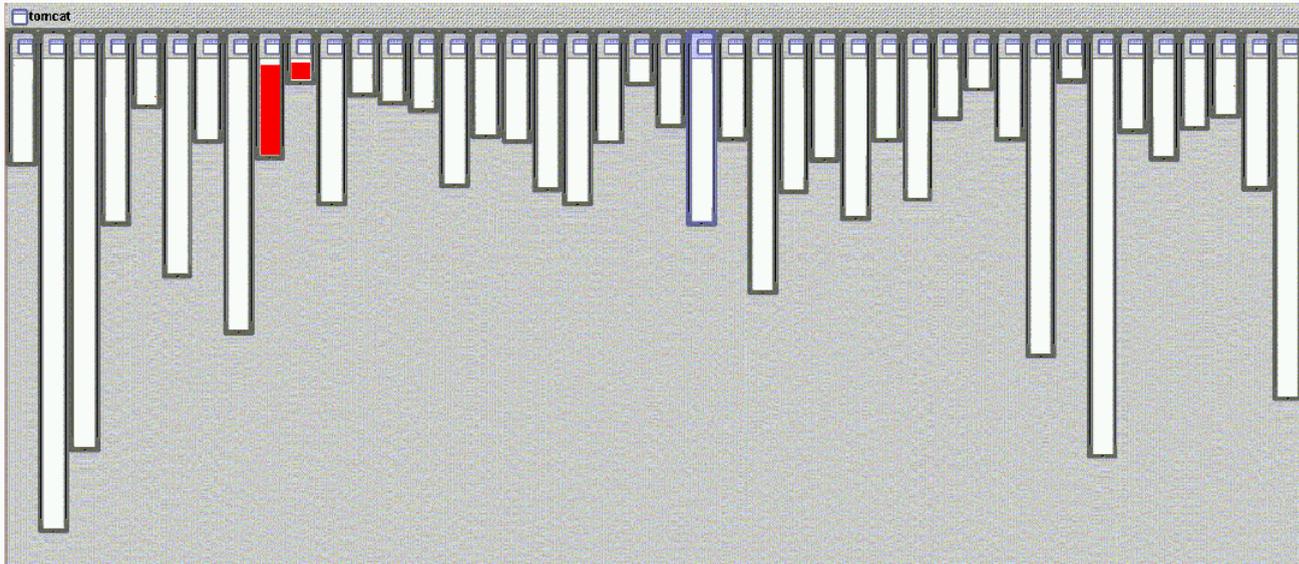
Der Code zum XML-Parsen steckt in einer eigenen Klasse (rot)





Aspekte in Apache (2)

Auch der Code zum Matchen von URLs steckt in zwei Klassen (Ober- und Unterklasse):



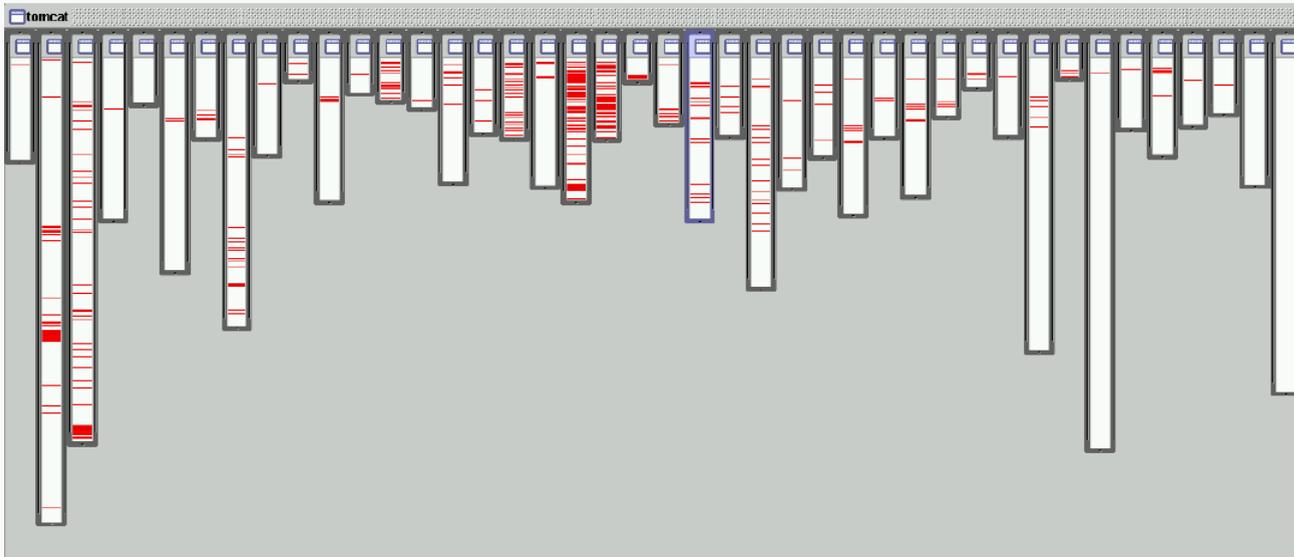
Gute Separation der Interessen!





Aspekte in Apache (3)

Log-Meldungen aber sind über den gesamten Code verteilt:



Frage: Kann man das Logging besser lokalisieren?





Aspekte in Apache (4)

In den Apache-Methoden sind *mehrere Aspekte* verwoben:

- die *eigentliche Funktionalität* wie XML- oder URL-Parsen (nach denen sie Klassen zugeordnet sind)
- das *Logging*, das den Programmablauf dokumentiert

In Apache sind die Methoden den Klassen *gemäß der eigentlichen Funktionalität* zugeordnet

⇒ zufriedenstellende Struktur, „dominante Dekomposition“





Aspekte in Apache (4)

In den Apache-Methoden sind *mehrere Aspekte* verwoben:

- die *eigentliche Funktionalität* wie XML- oder URL-Parsen (nach denen sie Klassen zugeordnet sind)
- das *Logging*, das den Programmablauf dokumentiert

In Apache sind die Methoden den Klassen *gemäß der eigentlichen Funktionalität* zugeordnet

⇒ zufriedenstellende Struktur, „dominante Dekomposition“

Würde man alternativ eine Klasse mit der Zuständigkeit „Logging“ einführen, müßten alle Methoden, die Log-Ausgaben produzieren, dieser Klasse zugeordnet werden

⇒ monolithische Struktur!





Aspekte in Apache (5)

Die verwobenen Aspekte der Apache-Methoden machen jedoch Probleme.

Beispiel: *Wir führen ein neues Log-Format ein.*

Folge: Wir müssen *alle Methoden ändern*, in denen Logging praktiziert wird.

Das Logging läßt sich nicht kapseln:

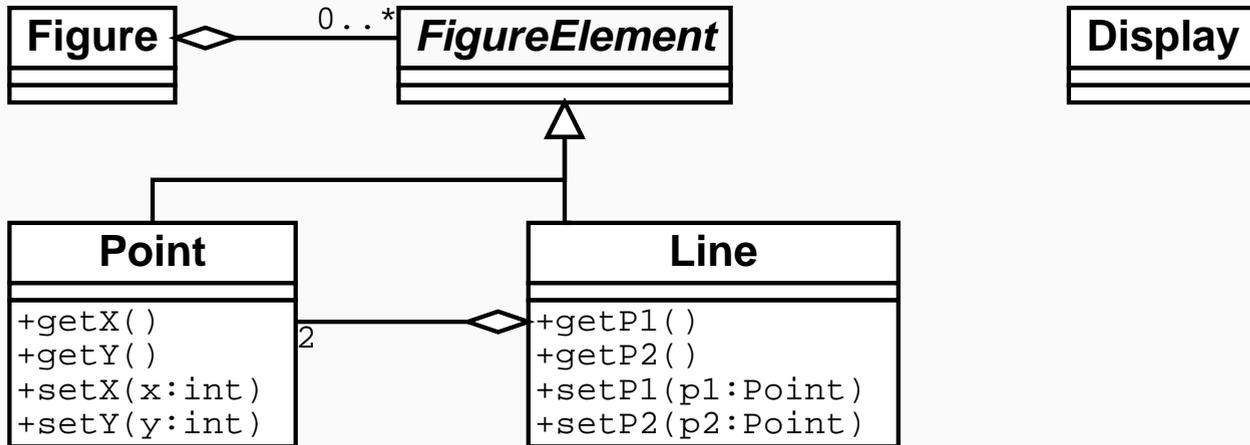
Tyrannie der dominanten Dekomposition!





Aspekte beim Figurenzeichnen

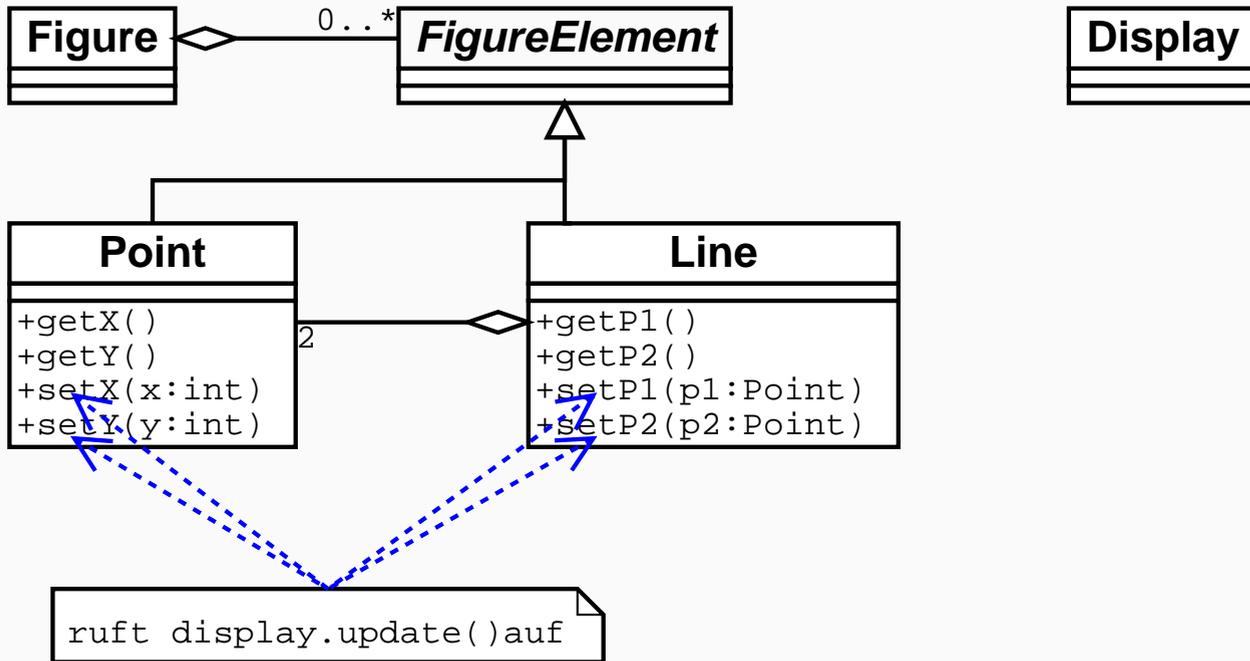
Beispiel: Einfache Figurenbibliothek





Aspekte beim Figurenzeichnen (2)

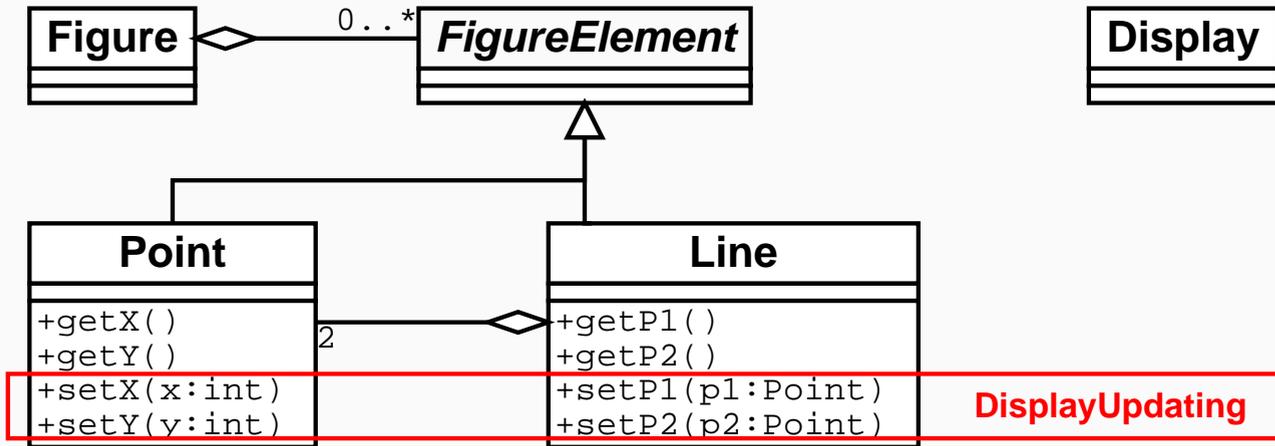
Jede set-Methode ruft `display.update()` auf:





Aspekte beim Figurenzeichnen (3)

Auch dies ist ein verwobener Aspekt:



Die set-Methoden

- ändern die jeweiligen Attribute (1. Aspekt) und
- aktualisieren die Anzeige (2. Aspekt)





Verwobene Aspekte

Generell machen verwobene Aspekte folgende Probleme:

Redundanter Code. Gleiche oder ähnliche Code-Fragmente finden sich an vielen Stellen im Programmtext

Schwer verständlicher Code. Aspekte wie „Logging“ oder „DisplayUpdating“ sind nicht als eigene syntaktische Einheit realisiert

Schwer änderbarer Code. Bei Änderungen müssen alle Code-Teile identifiziert werden (und konsistent und sicher geändert werden)





Aspekt-orientierte Programmierung

Modulübergreifende Sachverhalte („Aspekte“)

- treten in komplexen Systemen ständig auf
- haben eine klare Zuständigkeit
- haben eine klare Struktur:
 - bestimmte Menge von Methoden,
 - überschreiten Modulgrenzen,
 - werden an bestimmten Stellen benutzt

Also schaffen wir entsprechende *syntaktische Strukturen!*





Aspekt-orientierte Programmierung

Aspekt-orientierte Programmierung (AOP)

- führt Aspekte als *eigene syntaktische Strukturen* ein
- erhöht die Modularität von (OO-)Programmen in Bezug auf Aspekte.





Wozu ein neuer Ansatz?

Braucht man aspekt-orientierte Programmierung?

— *Nein, alles ist mit Objekten beschreibbar!*

Braucht man objekt-orientierte Programmierung?

— *Nein, alles ist mit Funktionen modellierbar!*

Braucht man Funktionen?

— *Nein, alles ist mit Sprüngen modellierbar!*

etc. etc.





Aspekte in AspectJ

AspectJ ist eine Erweiterung von Java (von XEROX PARC)

Grundideen:

- Aspekte (*aspects*) fassen Code (*advices*) zusammen, der an bestimmten Stellen im Programm (*join points*, *point cuts*) ausgeführt wird.
- Aspekte werden mit dem restlichen Code zu einem Java-Programm *verwoben*





Join Points

Ein *Join Point* ist ein wohldefinierter Punkt im Programmfluß.

AspectJ unterstützt zahlreiche Arten von Join Points. Wir betrachten zunächst nur einen, den *Aufruf einer Methode* (`call`):

Der Join Point `call(void Point.setX(int))` definiert den Aufruf der Methode `void Point.setX(int)`.

Weitere Arten: `get`, `set`, `within`, `cflow`...





Pointcuts

Ein *Pointcut* besteht aus bestimmten *Join Points* (und ggf. Werten an diesen join points).

In Pointcuts werden mehrere Join Points durch boolesche Operationen zusammengefaßt.

Beispiel - Der Pointcut `setter` faßt die Join Points bei `setX` und `setY` zusammen.

```
pointcut setter(): (call(void Point.setX(int)) ||  
                  call(void Point.setY(int)));
```





Pointcuts (2)

Der Pointcut move faßt alle Methodenaufrufe zusammen, die die Position eines Punktes oder einer Linie ändern:

```
pointcut move(): call(void Point.setX(int))    ||  
                  call(void Point.setY(int))    ||  
                  call(void Line.setP1(Point))  ||  
                  call(void Line.setP2(Point));
```





Advices

Ein *Advice* ist Code, der ausgeführt wird, wenn ein *Pointcut* erreicht wird.

Ein *After Advice* („after“) wird *nach* dem Methodenaufruf des Join Points ausgeführt.

Beispiel – nach jedem Ändern einer Position einen Text ausgeben:

```
after(): move() {  
    System.out.println("A figure element moved.");  
}
```

Weitere Arten: before, around





Advices (2)

Advices können auf den *Kontext* der Join Points zurückgreifen.

Der Pointcut `setP` merkt sich, welche Linie und welcher Punkt im Aufruf von `void a_line.setP*(p)` betroffen sind:

```
pointcut setP(Line a_line, Point p):  
    call(void a_line.setP*(p));
```

```
after(Line a_line, Point p): setP(a_line, p) {  
    System.out.println(  
        a_line + " moved to " + p + ".");  
}
```





Aspects

Ein *Aspect* faßt Advices zu einer syntaktischen Einheit zusammen:

```
aspect DisplayUpdating {
    pointcut move(): call(void Point.setX(int)) ||
                   call(void Point.setY(int)) ||
                   call(void Line.setP1(Point)) ||
                   call(void Line.setP2(Point));

    after(): move() {
        Display.update();
    }
}
```





Aspects (2)

Wir benutzen den *Before Advice* („before“), der *vor* dem Methodenaufruf des Join Points ausgeführt wird.

```
aspect CheckPointsRange {
    pointcut set_x(int x): call(void Point.setX(x));

    before(): set_x(p, new_x) {
        if (x < MIN_X || x > MAX_X)
            throw new IllegalArgumentException(
                "x is out of bounds.");
    }

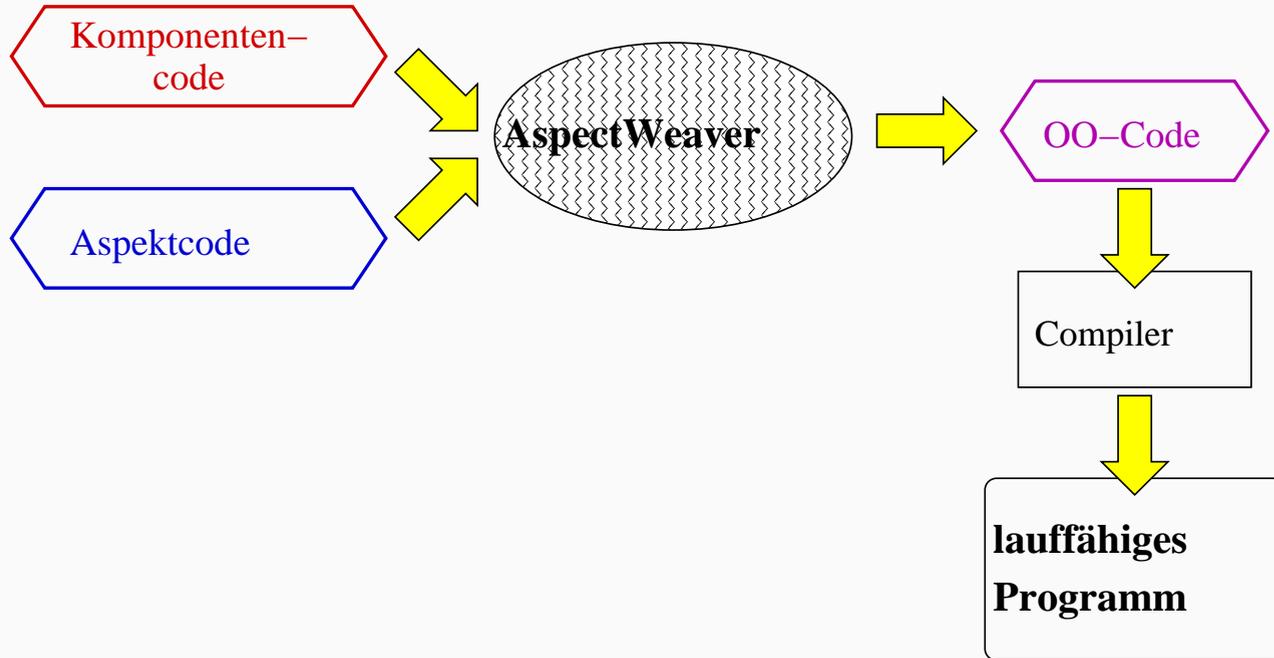
    // set_y analog
}
```





Aspect Weaver

Der *Aspect Weaver* faßt Aspekte und Code zusammen:





Anwendungen von Aspekten

Wir betrachten einige typische Anwendungen von Aspekten:

Entwicklung

- Logging
- Profiling

Produktion

- Änderungen verfolgen
- Konsistentes Verhalten





Logging

Wir definieren einen einfachen Aspekt zum Verfolgen von Aufrufen:

```
aspect SimpleTracing {
    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}
```

thisJoinPoint enthält den aktuellen Join Point





Profiling

Wir definieren einen Aspekt zum Zählen von Aufrufen:

```
aspect SetsInRotateCounting {
    int rotateCount = 0; // aspects are singletons
    int setCount = 0;

    before(): call(void Line.rotate(double)) {
        rotateCount++;
    }

    before(): call(void Point.set*(int)) &&
                cflow(call(void Line.rotate(double))) {
        setCount++;
    }
}
```

`cflow(x)`: Kontrollfluß des Join Points `x`





Aspekte in der Entwicklung

AOP bietet Vorteile in der Entwicklung:

- Diagnose-Code ist leicht definierbar
- Diagnose-Code ist in Aspekt eingekapselt
- Diagnose-Code ist leicht kombinierbar und abschaltbar





Änderungen verfolgen

MoveTracking merkt sich, ob ein Objekt bewegt wurde:

```
aspect MoveTracking {
    private static boolean dirty = false;

    public static boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;
    }

    pointcut move(): /* wie gesehen */;

    after() returning: move() { dirty = true; }
}
```





Konsistentes Verhalten

Wir möchten alle Fehler in Methoden `com.xerox.*` protokollieren:

```
aspect PublicErrorLogging {
    Log log = new Log();

    pointcut publicMethodCall():
        call(public * com.xerox.*.*(..));

    after() throwing (Error e):
        publicMethodCall() { log.write(e); }
}
```





Aspekte in der Produktion

Aspekte bieten Vorteile in der Produktion:

- Die Struktur der übergreifenden Zuständigkeit ist klar sichtbar
- Einfachere Evolution (z.B. MoveTracking merkt sich, welche Objekte bewegt wurden)
- Optionale Funktionalität (die in Aspekten gekapselt ist) und freie Konfigurierbarkeit
- Sicherere Implementierung (z.B. auch nach Hinzufügen einer Line-Unterklasse wird `Display.update()` aufgerufen)





Introduction

Mit AspectJ ist es auch möglich, bestehende Klassen zu verändern und zu erweitern (*introduction*):

```
aspect PointName {  
    public String Point.name;  
    public void Point.setName(String name) {  
        this.name = name;  
    }  
}
```

fügt der Klasse Point ein neues Attribut name und eine Methode setName hinzu.





Introduction (2)

Introduction geht auch für mehrere Klassen gleichzeitig.

Beispiel – wir erlauben das Klonen von Figuren:

```
aspect CloneableFigures {  
  
    declare parents: (Point || Line || Square)  
        implements Cloneable;  
  
    public Object (Point || Line || Square).clone()  
        throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```





Lohnt sich AOP?

Experiment (von Kiczales et al.): Überarbeitung eines Bildverarbeitungssystems

- Einsatz von Aspekten in der nichtoptimierten Variante, um Optimierungen einzufügen: Effizienzsteigerung um Faktor 100 (!)
- Überarbeitung der handoptimierten Variante: nur noch 1039 statt 35213 Zeilen





Lohnt sich AOP? (2)

Experiment (von Murphy et al.): Zwei Gruppen – eine AspectJ, eine Java

- Debugging: AspectJ-Gruppe schneller im Finden und Beheben von Fehlern
- Änderung eines existierenden Systems: Kein Unterschied





Zusammenfassung

- *Aspekt-Orientierte Programmierung* (AOP) führt mit *Aspekten* einen neuen Modularisierungsbegriff ein
- Aspekte ermöglichen es, existierenden Systemen nach Belieben Code hinzuzufügen, um das System *um genau definierte Funktionalitäten* zu erweitern
- Aspekte brechen die „Tyrannei der dominanten Dekomposition“
- Aspekte sind leicht wiederverwendbar und wartbar
- Aspekte sind (derzeit) rein syntaktische Erweiterungen
- Es gibt noch wenig Erfahrungen mit AOP.





Literatur

<http://aosd.net/> – Alles über aspekt-orientierte Software-Entwicklung

<http://aspectj.org/> – Alles über AspectJ

<http://www.research.ibm.com/hyperspace/> – Hyper/J, ein alternativer Ansatz („subjekt-orientierte Programmierung“)

Communications of the ACM, Oct. 2001 – Themenheft
„Aspektorientierte Programmierung“

Zugang über ACM Digital Library (via [Informatikbibliothek](#))

