04-2015 | Winter | www.ijug.eu

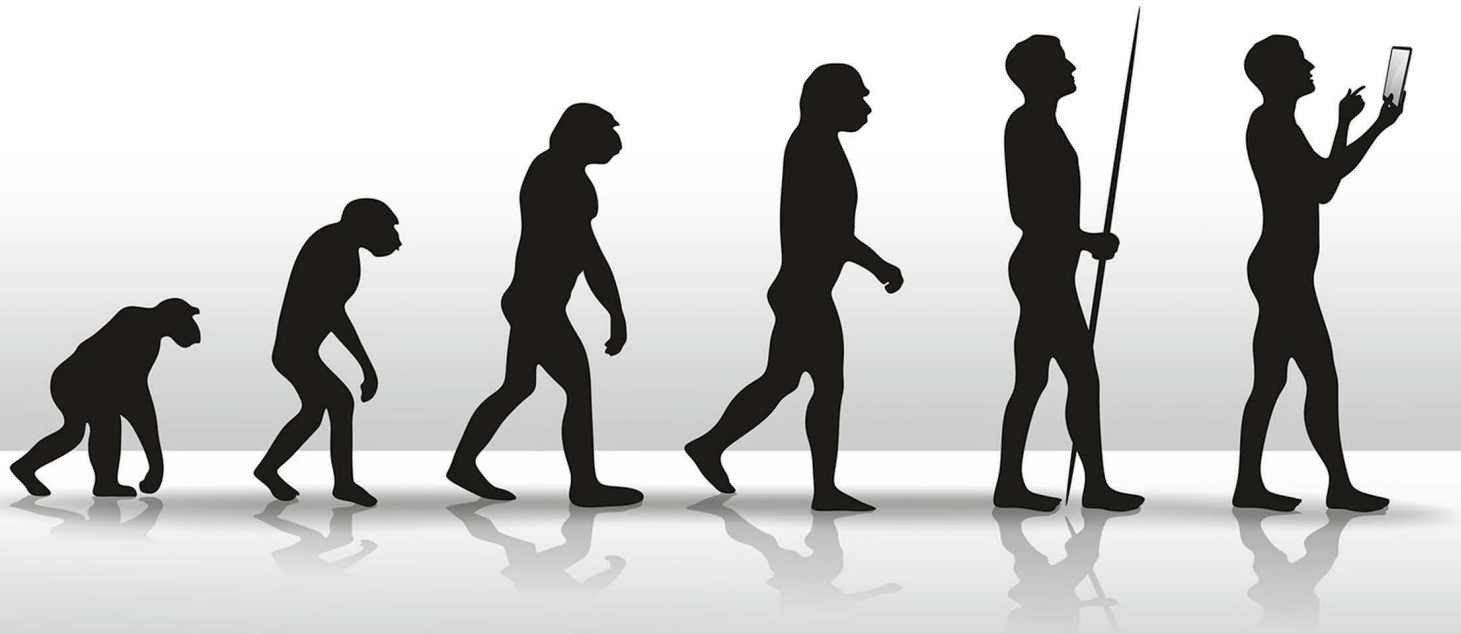
Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Java entwickelt sich weiter



D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Neue Frameworks
AspectJ, Eclipse Scout, Citrus

Raspberry Pi
Projekte mit Java

Java-Performance
Durch Parallelität verbessern

Web-Anwendungen
Hochverfügbar und performant

iJUG Verbund

4 191978 304903 04

Die Position „Software-Architekt" ist in der Software-Branche etabliert

Java-Champion Kirk Pepperdine gibt Performance-Tipps

3 Editorial

5 Das Java-Tagebuch
Andreas Badelt

7 Wo steht CDI 2.0?
Thorben Jannsen und Anatole Tresch

10 Der Software-Architekt in der heutigen Software-Entwicklung
Tobias Biermann

14 Hochverfügbare, performante und skalierbare Web-Anwendungen
Daniel Schulz

20 Software-Archäologie mit AspectJ
Oliver Böhm

25 Code-Review mit Gerrit, Git und Jenkins in der Praxis
Andreas Günzel

29 Kreative Signalgeber für Entwickler
Nicolas Byl

31 Java-Engines für die Labordaten-Konsolidierung
Matthias Faix

33 Performance durch Parallelität verbessern
Kirk Pepperdine

36 Kaffee und Kuchen: Projekte mit Java Embedded 8 auf dem Raspberry Pi
Jens Deter

39 MySQL und Java für die Regelung von Asynchronmaschinen
Eric Aristhide Nyamsi

43 Bean Testing mit CDI: Schnelles und produktives Testen von komplexen Java-EE-Anwendungen
Carlos Barragan

47 Vom proprietären Framework zum Open-Source-Projekt: Eclipse Scout
Matthias Zimmermann

50 Sofortkopien – minutenschnell in Selbstbedienung
Karsten Stöhr

53 Alles klar? Von wegen!
Von kleinen Zahlen, der Wahrnehmung von Risiken und der Angst vor Verlusten
Dr. Karl Kollischan

56 APM – Agiles Projektmanagement
gelesen von Daniel Grycman

57 Automatisierte Integrationstests mit Citrus
Christoph Deppisch

61 „Kommunikation ist der wichtigste Faktor …"
Interview mit der Java User Group Hamburg

63 Unbekannte Kostbarkeiten des SDK Heute: String-Padding
Bernd Müller

64 Der Weg zum Java-Profi
gelesen von Oliver Hock

66 Die iJUG-Mitglieder auf einen Blick

66 Impressum

66 Inserentenverzeichnis

Citrus bietet komplexe Integrationstests mit mehreren Schnittstellen

Software-Archäologie mit AspectJ

Oliver Böhm, Java User Group Stuttgart

Neue Projekte auf der grünen Wiese sind selten. Viel öfter wird man mit Altlasten konfrontiert und steht vor der undankbaren Aufgabe, vergangene Architekturen wieder freizulegen und unbekannten Code zu deuten, um das Projekt wieder zu vergangenen Hochkulturen zurückzuführen. Dabei gibt es verschiedene Möglichkeiten, dem Code seine Geheimnisse zu entreißen. Ein vielversprechender Ansatz ist dabei der Einsatz von Aspektorientierung und AspectJ, um unbekannte Code-Stellen zu erschließen und das Programmverhalten zu ergründen.

Historisch gewachsene SW-Systeme können viel erzählen: von Übernahmen durch andere Firmen (erkennbar an verschiedenen Package-Namen), von verschiedenen Programmier-Epochen (Outsourcing) oder vom (Un-)Kenntnisstand der verschiedenen Architekten. Oft enthalten sie aber auch viele Geheimnisse (Warum wurden sie erschaffen? Was machen sie eigentlich?), die die Einarbeitung in solche Relikte vergangener Hochkulturen erschweren, auch weil viele Zeugen des Entwicklungsgeschehens nicht mehr auffindbar sind und das Wissen aus Zweit- oder gar Dritt-Quellen und anderen Überlieferungen abgeleitet werden muss *(siehe Abbildung 1)*.

Die klassische Herangehensweise

Meist ist die Dokumentation die erste Anlaufstelle, um sich mit einer Alt-Anwendung vertraut zu machen. Auch wenn sie oftmals von der Realität abweicht, gibt sie doch einen wertvollen Einblick und wichtige Informationen, die den Einstieg in die Thematik erleichtern.

Im Ideal-Fall ist eine aktuelle Dokumentation vorhanden, die diesen Namen auch verdient und verschiedene Sichten auf die Anwendung erlaubt. Vor allem Übersichts-Dokumente über die Architektur, Infrastruktur und Randbedingen sind hier für die Einarbeitung, aber auch als Nachschlagewerk sehr hilfreich.

Um festzustellen, inwieweit der Code tatsächlich mit der Dokumentation übereinstimmt, sind Testfälle (sofern vorhanden) von eminenter Bedeutung. Sie bilden den Ausgangspunkt, um verschiedene Programmteile zu erkunden und deren Verhalten zu studieren. Die häufigsten Schwierigkeiten bestehen meist darin, die Testfälle zum Laufen zu bringen.

Hat man die Tests am Laufen, kann man sich an Refactoring-Maßnahmen wagen mit dem Ziel, die Business-Logik stärker zum Vorschein zu bringen und die Lesbarkeit und Wartbarkeit zu erhöhen. Vor allem JEE-Anwendungen sind oftmals „over designed", was eine Einarbeitung meist erschwert. Mit den neuen Erkenntnissen sollte nochmals die Dokumentation begutachtet werden: Welche Dokumente müssen überarbeitet, welche können aussortiert oder/und in Form von neuen Testfällen ausgedrückt werden.

Der Normal-Fall

Die Realität sieht oft ganz anders aus: Die Dokumentation ist häufig Teil des Problems und nicht der Lösung. Ehemalige Entwickler sind samt Know-how im Bermuda-Dreieck des Outsourcings verschwunden und als Test-Umgebung dient das Produktiv-System. Das bedeutet dann, im Code zu graben und die Mosaikstücke der ursprünglichen Architektur richtig zu deuten. Alles, was dabei hilft, den Source-Code zu verstehen, hilft auch bei der Rekonstruktion des Gesamt-Systems.

Auch die wenigen Test-Ruinen, die man oft noch findet, sind von unschätzbarem Wert, geben sie doch einen Einblick in die dynamischen Zusammenhänge. Zusammen mit aspektorientierten Code-Injektionen kann man darauf aufbauend neue Erkenntnisse über den Zusammenhalt der Code-Bereiche gewinnen.

Abbildung 1: Unbekannter Code

Die Welt der aspektorientierten Programmierung

Als Einstieg in die Begriffs- und Gedankenwelt von aspektorientierter Programmierung (AOP) dient ein Beispiel aus dem Banken-Bereich *(siehe Listing 1)*.

Dies ist ein sehr einfaches Modell einer Konto-Klasse, bei dem die Business-Logik (Einzahlung, Auszahlung, Überweisung) noch sehr gut erkennbar ist. Eine Warnung vorneweg: Für Real-World-Implementierungen bitte nie „double" als Datentyp verwenden, sonst kann es zu bösen Überraschungen aufgrund von Rundungsfehlern kommen *[1]*.

Schleichende Code-Verschmutzung

„Alle Kontobewegungen müssen protokolliert werden." Diese gesetzliche Anforderung soll jetzt umgesetzt werden. Dazu wird die Konto-Klasse erweitert *(siehe Listing 2)*.

Ganz langsam tritt damit die eigentliche Business-Logik in den Hintergrund. Zudem warten noch jede Menge weitere Anforderungen (oft auch als „Concerns" bezeichnet), die technischer Natur sind und mit der eigentlichen Fachlichkeit nichts zu tun haben: Autorisierung, Sicherheit, GUI, Transaktionen etc. *(siehe Abbildung 2)*.

Während des Informatik-Studiums bekommt man mit Dijkstras „Separations of Concerns" die Empfehlung mit auf den Weg, jeden „Concern" in einem eigenen Modul oder einer eigenen Klasse zu kapseln, aber bereits dieses einfache Logging-Beispiel zeigt, dass das mit den Mitteln der Objekt-Orientierung gar nicht so einfach ist *[2]*.

Separation of Concerns

Ein Ausweg aus diesem Dilemma bietet die Aspektorientierung. Sie verfügt über Konzepte, um solche meist technischen Anforderungen (im AOP-Jargon als „Crosscutting Concerns" bezeichnet) in eigene „Aspekte" kapseln zu können *(siehe Abbildung 3)*. Ein Vertreter aspektorientierter Sprachen ist AspectJ *[3]*, die Java um einige Sprachmittel erweitert und damit das Herauslösen der Logging-Funktionalität aus der Konto-Klasse ermöglicht *(siehe Listing 3)*. Übersetzt bedeutet dieser Code: „Wenn sich der Kontostand ändert, gib eine Log-Meldung aus." Für das Verständnis des Codes sind einige Begrifflichkeiten aus der AOP-Welt von Vorteil, die im nächsten Abschnitt vorgestellt werden.

```
public class Konto {

  private double kontostand = 0.0;

  public double abfragen() {
     return kontostand;
  }

  public void einzahlen(double betrag) {
     kontostand = kontostand + betrag;
  }

  public void abheben(double betrag) {
     kontostand = kontostand - betrag;
  }

  public void ueberweisen(double betrag,
        Konto anderesKonto) {
     abheben(betrag);
     anderesKonto.einzahlen(betrag);
  }

}
```

Listing 1: Eine einfache Konto-Klasse

```
public class Konto {

  private static Logger log = Logger.getLogger(Konto.class);
  private double kontostand = 0.0;

  public double abfragen() {
     return kontostand;
  }

  public void einzahlen(double betrag) {
     kontostand = kontostand + betrag;
     log.info("neuer Kontostand: " + kontostand);
  }

  public void abheben(double betrag) {
     kontostand = kontostand - betrag;
  }

  public void ueberweisen(double betrag,
        Konto anderesKonto) {
     abheben(betrag);
     anderesKonto.einzahlen(betrag);
     log.info("neuer Kontostand: " + kontostand);
  }

}
```

Listing 2: Konto-Klasse mit Logging

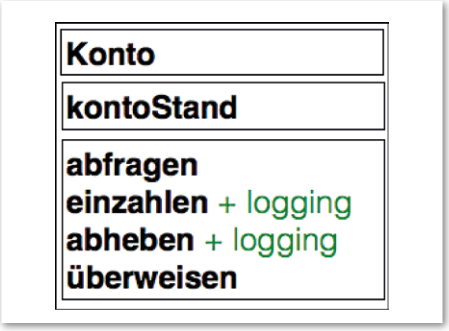


Abbildung 2: Separation of Concerns?

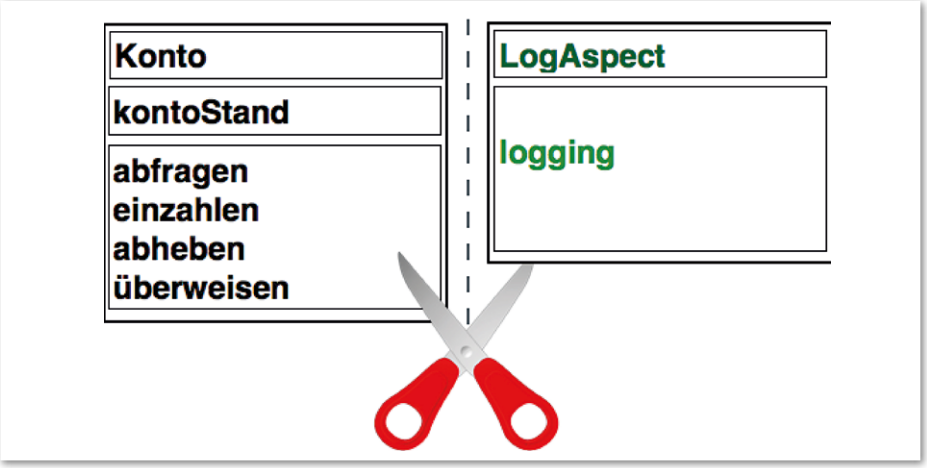


Abbildung 3: Separation of Concerns

Do you speak AOP?

Genauso wie SQL über eigene Begriffe wie „Relationen" und „Normalform" verfügt, grenzt sich die Aspektorientierung mit ihrer eigenen Begriffswelt von anderen Techniken ab. Die wichtigsten sind:

- Aspekt
- Joinpoints
- Pointcuts
- Advice

Der Aspekt ist für AOP das, was für OOP die Klasse bedeutet: der Container für die neuen Sprachmittel *(siehe Listing 4)*.

„Joinpoints" heißen im AOP-Jargon all die Punkte im Programm, auf die Einfluss genommen werden kann. Im Falle von AspectJ (und den meisten anderen AOP-Sprachen) sind dies:

- Aufruf („call") oder Ausführung („execution") einer Methode oder eines Konstruktors
- Initialisierung einer Klasse oder eines Objekts („initialization", „preinitialization", „staticinitialization")
- Zugriff auf eine Instanz-Variable („set", „get")
- Exception-Handling („handler")

Pointcuts adressieren dann diese Joinpoints *(siehe Listing 5)*.

Der ersten Pointcut („setKontostand") ist schon aus dem Beispiel des LogAspect in *Listing 3* bekannt. Er adressiert den schreibenden Zugriff auf das „kontostand"-Attribut. Interessanter ist der zweite Pointcut („callBankMethods"), der alle Methoden der verschiedenen Konto-Klassen anspricht (genauer: jene, die mit „...Konto" im Namen aufhören und im „bank"-Paket liegen) und beliebig viele Argumente hat. Möglich wird dies durch die Unterstützung verschiedener Wildcards („*" und „.."), die der Schlüssel für das Herausziehen der Crosscutting Concerns sind.

Das Ganze lässt sich noch mit booleschen Operatoren verknüpfen und mit weiteren Bedingungen kombinieren, sodass sich damit (fast) alle erdenklichen Szenarien realisieren lassen. Seit AspectJ 5 können auch noch Annotations zur Selektion herangezogen werden, was für die Lesbarkeit und Wartbarkeit enorme Vorteile hat.

„Advices" heißen die Methoden der Aspektorientierung, die Punkte im Programm, die über die Pointcuts adressiert sind, um zusätzliche Funktionalitäten anreichern zu können (oder kurz: zu manipulieren). Dabei besteht die Wahl, wann der Advice gegenüber dem Pointcut ausgeführt wird *(siehe Listing 6)*.

- vor (Before-Advice)
- nach (After-Advice)
- anstatt (Around-Advice)

Hier wird nach dem Pointcut „setKontostand" die entsprechende Log-Meldung ausgegeben. Die Verknüpfung mit dem „args"-Schlüsselwort von AspectJ dient in diesem Beispiel nur dazu, auf das Argument (also das Attribut) des Pointcut zugreifen zu können. Alternativ erlaubt AspectJ auch den Zugriff auf den Kontext des Joinpoint.

Der Webe-Vorgang

Damit die Aspekte in das fertige Programm kommen, gibt es zwei Techniken:

- Compile-Time Weaving (CTW)
- Load-Time Weaving (LTW)

Bei Compile-Time Weaving (CTW) werden die Aspekte nach Java übersetzt und mithilfe des bestehenden Java-Compilers in

```
public aspect LogAspect {

  private static Logger log =
     Logger.getLogger(LogAspect.class);

  pointcut setKontostand() :
     set(double bank.Konto.kontostand);

  after(double neu) : setKontostand() && args(neu) {
     log.info("neuer Kontostand: " + neu);
  }

}
```

Listing 3: LogAspect

```
public aspect LogAspect {
  // ...
}
```

Listing 4: Leerer „LogAspect"

```
pointcut setKontostand() :
  set(double bank.Konto.kontostand);
pointcut callBankMethods() :
  call(* bank.*Konto.*(..));
```

Listing 5: Definition zweier Pointcuts

den bestehenden Java-Code „eingewebt". Dabei muss der Java-Code nicht unbedingt als Source-Code vorliegen, sondern der AspectJ-Compiler verträgt auch Byte-Code beziehungsweise „.jar"-Dateien als Input.

Das Load-Time Weaving (LTW) „webt" die Aspekte während des Ladens der Java-Klassen ein. Diese Technik wird zum Beispiel vom Spring-AOP-Framework verwendet, das intern auf AspectJ aufsetzt. Der Vorteil hierbei ist, dass man mit den Aspekten eine größere Reichweite hat (prinzipiell jede Klasse, die geladen wird). Als Nachteil erkauft man sich Syntax-Fehler, die erst zur Laufzeit bemerkt werden.

AspectJ beherrscht beides und das AJDT-Plug-in für Eclipse unterstützt neben der inkrementellen Kompilierung die Visualisierung von Pointcuts und das Debuggen von Advices.

Grabungstechniken mit AOP

Nach diesem Ausflug in die wunderbare Welt der Aspekte, die uns einen kleinen Einblick in die Denkweise ermöglicht, wenden wir uns wieder unserem Fundstück aus der Steinzeit des Java-Zeitalters zu. Das größte Manko bei Alt-Anwendungen (und nicht nur dort) sind die Testfälle. Meistens fehlen sie oder sind genauso veraltet wie die Dokumentation. Ist die Anwendung noch in Betrieb, kann man versuchen, daraus Testfälle für die weitere Entwicklung abzuleiten. Und genau hier kann AOP helfen, fehlende Log-Informationen zu ergänzen oder die Kommunikation mit der Umgebung aufzuzeichnen.

Die interessanten Stellen sind vor allem die Schnittstellen zur Außenwelt. Bei JEE-Applikationen sind dies meist andere Systeme wie Legacy-Anwendungen oder Datenbanken, die über das Netzwerk angebunden werden. Hier reicht es oft aus, die Anwendung ohne Netzwerkverbindung zu starten und zu beobachten, wo überall Exceptions auftreten *(siehe Listing 7)*.

Aus dieser Exception kann man erkennen, dass die Anwendung auf eine MySQL-Datenbank zugreift, aber kein Connection-Objekt vom „DriverManager" bekommen hat. Mit diesem Wissen kann man alle Connection-Aufrufe genauer unter die Lupe nehmen *(siehe Listing 8)*.

Mit diesem Advice wird am Ende jedes Connection-Aufrufs der Aufruf selbst mitsamt seinen Parametern (wie „SELECT"-Statements) und dem Rückgabe-Wert ausgegeben (über die „getAsString()"-Methode, hier nicht abgebildet). Ähnlich kann man bei Netz- beziehungsweise Socket-Verbindungen verfahren, indem man die Programm-Punkte (Joinpoints) erweitert, über die Daten ausgetauscht werden.

Zugriff von außen

Handelt es sich um eine interne Bibliothek oder ein Framework, die es zu betrachten gilt, sind vor allem die öffentlichen Schnittstellen von Interesse. Hier kann man mit aspektorientierten Sprachmitteln all die Methoden-Aufrufe ermitteln, die tatsächlich von außerhalb aufgerufen werden – denn oft sind nicht alle Methoden, die als „public" deklariert sind, für den Aufruf von außen vorgesehen *(siehe Listing 9)*.

Hier werden alle Methoden eines Bank-Frameworks („bank"-Package) überwacht. Nur wenn der Aufruf nicht von innerhalb kam („!cflowbelow"), wird vor der Ausführung der Methode oder des Konstruktors der Aufrufer anhand des Stack-Trace ermittelt (Methode „getCaller()", hier nicht aufgelistet, *siehe Listing 10)*. Dies ist die Ausgabe, die den Aufruf aus einer JSP zeigt. Lässt man die Anwendung lang genug laufen, erhält man so alle Methoden, die von außerhalb aufgerufen werden.

Daten aufnehmen

Über Serialisierung in Java ist es relativ einfach möglich, Objekte abzuspeichern und wieder einzulesen. Damit lässt sich ein

```
after(double neu) : setKontostand() && args(neu) {
   log.info("neuer Kontostand: " + neu);
}
```

Listing 6: After-Advice

```
java.net.SocketException: java.net.ConnectException: Connection refused
  at com.mysql.jdbc.StandardSocketFactory.connect(StandardSocketFactory.java:156)
  at com.mysql.jdbc.MysqlIO.<init>(MysqlIO.java:283)
  at com.mysql.jdbc.Connection.createNewIO(Connection.java:2541)
  at com.mysql.jdbc.Connection.<init>(Connection.java:1474)
  at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:264)
  at java.sql.DriverManager.getConnection(DriverManager.java:525)
  at java.sql.DriverManager.getConnection(DriverManager.java:193)
  at bank.Archiv.init(Archiv.java:25)
  …
```

Listing 7: Exceptions

```
after() returning(Object ret) :
     call(* java.sql.Connection.*(..)) {
  log.debug(getAsString(thisJoinPoint) + " = " + ret);
}
```

Listing 8: Pointcut und Advice für einen SQL-Aufruf

```
public pointcut executePublic() :
  (execution(public * bank..*.*(..))
     || execution(public bank..*.new(..)))
  && !within(EnvironmentAspect);

public pointcut executeFramework() :
  execution(* bank..*.*(..)) || execution(bank..*.new(..));

public pointcut calledFromOutside() :
  executePublic() && !cflowbelow(executeFramework());

before() : calledFromOutside() {
  Signature sig = thisJoinPoint.getSignature();
  String caller = getCaller(Thread.currentThread()
            .getStackTrace(), sig);
  log.info(caller + " calls " + sig);
}
```

Listing 9: Pointcuts und Advice für Aufrufe von außerhalb

```
...
jsp.index_jsp._jspService(index_jsp.java:54) calls bank.Konto(int)
jsp.index_jsp._jspService(index_jsp.java:58) calls void bank.Konto.einzahlen(double)
jsp.index_jsp._jspService(index_jsp.java:60) calls double bank.Konto.abfragen()
...
```

Listing 10: Stack-Trace

einfacher Objekt-Recorder bauen, um damit die Schnittstellen-Daten aufzunehmen *(siehe Listing 11)*. Hinter „thisJoinPoint" verbirgt sich der Kontext der Aufrufstelle, die der AspectJ-Compiler als Objekt bereitstellt.

Daten einspielen

Wenn man die notwendigen Schnittstellen-Daten gesammelt hat, kann man anhand dieser Daten einen (einfachen) Simulator bauen. Man kennt die Punkte, über die diese Daten hereingekommen sind, man muss jetzt lediglich an diesen Punkten die aufgezeichneten Daten wieder einspielen *(siehe Listing 12)*.

Hat man so die Anwendung von der Außenwelt isoliert und durch Daten aus früheren Läufen simuliert, kann man das dynamische Verhalten der Anwendung genauer untersuchen. Damit lassen sich weitere Aspekte hinzufügen, die automatisch Sequenz-Diagramme erzeugen oder wichtige Programmzweige visualisieren (etwa mithilfe der UMLGraph-Bibliothek *[4]* oder über die Webseite von websequence diagrams *[5]*). So erhält man neben der statischen Sicht durch Klassen-Diagramme, die sich notfalls mittels Reverse-Engineering wiedergewinnen lassen,auch eine Visualisierung des dynamischen Verhaltens.

Code-Änderungen

Nach diesen Vorbereitungen kann mit den ersten Code-Änderungen (Refactorings *[6]*) begonnen werden. Soll der Original-Code (noch) unverändert bleiben, was bei unbekannten Abdeckungen vorhandener Testfälle sinnvoll sein kann, liefert die Aspektorientierung die Möglichkeit, den Code getrennt vom Original abzulegen. Man kann sogar verschiedene Varianten einer Komponente vorhalten und diese während der Laufzeit austauschen. Auf diese Weise kann man experimentell verschiedene Varianten und Code-Manipulationen ausprobieren, um deren Verhalten auf die Gesamtanwendung studieren zu können.

```
after() returning(Object ret) : sqlCall() {
  objLogger.log(thisJoinPoint);
  objLogger.log(ret);
}
```

Listing 11: Aufnahmepunkte für den Daten-Recorder

Fazit

„Don't patch bad code – rewrite it" (aus: „The Elements of Programming" *[7]*). Der Aufwand, sich in unbekannten Code einzuarbeiten, wird oft unterschätzt. Langfristig ist es meist wirtschaftlicher, vorhandenen Code komplett neu zu entwickeln, wenn die Dokumentation veraltet, der Code an vielen Stellen ausgewuchert ist und auch die Testfälle nicht vorhanden oder von zweifelhafter Qualität sind. Oftmals hat man allerdings keine andere Wahl, als auf den bestehenden Code aufzusetzen, weil er die einzig gültige Quelle der Dokumentation darstellt. Und hier bietet die Aspektorientierung eine Vielzahl von Möglichkeiten:

- Zusätzliche Log-Möglichkeiten einbauen
- Sequenz-Diagramme generieren *[8]*
- Schnittstellen überwachen
- Aufrufe und Events aufzeichnen und simulieren
- Exceptions provozieren

Daraus lassen sich weitere Testfälle schreiben und neue Erkenntnisse gewinnen, um Refactoring-Maßnahmen einzuleiten oder sich zu einer Neuentwicklung zu entschließen. Allerdings entbindet auch AOP nicht von der Aufgabe, bestehenden und neuen Code so zu gestalten und umzuformen, dass künftige Generationen damit glücklich werden.

```
Object around() : sqlCall() {
  Object logged = logAnalyzer.
getReturnValue(thisJoinPoint);
  return logged;
}
```

Listing 12: Aufnahme einspielen

Weitere Informationen

[1] http://haupz.blogspot.de/2009/01/ich-bin-reich.html
[2] Edsger Wybe Dijkstra, A Discipline of Programming, Prentice Hall (Juni 1976), ISBN 978-0132158718
[3] Oliver Böhm, Aspektorientierte Programmierung mit AspectJ 5, dpunkt.verlag, 1st edition, 2005. ISBN-10 3-89864-330-15
[4] UMLGraph, http://umlgraph.org
[5] websequence diagrams, https://www.websequencediagrams.com/
[6] Martin Fowler. Refactoring. Addison-Wesley, 7th edition, 2001, ISBN 0-201-48567-2
[7] Kernighan and Plauger, The Elements of Programming, McGraw-Hill, 2nd edition, 1974, 1978. ISBN 0-07-034207-5
[8] Generation of Sequence Diagrams, https://sourceforge.net/p/patterntesting/wiki/Generation%20of%20Sequence%20Diagrams

Oliver Böhm
ob@jugs.org

Oliver Böhm beschäftigt sich mit Java-Entwicklung unter Linux und Aspekt-Orientierter SW-Entwicklung. Neben seiner hauptberuflichen Tätigkeit als JEE-Architekt bei T-Systems ist er Buchautor, Projektleiter bei PatternTesting und Board-Mitglied der Java User Group Stuttgart.