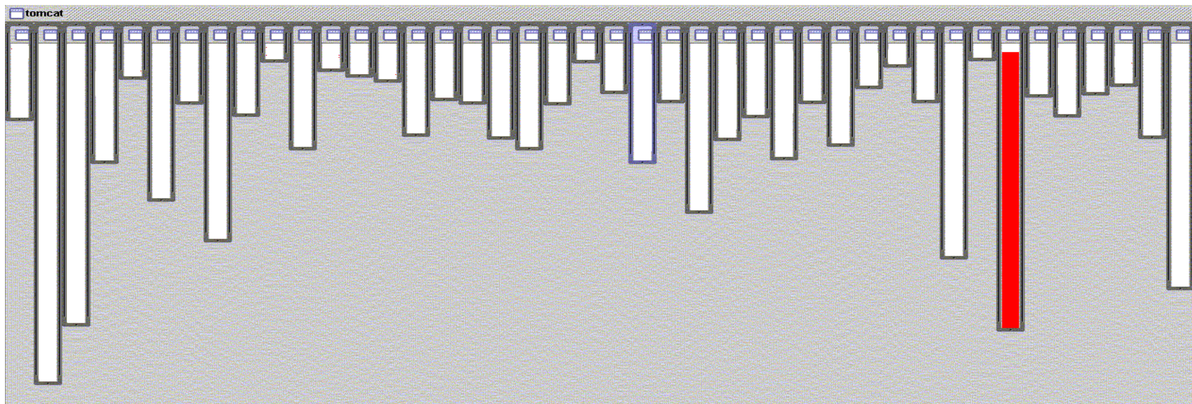
Kapitel 13

Aspekt Orientierte Programmierung

13.1 Aspekte in Apache

Wir betrachten den Quellcode des Apache-Servers:

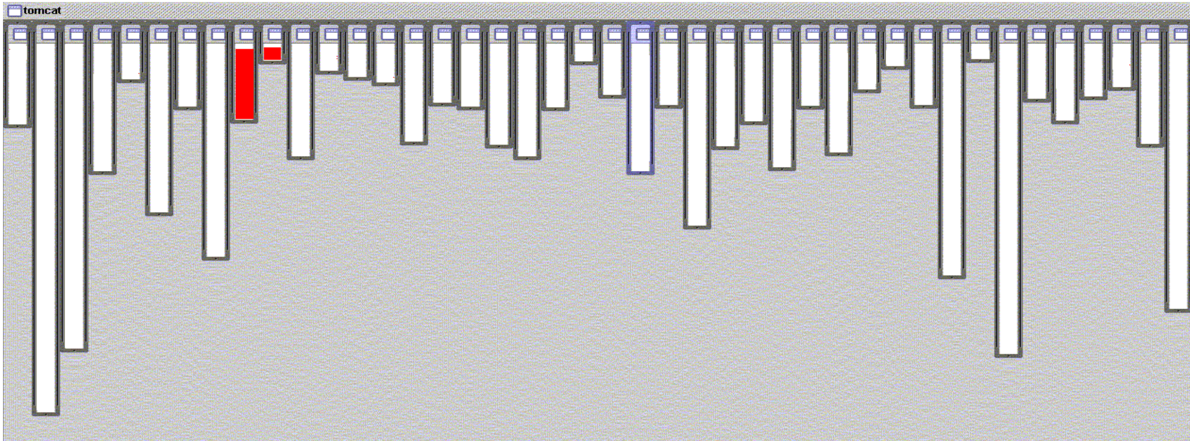


Der Code zum XML-Parsen steckt in einer eigenen Klasse (rot)

Quelle: aspectj.org

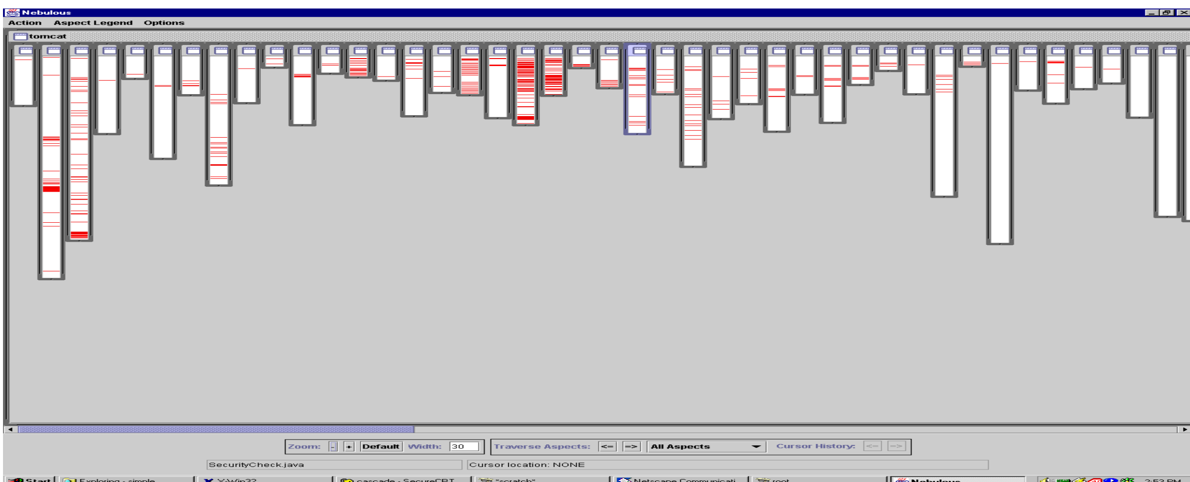
[0]ausgearbeitet von Andreas Zeller,
Lehrstuhl Softwaretechnik,
Universität des Saarlandes, Saarbrücken

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



Gute Seperation der Interessen!

Log-Meldungen aber sind über den gesamten Code verteilt:



Frage: Kann man das Logging besser lokalisieren?

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!

Die verwobenen Aspekte der Apache-Methoden machen jedoch Probleme.

Beispiel: Wir führen ein neues Log-Format ein.

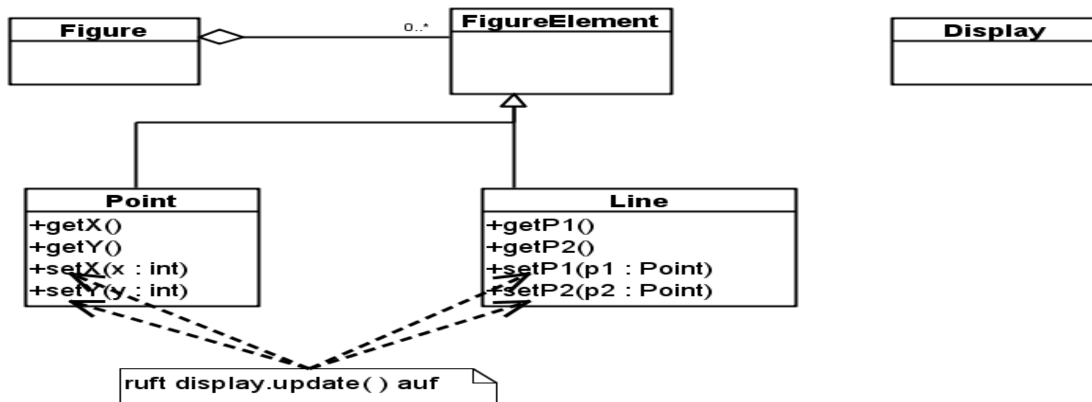
Folge: Wir müssen alle Methoden ändern, in denen Logging praktiziert wird.

Das Logging lässt sich nicht kapseln:

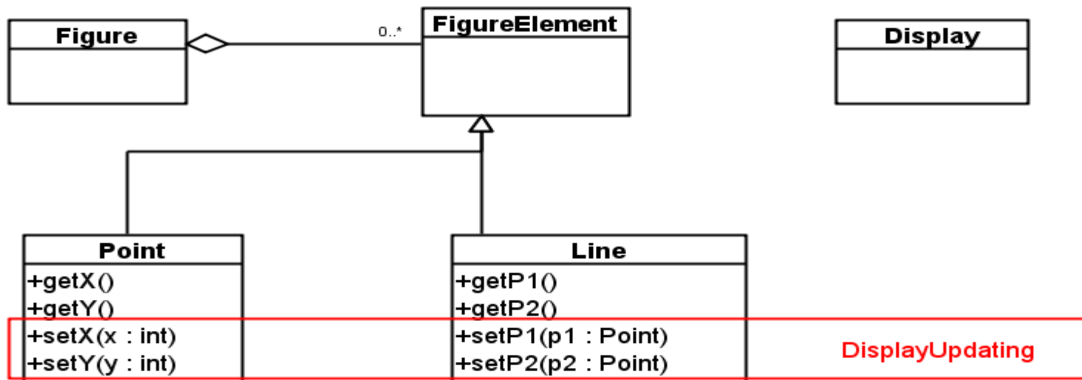
Tyrannei der dominanten Dekomposition!

13.2 Beispiel: Aspekte beim Figurenzeichnen

Beispiel: Einfache Figurenbibliothek
Jede set-Methode ruft display.update() auf:



Auch dies ist ein verwobener Aspekt:



Die set-Methoden

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

13.3 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 (AOP)

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

13.4 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, clfow...

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)));
```

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 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 Line.setP*(Point)) && args(a_line,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(); }
}
```

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(int)) &&
   args(x);

  before(): set_x(x) {
    if (x < MIN_X || x > MAX_X)
      throw new IllegalArgumentException(
        "x is out of bounds."); }
  // set_y analog
}
```

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 geht auch für mehrere Klassen gleichzeitig.

Beispiel - wir erlauben das Klonen von Figuren:

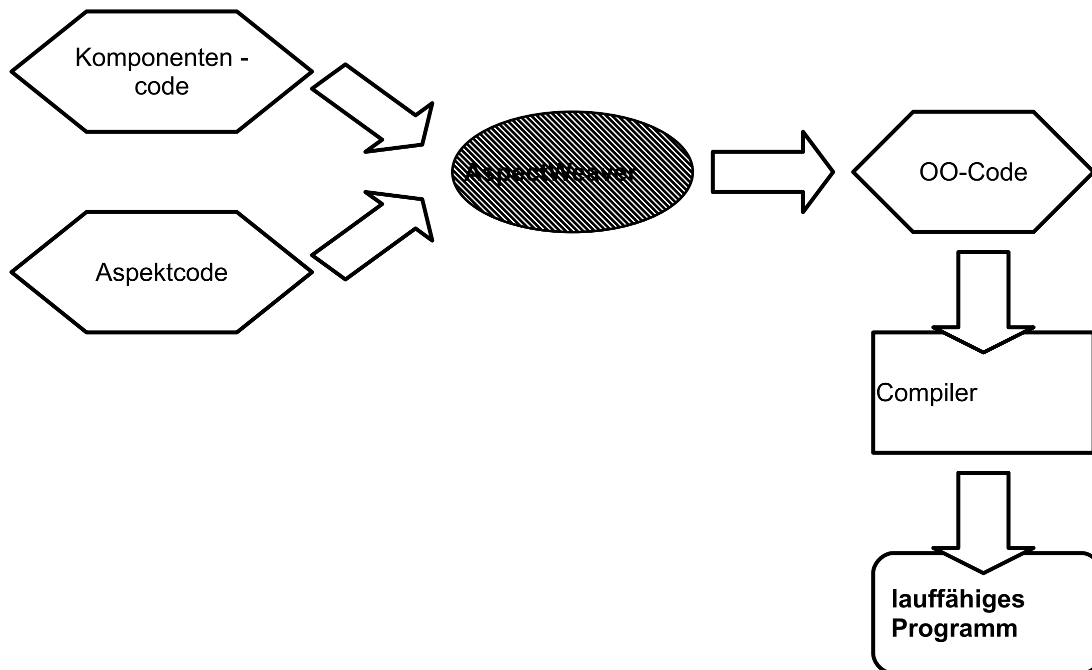
```
aspect Cloneable Figures {

  declare parents: (Point || Line || Square)
    implements Cloneable;

  public Object (Point || Line || Square).clone()
    throws CloneNotSupportedException {
      return super.clone();
  }
}
```

13.5 Aspect Weaver

Der *Aspect Weaver* fasst Aspekte und Code zusammen:



13.6 Typische Beispiele für Aspekte

Im Folgenden betrachten wir ein paar typische Beispiele für die Anwendung von Aspekten

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

Änderungen verfolgen

Move Tracking 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); }
```

13.7 Zusammenfassung

- Aspekt-Orientierte Programmierung (AOP) führt mit Aspekten einen neuen Modularisierungsbegriff ein
- Aspekte ermögichen 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.