04-2014 | Winter | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Mit Lambda-Ausdrücken einfacher programmieren

Go for the Money
Währungen und Geldbeträgen in Java

Oracle-Produkte
Die Vorteile von Forms und Java vereint

Pimp my Jenkins
Noch mehr praktische Plug-ins

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

4 191978 304903 02

iJUG Verbund

bis 26.09.
Call for Papers
Jetzt bewerben!

JavaLand

24.–25. März 2015
im Phantasialand
Brühl bei Köln

Die Konferenz der Java-Community!

- Seien Sie mit dabei, wenn die Konferenz zum Zentrum der deutschen Java-Szene wird!
- Wissenstransfer, Networking und gute Gespräche treffen auf spannende Abenteuer, Spaß und Action.
- Vom Einsteiger bis zum Experten haben alle die Gelegenheit, zwei Tage im JVM-Kosmos zu leben.

Zwei Tage lang das JavaLand besiedeln

www.JavaLand.eu

Präsentiert von: DOAG Deutsche ORACLE-Anwendergruppe e.V. | Heise Zeitschriften Verlag

Community Partner: iJUG Verbund

Wolfgang Taschner
Chefredakteur Java aktuell

Aus der Community – für die Community

Vor knapp fünf Jahren, im Dezember 2009, haben sieben Anwendergruppen den Interessenverbund der Java User Groups e.V. (iJUG) gegründet. Ziel war die umfassende Vertretung der gemeinsamen Interessen der Java User Groups sowie der Java-Anwender im deutschsprachigen Raum, insbesondere gegenüber Entwicklern, Herstellern, Vertriebsunternehmen sowie der Öffentlichkeit. Mittlerweile sind 22 User Groups Mitglied im iJUG e.V. geworden. Die vollständige Liste und die Kontaktdaten finden Sie auf Seite 66.

Ein Highlight war in diesem Jahr die JavaLand 2014, die vom 25. bis 26. März 2014 im Phantasialand in Brühl stattfand. Unter dem Motto „Zwei Tage lang das JavaLand besiedeln" nahmen mehr als 800 Teilnehmer aus 18 Ländern an der neuen Java-Konferenz teil. Das innovative Konzept bot der Java-Community einen attraktiven Rahmen zum Lernen, Erfahren und Austauschen, der von den im iJUG vertretenen Java User Groups gemeinsam gestaltet wurde.

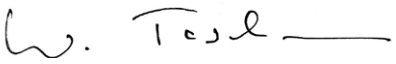
Ein weiteres Ziel des iJUG war die regelmäßige Herausgabe dieser Zeitschrift. Die Arbeit hat sich gelohnt: Die Java aktuell ist mittlerweile eine der größten Java-Zeitschriften im deutschsprachigen Raum. Alle drei Monate erscheint eine Ausgabe, gefüllt mit hervorragenden Artikeln.

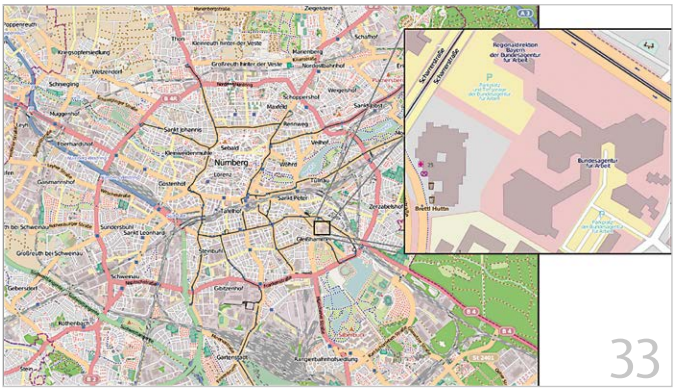
In dieser Ausgabe der Java aktuell können Sie zu den Artikeln Ihre Meinung abgeben, Fragen an den Autor stellen und uns Ergänzungen beziehungsweise Anregungen mitteilen. Scannen Sie dazu einfach den QR-Code am Ende des Artikels oder geben Sie die danebenstehende URL ein. Damit gelangen Sie auf die entsprechende Feedback-Seite.

Die erfolgreichen Aktivitäten des iJUG zeigen, dass die Java-Community zusammenwächst und sich weiterentwickelt. In diesem Sinne freue mich auf Ihr Feedback.

http://ja.ijug.eu/14/4/1

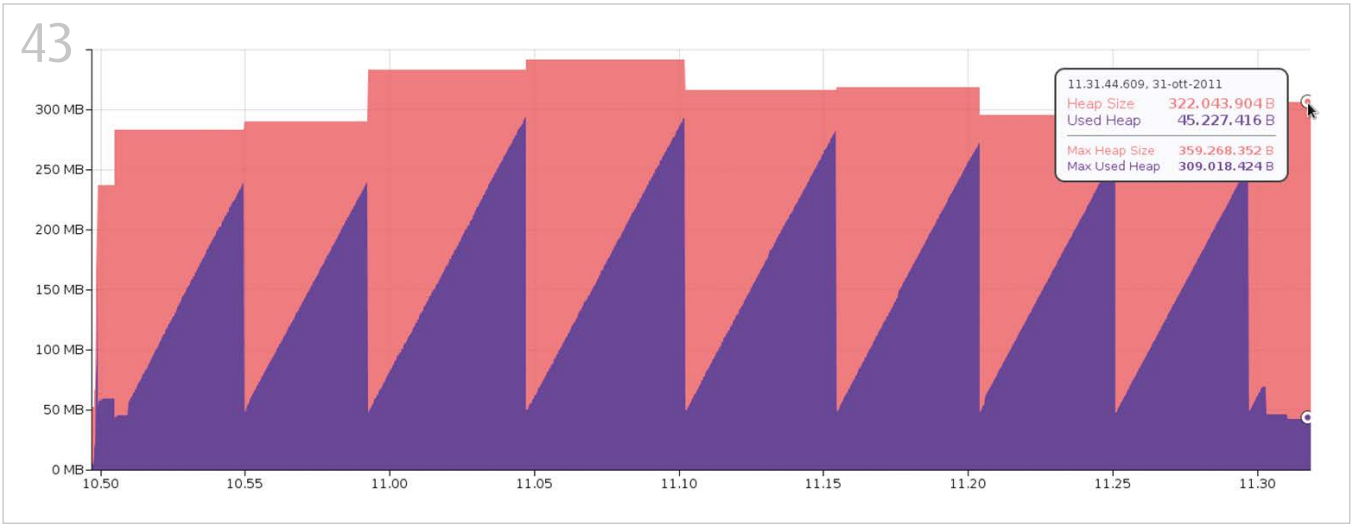
Ihr





OpenStreetMap ist vielen kommerziellen Diensten überlegen, Seite 33

Alles über Währungen und Geldbeträge in Java, Seite 20



WURFL ist verglichen mit selbst dem nicht reduzierten OpenDDR-Vokabular deutlich speicherhungriger, Seite 43

Das Java-Tagebuch

Andreas Badelt, Leiter der DOAG SIG Java

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im zweiten Quartal 2014.

28. April 2014

GlassFish 4 auf dem Raspberry Pi

Schon ein paar Monate alt, aber jetzt ausgegraben von Reza Rahman: Eine umfassende Anleitung zur Installation von GlassFish 4 auf einem Raspberry Pi. Dies ist ein weiterer Beweis, dass mit dem Übergang von J2EE zu Java EE einige Dinge verloren gegangen sind, darunter auch solche, die wir nicht vermissen werden, wie ein „big footprint".
https://blogs.oracle.com/theaquarium/entry/glassfish_4_on_the_raspberry

30. April 2014

Java Micro Edition 8 freigegeben

Das Java-ME-8-Release mit der Embedded Runtime ist offiziell freigegeben. Damit steht Entwicklern jetzt eine deutlich mächtigere Embedded-Plattform zur Verfügung. Insbesondere das Alignment zwischen SE und ME dürfte aber eine deutliche Erleichterung für diejenigen sein, die von SE umsteigen oder parallel für beide Plattformen entwickeln wollen. Ursprünglich sollte die Freigabe gemeinsam mit Java SE 8 und Java SE 8 Embedded erfolgen, was das immer weitere Verschmelzen der Plattformen betont hätte. Die kleine Verzögerung ist verzeihlich, Hauptsache die Qualität der Implementierung stimmt.
http://terrencebarr.wordpress.com/2014/04/30/java-me-8-released

2. Mai 2014

Hilfe für Applet-gestresste System-Administratoren

Trotz Sicherheitslücken und Sicherheitswahn sind Applets offensichtlich weiterhin verbreitet. Sonst wären Deployment Rule Sets vermutlich nie entstanden. Sie wurden auf Wunsch von Desktop-Administratoren im Zuge der neuen Sicherheitsmaßnahmen für Java SE in Release 7 Update 40 herausgegeben und ermöglichen eine feingranulare Kontrolle darüber, welche Applets mit welcher Java-Version ausgeführt werden und welche Default-Sicherheitseinstellungen gelten sollen. Sie sind beispielsweise notwendig, wenn ein firmeneigenes Applet eine ältere Java-Version benötigt, was normalerweise gar nicht möglich wäre. Im Blog der Java Platform Group wird am Beispiel erklärt, wie das geht.
https://blogs.oracle.com/java-platform-group/entry/deployment_rule_set_by_example

12. Mai 2014

HTTP PATCH mit JAX-RS 2

Wer kennt HTTP PATCH? Nein, es geht nicht um einen Nachtrag zum RFC 2616, sondern um die Request-Methode, die „partielle Updates" auf standardisierte Weise ermöglicht. Wie das Ganze mit JAX-RS 2 und Jersey funktioniert, erklärt ein Eintrag von Reza Rahman im GlassFish-Blog „Aquarium".
https://blogs.oracle.com/theaquarium/entry/using_http_patch_with_jax

14. Mai 2014

Von der Spring Pet Clinic zur Java EE 7 Pet Clinic

Bist du der Spring- oder der EE-Typ? Ein bisschen Futter für diejenigen, die sich noch nicht entschieden haben, oder (Ketzerei!) beides machen: Ein deutscher Entwickler hat die Beispiel-Applikation „Spring Pet Clinic" auf Java EE 7 und RichFaces migriert, wie im „Aquarium" zu lesen ist.
https://blogs.oracle.com/theaquarium/entry/migrating_the_spring_pet_clinic

21. Mai 2014

Compact Profiles für Java SE Embedded selbst erzeugen

Um die Größe (oder auch die Angriffsfläche) von Java SE for Embedded Runtimes anzupassen, gibt es die Compact Profiles. Mit Hilfe von „jrecreate" lassen sich diese auf den eigenen Bedarf zugeschnitten selbst erzeugen. Das Gegenstück „jdeps –profile" liefert zu einem existierenden „jar"-File die Abhängigkeiten und welches Profil benötigt wird. Eine ausführliche Anleitung gibt ein Blog-Eintrag der Java Platform Group.
https://blogs.oracle.com/java-platform-group/entry/compact_profiles_space_and_security

23. Mai 2014

Eine weitere Kooperation für den Embedded-Markt

Henrik Ståhl, oberster Produktmanager für die Java-Plattform bei Oracle, kündigt die nächste Kooperation mit einem für den Embedded-Markt wichtigen Hersteller an. Oracle will mit Imagination, dem neuen Eigner der MIPS-Technologie, an Implementierungen für diese Prozessor-Architektur arbeiten. Insbesondere Java SE Embedded 8 für die 32-Bit-Versionen wird im Fokus sein.
https://blogs.oracle.com/henrik/entry/oracle_announces_plans_for_java

23. Mai 2014

Quell-Code der „IoT Developer Challenge" auf „java.net"

Lust auf ein IoT-Projekt, aber ein paar Anregungen fehlen? Auf „java.net" ist jetzt Quellcode von einigen Projekten der „IoT Developer Challenge" zu finden – neben dem sonstigen umfangreichen Trainingsmaterial.
https://www.java.net/challenge/code-samples

28. Mai 2014

Server-Sent Events SE für Java EE 8

Java EE 8 soll neben dem existierenden WebSocket-Support auch Unterstützung für das uni-direktionale SSE (Server-Sent Events) bekommen. Noch eine Kommunikations-Technologie, die aber auch ihre Berechtigung hat und wie WebSockets Teil von HTML 5 ist. Die Frage scheint daher nur zu sein, auf welche Weise dies geschehen soll: Als Teil des Servlet-API, des WebSocket-API oder von JAX-RS? Oder eine komplett neues API nur für SSE, in einem eigenen JSR spezifiziert? Santiago Pericas-Geertsen, Spec-Lead von JAX-RS (JSR 339), hat sich die Optionen angesehen und empfiehlt eine Integration in JAX-RS. Aber die Diskussion ist sicher noch nicht beendet und die jeweiligen Spec-Leads bestimmt dankbar für fundierten Input aus der Community.
https://java.net/downloads/javaee-spec/SSE-in-EE8.pdf

2. Juni 2014

CDI 2 soll deutliche Neuerungen bringen

Context and Dependency Injection soll mit einer neuen Version 2 Bestandteil von Java EE 8 werden und dafür deutlich überarbeitet werden. Unter anderem wird über asynchrone Events und Methodenaufrufe, Class Hot Swapping und eine Modularisierung der Spezifikation nachgedacht. Auf der eigenen Webseite hat die Expert Group dazu eine Umfrage veröffentlicht, um die Präferenzen der Community zu erfahren. Falls das Formular zum Zeitpunkt der Veröffentlichung des Java-Tagebuchs nicht mehr freigeschaltet ist, sollten auf der Webseite zumindest die Ergebnisse stehen.
http://www.cdi-spec.org/news/2014/05/28/CDI-2_0-survey

3. Juni 2014

Henrik Ståhl: Oracle baut JDK 8 Port für 64-bit ARMv8

Oracle bastelt mit Unterstützung von Cavium an einem „Standard Oracle JDK 8 Port" für die neue Generation 64-bit ARMv8-basierter Server. Cavium ist führender Halbleiter-Hersteller und langjähriger Unterstützer des Java-Ökosystems. Geplante Fertigstellung ist laut Henrik Stahl „early 2015". Der Port soll auf der gebührenfreien Binary Code License basieren, aber eine Freigabe als Open Source-Version ist nicht geplant.
https://blogs.oracle.com/henrik/entry/oracle_and_cavium_to_work

4. Juni 2014

Java EE 8 Update

Die vorbereitenden Arbeiten an Java EE 8 laufen auf Hochtouren und viele Rahmendaten sind geplant. Vorsicht ist aber angebracht, gerade die Erfahrung mit Java hat zuletzt ja gezeigt, dass Pläne nur dazu da sind, sie zu ändern. Auf den Java Days Tokyo hat Cameron Purdy, Vice President of Development bei Oracle, einen Zeitplan präsentiert: Die Einzel-JSRs sollen noch im Q2 2014 eingereicht werden, das „Early Draft Review" soll Anfang 2015 folgen, und der „Proposed Final Draft" Ende 2015. Bis zum Final Release der Referenz-Implementierung und des SDK dauert es dann aber noch bis Q3 2016 (laut Plan, wohlgemerkt). Als neue JSRs sind geplant: J-Cache (die es ja leider nur fast in Java EE 7 geschafft hatten), Java-API for JSON Binding und Java Configuration. Als Leit-Themen für das Release sollen die folgenden Ziele dienen: Unterstützung der neuesten Web-Standards (wie HTTP 2.0), „Ease of development" (nicht neu, aber weiter wichtig), eine verbesserte Infrastruktur für Cloud-Unterstützung und weiteres Alignment mit Java SE 8.
https://blogs.oracle.com/theaquarium/entry/java_ee_8_update

12. Juni 2014

Java EE 7 hat Geburtstag!

Wie die Zeit vergeht: Java EE 7 wird schon ein Jahr alt. Unterstützt wird es in seiner ganzen Bandbreite bislang nur von GlassFish 4, WildFly 8 (ehemals JBoss AS) und TMAX JEUS 8. Aus gegebenem Anlass das Ergebnis einer Umfrage von RebelLabs zur Verbreitung der Java EE-Versionen: Demnach liegt der Anteil von Java EE 6 noch bei 49 Prozent. Java EE 7 folgt bereits mit 35 Prozent, was laut RebelLabs mit einem 100-prozentigen Anstieg der Nutzerzahlen von GlassFish einhergeht – bemerkenswert nach der Abkündigung des kommerziellen Supports durch Oracle.
https://blogs.oracle.com/theaquarium/entry/java_ee_7_turns_one

30. Juni 2014

Die Gewinner der IoT Developer Challenge

Die sechs Gewinner der IoT Developer Challenge stehen fest, jeweils drei professionelle und drei studentische Projekte. In beiden Kategorien haben sich die Gewinner mit dem Thema „Pflanzen" beschäftigt und ein Fernüberwachungssystem für Heimpflanzen mit Gießfunktion beziehungsweis gleich ein vollautomatisches Gewächshaus konstruiert. Daneben gibt es unter anderem einen Überwachungsroboter, der per Twitter kommuniziert, und ein Spracherkennungssystem zur Zugangskontrolle.
http://www.java.net/challenge

8. Juli 2014

JMS 2.1 wird geplant

Mit Java EE 8 soll auch der Java Message Service eine Aktualisierung erfahren. Spec-Lead Nigel Deakin hat jetzt einen Entwurf für den JSR zu JMS 2.1 veröffentlicht und hofft auf Feedback aus der Community. Einige der Themen, die für 2.1 geplant sind: „Ease of use" entsprechend dem übergeordneten Fokus von Java EE 8; Verbesserung der Portabilität von JMS-Providern; Unterstützung von Java-SE-8-Neuerungen im JMS-API.
https://java.net/projects/jms-spec/pages/JMS21JSRDraft1

8. Juli 2014

Java EE 7 und WebLogic 12.1.3

Nein, die neueste Weblogic-Version ist immer noch nicht vollständig Java-EE-7-zertifiziert. Aber um das Warten ein wenig zu verkürzen, hat Bruno Borges eine Liste der EE-7-JSRs gepostet, die bereits unterstützt werden (JPA 2.1, JAX-RS 2.0, JSON-P 1.0 und WebSockets 1.0), plus Links zu einigen anderen wissenswerten Dingen.
http://blogs.oracle.com/theaquarium/entry/java_ee_7_support_comes

12. Juli 2014

Java wird weiter auf Windows XP laufen – mit Einschränkungen

Henrik Ståhl tritt der Gerüchteküche entgegen, nachdem erst Microsoft Windows XP abgekündigt und daraufhin Oracle den offiziellen Java-Support für XP eingestellt hat. Java werde weiterhin auf XP laufen und Updates (natürlich auch Security-Updates) im Rahmen des normalen Supports geliefert werden – zumindest für alle Java-Versionen, die zum Zeitpunkt der Abkündigung auf XP unterstützt waren. Aber selbst für das JDK 7 ist ja das Ende der Public-Updates in Sicht (April 2015).

Ob das JDK 7, das momentan (noch) nicht auf XP unterstützt wird, dann für XP fit gemacht wird, ist noch nicht entschieden. Zwar werde an den Problemen mit dem Installer unter XP gearbeitet, aber es sei „nicht klar, warum man auf Java 8 migrieren sollte, ohne gleichzeitig das Betriebssystem zu aktualisieren", fasst Henrik Ståhl zusammen.
https://blogs.oracle.com/henrik/entry/the_future_of_java_on

Andreas Badelt
Leiter der DOAG SIG Java

Andreas Badelt ist Senior Technology Architect bei Infosys Limited. Daneben organisiert er seit 2001 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V.
http://ja.ijug.eu/14/4/2

UPDATE An dieser Stelle erhalten Sie in jeder Ausgabe ein Update über das Geschehen in der Java-Community

JDK 8 im Fokus der Entwickler

Wolfgang Weigend, ORACLE Deutschland B.V. & Co. KG

Das offene und frei verfügbare OpenJDK mit seiner einheitlichen Entwicklungsbasis bildet die zentrale Grundlage für die Aktivitäten der Java Standard Edition 8 (Java SE 8) und der Java Standard Edition 9 (Java SE 9).

Java ist der technologische Ausgangspunkt der meisten Hard- und Software-Hersteller und bildet auch die Basis für kommerzielle Middleware und unternehmensweit genutzte Anwendungen. Dies verdeutlicht auch das Geschäftsmodell für Java-Entwickler, die anhand der erlernten Programmiersprache und der frei zugänglichen Java-Technologie die erstellte Programmierlogik für Anwendungen und Services in die kommerzielle Vermarktung bringen.

Die Verwendung von Java in Open-Source-Projekten macht einen Großteil der IT-Landschaft aus, bietet doch der kommerzielle Einsatz des Java-Programmier-Codes die Möglichkeit einer Einnahmequelle für die Entwickler. Bereits bei der Verwendung des OpenJDK ist der Entwickler integraler Bestandteil einer klar umrissenen Java-Strategie.

Im März dieses Jahres wurde das JDK 8 freigegeben und die Entwickler konnten erste Erfahrungen sammeln. Die Funktionalität des JDK 8 erstreckt sich über die neuen Lambda-Ausdrücke für funktionale Programmierung, die Nashorn-JavaScript-Engine für JavaScript-JVM-Interoperabilität und zahlreiche Neuerungen in der Java-Sprache sowie den Core-Bibliotheken bis hin zu Java-Security und JVM. Weitere Verbesserungen sind im JDK 8 bei Typ-Annotationen und dem „Date & Time"-API zu finden. Die mit Java SE 8 eingeführten Compact-Profile stehen als Subsets von Java SE 8 in drei Varianten zur Verfügung, um die Modularisierung für Java Embedded bereits im JDK 8 vorzubereiten. Die Java-Plattform-Modularisierung mit dem Projekt Jigsaw ist für das JDK 9 vorgesehen.

Versionsnummerierung

Die Nummerierung von JDK-8-Update-Versionen erfolgt nach dem bisherigen Prinzip: Nach dem JDK 8 Update 5 (Critical-Patch-Update) folgt das nächste Critical-Patch-Update mit JDK 8 Update 11, dann kommt das nächste Java-Limited-Update JDK 7 Update 20. Darauf können mit JDK 8 Update 25, JDK 8 Update 31 und JDK 8 Update 35 die nächsten drei Critical-Patch-Updates folgen, dann das nächste Java-Limited-Update JDK 8 Update 40. Das Muster sieht drei Critical-Patch-Updates vor, bis zum nächsten Limited-Java-Update. Nach Bedarf können zwischendurch weitere Java-Updates veröffentlicht werden. Danach folgen die nächsten drei Critical-Patch-Updates, mit JDK 8 Update 45, JDK 8 Update 51 und JDK 8 Update 55, bis zum nächsten Java-Limited-Update, dem JDK 8 Update 60 usw. (siehe Abbildung 1).

Um das künftige Release vorab besser testen zu können, sind vor der Freigabe eines geplanten Java-Limited-Updates üblicherweise acht Wochen vorher die Binary-Snapshot-Releases und Early-Access-Releases per Download verfügbar. Die Early-Access-Releases von JDK und JRE basieren auf dem zu diesem Zeitpunkt verfügbaren OpenJDK-Code und dem daraus erzeugten Build. Deshalb ist nicht gewährleistet, dass die neuesten Sicherheits-Patches darin enthalten sind.

Das Oracle JDK 8u5 und die vollständige JRE werden auf den folgenden Betriebssystem-Plattformen unterstützt: Windows 32- und 64-bit, Linux 32- und 64-bit, Mac OS X 64-bit sowie Solaris SPARC 64-bit & Solaris 64-bit. Die mit Java SE Embedded 8 eingeführten Compact-Profile 1, 2 und 3 ermöglichen auch den Einsatz von Java für kleinere Geräte, mit reduziertem Speicherbedarf. Der Speicherbedarf für Compact Profile 1 beträgt 13,8 MB, Compact Profile 2 benötigt 17,5 MB, für Compact Profile 3 werden 19,5 MB gebraucht und die vollständige JRE erfordert 38,4 MB. Die entsprechende Referenz-Implementierung wird jeweils als Binärdatei mit der GNU-General-Public-Lizenz Version 2 und der Oracle-Binary-Code-Lizenz per Download angeboten.

Lambda-Ausdrücke

Mit der Einführung von Lambda-Ausdrücken und dem Stream-API erfährt Java SE 8 seine größte Sprachänderung seit der Einführung von Generics in Java SE 1.5, um die funktionale Programmierung auch für Java zu ermöglichen und dabei maximale Abwärtskompatibilität sicherzustellen. Bisher wurden in Java oftmals Single-Abstract-Method-Typen (SAM) verwendet – als Interface mit einer einzigen abstrakten Methode. Damit kann eine Klasse erzeugt werden, die das Interface implementiert und ein Objekt dieses Typs instanziiert. Da dies relativ schwerfällig ist, erlaubt Java stattdessen die Verwendung einer anonymen inneren Klasse („Inner Class"); das ist allerdings unvorteilhaft, weil zu viel zusätzlicher Code verwendet werden muss. Um dieses Problem zu lösen, wurden die Lambda-Ausdrücke in die Java-Sprachsyntax aufgenommen und mittels „invokedynamic"-Bytecode implementiert. Die Verwendung von „MethodHandles" verbessert die Leistungsfähigkeit von Lambda-Ausdrücken gegenüber den gleichwertigen anonymen inneren Klassen. In den Java-Klassenbibliotheken wurden SAM-Typen statt aller anonymen inneren Klassen verwendet; damit erhält man ohne

Anwendungs-Code-Änderung schnellere Java-Anwendungen.

Das neue Lambda-Sprachfeature verändert die Art und Weise, wie Java-Programme künftig geschrieben werden, weil dadurch einige neue Konzepte in Java SE 8 eingeflossen sind. Es stellt sich die Frage, was ein Lambda ist. Im Grunde repräsentiert ein Lambda eine Funktion. Wo liegt jetzt der Unterschied zwischen einer Methode und einer Funktion? Eine Funktion ist eine Berechnung, die eine Anzahl von Parametern zugeführt bekommt und dann einen Wert zurückliefert. Damit ist es eher ein mathematisches Konzept. Funktionen können auch Seiteneffekte haben und deshalb war es bei der Implementierung von Lambda-Ausdrücken in Java SE 8 besonders wichtig, diese Seiteneffekte zu vermeiden. Es geht hauptsächlich darum, die Eingangsparameter zu berechnen und dann ein Ergebnis zurückzugeben.

Bevor es Java SE 8 gab, musste jede Berechnung in einer Methode enthalten sein, die wiederum in einer Klasse enthalten war; der Programmcode war Bestandteil einer Methode. Diese hat einen Namen und der Methoden-Aufruf erfolgte über die Namens-Referenz und möglicherweise auch über die Klasse mit der enthaltenen Methode oder mit einer Instanz, die mit der Klasse und der darin enthaltenen Methode verbunden war. Somit konnte man Funktionen bisher nur mithilfe von Methoden in Java implementieren.

Mit den Lambdas in Java SE 8 bekommt man Funktionen, die nicht notwendigerweise fest mit einem Methoden-Namen innerhalb einer benannten Klasse verbunden sind. Wird die Funktion aus dem strikten Korsett von Methoden-Name und Klasse herausgelöst, so kann man eine Lambda-Funktion verwenden und sie als Parameter weiterreichen, als Rückgabewert zurückgeben oder in eine Datenstruktur einbinden. Damit besteht jetzt auch mit Java die Möglichkeit, Funktionen zu benutzen, wie es bereits in anderen Programmiersprachen der Fall ist. Funktionen können ineinander verschachtelt sein und Funktionen können wiederum Funktionen zurückliefern. Die Beispiele in Listing 1 zeigen eine formale Parameterliste, den Pfeiloperator „->“ und den darauffolgenden Lambda-Ausdruck, der den Term aus den Parametern bildet.

Abbildung 1: Java-Limited-Updates und Critical-Patch-Updates

```
(int x, int y) -> x+y           // Multiple declared-type parameters

(x,y) -> x+y                    // Multiple inferred-type parameters

(final int x) -> x+1            // Modified declared-type parameter

(x, final y) -> x+y             // Illegal: can‘t modify inferred-type
parameters

(x, int y) -> x+y               // Illegal: can‘t mix inferred and de-
clared types

(x, y, z) -> {
  if (true) return x;
  else {
    int result = y;
    for (int i = 1; i < z; i++)
      result *= i;
    return result;
  }
}
```

Listing 1: Beispiele für Lambda-Ausdrücke in der Java-SE-8-Spezifikation

Modifier und Type	Methode und Beschreibung
default Predicate<T>	and(Predicate<? super T> other) Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>	isEqual(Object targetRef) Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).
default Predicate<T>	negate() Returns a predicate that represents the logical negation of this predicate.
default Predicate<T>	or(Predicate<? super T> other) Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	test(T t) Evaluates this predicate on the given argument.

Tabelle 1: Interface „Predicate<T>“ mit Methoden

Anonyme Funktionen und „Functional Interfaces“

Der Lambda-Ausdruck repräsentiert eine anonyme Funktion und seine Struktur ist mit der einer Methode vergleichbar, aber sie ist eben nicht mit der Klasse verbunden. Lambda-Ausdrücke erhalten typisierte Parameter von links, auf der rechten Seite stehen der Rückgabe-Typ und die Fehlerbehandlung. Beim Rückgabe-Typ kann explizit „return“ verwendet werden, jedoch kann in den meisten Fällen der Compiler den Rückgabe-Typ herleiten.

Der Typ des Lambda-Ausdrucks ist ein „Functional Interface“, das als Interface mit einer einzigen abstrakten Methode definiert ist. Dabei wurden feine Unterschiede gemacht, weil „Functional Interfaces“ mehr

```
package exercise;
//define a functional interface
public class WorkerInterfaceTest {
     public static void execute(WorkerInterface worker) {
        worker.doSomeWork();
    }
     public static void main(String [] args) {
         //invoke doSomeWork using Annonymous class
        execute(new WorkerInterface() {
            @Override
            public void doSomeWork() {
                System.out.println('Worker invoked using Anonymous
class');
            }
        });
          //invoke doSomeWork using Lambda expression
        execute( () -> System.out.println("Worker invoked using
Lambda expression') );
    }
 }
```

Listing 2: „WorkerInterfaceTest.java" mit „Functional Interface"-Definition

```
package exercise;
    @FunctionalInterface
public interface WorkerInterface {
     public void doSomeWork();
 }
```

Listing 3: „WorkerInterface.java"

```
Worker invoked using Anonymous class
Worker invoked using Lambda expression
```

Listing 4: Datenausgabe von „WorkerInterfaceTest.java"

```
package exercise;
    @FunctionalInterface
public interface WorkerInterface {
     public void doSomeWork();
    public void doSomeMoreWork();
   }
```

Listing 5: Fehlerhaft verändertes „WorkerInterface.java"

als eine Methode haben. Beispielsweise besitzt das „Predicate Interface" mehrere Methoden, aber nur eine ist abstrakt, „test(T t)". Die anderen Methoden haben Default-Implementierungen und die Methode „isEqual" ist statisch (siehe Tabelle 1). Die Fähigkeit eines Interface, statische Methoden zu enthalten, ist ebenfalls ein neues Java-Feature. Ein eigenes „Functional Interface" kann mit der Annotation „@FunctionalInterface" erzeugt werden und der Compiler überprüft, ob das Interface die korrekte Struktur besitzt. Das Functional Interface wird in der Datei „WorkerInterfaceTest.java" definiert (siehe Listing 2), die korrekte Annotation „@FunctionalInterface" steht in der Datei „WorkerInterface.java" (siehe Listing 3) und zeigt in der Datenausgabe (siehe Listing 4) den Hinweis auf die anonyme Klasse mit Ausführung des Lambda-Ausdrucks.

Wird dem Interface der Annotation „@FunctionalInterface" die Zeile „public void doSomeMoreWork();" zur Veranschaulichung hinzugefügt (siehe Listing 5), so bekommt man die Fehlermeldungen „Unexpected @FunctionalInterface annotation, WorkerInterface is not a functional interface .." beim Programm-Build aus der Datei „WorkerInterface.java" vom Compiler (siehe Listing 6). Mit dem Lambda-Ausdruck wird die Instanz vom „Functional Interface" erzeugt und kann in den Methoden-Aufrufen und -Zuweisungen verwendet werden. Über das „Functional Interface" verkörpert der Lambda-Ausdruck den abgeleiteten Kontext.

Stream-API

Mit den Lambda-Ausdrücken, den Extension-Methoden für Interfaces und dem Stream-API sind funktionale Programmier-Eigenschaften in Java SE 8 eingeflossen, die eine Manipulation von Java-Collections und Daten ermöglichen. Das Package „java.util.stream" beinhaltet Klassen zur Unterstützung funktionaler Operationen, angewendet auf Streams, eine Gruppe von aneinandergereihten Elementen, vergleichbar mit MapReduce-Transformationen bei Java-Collections. Dabei stellen die Streams die Abfolge der Elemente dar.

Streams besitzen Operationen, die funktionale Bestandteile in Form von Lambdas als Argument nehmen können. Die Operationen sind auf alle Elemente der Folge anwendbar. Die Verarbeitung kann mit mehreren Threads parallel oder wahlweise mit einem Thread sequenziell erfolgen. Ein Stream ist keine Datenstruktur, die Elemente speichert, sondern er reicht die Elemente von unterschiedlichen Quellen wie Datenstrukturen, Arrays, Generator-Funktion oder I/O-Channel an eine Pipeline weiter, bestehend aus Berechnungsoperationen. Eine Operation, die auf einen Stream angewandt wird, erzeugt ein Resultat, aber sie verändert die Quelle nicht. Wird beispielsweise ein Stream gefiltert, den man aus einer Java-Collection erhalten hat, so wird ein neuer Stream ohne die gefilterten Elemente erzeugt – ohne Elemente aus der Java-Collection-Quelle zu entfernen.

Einige Stream-Operationen wie Filtering, Mapping oder das Entfernen von Duplikaten können über die Lazy-Implementierung verwendet werden, um eine Optimierung erst bei tatsächlichem Verarbeitungsbedarf vorzunehmen. Um beispielsweise das erste Wort mit drei gleichen aufeinanderfolgenden Buchstaben in einem String zu finden, müssen nicht alle Input-Strings komplett untersucht werden.

Die Stream-Operationen bestehen aus beliebigen Intermediate-Operationen („filter", „map") mit Lazy-Implementierung und

einer Terminal-Operation („reduce“, „sum“). Intermediate-Operationen erzeugen einen Stream, sie verwenden keine Heap-Objekte, sondern Intermediate-Memory, und die abschließende Terminal-Operation erzeugt einen Wert oder hat einen Seiteneffekt. Während Java-Collections eine feste Größe haben, ist dies bei den Streams nicht der Fall. Kurz laufende Operationen, wie „limit(n)“ im Listing 7 oder „findFirst()“, erlauben Berechnungen auf unbegrenzten Streams, um sie in einer endlich bestimmten Zeit abzuschließen. Die Stream-Elemente können nur während des Stream-Lebenszyklus benutzt werden. Soll dasselbe Element der Quelle nochmal verwendet werden, so muss ein neuer Stream erzeugt werden, vergleichbar mit dem Iterator.

Berechnung mit Lambdas und Parallelisierung

Im Beispiel vom Listing 8 werden eine definierte Array-Liste über Lambdas ausgewertet und die jeweiligen Ergebnisse ausgegeben. Mit der bisherigen Vorgehensweise – ohne Lambdas – wird die Summe in der For-Schleife berechnet (siehe Listing 9) und man erhält als Summe 285. Im Listing 10 wird die Berechnung der Summe mit den beiden Lambda-Ausdrücken jeweils nach „.map“ und nach „.reduce“ in einer einzigen Zeile geschrieben. Damit erhält man auch das Ergebnis Sum = 285, jedoch mit dem Vorteil einer höheren Abstraktion und besseren Lesbarkeit des Codes.

Verwendet man „parallelStream()“, wie im Listing 11 dargestellt, und schaut sich die Threads der entsprechenden Prozess-ID mit Java VisualVM oder Java Mission Control an, so sieht man, wie sich die Verarbeitung auf mehrere Threads verteilt, beispielsweise auf die „ForkJoinPool.commonPool-worker-0“ bis„ForkJoinPool.commonPool-worker-3“ im darunterliegenden Fork-Join-Framework.

„Date & Time“-API

Aufgrund der Fehleranfälligkeit der bisherigen Datums- und Zeitmechanismen wurde das „Date, Time und Kalender“-API (JSR 310) für Java SE 8 mit JEP 150 neu definiert und in die existierenden Java-APIs integriert. Maßgebliche Ideen und Erkenntnisse vom Joda-Time-API sind durch Stephen Colebourne über JEP 150 in das Java SE 8 „Date & Time API“-übertragen worden.

```
Error: Unexpected @FunctionalInterface annotation
WorkerInterface is not a functional interface
Multiple non-overriding abstract methods found in interface Work-
erInterface

Error: <anonymous exercise.WorkerInterfaceTest$1> is not abstract
and does not override abstract method doSomeMoreWork() in Worker-
Interface
execute(new WorkerInterface() {

Error: incompatible types: WorkerInterface is not a functional
interface
execute( () -> System.out.println("Worker invoked using Lambda
expression") );
multiple non-overriding abstract methods found in interface Work-
erInterface
```

Listing 6: Fehlermeldungen beim Build vom Programm „WorkerInterfaceTest.java“

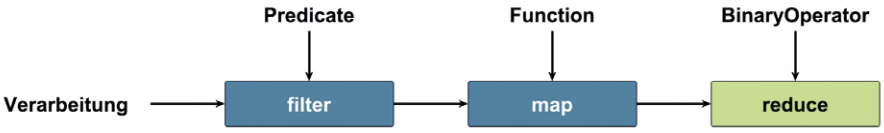


Abbildung 2: Pipeline mit Intermediate-Operationen („filter“, „map“) und Terminal-Operation („reduce“)

```
import java.util.function.Supplier;
Supplier<Double> random = Math::random;
System.out.println("Zwei Zufallszahlen:");
Stream.generate(random).limit(2).forEach((x) -> System.out.
println(x));
System.out.println("Vier Zufallszahlen:");
Stream.generate(random).limit(4).forEach((x) -> System.out.
println(x));

Zwei Zufallszahlen:
0.3405937451886095
0.5720233570895665

Vier Zufallszahlen:
0.023603094775235145
0.6088330495614978
0.257001300043167
0.5215687978948829
```

Listing 7: Erzeugen von Zufallszahlen und Ausgabe des Ergebnisses

Die neue Implementierung von „Date & Time“ im JDK 8 unterstützt ein standardisiertes Zeitkonzept einschließlich „Date“, „Time“, „Instant“, „Duration“ und „Zeitzonen“. Ein zentrales Kalendersystem bildet das Fundament für die Date- und Time-Klassen und ist auch für andere Kalender erweiterbar. Das Kalendersystem ist nach dem ISO-8601-Standard definiert und weltweit einsetzbar.

Das „java.time“-API beinhaltet fünf Packages (siehe Tabelle 2). In Tabelle 3 sind die Basisklassen des Package „java.time“ dargestellt. Ausgehend von einem Abschnitt der Zeitrechnung verwendet die interne Speicherung einen Offset in Nanosekunden; die Zeiteinteilung wurde anhand der Coordinated Universal Time (UTC) ausgerichtet. Die „java.time“-Basisklassen sind als unveränderliche Value-Objekte implementiert, die

```
package exercise;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
public class Main {
    public static void main(String [] a)  {

        List<Integer> list = Arrays.asList(1 , 2, 3, 4, 5, 6, 7,
8, 9);

        System.out.print("\n" + "Print all numbers: ");
        evaluate(list, (n)->true);
        System.out.print("\n" + "Print no numbers: ");
        evaluate(list, (n)->false);
        System.out.print("\n" + "Print even numbers: ");
        evaluate(list, (n)-> n%2 == 0 );
        System.out.print("\n" + "Print odd numbers: ");
        evaluate(list, (n)-> n%2 == 1 );
        System.out.print("\n" + "Print numbers greater than 6: ");
        evaluate(list, (n)-> n > 6 );
        System.out.print("\n" + "Print numbers lower than 6: ");
        evaluate(list, (n)-> n < 6 );
    }

    public static void evaluate(List<Integer> list,
Predicate<Integer> predicate) {
        for(Integer n: list)  {
            if(predicate.test(n)) {
                System.out.print(n + " ");
            }
        }

    }
}

Print all numbers: 1 2 3 4 5 6 7 8 9
Print no numbers:
Print even numbers: 2 4 6 8
Print odd numbers: 1 3 5 7 9
Print numbers greater than 6: 7 8 9
Print numbers lower than 6: 1 2 3 4 5
```

Listing 8: Array-Liste mit Lambda-Ausdrücken und Ausgabe des Ergebnisses

```
        // Bisherige Progammierweise, ohne Lambdas
        int summe = 0;
            for(Integer n : list) {
                int x = n * n;
                summe = summe + x;
            }
        System.out.println("\n" + "Summe = " + summe);

Summe = 285
```

Listing 9: Berechnung der Summe in einer For-Schleife und Ausgabe der Summe

```
// Verwendung von Lambdas mit Java SE 8
int sum = list.stream().map(x -> x*x).reduce((x,y) -> x +
y).get();
System.out.println("Sum = " + sum);

Sum = 285
```

Listing 10: Berechnung der Summe mit Lambdas im JDK 8 und Ausgabe der Summe

auf mehrere Threads verteilt werden können. Im Listing 12 wird die Tageszeit ohne Datum mit „LocalTime" verwendet, im Listing 13 ein Datum ohne Zeit mit„LocalDate".

JavaFX-User-Interface

Mit dem JDK 8 wird auch die überarbeitete JavaFX-UI-Technologie ihre steigende Verbreitung in der Java-Community erfahren und es deutet sich ein positiver Trend für den zu erwartenden Einsatz von JavaFX in Desktop-Anwendungen an. Bisher entscheiden die Entwicklererfahrung und der Reifegrad von JavaFX über die künftige Verwendung dieser neuen UI-Technologie für geschäftskritische Desktop-Anwendungen und dies wird auch wesentlich für die Ablöse bestehender Swing-Technologie in Java-Anwendungen sein. Die künftige Standardisierung in einem JSR für JavaFX ist mit Java SE 9 vorgesehen.

JavaScript-Engine „Nashorn"

Nashorn ist die neue, leistungsstarke und standardkonforme JavaScript-Engine im JDK 8, die auch weitere Versionen von JavaScript und ECMAScript 6 unterstützen wird. Die künftigen Implementierungsschritte umfassen eine Leistungssteigerung von Nashorn und der JVM, sodass die Verarbeitungsgeschwindigkeit von JavaScript mit Nashorn auf der JVM nicht mehr infrage gestellt wird.

Mit dem in Nashorn neu implementierten Meta-Objekt-Protokoll werden die JavaScript-Aufrufe von Java-APIs vereinfacht. Dies ermöglicht die nahtlose Interaktion von JavaScript und Java. Denkbar ist auch eine Alternative am Browser, bei der die Entwickler wählen können, ob sie die Google V8 JavaScript Engine mit WebKit oder auch dort Nashorn verwenden möchten. Es bleibt offen, wie stark die Verbreitung von Anwendungen mit JavaScript am Browser und auf dem Server tatsächlich zunehmen wird. Aber Nashorn ermöglicht den Infrastrukturanbietern, die Verarbeitung von JavaScript und Java auf eine zentrale JVM zu konsolidieren.

Die JVM bietet neben Memory-Management, Code-Optimierung und Scripting für die Java-Plattform (JSR-223) auch viele nützliche JVM-Management-Werkzeuge mit „Debugging & Profiling"-Fähigkeiten. Nashorn verfolgt das Ziel, die Attraktivität der JVM zu erhöhen und sie langfristig als

mehrsprachige Laufzeitumgebung anzubieten, wie im OpenJDK-Projekt Multi-Language VM (MVLM, oder auch „Da Vinci Machine"-Projekt) dargestellt. Die daraus gewonnenen Forschungsergebnisse dienen dazu, existierenden Bytecode zu verwenden, um die Geschwindigkeitsvorteile von Nashorn im JDK 9 zu erhöhen und den HotSpot-Code-Generator für eine verbesserte Unterstützung dynamischer Sprachen anzupassen.

Ausblick auf Jigsaw, Sumatra und Panama im JDK 9

Das Projekt Jigsaw hat die Aufgabe, das Design und die Implementierung eines Standard-Modulsystems für die Java-Plattform und für das JDK bereitzustellen. Das Projekt befand sich einige Jahre in einer Forschungsphase, woraus der Prototyp eines individuellen Ansatzes entstanden ist, um die vorläufigen Anforderungen darzustellen. Nun sollen diese Erkenntnisse für eine produktiv ausgereifte Implementierung passend für JDK 9 mit hoher Qualität umgesetzt werden. Die dabei entstandene Dokumentation beschreibt den Prototyp, jedoch ohne ihn zu überspezifizieren oder die angestrebte Lösung einzuschränken, sondern sie dient als Startpunkt für den künftigen JSR vom Java-Platform-Module-System.

Jigsaw wird erhebliche Änderungen für das JDK mit sich bringen, sodass es unklug wäre, alles komplett fertigzustellen, bevor man es zusammengefügt hat. Aus diesem Grund wird der Erstellungsprozess in großen Schritten durchgeführt und von mehreren JEPs begleitet. Mit den ersten drei JEPs werden die Vorschläge für eine spezifische modulare Struktur für das JDK gemacht und die Reorganisation des JDK-Source-Codes an diesen Stellen vollzogen, jedoch ohne die Binaries anzutasten; später werden die Binär-Images modularisiert. In einem vierten JEP wird das Modulsystem selbst vorgestellt, das anhand der Modulsystem-Spezifikation (JSR) ausgerichtet sein wird.

Es erscheint ungewöhnlich, dass der Modulsystem-JEP am Ende kommt, aber vorangegangene JEPs brauchen nur minimale Annahmen über ihre Fähigkeiten zu treffen, deshalb kann die Arbeit an der Funktionalität parallel zu den Arbeiten am Modulsystem-JEP und dem JSR erfolgen. So lauten die primären Ziele von Jigsaw:

```
int s = list.parallelStream().map(x -> x*x).reduce((x,y) -> x +
y).get();
System.out.println("Sum = " + s);
```

Listing 11: Berechnung der Summe mit Lambdas und Parallelisierung

Package	Beschreibung
java.time	The main API for dates, times, instants, and durations
java.time.chrono	Generic API for calendar systems other than the default ISO
java.time.format	Provides classes to print and parse dates and times
java.time.temporal	Access to date and time using fields and units, and date time adjusters
java.time.zone	Support for time-zones and their rules

Tabelle 2: Packages des „java.time"-API

Klasse	Beschreibung
Clock	A clock providing access to the current instant, date and time using a time-zone.
Duration	A time-based amount of time, such as‚34.5 seconds'.
Instant	An instantaneous point on the time-line.
LocalDate	A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.
LocalDateTime	A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.
LocalTime	A time without time-zone in the ISO-8601 calendar system, such as 10:15:30.
MonthDay	A month-day in the ISO-8601 calendar system, such as --12-03.
OffsetDateTime	A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00.
OffsetTime	A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00.
Period	A date-based amount of time in the ISO-8601 calendar system, such as‚2 years, 3 months and 4 days'.
Year	A year in the ISO-8601 calendar system, such as 2007.
YearMonth	A year-month in the ISO-8601 calendar system, such as 2007-12.
ZonedDateTime	A date-time with a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00 Europe/Paris.
ZoneId	A time-zone ID, such as Europe/Paris.
ZoneOffset	A time-zone offset from Greenwich/UTC, such as +02:00.

Tabelle 3: Basisklassen im Package „java.time"

- Durchgängige, dynamische und einfache Anpassbarkeit der Java-SE-Plattform und des JDK, auch für kleine Endgeräte
- Allgemeine Verbesserung von Sicherheit und Wartbarkeit von Java-SE-Plattform-Implementierungen, speziell vom JDK
- Bessere Anwendungsperformance
- Einfache Erstellung und Wartung von Bibliotheken und großen Anwendungen für Java SE und Java EE

Sumatra

Das für JDK 9 vorgesehene Projekt Sumatra zielt auf die bessere Ausnutzung von grafischen Prozessoren, GPUs und Grafikbeschleuniger-Einheiten-APUs zur Leistungssteigerung für Java-Anwendungen ab. Dies ist unabhängig davon, ob einzelne grafische Prozessoren oder integrierte Prozessoren im Einsatz sind. Der Schwerpunkt liegt auf der HotSpot-JVM um die Code-Ge-

```
LocalTime now = LocalTime.now();
System.out.println("Time now = " + now);
LocalTime later = now.plusHours(8);
System.out.println("Time later = " + later);

Time now = 11:03:30.889
Time later = 19:03:30.889
```

Listing 12: „LocalTime"

```
day = LocalDate.of(2014, Month.JULY, 4);
System.out.println("So ein schöner Tag = " + day);

So ein schöner Tag = 2014-07-04
```

Listing 13: „LocalDate"

nerierung, die Laufzeitunterstützung und die Garbage Collection für GPUs zu ermöglichen. Sobald erste Basisfähigkeiten in der JVM vorhanden sind, wird untersucht, wie die bestmögliche GPU-Unterstützung für die Anwendungen, die Bibliotheken-Entwicklung, die Nutzbarkeit der neuen Lambda-Sprachausdrücke und zusätzliche Library-Merkmale aussehen wird.

Sollten während des Projektverlaufs neue Anforderungen in Bezug auf das Java API aufkommen, die Erweiterungen für die Sprache, die JVM und die Bibliotheken erfordern, so wird dies unter dem standardisierten Prozess im JCP geführt. Für das Projekt Sumatra steht die Sprache Java im Vordergrund, aber andere JVM-basierte Sprachen wie JavaScript mit Nashorn, Scala und JRuby können ebenfalls davon profitieren.

Panama

Hinter dem Namen Panama steht der Vorschlag für ein neues OpenJDK-Projekt, um einen nativen Interconnect zu schaffen, der den Code, der von einer JVM verwaltet wird, mit den Library-APIs, die nicht durch die JVM verwaltet werden, verbinden soll. Damit sollen Verbindungen von Non-Java-APIs mit JVM-managed-Code ermöglicht werden, sodass Java-Entwickler beispielsweise auch C-APIs nutzen können. So erleichtert sich auch die Anbindung der JVM mit dem Programmiercode und dem jeweiligen Datenzugriff an die eigenständigen APIs. Gleichfalls wird ein grundlegendes Binding für die nativen APIs betrachtet. Je nach Entwicklungsfortschritt im Projektvorschlag Panama könnten die Ergebnisse noch in das JDK 9 gelangen.

Fazit

In Java SE 8 sind mehr als fünfzig neue Merkmale enthalten, die zuallererst für die Entwickler von großem Interesse sind. Sie können so neue Java-Anwendungen programmieren, die nach erfolgten Tests in den IT-Betrieb gelangen. Es hängt einerseits von den Entwicklern ab, wie schnell sie die eingeforderten neuen Java-Features bei der Anwendungsentwicklung umsetzen können, und andererseits vom JDK-Reifegrad, der für den Produktivbetrieb von unternehmenskritischen Anwendungen maßgeblich ist. Dabei sind ausführliche Testverfahren zwingend notwendig, um den Anwendern neue und innovative Java-Programme zur Verfügung zu stellen. Ist der Programm-Code mit den neuen Lambda-Ausdrücken besser lesbar, können fachliche Änderungen schneller und effektiver vom Entwickler implementiert werden.

Der IT-Betrieb besitzt mit dem JDK 8 bessere Möglichkeiten, bei Bedarf eigene Maßnahmen zur Performanzsteigerung von Java-Anwendungen zu ergreifen, beispielsweise mit der Parallelisierung von Threads und der optimierten Auslastung vorhandener physikalischer Prozessorknoten, die in der Vergangenheit oftmals ungenutzt blieben. Es hängt also von einem detailliert abgestimmten Einführungsplan ab, wie erfolgreich man auf das neue JDK 8 umsteigen kann. Vor allem sollte dafür ein exakter Zeitplan mit den Entwicklern, dem IT-Betrieb und den Fachabteilungen erstellt werden.

Referenzen und Quellen

[1] http://docs.oracle.com/javase/8
[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[3] https://jdk8.java.net/download.html
[4] https://jdk8.java.net/java-se-8-ri
[5] http://viralpatel.net/blogs/lambda-expressions-java-tutorial
[6] http://www.angelikalanger.com/Lambdas/Lambdas.html
[7] http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
[8] http://www.slideshare.net/martyhall/java-8-tutorial-streams-part-2-map-reduce-parallel-streams-and-infinite-streams
[9] Oracle Java magazine May/June 2014 Processing Data with Java SE 8 Streams
[10] http://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html
[11] http://jaxenter.de/Java-8-Nashorn-Weg-zur-polyglotten-VM-172134
[12] http://openjdk.java.net/projects/jigsaw/dashboards/module-system
[13] http://jdk8.java.net/jigsaw
[14] http://openjdk.java.net/projects/jigsaw/goals-reqs/03
[15] https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm

Wolfgang Weigend
wolfgang.weigend@oracle.com

Wolfgang Weigend, Systemberater für die Oracle Fusion Middleware bei der ORACLE Deutschland B.V. & Co. KG, ist zuständig für Java-Technologie und -Architektur mit strategischem Einsatz bei Großkunden. Er verfügt über langjährige Erfahrung in der Systemberatung und im Bereich objektorientierter Softwareentwicklung mit Java.

http://ja.ijug.eu/14/4/3

Einmal Lambda und zurück – die Vereinfachung des TestRule-API

Günter Jantzen, Capgemini

Anfangs sprach man von Closures, später von Lambda-Expressions, dann von Lambda-Ausdrücken und dem Java-8-Streams-API. Wir bekommen neue Werkzeuge, mit denen wir zunächst einmal umzugehen lernen müssen. Wie wirken neue und alte Sprachmittel zusammen, wie sind sie am besten einzusetzen? Wir erwarten hier einen regen Erfahrungsaustausch. Denn eines ist sicher: Die Lambdas werden Java verändern.

Heute wenden wir uns wieder den Closures zu, die schon mit Java 7 kommen sollten, deren Einführung dann aber verschoben wurde. Es geht los mit etwas Theorie, die dann auch gleich zum Einsatz gebracht wird. Ziel soll sein, Closures in der Anwendung zu verstehen und nebenbei die JUnit-Schnittstelle zur Erstellung von Testregeln zu vereinfachen.

Closures in Java

Als „anonyme Funktion" (manchmal auch „Codeblock") bezeichnet man allgemein in Programmiersprachen ein ausführbares Stück Code, das man herumreichen und später aufrufen kann. Wir können uns eine anonyme Funktion als einen parametrisierbaren Ausdruck vorstellen. Die Parameter und lokalen Variablen dieses Ausdrucks bezeichnen wir als „gebundene Variable". Die verbleibenden ungebundenen Variablen dieses Ausdrucks heißen „freie Variable". Die Bindung der freien Variablen einer anonymen Funktion erfolgt nicht in der anonymen Funktion selbst, sondern in dem definierenden lexikalischen Kontext, in den sie textuell eingebettet ist.

```
public abstract class Verifier implements TestRule {
  public Statement apply(final Statement base, Description de-
scription) {
     return new Statement() {
         @Override
         public void evaluate() throws Throwable {
             base.evaluate();
             verify();
         }
     };
  }
  /*** Override this to add verification logic. Overrides should
throw an exception to indicate that verification failed.*/
  protected void verify() throws Throwable {
  }
}
```

Listing 1

```
public interface TestRule {
    Statement apply(Statement base, Description description);
}
```

Listing 2

Mit dieser Begrifflichkeit dürfte eine anonyme Funktion keine freien Variablen enthalten. Denn wenn sie später und an anderer Stelle aufgerufen wird, ist der definierende Scope nicht mehr verfügbar, die freien Variablen sind undefiniert. Das ist in vieler Hinsicht eine zu starke Beschränkung und darum gibt es das schon lange und in vielen Programmiersprachen verbreitete Konstrukt der Closures.

Ein Closure ist eine anonyme Funktion mit der besonderen Eigenschaft, dass die Bindung der freien Variablen an ihre ursprüngliche Definition nicht verloren geht, wenn sie später an anderer Stelle aufgerufen wird. Diese dauerhafte Bindung einer freien Variablen wird bildhaft als „capture" bezeichnet. Listing 1 zeigt ein Beispiel dazu.

Dort verwendet die anonyme Klasse „Statement" in der Methode „evaluate" eine freie Variable „base". Diese wird außerhalb der Statement-Definition als Parameter der Methode „apply" gebunden, in die sie textuell eingebettet ist. Ein von der Methode „apply" zurückgegebenes Statement kann die Variable „base" beim Aufruf von „evaluate" auswerten, obwohl die ursprüngliche Definition nicht mehr zur Verfügung steht.

Closures waren in Java schon immer als innere Klassen realisierbar, also als anonyme oder als nicht statische eingebettete Klasse. Dies aber mit einer geradezu sprichwörtlichen Umständlichkeit, die bestenfalls als Notlösung gesehen werden konnte. In Java 8 wurde alles getan, um dieses vertikale Problem zu lösen.

Eine Besonderheit in Java ist, dass die Bindung einer freien Closure-Variablen nur möglich ist, wenn diese nach ihrer Initialisierung nicht mehr verändert wird. Seit Java 8 ist das Schlüsselwort „final" nicht mehr erforderlich, wenn der Compiler die Variable als effektiv final erkennt. Dies mag manchem immer noch als starke Einschränkung erscheinen, dient aber dem Schutz des Programmierers vor sich selbst.

Die Lambda-Ausdrücke in Java wurden für den Einsatz in Multicore-Umgebungen konzipiert. Dort ist Regel Nummer eins, mit unveränderlichen Daten und möglichst ohne Seiteneffekte zu programmieren. Nicht mehr die Daten werden herumgereicht, sondern der Code. Ohne die Einschränkung des Zugriffs auf unveränderliche Variable wären die Lambdas nicht „threadsafe".

Ein verhaltensbasiertes API im JUnit-Framework

Sind Closures wirklich so neu? Vielleicht hatten wir sie schon immer und wussten es bloß nicht so genau? Dieser Gedanke kam dem Autor beim Lesen des Buchs „JUnit-Profiwissen" von Michael Tamm. Es ist nicht nur ein schönes Buch zum Thema „Testen", sondern enthält nebenbei eine Fülle von Anwendungsbeispielen für Closures. Es sind Beispiele aus der Praxis, die kein JDK 8 benötigen, sondern mit den Compiler-Versionen, die man in seinen Projekten verwendet, laufen.

Closures werden mit anonymen Klassen realisiert und genutzt, um Schnittstellen mit Verhalten zu parametrisieren. Kein Zufall, dass derartige Beispiele beim Testen zu finden sind, denn dort geht es ja vor allem um Verhalten, nämlich das Verhalten der zu testenden Funktionalität. So beschreibt Michael Tamm, wie mit Closures verhaltensbasierte Schnittstellen für Mocking-Konfigurationen und Hamcrest-Matcher für maßgeschneiderte Asserts realisiert werden.

Sehr viele Beispiele bietet das Buch jedoch für die Verwendung von Testregeln. Das Rules-Framework „org.junit.rules" behandelt das Problem der Ressourcen-Verwaltung. Beim Testen sind oft verschachtelte Initialisierungs- und Aufräumarbeiten erforderlich, bei denen es auch auf die Einhaltung der Reihenfolge ankommen kann. Da dies ein viel genutzter Anwendungsfall ist, sollte es dem Anwender so leicht wie möglich gemacht werden, eigene Testregeln zu implementieren. Dies ist jedoch nicht der Fall.

Nachfolgend ist der Versuch beschrieben, das API für die Erstellung von Testregeln zu vereinfachen. Dabei geht es weniger um die Zukunft des JUnit-Projekts, sondern doch eher um den Erkenntnisgewinn am praktischen Beispiel. Das in Listing 2 gezeigte Interface „org.junit.rules.TestRules" beschreibt die zu implementierende Schnittstelle. Die ausführliche Java-Dokumentation der Sourcen bietet einen guten Einstieg und verweist auch auf verschiedene Beispiele, die Bestandteil des Projekts sind.

Zu sehen ist eine Methode „apply", die aus einem Statement „base" ein neues Statement bildet. Eine „TestRule" bildet eine Klammer für das Initialisieren und Entfernen einer einzelnen Ressource, die um den eigentlichen Test quasi herumgewickelt wird.

Eines der mitgelieferten Beispiele aus „org.junit.rules" ist der Verifier aus Listing 1. Der Aufbau ist schon etwas verwickelt, obwohl der Kern der Regel schlicht ist: Nach dem Aufruf eines Tests „base.evaluate" ist eine Methode „verify" aufzurufen. Dabei kann „base" für einen Testfall aus der Testklasse stehen, in der die Regel angewandt wird. Sofern eine Testklasse mehrere Regeln enthält, kann „base" aber bereits aus der Anwendung einer anderen Regel hervorgegangen sein. Der rekursive Aufbau der Definition erlaubt es, mehrere Regeln zu verschachteln. Initialisierungen erfolgen vor dem Aufruf von „base.evaluate", Aufräumarbeiten danach. Listing 3 zeigt, wie eine Verifier-Testregel typischerweise in einem Testfall verwendet wird.

Es ist ein Ausschnitt aus einer Testklasse zu sehen. Getestet wird ein Parser, der Fehlermeldungen in ein Error-Log schreibt. Die Testklasse enthält bereits viele Tests, beispielsweise für Gut-Fälle, deren eigentlicher Fokus nicht die Überprüfung des Error-Logs ist. Der Umgang mit dem Error-Log wird nicht in jedem einzelnen Test der Testklasse behandelt, sondern orthogonal dazu, als separater „Testaspekt" in einer Regel. Um diese technisch zu realisieren, ist sie mit der Annotation „@Rule" zu versehen. Sie wird dann auf alle in der Testklasse implementierten Tests angewandt.

Vereinfachung des TestRule-API

Eine schlichtere Anwendung einer Testregel ist in Listing 4 zu finden. Die Klasse enthält einen Testfall „test", der nichts anderes macht, als eine Meldung auszugeben. In der TestRule „rule" ist der Einfachheit halber ein „Verifier" wie in Listing 1 direkt in der Testklasse implementiert. Die Annotation „@Rule" macht den Testfall wieder dem JUnit-Framework bekannt.

Das Interface „TestRule" enthält nur eine abstrakte Methode, ist also ein funktionales Interface und kann damit durch einen Lambda-Ausdruck implementiert werden. Zudem wurde ausgenutzt, dass ein Lambda-Ausdruck „(b,d)->{return c;}", dessen Body aus einer einzigen Return-Anweisung besteht, sich einfacher schreiben lässt als „(b,d) -> c" .

Die in Listing 4 gezeigte Inline-Implementation eines Verifier mit einem Lambda-Ausdruck ist kompakter, als dies mit einer anonymen Klasse möglich gewesen wäre. Listing 5 zeigt die traditionelle Variante zum Vergleich.

Als Nächstes wollen wir auch die anonyme Klasse „Statement" loswerden und durch einen Lambda-Ausdruck ersetzen. Jedoch ist „org.junit.runners.model.State-

```
@Rule
public TestRule verifier = new Verifier() {
    @Override
    public void verify() {
        assertTrue(errorLog.isEmpty());
    }
};
```

Listing 3

```
public class TestRulePart1 {
    @Test
    public void test() {
        System.out.println("not implemented yet");
    }

    private void verify() {
        System.out.println("verify");
    }

    @Rule
    public TestRule rule = (base, d) -> new Statement() {
        @Override
        public void evaluate() throws Throwable {
            base.evaluate();
            verify();
        }
    };
}
```

Listing 4

```
public TestRule rule = new TestRule() {
    public Statement apply(final Statement base, final Description
d) {
        return new Statement() {
            @Override
            public void evaluate() throws Throwable {
                base.evaluate();
                verify();
            }
        };
    }
};
```

Listing 5

```
@FunctionalInterface
/** Ein Evaluable ist konzeptionell (nicht technisch) ein Inter-
face fuer org.junit.runners.model.Statement */

interface Evaluable {
    // Signatur wie Methode evaluate() in org.junit.runners.model.
Statement
    void evaluate() throws Throwable;
}
```

Listing 6

```
public class StatementFactory {
    static Statement makeStatement(final Evaluable e) {
        return new Statement() {
            public void evaluate() throws Throwable {
                e.evaluate();
            }
        };
    }
}
```

Listing 7

ment" kein Interface, sondern eine abstrakte Klasse. Lambdas sind so leicht, sie erben nicht von „Object", sie erben überhaupt nicht. Sie haben keine eigene Identität und kennen nur das „this" und „super" ihrer lexikalischen Wirtsumgebung. Der Preis dieser Leichtigkeit ist, dass Lambdas ausschließlich funktionale Interfaces implementieren können, und damit eben keine abstrakte Klassen.

Also schreiben wir, wie in Listing 6 zu sehen ist, „Evaluable" als neues funktionales Interface. Die Methode „evaluate" bekommt die gleiche Signatur wie die abstrakte Methode „evaluate" der JUnit-Klasse „Statement". Damit formulieren wir eine konzeptionelle Schnittstelle. Technisch besteht noch kein Bezug.

Listing 7 zeigt, wie dieser Bezug durch Delegation in einer „StatementFactory" realisiert wird. Die Factory-Methode „makeStatement" erzeugt eine Instanz einer anonymen, von „Statement" abgeleiteten Klasse. Die Methode „evaluate" delegiert den Aufruf an das als Parameter übergebene „Evaluable".

In Listing 8 sehen wir diese Factory-Methode in der Anwendung. Der Ausdruck zum Anlegen der Testregel „rule" ist einfacher geworden, wir brauchen keine anonyme Klasse mehr. Der Versuch, den größten Teil der Regel, also alles bis auf das Argument von „makeStatement", in eine „RuleFactory" auszulagern, scheitert jedoch.

Wir können zwar, wie in Listing 9 zu sehen, eine Factory-Methode „notUseful" schreiben, die aus einem „Evaluable e" eine Regel erstellt. Bei der Anwendung dieser Methode hätten wir dann allerdings das Problem, dass dieses „Evaluable" außerhalb des obigen Regelkontextes steht und damit kein sinnvoller Ausdruck ist. Der Compiler moniert zurecht „base cannot be resolved".

Wir verfolgen den Ansatz weiter, eine „RuleFactory" zu verwenden. Ziel ist, das richtige Argument für die Factory-Methode zu finden. Es sollte so einfach wie möglich sein, aber nicht zu einfach. Schauen wir uns noch einmal an, wie in Listing 1 eine „TestRule" implementiert wurde. Die anonyme Instanz eines Statements ist ein Closure, das die freien Variablen „base" und „description" enthalten kann. Diese werden an Parameter der Methode „apply" gebunden.

```
TestRule rule = (base, d) ->
makeStatement(() -> {
    base.evaluate();
    verify();
});

Evaluable e = () -> {
    // base.evaluate(); /*Fe-
hler: base cannot be resolved
*/
    verify();
};
TestRule notUsefulRule = Rule-
Factory.notUseful(e);
```

Listing 8

```
public class RuleFactory {
    static TestRule
notUseful(Evaluable e) {
        return (base, d) ->
makeStatement(e);
    }

    static TestRule
makeRule(BiFunction<Statement,
Description, Evaluable> f2) {
        return (base, d) ->
makeStatement(f2.apply(base,
d));
    }

    static TestRule
makeRule(OuterEvaluable oe) {
        return (base, d)
-> makeStatement(() ->
oe.evaluate(base, d));
    }
}
```

Listing 9

```
BiFunction<Statement, Description, Evaluable> f2 = (base, d)
-> () -> {
    base.evaluate();
    verify();
};
TestRule rule5 = RuleFactory.makeRule(f2);
```

Listing 10

```
OuterEvaluable oe = (base, d) -> {
    base.evaluate();
    verify();
};
TestRule rule6 = RuleFactory.makeRule(oe);
```

Listing 11

```
@FunctionalInterface
/** Ein OuterEvaluable ist auswertbar (um ein auswertbares State-
ment herum) */
interface OuterEvaluable {
    void evaluate(Statement base, Description d) throws Throwable;
}
```

Listing 12

```
public abstract class BadKarmaVerifier extends SmartTestRule {
    BadKarmaVerifier() {
        setOuterEvaluable((base, d) -> {
            base.evaluate();
            verify();
        });
    }

    // Override this to add verification logic. ...
    abstract void verify();
}
```

Listing 13

Auch wenn wir eine „TestRule“ einfacher bauen wollen, müssen wir trotzdem grundsätzlich genauso vorgehen. Wir übergeben also an eine verbesserte Factory-Methode nicht nur ein „Evaluable e“, sondern als geeignetes Ökosystem einen minimalen lexikalischen Kontext, der in der Lage ist, freie Variablen „base“ und gegebenenfalls „description“ zu binden.

Das Package „java.util.function“ stellt einen Baukasten mit vorgefertigten funktionalen Schnittstellen zur Verfügung. „BiFunction<T, U, R>“ beschreibt eine allgemeine Funktion mit zwei Parametern vom Typ „T“, „U“ und Rückgabetyp „R“. In Listing 10 wird also eine Funktion mit zwei Parametern vom Typ „Statement“ und „Description“ sowie einem Rückgabetyp „Evaluable“ definiert.

Die Parameter „base“ und „description“ spannen einen lexikalischen Scope um das zurückgegebene „Evaluable“. Das dort verwendete „base“ ist in diesem Scope gebunden. Der Compiler kann mit diesem Ausdruck etwas anfangen, er moniert nicht mehr.

Die Musik dieses Lambda-Ausdrucks spielt in den inneren geschweiften Klammern. Nur dort wird ein Effekt bewirkt. Die Parameterzuführung von zwei Parametergruppen erfolgt jedoch ohne Seiteneffekt. In der einen Gruppe werden zwei, in der anderen null Parameter zugeführt. Die seiteneffektfreie Zuführung einer leeren Parametergruppe ist ein Artefakt, das wir getrost entfernen dürfen. Der in Listing 11 gezeigte Lambda-Ausdruck benötigt ein neues funktionales Interface „OuterEvaluable“, das in Listing 12 definiert ist.

Das Interface „OuterEvaluable“ macht den rekursiven Charakter der Statement-Verwendung explizit. Die Erstellung eines „OuterEvaluable“ ist jetzt einfacher geworden, da die Parametrisierung alle Variablen bindet. Es wird keine entfernte Bindung an das „TestRule“-Interface mehr benötigt. Doch haben wir noch ein kleines Problem zu lösen. Angenommen, wir implementie-

```
abstract class FunctionalTestRule implements TestRule, OuterEvalu-
able {
    private TestRule rule = RuleFactory.makeRule(this);

    public Statement apply(Statement base, Description d) {
        return rule.apply(base, d);
    }
}
```

Listing 14

```
public abstract class FunctionalVerifier extends Functional-
TestRule {
    public void evaluate(Statement base, Description d) throws
Throwable {
        base.evaluate();
        verify();
    }

    // Override this to add verification logic. ...
    abstract void verify();
}
```

Listing 15

```
@Rule
public TestRule verifier = new FunctionalVerifier() {
    @Override
    public void verify() {
        assertTrue(errorLog.isEmpty());
    }
};
```

Listing 16

```
static TestRule makeRule(final
OuterEvaluable oe) {
    return new TestRule() {
        public Statement
apply(final Statement base,
final Description d) {
            return
makeStatement(new Evaluable()
{
                public void
evaluate() throws Throwable {

oe.evaluate(base, d);
                }
            });
        }
    };
}
```

Listing 17

ren die hier entwickelte Logik zur Erstellung von Testregeln in einer (hier nicht gezeigten) Basisklasse „SmartTestRule“, die das Interface „TestRule“ implementiert.

Wenn wir jetzt analog wie im JUnit-Framework einen „Verifier“ mit der Methode „verify“ erstellen, müssen wir ein „OuterEvaluable“ so an die „SmartTestRule“ übergeben, dass es auch Zugriff auf die Methode „verify“ hat. Die Übergabe über den Konstruktor funktioniert nicht, da im Konstruktor der Basisklasse die Methoden der abgeleiteten Klasse noch nicht bekannt sind. Also Übergabe über einen Setter der Basisklasse? Das führt dann zu der uneleganten Lösung (siehe Listing 13), die man sich wahrscheinlich auch nicht besser merken kann als die Konstruktion mit den verschachtelten anonymen Klassen.

Listing 14 zeigt einen anderen Ansatz. Die abstrakte Klasse „FunctionalTestRule“ implementiert die Schnittstellen „TestRule“ und „OuterEvaluable“. Die „TestRule“-Funktionalität wird komplett aus dem „OuterEvaluable“ abgeleitet. Dieses wird jedoch in dieser Basisklasse noch nicht implementiert. Der „FunctionalVerifier“ in Listing 15 wird von „FunctionalTestRule“ abgeleitet. Er bekommt kein „OuterEvaluable“ übergeben, er muss es implementieren. Listing 16 zeigt, dass der „FunctionalVerifier“ genauso verwendet werden kann wie der „Verifier“ aus dem JUnit-Framework.

Es ist bemerkenswert, dass in den letzten Listings nicht mehr so viele Lambda-Ausdrücke zu sehen sind. Das einzig verbliebene Lambda ist in der „RuleFactory“ zu finden. Dies lässt sich ohne Weiteres durch eine anonyme Klasse ersetzen, wie in Listing 17 zu sehen ist. Die hier gezeigte Lösung lässt sich also auch in Java 5 verwenden. Es sind lediglich die Annotationen „@FunctionalInterface“ zu entfernen.

Günter Jantzen
guenter.jantzen@capgemini.com

Günter Jantzen, Mathematiker, ist fast zwanzig Jahre unterwegs mit Java, Python, C++. Er entwickelt aktuell Tools für die QS von ETL-Prozessen beim Zoll. Günter Jantzen läuft gern um Maschsee und Alster, auch zum nächsten JUG-Treffen. Seit dem Startschuss im Jahr 2012 ist er aktiv in der JUG Hannover (siehe „www.jug-h.de“), einer ideenreichen Gruppe mit regionaler Verankerung.

http://ja.ijug.eu/14/4/4

Go for the Money – eine Einführung in JSR 354

Anatole Tresch, Credit Suisse

Der Java Specification Request (JSR) 354 befasst sich mit der Standardisierung von Währungen und Geldbeträgen in Java. Ziel ist es, das Java-Eco-System durch einen flexiblen und erweiterbaren API zu ergänzen, um das Arbeiten mit monetären Beträgen einfacher, aber auch sicherer zu machen. Es löst das Problem unterschiedlicher numerischer Anforderungen, definiert Erweiterungspunkte für weitergehende Funktionen und stellt auch Funktionen für die Währungsumrechnung und Formatierung zur Verfügung.

Der JSR 354 [1] wurde im Frühjahr 2012 gestartet, ein Jahr später kam der Early Draft Review heraus. Ursprünglich war die Absicht, den JSR in Java 8 als Teil der Plattform zu integrieren. Es hatte sich aber schnell gezeigt, dass dies aus verschiedenen Gründen kurzfristig nicht möglich war. Somit wird der JSR als eigenständige Spezifikation getrieben. Im Schatten von Java EE 7 und Java 8 wurde er kontinuierlich weiterentwickelt, sodass kürzlich der Public Review erfolgreich veröffentlicht werden konnte. Es ist geplant, den JSR bis Ende des Jahres zu finalisieren.

Ausgangslage und Motivation

Die meisten Java-Entwickler sind früher oder später in irgendeiner Form mit Geldbeträgen konfrontiert, da Geld doch ein weit verbreitetes Mittel darstellt. Somit dürfte den meisten auch die Klasse „java.util.Currency“ im JDK ein Begriff sein. Diese ist seit Java 1.4 Bestandteil der Plattform und modelliert Währungen auf Basis des ISO-4217-Standards [2].

Darüber hinaus unterstützt „java.text.DecimalFormat“ die Formatierung von Geldbeträgen. Wer jedoch etwas genauer hinschaut, wird schnell merken, dass viele Anwendungsfälle nur ungenügend behandelt sind. So enthält die Klasse „java.util.Currency“ nicht alle ISO-Währungen; es fehlen die Codes „CHE“ (Schweizer Franken Euro) und „CHW“ (Schweizer Franken WIR). Die unterschiedlichen Codes für US-Dollar („USD“, „USS“ und „USN“) sind im Gegensatz dazu vorhanden. Seit Java 7 ist es mit vertretbarem Aufwand möglich, weitere Währungen hinzuzufügen, es muss dazu jedoch eine sogenannte „Java Extension“ implementiert werden, ein Java-Archiv, das in den „lib“-Ordner der JRE zu installieren ist – für viele Unternehmungen kein akzeptabler Weg.

Zudem ist es nicht möglich, Anwendungsfälle wie Mandantenfähigkeit, virtuelle Währungen oder kontext-abhängiges Verhalten im Java-EE-Umfeld zu modellieren. Auch die Formatierung von Geldbeträgen lässt zu wünschen übrig. Die Klasse „java.text.DecimalFormat“ bietet zwar viele grundlegende Funktionen an, leider sind diese für viele Anwendungen in der Finanzwelt zu starr, da sie nur eine Unterscheidung in Patterns für positive und negative Zahlen kennt. Viele Anwendungen erfordern jedoch dynamisches Verhalten, etwa in Abhängigkeit von der Größe des Geldbetrags. Es lassen sich auch keine unterschiedlichen Gruppierungen definieren, wie sie zum Beispiel bei Indischen Rupien nötig wären (wie INR 12,23,123.34). An die fehlende Thread-Sicherheit der For-

```
public interface CurrencyUnit {
   public String getCurrencyCode();
   public int getNumericCode();
   public int getDefaultFractionDigits();
}
```

Listing 1

```
ISO:CHF    // ISO mit Gruppierung
CS:23345   // Eigener Namespace mit Gruppierung
23345-1    // Eigener Code
```

Listing 2

```
CurrencyUnit eur = MonetaryCurrencies.getCurrency("EUR");
Set<CurrencyUnit> usCurrencies =
                  MonetaryCurrencies.getCurrencies(Locale.US);
```

Listing 3

```
Set<CurrencyUnit> currencies =  MonetaryCurrencies
   .getCurrencies(new CurrencyContext()
                      .setAttribute("continent", "Europe")
                      .setInt("year",1970).create());
```

Listing 4

matklassen haben sich die meisten Nutzer schon fast gewöhnt.

Eine Abstraktion für Geldbeträge fehlt komplett. Natürlich kann man „BigDecimal" verwenden, aber jeder Entwickler muss selbst dafür sorgen, dass die Währung immer mit ihrem Betrag transportiert wird. Sollte sich jemand auf „double" als numerischen Typ stützen, hat das fehlerhafte Resultate zur Folge, da sich die Fließkomma-Arithmetik nicht für Finanzanwendungen eignet [3]. Dass weitergehende Funktionen wie Währungsumrechnung und finanzmathematisches Runden ebenfalls fehlen, ist letztendlich nur eine logische Konsequenz.

Modellierung von Währungen

Die bestehende Klasse „Currency" basiert auf ISO 4217 [2]. Im JSR ist intensiv diskutiert worden, welche Eigenschaften zusätzlich notwendig sind, um auch soziale und virtuelle Währungen unterstützen zu können. Zudem hat ISO 4217 selbst Limitierungen: Einerseits hinkt ein Standard typischerweise eine gewisse Zeit hinter der Realität her, andererseits gibt es auch Mehrdeutigkeiten (der Code CFA gilt für zwei Länder) oder nicht modellierte Aspekte (unterschiedliches Runden, Legal Tender, historische Währungen). Hinzu kommt noch, dass nicht benötigte Codes nach einer gewissen Zeit neu vergeben werden dürfen. Dennoch hat sich gezeigt, dass sich die Essenz einer Währung mit nur drei Methoden abbilden lässt. Diese sind bereits in der bestehenden Currency-Klasse vorhanden (siehe Listing 1).

Im Gegensatz zur aktuellen Klasse „Currency" ist der Währungscode für Nicht-ISO-Währungen frei wählbar. Somit können auch eigene Code-Systeme integriert oder Codes gruppiert werden. Listing 2 zeigt alle möglich Codes. Währungen selbst lassen sich vom „MonetaryCurrencies"-Singleton analog wie bei „java.util.Currency" beziehen (siehe Listing 3).

Bemerkenswert ist der Unterschied zur bisherigen Klasse „Currency", in der immer nur eine Währung für eine „Locale" zurückgegeben wird. Das API von JSR 354 unterstützt auch wesentlich komplexere Szenarien, in denen ein sogenannter „CurrencyContext" übergeben werden kann. Ein solcher Kontext lässt sich mit beliebigen Attributen jeglichen Typs konfigurieren. Somit könnten beispielsweise alle europäischen Währungen, die im Jahre 1970 gültig waren, abgefragt werden (siehe Listing 4).

Im Hintergrund arbeitet ein zugehöriges SPI, wobei mehrere Provider entsprechend passende Währungen liefern können. Die Abfragemöglichkeiten hängen dabei einzig von den registrierten Providern ab. Für weitere Details ist auf die Dokumentation [4] sowie die Referenz-Implementation [5] verwiesen.

Modellierung von monetären Beträgen

Intuitiv würde man wohl einfach eine Währung mit einem numerischen Wert, etwa „BigDecimal", zu einem neuen „immutable" Wert-Typ verbinden. Leider hat sich jedoch gezeigt, dass dieses Modell an der enormen Bandbreite von Anforderungen an einen Betrags-Typ scheitert. Ein idealer Typ muss extrem kurze Rechenzeiten bei geringem Speicherverbrauch garantieren (Handel), dafür können aber gewisse Abstriche bei den numerischen Fähigkeiten in Kauf genommen werden (wie Nachkommastellen). Produkt-Rechnungen benötigen oft sehr hohe Genauigkeiten, wobei die Laufzeit weit weniger kritisch ist. Risiko-Berechnungen oder Statistiken hingegen produzieren sehr große Zahlen.

Zusammenfassend wird es kaum möglich sein, alle Anforderungen mit nur einem einzigen Implementations-Typ abzudecken. Aus diesem Grunde hat man sich im JSR dazu entschieden, mehrere Implementationen gleichzeitig in einem System zu unterstützen. Dabei wird ein Betrag durch das „MonetaryAmount"-Interface modelliert. Interoperabilitäts-Regeln verhindern das unerwartete Auftreten von Rundungsfehlern. Damit diese sinnvoll implementiert werden können, hat der Geldbetrag nebst Währung und numerischem Wert einen sogenannten „MonetaryContext" erhalten. Dieser kann die numerischen Grenzen einer Implementation (Präzision und Skalie-

rung) sowie weitere Eigenschaften aufnehmen (siehe Listing 5).

Der numerische Wert eines „MonetaryContext“ wird als „javax.money.NumberValue“ zurückgegeben. „NumberValue“ erweitert „java.lang.Number“ um weitere Funktionen, etwa um den numerischen Wert korrekt und verlustfrei exportieren zu können. Zudem lohnt sich auch ein Blick auf die beiden Methoden „with“ und „query“, die Erweiterungspunkte definieren, um zusätzliche externe Funktionen wie Runden, Währungsumrechnung oder Prädikate mit einem Geldbetrag zu kombinieren. „MonetaryAmount“ bietet auch Operationen, um Beträge miteinander zu vergleichen oder arithmetische Operationen analog zu „BigDecimal“ anzuwenden. Schließlich stellt jeder Betrag eine „MonetaryAmountFactory“ zur Verfügung, die auf Basis des gleichen Implementations-Typs beliebige weitere Beträge erzeugen kann.

```
public interface MonetaryAmount extends CurrencySupplier, Number-
Supplier, Comparable<MonetaryAmount> {
  CurrencyUnit getCurrency();
  NumberValue getNumber();
  MonetaryContext getMonetaryContext();

  <R> R query(MonetaryQuery<R> query){
  MonetaryAmount with(MonetaryOperator operator);
  MonetaryAmountFactory<? extends MonetaryAmount>
                                        getFactory();
  …
}
```

Listing 5

```
MonetaryAmount amt = MonetaryAmounts.getAmountFactory()
                      .setCurrency(“EUR”)
                      .setNumber(200.5).create();
```

Listing 6

```
MonetaryAmountFactory<?> factory = MonetaryAmounts
  .queryAmountFactory(new MonetaryContext.Builder()
             .setPrecision(200).setMaxScale(10).build());
```

Listing 7

```
@FunctionalInterface
public interface MonetaryOperator
extends UnaryOperator<MonetaryAmount> {}
```

Listing 8

Erzeugen von monetären Beträgen

Geldbeträge werden mithilfe einer „MonetaryAmountFactory“ erzeugt. Neben einem konkreten Betrag sind „Factory“-Instanzen auch über das „MonetaryAmounts“-Singleton möglich. Im einfachsten Fall holt man eine „default“-Factory und baut den Betrag zusammen (siehe Listing 6).

Zudem lässt sich eine „MonetaryAmountFactory“ direkt adressieren, indem man den gewünschten Implementationstyp als Parameter an „MonetaryAmounts“ übergibt. Ein anderes Feature bietet den Bezug einer Factory aufgrund der benötigten numerischen Anforderungen. Diese können mit einer Instanz von „MonetaryContext“ definiert werden, um eine geeignete Implementation abzufragen oder direkt eine passende Factory zu erzeugen (siehe Listing 7).

Runden von monetären Beträgen

Jedem „MonetaryAmount“ kann eine Instanz eines „MonetaryOperator“ übergeben werden, um beliebige weitere externe Funktionen auf einem Betrag auszuführen (siehe Listing 8). Dieser Mechanismus wird auch für das Runden von Geldbeträgen benutzt. Dabei sind folgende Arten des Rundens berücksichtigt:

- *Internes Runden*
  Geschieht implizit aufgrund des numerischen Modells des benutzten Implementations-Typs. Wenn eine Implementation definiert, dass sie eine Skalierung von maximal fünf Stellen nach dem Komma unterstützt, darf sie das Resultat einer Division durch sieben auf genau diese fünf Stellen runden.
- *Externes Runden*
  Diese Funktion wird benötigt, um einen numerischen Wert aus „NumberValue“ in eine numerische Repräsentation zu exportieren, die weniger numerische Fähigkeiten besitzt. So darf ein Betrag von „255.15“ auf „255“ gerundet werden, wenn der Wert als „Byte“ exportiert wird (was allerdings zu hinterfragen ist).
- *Formatierungsrunden*
  Hier wird ein Betrag auf eine arbiträre andere Form gerundet, beispielsweise kann CHF „2’030’043“ einfach als „CHF > 2 Mio“ angezeigt werden.

Grundsätzlich ist internes (beziehungsweise implizites) Runden nur in wenigen Fällen erlaubt, ähnlich dem oben erwähnten. Alle anderen Rundungsvarianten sind hingegen explizit, der Benutzer wendet diese also aktiv an. Dies ergibt Sinn, da je nach Anwendungsfall zu unterschiedlichen Zeitpunkten und in verschiedener Weise gerundet werden muss. Somit hat der Entwickler maximale Kontrolle über das Runden.

Unterschiedliche Rundungen können über das „MonetaryRoundings“-Singleton bezogen werden, etwa Rundungen für Währungen oder auch passend zu einem „MathContext“. Analog zu anderen Bereichen kann für komplexe Fälle auch ein „RoundingContext“ übergeben werden. Ein Beispiel ist das Runden von CHF bei Barzahlungen in der Schweiz. Die kleinste Münzeinheit sind fünf Rappen, somit ist

```
MonetaryOperator rounding = MonetaryRoundings.getRounding(
         new RoundingContext.Builder()
          .setCurrency(MonetaryCurrencies.getCurrency("CHF"))
          .setAttribute("cashRounding", true).build());
```

Listing 9

```
MonetaryAmount amount = …;
MonetaryAmount roundedCHFCashAmount = amount.with(rounding);
```

Listing 10

```
CurrencyConversion conversion = MonetaryConversions
                                    .getConversion("USD");
ExchangeRateProvider prov = MonetaryConversions
                                .getExchangeRateProvider();
```

Listing 11

```
MonetaryAmount amountCHF = …;
MonetaryAmount convertedAmountUSD = amount.with(conversion);
```

Listing 12

```
public interface MonetaryAmountFormat
extends MonetaryQuery<String>{
  AmountFormatContext getAmountFormatContext();
  String format(MonetaryAmount amount);
  void print(Appendable appendable, MonetaryAmount amount)
    throws IOException;
  MonetaryAmount parse(CharSequence text)
    throws MonetaryParseException;
}
```

Listing 13

```
MonetaryAmountFormat fmt =
             MonetaryFormats.getAmountFormat(Locale.US);

DecimalFormatSymbols symbols = …;
MonetaryAmountFormat fmt =
  MonetaryFormats.getAmountFormat(
    new AmountFormatContext.Builder(Locale.US)
      .setObject(symbols).build());
```

Listing 14

auf eben diese fünf Rappen zu runden. Listing 8 zeigt, wie auch dies mit dem API gelöst werden kann (siehe Listing 9). Die Rundung selbst kann sehr einfach auf jeden Betrag angewendet werden (siehe Listing 10).

Währungsumrechnung

Herzstück einer Währungsumrechnung bildet die „ExchangeRate". Diese beinhaltet nebst den beteiligten Quell- und Zielwährungen auch den Umrechnungsfaktor und weitere Referenzen. Auch mehrstufige Umrechnungen (wie „Triangular Rates") sind unterstützt. Eine „Rate" ist dabei immer unidirektional. Die „ExchangeRate" wird schließlich von einem sogenannten „ExchangeRateProvider" geliefert. Dabei ist optional ein „ConversionContext" möglich, um die gewünschte Umrechnung weiter zu konfigurieren.

Die Währungsumrechnung selbst wird als „CurrencyConversion" modelliert, die „MonetaryOperator" erweitert sowie an einen „ExchangeRateProvider" und eine Zielwährung gebunden ist. Beide Interfaces lassen sich über das „MonetaryConversions"-Singleton beziehen (siehe Listing 11).

Zusätzlich kann beim Bezug einer „CurrencyConversion" oder eines „ExchangeRateProvider" auch die gewünschte Kette von Providern dynamisch konfiguriert und ein „ConversionContext" übergeben werden, um die entsprechenden Provider zu konfigurieren. Die Anwendung einer Konversion funktioniert analog wie das Runden (siehe Listing 12).

Die aktuelle Referenz-Implementation bringt zwei vorkonfigurierte Provider, die Umrechnungsfaktoren auf Basis der öffentlichen Datenfeeds der Europäischen Zentralbank und des Internationalen Währungsfonds zur Verfügung stellen. Für gewisse Währungen reichen diese zurück bis in das Jahr 1990.

Formatierung

Beim Formatieren von Beträgen wurde ebenfalls darauf geachtet, das API möglichst einfach und flexibel zu halten (siehe Listing 12). Analog zu den anderen Bereichen lassen sich entsprechende Instanzen über ein „MonetaryFormats"-Singleton beziehen (siehe Listing 13). Auch hier gibt es die Möglichkeit, einen „AmountFormatContext" zu übergeben, der beliebige Attribute zur Kontrolle der Formatierung erhalten kann (siehe Listing 14). Somit lassen sich Formate beispielsweise mit Namen oder „Enum"-Typen identifizieren und beliebig konfigurieren. Die Fähigkeiten der registrierten SPI-Instanzen legen dabei fest was alles möglich ist.

SPIs

Zusätzlich zum beschriebenen API definiert der JSR auch ein komplettes SPI, mit dem sich die gesamte angebotene Funk-

tionalität an die konkreten Bedürfnisse anpassen lässt. So können zusätzliche Währungen, Konversionen, Rundungen, Formate oder Implementationen für Beträge ergänzt oder das Komponenten-Bootstrapping angepasst werden.

Fazit

JSR 354 definiert ein übersichtliches und zugleich mächtiges API, das den Umgang mit Währungen und Geldbeträgen stark vereinfacht, aber auch die teilweise widersprüchlichen Anforderungen. Auch fortgeschrittene Themen wie Runden und Währungsumrechnung sind adressiert und es ist möglich, Geldbeträge beliebig zu formatieren. Die bestehenden Erweiterungspunkte schließlich erlauben es, weitere Funktionen elegant zu integrieren. Diesbezüglich lohnt sich auch ein Blick in das OSS-Projekt [6], wo neben finanzmathematischen Formeln auch eine mögliche Integration mit CDI experimentell zur Verfügung steht.

JSR 354 sollte bis spätestens Ende des Jahres finalisiert sein. Für Anwender, die noch mit Java 7 arbeiten, wird zusätzlich ein vorwärtskompatibler Backport zur Verfügung stehen.

Weiterführende Links

[1] https://jcp.org/en/jsr/detail?id=354, https://java.net/projects/javamoney
[2] http://www.iso.org/iso/home/standards/currency_codes.htm
[3] http://stackoverflow.com/questions/3730019/why-not-use-double-or-float-to-represent-currency
[4] https://github.com/JavaMoney/jsr354-api
[5] https://github.com/JavaMoney/jsr354-ri
[6] http://javamoney.org

Anatole Tresch
anatole.tresch@credit-suisse.com

Nach dem Wirtschaftsinformatik-Studium an der Universität Zürich war Anatole Tresch mehrere Jahre lang als Managing-Partner und Berater tätig. Er sammelte weitreichende Erfahrungen in allen Bereichen des Java-Ökosystems vom Kleinunternehmen bis zu komplexen Enterprise-Systemen. Schwerpunkte sind verteilte Systeme sowie die effiziente Entwicklung und der Betrieb von Java EE in der Cloud. Aktuell arbeitet Anatole Tresch als technischer Architekt und Koordinator bei der Credit Suisse. Daneben ist er Specification Lead des JSR 354 (Java Money & Currency) und Co-Spec Lead beim Java EE Configuration JSR.

http://ja.ijug.eu/14/4/5

Scripting in Java 8

Lars Gregori, msgGillardon AG

Die Java Virtual Machine (JVM) unterstützt neben unterschiedlichen Programmiersprachen seit Längerem auch die Integration von Skriptsprachen. Mit Java 8 und der komplett überarbeiteten JavaScript-Engine „Nashorn" geht die JVM einen Schritt weiter in Richtung „Multi-Sprachen-Unterstützung". Für den Entwickler und Architekten lohnt es sich jetzt, sich mit dem Thema „Skriptsprachen" und der Motivation für deren Verwendung zu befassen. Die bisherige Performance galt für viele als Hinderungsgrund – Java 8 und die Einführung der Lambda-Ausdrücke nehmen hier positiven Einfluss.

Skriptsprachen werden für kleine und überschaubare Programme benutzt. Die Syntax der Programmiersprache verzichtet meistens auf bestimmte Sprachelemente wie zum Beispiel die der Variablen-Deklaration. Man spricht in diesem Fall von einer dynamischen Typisierung. Der Typ wird zur Laufzeit an den Wert gehängt. Dadurch lassen sich Prototypen und kleine Programme schnell erstellen. Java hingegen ist eine stark statisch typisierte Sprache. Der Compiler überprüft den Typ der Variable und somit benötigt jede Variable einen Typ. Zudem lassen sich lediglich Werte von diesem Typ zuweisen. Schwach typisierte Sprachen hingegen erlauben eine implizite Umwandlung unterschiedlicher Datentypen.

Die Betrachtung der daraus resultierenden Code-Qualität ist eher subjektiv. Skriptsprachen sind kompakter und lesbarer, da es weniger Codezeilen gibt. Zudem entfällt der zusätzliche Aufwand der Typ-Deklaration. Ob man sich der Argumentation hingeben soll, dass weniger Code und weniger Komplexität zu weniger Fehlern führen? Andersherum werden durch die Typ-Deklarationen zwar Schreibfehler beim Kompilieren des Quellcodes entdeckt, dadurch wird allerdings die Qualität der Anwendung nicht zwangsläufig besser. Qualität entsteht hauptsächlich durch Tests. Deren Erstellung ist weder durch eine dynamische noch durch eine statische Typisierung eingeschränkt. Dabei können aber die kleinen und kompakten

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("rhino");
Integer result = (Integer) engine.eval("6 * 7");
System.out.println(result);
```

Listing 1

Code-Stücke einer dynamischen Sprache das Schreiben von Tests fördern.

Bei der Performance sieht die Sache klarer aus. Java-Code, der als Byte-Code kompiliert ist, ist schneller als Skript-Code, der in Java ausgeführt wird. Jedoch hat die Einführung von „invokedynamic" in Java 7 zu einer deutlichen Verbesserung geführt. Dazu später.

Warum Skriptsprachen?

In „The Pragmatic Programmer" wird empfohlen, dass man jedes Jahr eine neue Sprache lernt. Skriptsprachen können etwas Neues und Anderes sein und auch Spaß machen. Mit ihnen lassen sich Programmteile in Konfigurationsdateien und Datenbanken auslagern und Anpassungen könnten zum Beispiel durch den Fachbereich erfolgen. Dadurch kann sich auch das Deployment der Anwendung erleichtern, indem sie bei kleineren Regeländerungen nicht komplett neu ausgerollt werden muss. Trotzdem sollte ein Prozess hinterlegt werden, der sicherstellt, dass Änderungen auf dem Produktivsystem nicht zu dessen Ausfall führen.

Trotz der vielen Java-Bibliotheken bieten verschiedene Programmiersprachen im Gegensatz zu Java die passendere Lösung zu einem Problem. Jython, die Java-Version von Python, und Sleep (Perl) haben ihre Stärke in der String-Manipulation. ABCL (Common Lisp) und Clojure bringen die „map"- und „reduce"-Methoden mit, die mit den Lambda-Ausdrücken nun allerdings auch in Java 8 Einzug halten. Groovy hat mit GroovySQL und GroovyMarkup eigene Klassen für die SQL- und HTML-Erzeugung.

Skriptsprachen bieten auch eine Vielzahl von Bibliotheken an, die verschiedene Aufgaben in bestehenden Anwendungen lösen. In einem nachfolgenden Beispiel wird auf das Template-Framework Mustache eingegangen. Man sollte aber trotzdem bedenken, dass Skriptsprachen mächtig sind und prinzipiell auf alles zugreifen können.

In Web- und Browser-Anwendungen ist JavaScript im Frontend als Standard gesetzt. Durch die Integration ins Backend, die einer der Gründe für den Erfolg von „Node.js" ist, erfordert die Frontend- und Backend-Entwicklung keine Trennung anhand der benutzten Programmiersprachen. Der Frontend-Entwickler kann Aufgaben aus dem Backend übernehmen, ohne dass er eine neue Sprache lernen muss. So lässt sich zum Beispiel die im Browser für die Überprüfung der Eingabefelder verwendete JavaScript-Datei auch im Backend nutzen. Änderungen bei der Eingabe-Validierung müssen nicht in Java nachgezogen werden. Es wird auch nicht automatisch das Backend vergessen, da zum Beispiel in einer Anforderungsänderung für die Altersprüfung nur von der Oberfläche gesprochen wird.

Skriptsprachen lassen sich auch in den Build- und Test-Prozess einbinden. Hier werden die notwendigen Bibliotheken meist einfacher hinzugefügt. Für die Verwendung in der eigentlichen Anwendung können bestimmte Freigaberegeln gelten.

Scripting in Java

Für die Integration von Skriptsprachen gibt es verschiedene Möglichkeiten. Groovy bietet zum einen mit „groovyc" einen Compiler, um „groovy"-Dateien in „class"-Dateien zu kompilieren. Auf diese Klassen kann wie in Java zugegriffen werden. Typenlose Rückgabewerte müssen aber in Java in einen entsprechenden Java-Typ umgewandelt werden. Zudem implementiert die kompilierte Klasse das „GroovyObject"-Interface, wodurch weiterhin eine Abhängigkeit zu Groovy bestehen bleibt.

Eine weitere Möglichkeit ist eine eigene Skript-Engine der Skript-Sprachen. Die „GroovyShell"-Klasse ermöglicht es, durch die „evaluate"-Methode Groovy-Skripte auszuführen. Java- und Groovy-Variablen können mit der „Binding"-Klasse gebunden werden. Dadurch lassen sich Werte in beide Richtungen austauschen.

Das „Bean Scripting"-Framework (BSF), ein Apache-Projekt, definiert eine Schnittstelle für Skriptsprachen, um in Java ausge-

führt zu werden und auf Java-Objekte und -Methoden zuzugreifen. Dabei wird beim „eval"-Methodenaufruf der „BSFManager"-Klasse der Name der Skript-Engine übergeben. Dieser ist innerhalb der BSF-Umgebung registriert und ruft dann die entsprechende Klasse mit dem übergebenen Skript auf.

Der Java Community Process, der Java Specification Requests (JSR) definiert, hat mit dem JSR 223 (Scripting for the Java Platform) ein ähnliches Prinzip wie BSF in Java integriert. Dadurch steht ab Java 6 das „javax.script"-Package zur Verfügung. Der „ScriptEngineManager" liefert mit der „getEngineByName"-Methode die entsprechende „ScriptEngine" zurück. Diese kann dann mit „eval" das Skript ausführen. Sowohl Skript- als auch Java-Variablen können gebunden werden und stehen beiden Seiten zur Verfügung. Die Java-Integration bietet auch für JVM-Skriptsprachen die Möglichkeit, auf andere JVM-Skriptsprachen direkt zuzugreifen. So kann zum Beispiel ein Groovy-Skript JavaScript-Code ausführen und ebenfalls auf die Variablen zugreifen.

Wer mehr über Scripting in Java erfahren möchte, dem sei das Buch „Scripting in Java: Languages, Frameworks, and Patterns" von Dejan Bosanac (ISBN 978-0321321930) empfohlen. Nachfolgend ist ein JSR-223-Beispiel aufgeführt, das eine Berechnung in JavaScript ausführt (siehe Listing 1).

Das Beispiel verwendet „Rhino" als Engine. Dabei handelt es sich um die in Java 6 integrierte JavaScript-Engine. Sie wurde in Java 8 komplett überarbeitet und heißt jetzt „Nashorn".

Nashorn

Die JavaScript-Engine „Nashorn" wurde im ECMAScript-Standard (siehe „http://www.ecma-international.org/ecma-262/5.1") implementiert. Der Unterschied zwischen ECMAScript und JavaScript besteht darin, dass JavaScript eine Implementierung des ECMAScript-Standards ist.

Nashorn ist die Referenz-Implementierung für den JSR 292 (Supporting Dynamically Typed Languages on the Java Platform), um dynamische Sprachen auf der Java-Plattform zu unterstützen. Der Weg dorthin wurde bereits mit Java 7 gelegt, indem mit der JVM der Byte-Code-Befehl „invokedynamic" eingeführt wurde. Dabei handelt es sich um eine Erweiterung der bereits bestehenden Invocation-Operatoren. So wird zum Beispiel „invokestatic" für statische Methodenaufrufe und „invokevirtual" für Methodenaufrufe einer Instanz-Klasse verwendet.

Im Gegensatz zu „invokedynamic" prüft der Compiler die Verwendung des richtigen Daten-Typs und bindet die entsprechenden Methodenaufrufe fest im erzeugten Byte-Code. Bei dynamisch typisierten Sprachen hingegen ermittelt „invokedynamic" den Typ zur Laufzeit und bindet die entsprechende Klasse und Methode dynamisch.

Beispiel „Mustache"

Mustache (siehe „http://mustache.github.io") ist ein Template-System, das für unterschiedliche Sprachen zur Verfügung steht. Da die Syntax auf „if" else"-Ausdrücke und Schleifen verzichtet, bezeichnet sich Mustache als „logic-less" Template-System. Der Name „Mustache", zu deutsch Oberlippenbart oder Schnauzbart, rührt daher, dass geschweifte Klammern für die Platzhalter verwendet werden. Dreht man die öffnende Klammer um 90 Grad, ähnelt dies einem Mustache. Das folgende Beispiel Template (template.mst) zeigt zu einem Kontakt den Namen und die dazugehörigen Telefonnummern (siehe Listing 2). Die Kontaktdaten sind als JSON-Daten in diesem Format hinterlegt (siehe Listing 3).

Wie im obigen Beispiel ermittelt der „ScriptEngineManager" die „ScriptEngine". Diese ist in dem Fall eine „NashornScript Engine"-Instanz, sie implementiert das „javax.script.Invocable"-Interface und stellt die Methode „invokeMethod" zur Verfügung. Zunächst wird das Template eingelesen und dann die Kontaktdaten. Diese liegen im JSON-Format in der Datei „contact.json" vor und lassen sich über das

```
Telefonnummer von {{contact.
name}}:
{{#contact.phone_numbers}}
* {{type}}: {{number}}
{{/contact.phone_numbers}}
```

Listing 2

```
{
 "contact":{
   "name":"Lars Gregori",
   "phone_numbers":[
     {
       "type":"daheim",
       "number":"+49 555 5555"
     },
     {
       "type":"privat",
       "number":"+49 555 2309"
     }
   ]
 }
}
```

Listing 3

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine nashorn = manager.getEngineByName("nashorn");
Invocable invocable = (Invocable) nashorn;

String template = new String(
         Files.readAllBytes(
                     Paths.get("template.mst")));

String contactJson = new String(
         Files.readAllBytes(
                     Paths.get("contact.json")));
Object json = nashorn.eval("JSON");
Object data = invocable.invokeMethod(
                     json, "parse", contactJson);

nashorn.eval(new FileReader("src/mustache.js"));
Object mustache = nashorn.eval("Mustache");

Object output = invocable.invokeMethod(
         mustache, "render", template, data);
System.out.println(output);
```

Listing 4

```
Telefonnummer von Lars Gregori:
* daheim: +49 555 5555
* privat: +49 555 2309
```

Listing 5

```
List<Object> someObjects = new ArrayList<Object>();
someObjects.add(22);
someObjects.add(„hallo“);
someObjects.forEach(obj -> System.out.println(obj));
```

Listing 6

```
...
30: invokedynamic #36,  0
// InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
...
```

Listing 7

Nashorn-interne JSON-Objekt in Objektdaten umwandeln. Dazu wird mit der „eval“-Methode das interne JSON-Objekt geholt und der „invokeMethode“ übergeben. Als zweiter Parameter des „invokeMethode“-Aufrufs wird der aufzurufende Methoden-Name angegeben. In diesem Fall ist es die „parse“-Methode. Als Parameter für den „parse“-Aufruf werden die eingelesenen Kontaktdaten als String mitgegeben (siehe Listing 4).

Die „eval“-Methode liest die eigentliche Mustache-JavaScript-Datei „mustache.js“ (siehe „https://github.com/janl/mustache.js“) ein und führt sie intern aus. Durch den „eval(„Mustache“)“-Aufruf steht das Mustache-Objekt zur Verfügung. An diesem kann wieder mithilfe von „invokeMethod“ die „render“-Methode mit den Template- und Kontaktdaten als Parameter aufgerufen werden. Das „render“-Ergebnis lässt sich dann weiterverwenden. In diesem Fall wird es mit „println“ ausgegeben (siehe Listing 5).

Nashorn intern

Der Nashorn-Parser parst zunächst den JavaScript-Code und baut einen internen Baum auf, der das Programm repräsentiert. Innerhalb des Baums werden Ausdrücke durch Java-Aufrufe ersetzt und Aufrufe vereinfacht. Für die Daten werden die Verwendung, der Speicher und der Typ ermittelt. Im nächsten Schritt erzeugt der Code-Generator über die ASM-Bibliothek (siehe „http://asm.ow2.org“) Byte-Code. Dieser bleibt dann wie bei einem kompilierten Java-Programm im Speicher, wird mit dem Class Loader geladen und steht der Anwendung zur Verfügung. Er verhält sich wie kompilierter Java-Code, den die JVM automatisch optimieren kann.

Lambda-Ausdrücke

Das in Java 7 eingeführte „invokedynamic“ war zunächst nur eine Erweiterung der JVM für Skriptsprachen. Java als Sprache profitiert seit Java 8 mit den neu eingeführten Lambda-Ausdrücken auch davon. Dabei handelt es sich um anonyme Methoden, deren Typ erst zur Laufzeit ermittelt wird und die somit bestens für „invokedynamic“ geeignet sind (siehe Listing 6).

Im Beispiel wird eine Liste von unterschiedlichen Objekten angelegt. Mit der in Java 8 neu hinzugekommenen „forEach“-Methode wird über die einzelnen Elemente der Liste gegangen und an einen Lambda-Ausdruck übergeben. Die Syntax hierfür besteht aus einer Parameterliste, einem Pfeil-Operator und einem Ausdruck, der auf die Parameter zugreifen kann. Im Beispiel wird das Element als „obj“-Parameter übergeben und mit der „println“-Methode ausgegeben.

Mit dem im JDK enthaltenen Kommandozeilen-Tool „javap“ lässt sich eine kompilierte Klasse „disassembeln“ und deren Bytecode betrachten. Im Listing 7 ist ein stark gekürzter Ausschnitt des Lambda-Beispiels, das mit „javap -c -v SomeObjects.class“ aufgerufen wurde und in dem der „invokedynamic“-Aufruf zu sehen ist.

Fazit

Die JVM entwickelt sich mehr und mehr zu einer Multi-Sprachen-Umgebung, in der dynamische Skriptsprachen ihre Unterstützung finden. Zudem wurde in Java 8 mit den Lambda-Ausdrücken ein neues Sprachelement aus der funktionalen Programmierung übernommen. Der Schlüssel hierzu war die Einführung von „invokedynamic“ und die damit verbundene Steigerung der Laufzeit. Dabei handelt es sich aber nur um den ersten Schritt des noch von Sun gegründeten „Da Vinci Machine“-Projekts (siehe „http://openjdk.java.net/projects/mlvm“). Die Idee dahinter ist, dass die JVM den Entwickler einer Programmiersprache unterstützt, um bestimmte Konstrukte der Programmiersprache nicht mehr umständlich nachbauen zu müssen. Hier war „invokedynamic“ ein erster Schritt, um dynamische Aufrufe zur Laufzeit zu binden und so die Performance zu steigern. Mit Nashorn ist damit eine konkurrenzfähige JavaScript-Engine entstanden. Dies ermöglicht es, „Node.js“-Anwendungen mit „Avatar.js“ (siehe „https://avatar-js.java.net“) in Java integrieren zu können und somit die JVM weiter in Richtung „Multi-Sprachen-Umgebung“ zu bringen.

Lars Gregori
lars.gregori@msg-gillardon.de

Lars Gregori ist Lead IT Consultant bei der msgGillardon AG in Ismaning/München. Als Software-Architekt beschäftigt er sich im JEE- und mobilen Bereich für Finanzunternehmen. Im Center of Competence für IT-Architekturen hat er Technologien von heute und morgen im Fokus. Zudem gilt sein Interesse unterschiedlichen Programmiersprachen.

http://ja.ijug.eu/14/4/6

JGiven: Pragmatisches Behavioral-Driven-Development für Java

Dr. Jan Schäfer, TNG Technology Consulting GmbH

JGiven ist ein Open-Source -Tool für Java, das Test-Szenarien in einer Java-DSL anstatt in Text-Dateien schreibt. JGiven erzeugt daraus eine Verhaltensdokumentation, die dann von der Fachabteilung beurteilt und mit den Anforderungen verglichen werden kann. Szenarien in JGiven sind modular aufgebaut, wodurch eine hohe Wiederverwendbarkeit von Test-Code entsteht.

Behavioral-Driven-Development (BDD) ist eine Testmethode, bei der automatisierte Tests in der Sprache des jeweiligen Fachgebiets der Anwendung geschrieben werden. Dadurch können BDD-Tests als Spezifikation und Dokumentation an die Fachabteilung gegeben und dort mit den Anforderungen abgeglichen werden. Klassische BDD-Tools für Java wie JBehave [1] oder Cucumber [2] formulieren Test-Szenarien in einfachen Text-Dateien. Dadurch soll es der Fachabteilung sogar möglich sein, automatisierte Tests selbst zu schreiben. Obwohl dies natürlich ideal wäre, sind es am Ende jedoch oft die Entwickler, die diese Text-Dateien schreiben und warten müssen.

Szenarien in Text-Dateien zu schreiben, hat jedoch diverse Nachteile. Erstens stehen einem nicht die bekannten Hilfsmittel einer modernen Entwicklungsumgebung (IDE) zur Verfügung, etwa automatische Vervollständigung, Typsicherheit und Refactoring. Plug-ins liefern zwar zumindest Autovervollständigung und Syntax-Highlighting nach, kommen aber ansonsten nicht an den Komfort heran, den man von einer Java-IDE gewöhnt ist. Zweitens ist man in der Text-Sprache gefangen und kann nicht auf die Mächtigkeit einer vollen Programmiersprache zurückgreifen. Dies zwingt oft dazu, Szenarien per „Copy & Paste“ zu erstellen, was die Wartbarkeit erheblich erschwert. Drittens müssen die Schritte der jeweiligen in Textform geschriebenen Szenarien durch reguläre Ausdrücke an ausführbaren Java-Code gebunden werden. Dies stellt in der Praxis einen nicht unerheblichen Aufwand dar.

Da nicht selten der Entwicklungs- und Wartungsaufwand von automatisierten Tests den des eigentlichen Produktiv-Codes übersteigt, bedeutet jeder zusätzliche Aufwand, der für Tests erforderlich ist, erhebliche Zusatzkosten für die gesamte Entwicklung. Schlussendlich führt dies oft dazu, dass BDD ganz aufgegeben wird. Aus diesen Gründen hat der Autor JGiven entwickelt. Es legt den Fokus auf den Entwickler und macht es so einfach wie möglich, BDD-Tests zu schreiben und zu warten.

Prinzipien

JGiven verabschiedet sich von der Idee, Szenarien in Text-Dateien zu formulieren. Stattdessen werden diese in einer Java-DSL geschrieben. Sämtliche der oben genannten Nachteile entfallen dadurch, da bei der Szenario-Erstellung auf die volle Mächtigkeit der Java-IDE zurückgegriffen werden kann. Die DSL selbst wird von den jeweiligen Anwendungsentwicklern definiert, JGiven liefert nur die nötigen Hilfsmittel dazu mit.

JGiven vermeidet – soweit wie möglich – unnötige Zusatz-Annotationen. Dies wird dadurch erreicht, dass Methoden-Aufrufe zur Laufzeit von JGiven abgefangen und eingelesen werden. Technisch funktioniert dies ähnlich wie die aus dem Mockito-Framework [3] bekannten „Spys“. Ein weiteres wichtiges Prinzip ist der modulare Aufbau von Szenarien aus wiederverwendbaren Einheiten. Dadurch wird das Erstellen neuer Szenarien deutlich vereinfacht und gleichzeitig die Duplikation von Test-Code vermieden.

JGiven selbst ist ein kleines, leichtgewichtiges Framework. Es überwacht lediglich die Ausführung von Szenarien und erstellt daraus entsprechende Berichte. Die Tests selbst werden entweder durch „JUnit“ oder „TestNG“ ausgeführt. JGiven ist daher sehr leicht in existierende Test-Infrastrukturen zu integrieren und ermöglicht es so, bestehende Tests Stück für Stück in Szenario-Tests umzuwandeln.

In den folgenden Beispielen wird ein imaginärer Web-Shop mit JGiven getestet. Alle Beispiele in diesem Artikel sind auf Deutsch geschrieben, es ist aber selbstverständlich genauso gut möglich, Test-Szenarien auf Englisch oder in jeder anderen Sprache zu schreiben. Bei Verwendung von Nicht-ASCII-Zeichen muss das Datei-Encoding allerdings „UTF-8“ sein. Das gesamte Beispiel-Projekt ist auf „GitHub“ verfügbar [4]. In der ersten Story „ABC-1“ soll sich ein Kunde registrieren können. Dazu formulieren wir ein erstes Szenario (siehe Listing 1).

Das Beispiel zeigt eine „JUnit 4“-Testmethode („@Test“), in der das JGiven-Szenario definiert ist. Der Methodenname selbst ist die Beschreibung des Szenarios in „Snake_Case“. Innerhalb der Testmethode werden dann die einzelnen Schritte des Szenarios in der „Gegeben, Wenn, Dann“-Notation als Methodenaufrufe geschrieben, jeweils wieder in „Snake_Case“. Abgesehen von ein paar von Java benötigten Klammern, Punkten und Unterstrichen enthält der Code keinerlei unnötige Boilerplates und ist für sich allein schon fast so gut lesbar wie ein in reinem Text geschriebenes Szenario. Wenn der Test nun von JUnit ausgeführt wird, generiert JGiven auf der Konsole eine Text-Ausgabe (siehe Listing 2).

```
@Test @Story("ABC-1")
public void Kunden_können_sich_registrieren() throws Exception {
   gegeben().die_Registrierungsseite_ist_geöffnet()
      .und().eine_valide_Email_ist_angegeben()
      .und().ein_valides_Passwort_ist_angegeben();
   wenn().der_Kunde_auf_den_Registrieren_Knopf_drückt();
   dann().ist_der_Kunde_registriert()
     .und().der_Kunde_erhält_eine_Bestätigungsemail();
}
```

Listing 1: Definition des Registrierungs-Szenarios in JGiven

```
Scenario: Kunden können sich registrieren

  Gegeben die Registrierungsseite ist geöffnet
      Und eine valide Email ist angegeben
      Und ein valides Passwort ist angegeben
     Wenn der Kunde auf den Registrieren Knopf drückt
     Dann ist der Kunde registriert
      Und der Kunde erhält eine Bestätigungsemail
```

Listing 2: Text-Ausgabe während der Test-Ausführung

```
public class RegistrierungsTest extends
SzenarioTest<GegebenRegistrierungsSeite<?>,
WennRegistrierungsSeite<?>, DannRegistrierungsSeite<?>> {
      // Szenarien ...
}
```

Listing 3: Definition der „RegistrierungsTest"-Klasse

Die Text-Ausgabe von JGiven dient im Wesentlichen dem Entwickler während der Entwicklung des Szenarios. Zusätzlich generiert JGiven auch JSON-Dateien während der Ausführung, die dann in einem späteren Schritt in HTML-Berichte konvertiert werden. Abbildung 1 zeigt die HTML-Darstellung des Registrierungsszenarios.

Snake_Case

Für Java-Entwickler etwas ungewöhnlich und auch gegen die Java-Code-Konventionen ist die Verwendung von „Snake_Case" in Methodennamen. „Snake_Case" ist essenziell für JGiven, da dadurch die korrekte Groß- und Kleinschreibung verwendet werden kann, was entscheidend zur Lesbarkeit der generierten Berichte beiträgt. In der Praxis hat sich die parallele Verwendung von „CamelCase" für normalen Java-Code und „Snake_Case" für Test-Szenarien als problemlos herausgestellt.

Falls die Verwendung von „Snake_Case" aufgrund von Projekt-Vorgaben unmöglich sein sollte, lässt sich die korrekte Schreibweise mit der „@Description"-Annotation angeben. Dies ist auch nützlich, wenn Sonderzeichen im Bericht erscheinen sollen, die nicht in Java-Methoden-Namen erlaubt sind.

Stage-Klassen

Schrittmethoden sind in JGiven in sogenannten „Stage-Klassen" definiert. Entsprechend der „Gegeben, Wenn, Dann"-Notation besteht ein Szenario in der Regel aus drei Stage-Klassen: eine Klasse für die „Gegeben"-, eine für die „Wenn"- und eine für die „Dann"-Schritte. Diese Modularisierung der Szenarien hat insbesondere für die Wiederverwendung von Test-Code große Vorteile. Szenarien können so nach dem Baukastenprinzip aus den Stage-Klassen zusammengesetzt werden.

Bewährt hat sich auch die Verwendung von Vererbung innerhalb der Stage-Klassen, bei denen speziellere, seltener gebrauchte Stages von allgemeineren, öfter gebrauchten Stages erben. Ein Szenario ist

Abbildung 1: HTML-Darstellung des Beispiel-Szenarios

nicht auf drei Stage-Klassen beschränkt, sondern kann aus beliebig vielen Stages bestehen. Für unser Beispiel definieren wir drei Stage-Klassen: „GegebenRegistrierungsSeite", „WennRegistrierungsSeite" und „DannRegistrierungsSeite". Damit die Stage-Klassen in unserem Test verwendet werden können, muss die Test-Klasse von der von JGiven bereitgestellten „SzenarioTest"-Klasse erben und deren Typ-Parameter entsprechend setzen (siehe Listing 3). Für englische Szenarien erbt man von „ScenarioTest".

Die „Gegeben"-Stage

Kommen wir nun zu den Stage-Klassen. Listing 4 zeigt die „GegebenRegistrierungsSeite"-Klasse, die wir für obiges Szenario benötigen. Der Einfachheit halber simulieren wir den Web-Shop mithilfe von statischen HTML-Seiten. Beim Testen kommt der Selenium WebDriver [5] zum Einsatz.

Das Beispiel zeigt mehrere wichtige Konzepte. Das erste ist, dass Stage-Klassen in der Regel von einer von JGiven vordefinierten Klasse erben, in unserem Fall der „Stufe"-Klasse (die deutsche Variante der „Stage"-Klasse). Sie stellt eine Reihe von vordefinierten Hilfsmethoden zur Verfügung, wie „und()", „aber()" etc., die in der Regel für Szenarien gebraucht werden. Stage-Klassen folgen dem „Fluent Interface"-Pattern, jede Methode gibt also als Rückgabewert das aufgerufene Objekt wieder zurück.

Damit das „Fluent Interface"-Pattern auch unter Vererbung korrekt funktioniert, wird außerdem ein Typ-Parameter an die Super-Klasse durchgereicht, der dem eigenen Typ der Klasse entspricht. Die „self()"-Methode liefert nun diesen Typ wieder zurück. Die Annotationen „@ProvidedScenarioState" und „@AfterScenario" sind nachfolgend erläutert.

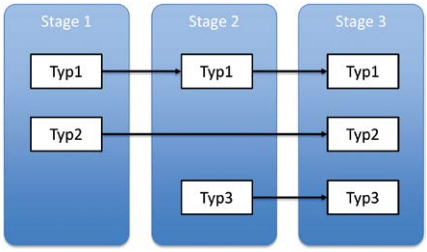


Abbildung 2: Zustand von Stage-Feldern wird von Stage zu Stage übertragen

Zustands-Transfer

Ein typisches Szenario funktioniert in der Regel so: In der „Gegeben"-Stage wird ein bestimmter Zustand hergestellt. Darauf wird dann in der „Wenn"-Stage zugegriffen, eine Aktion ausgeführt und ein Ergebniszustand produziert. Schließlich wird in der „Dann"-Stage der Ergebniszustand ausgewertet.

Um einen Zustand zwischen Stages zu transportieren, besitzt JGiven einen Mechanismus, der automatisch Felder von Stages ausliest und in nachfolgende Stages schreibt. Felder, die mit „@ScenarioState", „@ProvidedScenarioState" oder „@ExpectedScenarioState" annotiert sind, werden dabei berücksichtigt. Man kann den Mechanismus mit Dependency Injection vergleichen, mit dem Unterschied, dass der Zustand nicht nur injiziert, sondern auch extrahiert wird (siehe Abbildung 2).

Im Beispiel wird die WebDriver-Instanz für die nachfolgenden Stages verfügbar gemacht und deswegen das entsprechende Feld mit „@ProvidedScenarioState" annotiert. Nachfolgende Stages können sich nun die WebDriver-Instanz injizieren lassen, indem sie ein entsprechendes Feld deklarieren und es mit „@ExpectedScenarioState" annotieren. Der Zustandstransfer-Mechanismus von JGiven ist entscheidend für die Modularität. Dadurch, dass eine Stage nur deklariert, welchen Zustand sie benötigt, und nicht, von welcher Stage der Zustand kommen muss, lassen sich beliebige Stages miteinander kombinieren, solange die jeweils benötigten Zustände von den vorherigen Stages bereitgestellt sind.

Life-Cycle-Annotationen

Ähnlich wie bei JUnit bietet JGiven die Möglichkeit, bestimmte Aktionen zu bestimmten Zeitpunkten des Szenarios auszuführen. Dazu gibt es eine Reihe von Methoden-Annotationen: „@BeforeStage" und „@AfterStage", um Aktionen vor beziehungsweise direkt nach einer Stage auszuführen, „@BeforeScenario" und „@After Scenario", um Aktionen vor beziehungsweise nach dem ganzen Szenario auszuführen. In unserem Beispiel muss nach dem Szenario der Browser wieder geschlossen werden. Entsprechend wird die „closeBrowser"-Methode definiert und mit „@AfterScenario" annotiert.

```
public class GegebenRegistrierungsSeite<SELF extends GegebenRegis-
trierungsSeite<?>> extends Stufe<SELF> {
   @ProvidedScenarioState
   WebDriver webDriver = new HtmlUnitDriver(true);
   @ProvidedScenarioState
   String email = "testuser@test.com";
   @ProvidedScenarioState
   String passwort = "Passwort1234!$";
   public SELF die_Registrierungsseite_ist_geöffnet() throws Ex-
ception {
      File file = new File( "src/main/resources/webshop/regis-
trierung.html" );

      webDriver.get( file.toURI().toURL().toString() );
      return self();
   }

   public SELF eine_valide_Email_ist_angegeben() {
      webDriver.findElement( By.id( "emailInput" ) )
         .sendKeys( email );
      return self();
   }
   public SELF ein_valides_Passwort_ist_angegeben() {
      webDriver.findElement( By.id( "passwortInput" ) )
         .sendKeys( passwort );
      return self();
   }
   @AfterScenario
   public void closeBrowser() {
      webDriver.close();
   }
}
```

Listing 4: Definition der „GegebenRegistrierungsSeite"-Klasse

Die „Wenn"-Stage

Die Klasse „WennRegistrierungsSeite" ist sehr übersichtlich (siehe Listing 5). Wie oben schon kurz erwähnt, erwartet die Stage eine „WebDriver"-Instanz und besitzt ein entsprechend mit „@ExpectedScenarioState" annotiertes Feld. Typischerweise würde die „Wenn"-Stage selbst auch den Zustand für folgende Stages bereitstellen, in unserem Beispiel ist allerdings der Ergebnis-Zustand wiederum im WebDriver gekapselt. Ansonsten implementiert die Klasse die Registrierungsaktion, indem sie einen Knopfdruck durchführt.

Die „Dann"-Stage

Da wir in unserem Beispiel nur gegen statische HTML-Seiten testen, ist die „Dann"-Stage nicht vollständig implementiert. Ob der Kunde registriert ist oder eine E-Mail versendet wurde, würde man in der Praxis normalerweise gegen die Datenbank testen. Listing 6 zeigt die implementierten Teile. In „Dann"-Klassen sind typischerweise die Assertions eines Tests implementiert, wie auch in diesem Beispiel.

Parametrisierte Schrittmethoden

Nachdem die erfolgreiche Registrierung getestet wurde, kommen nun die Fehlerfälle an die Reihe. Unter anderem soll die E-Mail-Adresse bei der Registrierung validiert werden. Da es mühselig wäre, für jeden Wert eine eigene Schrittmethode zu schreiben, kön-

```
public class WennRegistrierungsSeite<SELF extends WennRegis-
trierungsSeite<?>> extends Schritte<SELF> {
   @ExpectedScenarioState
   WebDriver webDriver;

   public SELF der_Kunde_auf_den_Registrieren_Knopf_drückt() {
      webDriver.findElement( By.id( "registrierenButton" ) )
         .click();
      return self();
   }
}
```

Listing 5: Die „WennRegistrierungsSeite"-Klasse

```
public class DannRegistrierungsSeite<SELF extends DannRegis-
trierungsSeite<?>> extends Schritte<SELF> {
   @ExpectedScenarioState
   WebDriver webDriver;

   public SELF wird_die_Willkommensseite_geöffnet() {
      assertThat( webDriver.getTitle() )
         .isEqualTo( "Willkommen beim WebShop!" );
      return self();
   }
   // ...
}
```

Listing 6: Ausschnitt aus der „DannRegistrierungsSeite"-Klasse

```
public SELF als_Email_ist_$_angegeben( String email ) {
   this.email = email;
   webDriver.findElement( By.id( "emailInput" ) )
      .sendKeys( email );
   return self();
}
```

Listing 7: Parametrisierte Schrittmethode

```
public SELF wird_die_Registrierung_mit_der_Fehlermeldung_$_abge-
lehnt( String fehlerMeldung ) {
     assertThat(
        webDriver.findElement( By.id( "fehlerMeldung" ) )
           .getText() ).isEqualTo( fehlerMeldung );
   return self();
}
```

Listing 8: Parametrisierte Schrittmethode in der „Dann"-Stage

nen Schrittmethoden Parameter haben. Die „GegebenRegistrierungsSeite"-Klasse erhält nun eine neue Methode (siehe Listing 7).

Das Dollarzeichen im Methodennamen ist ein Platzhalter, um den Parameter an die richtige Stelle des Satzes setzen zu können. Im generierten Bericht wird es durch den Wert des Parameters ersetzt. Falls kein „$" vorkommt, werden Argumente einfach am Ende angefügt. Zusätzlich ist noch eine neue Schrittmethode in der „DannRegistrierungsSeite"-Klasse notwendig, die ebenfalls parametrisiert ist (siehe Listing 8).

Parametrisierte Szenarien

Parametrisierte Schrittmethoden sind für sich schon nützlich, um in verschiedenen Szenarien verschiedene Werte übergeben zu können. Oft möchte man allerdings das Szenario selbst parametrisieren. Im Beispiel werden verschiedene ungültige E-Mail-Adressen mit dem JUnit-DataProvider [6] getestet. Dazu wird eine neue Testmethode erstellt, die einen Data-Provider verwendet und eine E-Mail-Adresse als Parameter erwartet (siehe Listing 9). JGiven erzeugt daraus einen Bericht, der die verschiedenen Fälle in einer Tabelle übersichtlich darstellt (siehe Abbildung 3).

Tags

Es ist sicher aufgefallen, dass die bisherigen Tests alle mit „@Story(„ABC-1")" annotiert sind. Dies ist keine spezielle JGiven-Annotation, sondern eine für das Beispiel definierte Annotation. „@Story" ist selbst wiederum mit „@IsTag" annotiert, wodurch JGiven es als Tag erkennt. Tags erscheinen im generierten HTML-Bericht und für jedes Tag wird eine extra Seite generiert. Dadurch erhält man nach Themen gruppierte Szenarien, unabhängig von der Test-Klasse, in der sie definiert sind.

Anders als in anderen BDD-Tools hat JGiven keinen Mechanismus, um Features zu beschreiben. In der Regel werden Features beziehungsweise Stories (oder auch Bugs) schon in anderen Tools verwaltet. Anstatt die Beschreibung der Features in JGiven zu wiederholen, können Tags verwendet werden, um eine Zuordnung zwischen Szenarien und Features zu ermöglichen. Idealerweise enthalten die Story-Tickets auch die entsprechenden Akzeptanzkriterien, die dann direkt mit den geschriebenen Szenarien verglichen werden können.

Einbindung ins Projekt

JGiven wird entweder zusammen mit JUnit oder TestNG verwendet. Für JUnit benötigt man folgende Maven-Abhängigkeit (siehe Listing 10). Für „TestNG" hat die „artifactId" den Namen „jgiven-testng". Um nach der Test-Ausführung den HTML-Bericht zu generieren, bindet man noch das JGiven-Maven-Plug-in ein (siehe Listing 11). Ein „mvn verify" führt nun die Tests aus und generiert anschließend die JGiven-HTML-Berichte.

Fazit

Hinter Behavioral-Driven-Development steht die Idee einer gemeinsam verwendeten Sprache zwischen Product-Ownern,

```
@Test @Story("ABC-1")
@DataProvider( { "abc.com", "ungültig", "1234" } )
public void Kunden_können_sich_nur_mit_valider_
Emailadresse_registrieren( String email ) {
  gegeben().die_Registrierungsseite_ist_geöffnet()
    .und().als_Email_ist_$_angegeben( email )
    .und().ein_valides_Passwort_ist_angegeben();
  wenn().der_Kunde_auf_den_Registrieren_Knopf_
drückt();

dann().wird_die_Registrierung_mit_der_Fehlermel-
dung_$_abgelehnt( "Ungültige Emailadresse" )
   .und().der_Kunde_ist_nicht_registriert();
}
```

Listing 9: Parametrisiertes Szenario

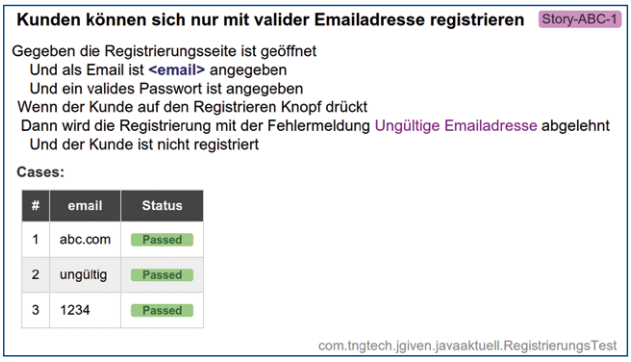


Abbildung 3: HTML-Darstellung von parametrisierten Szenarien

```
<dependency>
  <groupId>com.tngtech.jgiven</groupId>
  <artifactId>jgiven-junit</artifactId>
  <version>0.3.0</version>
  <scope>test</scope>
</dependency>
```

Listing 10

```
<build>
  <plugins>
    <plugin>
      <groupId>com.tngtech.jgiven</groupId>
      <artifactId>jgiven-maven-plugin</artifactId>
      <version>0.3.0</version>
      <executions>
        <execution>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 11: Maven-Plug-in zur Erzeugung des HTML-Berichts

Business-Analysten, Entwicklern und Testern. Dadurch werden Kommunikationsfehler vermieden und sichergestellt, dass Anforderungen so wie gewünscht umgesetzt werden. Bestehende BDD-Tools für Java zwingen Entwickler, mit einfachen Text-Dateien zu arbeiten, was den Aufwand der automatisierten Testerstellung deutlich erhöht. JGiven geht einen anderen Weg, indem es, der TDD-Philosophie folgend, den Entwickler in den Mittelpunkt stellt. Dies steht nicht im Gegensatz zur BDD-Philosophie. Business-Analysten schreiben die Anforderungen weiterhin in der „Gegeben, Wenn, Dann"-Notation. Anstatt deren Eingabe allerdings direkt auszuführen, werden die generierten Szenario-Berichte vom Business-Analysten beziehungsweise Product-Owner abgenommen und gegen die Akzeptanz-Kriterien abgeglichen, idealerweise zusammen mit den Entwicklern. Die von JGiven generierten Berichte dienen zusätzlich als automatisch validierte Verhaltens-Dokumentation der Anwendung. Aktuelle Informationen, weitere Beispiele und natürlich der Source-Code von JGiven sind über die offizielle Webseite [7] verfügbar.

Quellen

[1] jbehave.org
[2] cukes.info
[3] code.google.com/p/mockito
[4] github.com/janschaefer/jgiven-java-aktuell
[5] docs.seleniumhq.org/projects/webdriver
[6] github.com/TNG/junit-dataprovider
[7] jgiven.org

Dr. Jan Schäfer
jan.schäfer@tngtech.com

Dr. Jan Schäfer ist Senior Consultant und seit drei Jahren bei der TNG Technology Consulting GmbH in München tätig. Er ist leidenschaftlicher Software-Entwickler und programmiert seit mehr als fünfzehn Jahren in Java. Während seiner Promotion befasste er sich mit der formalen Definition von Objekt-orientierten Programmiersprachen und Aktor-basierten Nebenläufigkeitsmodellen.

http://ja.ijug.eu/14/4/7

Geodatensuche und Datenanreicherung mit Quelldaten von OpenStreetMap

Dr. Christian Winkler, mgm technology partners GmbH

Im Gegensatz zu Wikipedia ist die Verbreitung von OpenStreetMap als Standard-Kartendienst noch deutlich kleiner. Dabei ist sowohl die Qualität als auch der Umfang der Daten vielen kommerziellen Diensten bereits jetzt deutlich überlegen. Attraktiv sind außerdem die kostenlosen Nutzungs- und Verbreitungsmöglichkeiten. Der Artikel zeigt, wie man die offenen Kartendaten umformen kann, um sie in Echtzeit durchsuchen zu können. Dies erfordert zwar einige Vorarbeiten, bietet aber sehr interessante Möglichkeiten zur Anreicherung von geokodierten Daten und erlaubt weitreichende statistische Analysen.

Beim Auffinden von Informationen haben wir uns an die Suche mit Suchmaschinen im Internet gewöhnt. Google und Wikipedia sind dabei eine große Hilfe und machen Wissen in einer Weise verfügbar, wie das noch nie zuvor in der Geschichte der Fall gewesen ist. Die Technik hält allerdings auch in anderen Bereichen Einzug, viele Menschen können sich heute ein Fahren ohne Navigationssystem gar nicht mehr vorstellen. Dazu trägt natürlich auch die breite Verfügbarkeit der Geräte bei, da die Technologie einfacher und preisgünstiger geworden ist.

Etwas schwieriger wird es aber bei alltäglichen Fragestellungen. Jeder kennt an seinem Wohnort die nächste Apotheke oder den nächsten Tierarzt. Anders sieht es aus, wenn man unterwegs ist und das nächste Schwimmbad, Restaurant oder auch nur einen Supermarkt finden möchte. Manche Navigationsgeräte bieten zwar entsprechende Funktionen, man hat sie jedoch nicht immer dabei, wenn man zu Fuß unterwegs ist, oder nutzt sie nicht zur Planung einer Tour.

Auch für kommerzielle Anwendungen kann eine solche Suche sinnvoll sein. So lassen sich damit beispielsweise Adressen von Immobilien weiter qualifizieren, indem man Entfernungen zu Kindergärten, Schulen, Ärzten etc. berechnet. Nur – wer bietet solche Informationen an? In der Google-Suche findet man zwar gelegentlich Geo-Resultate, aber spezifisch danach suchen kann man auch in Google Maps [1] nur bedingt. Zudem möchte man sich nicht von einem kommerziellen Anbieter wie Google abhängig machen.

Wikipedia für Geodaten

Für Wissen und (relativ) strukturierte Informationen spielt Wikipedia eine Vorreiterrolle und hat das ehemals hehre Gut für die Masse verfügbar gemacht. Man bräuchte jetzt eine Art Wikipedia für geografische Daten, die ähnlich unabhängig durch eine Community agiert und ihre Kenntnisse frei verfügbar macht.

Von der Öffentlichkeit relativ unbemerkt hat sich schon lange der Dienst OpenStreetMap [2] etabliert und stellt in der Zwischenzeit sehr umfangreiche Daten in einer exzellenten Qualität zur Verfügung. Jeder kann sich dort anmelden und Karten mit einem einfach zu bedienenden Editor verbessern.

Sehr viele Freiwillige haben das mit GPS-Trackern [3] schon getan und daher stehen bereits extrem umfassende Informationen zur Verfügung. Auf der Website „www.openstreetmap.org“ kann man sich ein Bild darüber verschaffen (siehe Abbildung 1). Oft sind die Karten so detailliert, dass sogar Konturen von Häusern angezeigt werden. Warum das eine Rolle spielt, werden wir später sehen.

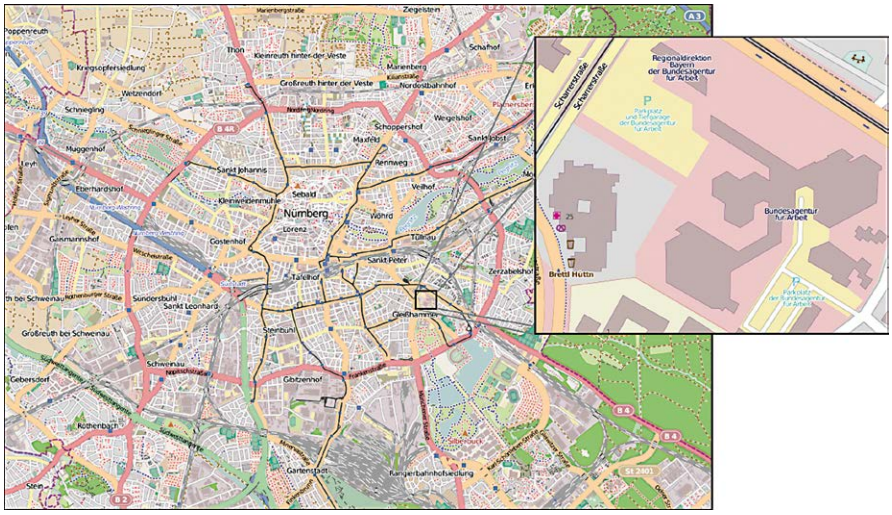


Abbildung 1: OpenStreetMap von Nürnberg mit dem Zoom auf die Bundesagentur für Arbeit, bei der man gut die Kontur der Gebäude erkennen kann. Viele Details sind in den Karten gar nicht dargestellt.

Die Datenqualität von OpenStreetMap steigt ständig. Das Wissen über geografische Gegebenheiten geht nicht verloren, sondern wird laufend ergänzt und verfeinert. Durch die (nach der Lizenz gestattete) Verwendung in kommerziellen Anwendungen gibt es auch ein hohes Interesse der Wirtschaft, diese Daten immer weiter zu pflegen. Je mehr Menschen OpenStreetMap verwenden, desto höher ist natürlich auch die Bereitschaft, sich selbst einzubringen, Fehler zu melden oder Ergänzungen vorzunehmen.

Noch spannender als die Karten ist bei OpenStreetMap die Möglichkeit, sich die Quelldaten herunterzuladen und für andere Zwecke zu verwenden. Nachfolgend ist beschrieben, wie man diese zugänglich machen und für viele unterschiedliche Dinge nutzen kann.

Indizierung von Geodaten

Die OpenStreetMap-Website bietet die gesamten Kartendaten zum Herunterladen an. Allerdings stehen nur entweder rechteckige Ausschnitte oder die ganze Erde als Download zur Verfügung, was in beiden Fällen zu Problemen führt. Die Daten der ganzen Erde sind extrem umfangreich, während die Daten eines Rechtecks oft nicht komplett konsistent sind, weil gewisse Flächen den Rand des Rechtecks überschneiden.

Glücklicherweise gibt es auch konsistente Ausschnitte, beispielsweise für ganze Länder wie Deutschland, einzelne Bundesländer oder Regierungsbezirke. Diese werden aber nur periodisch erzeugt und reflektieren damit nicht automatisch den Stand, den man in den Karten auf [1] sieht.

Die Download-Formate sind XML (gepackt) oder ProtocolBuffer [4]. Bei Letzterem handelt es sich um ein von Google erdachtes, sehr stark komprimiertes Format, das ursprünglich zum Nachrichtenaustausch konzipiert war. Es gibt Parser für viele gängige Sprachen. Allerdings eignet sich das XML-Format eher zum Experimentieren, weil man es leichter lesen und die Inhalte als Mensch semantisch erfassen kann.

Die Quelldaten von OpenStreetMap müssen so abgespeichert werden, dass Suchen nach Entfernungen („geospatial search“) möglich ist. Grundsätzlich bieten bereits einige Datenbank-Systeme wie PostgreSQL, Oracle oder Microsoft SQL Server entsprechende Funktionen zum Indizieren und Suchen an.

```
<field name="id" type="string" indexed="true" stored="true"
required="true" />
<field name="geo" type="location_rpt" indexed="true" stored="true"
/>
<fieldType name="location_rpt"
class ="solr.SpatialRecursivePrefixTreeFieldType"
spatialContextFactory="com.spatial4j.core.context.jts.\
JtsSpatialcontextFactory"
geo="true" distErrPct="0.01" maxDistErr="0.0009" units="degrees"
/>
```

Listing 1

```
<node id="26212605" lat="49.588555" lon="11.0014352" version="7"
timestamp="2013-01-08T17:55:18Z" changeset="14577549" uid="479256"
user="geodreieck4711"/>
<way id="2293021" version="12" timestamp="2013-01-08T17:55:13Z"
  changeset="14577549" uid="479256" user="geodreieck4711">
  <nd ref="26212605"/>
  <nd ref="9919314"/>
  <nd ref="2101861553"/>
  <nd ref="10443807"/>
  <tag k="bicycle" v="designated"/>
  <tag k="cycleway" v="segregated"/>
  <tag k="foot" v="designated"/>
  <tag k="highway" v="cycleway"/>
  <tag k="segregated" v="yes"/>
  <tag k="surface" v="asphalt"/>
</way>
```

Listing 2

Da wir uns hauptsächlich für die Suche (und Facettierung) interessieren, nutzt die vorgestellte Lösung eine genau darauf optimierte Software, nämlich Apache Solr [5]. Diese verfügt seit der Version 4.0 über umfangreiche Möglichkeiten zur Suche in Geodaten. Selbstverständlich kann Solr die indizierten OpenStreetMap-Daten auch als XML-Quelltext anzeigen, lediglich eine Visualisierung in Karten erfordert mehr Infrastruktur und ist (bisher) nicht Bestandteil der Lösung.

Ähnlich wie relationale Datenbanken eine Definition der Entitäten und Relationen benötigen, braucht Solr ein sogenanntes „Schema“, um Daten zu indizieren. Im Gegensatz zu den relationalen Systemen ist der Datenbestand von Solr aber flach, es gibt also vereinfacht gesprochen nur eine einzige Tabelle.

Solr beherrscht unterschiedliche Datenformate, in unserem Fall ist neben den eigentlichen Koordinaten vor allem String relevant, weil hiermit viele Attribute (siehe unten) abgebildet werden können. Sehr spannend ist allerdings der Support für Geokoordinaten, für den es zwei unterschiedliche Ansätze gibt:

- *LatLonType*
  Dieser Datentyp kann einen (oder mehrere) Punkte im Solr-Index abspeichern. Die Abspeicherung erfolgt kompakt in Form von zwei Float-Werten, die sich sehr schnell abfragen lassen.
- *SpatialRecursivePrefixTreeFieldType*
  Dieser Datentyp ist deutlich flexibler und nutzt sogenannte „Geohashes“ [6] zum Abspeichern der Daten. Dafür wird zwar mehr Speicherplatz benötigt, aber es stehen auch mehr Suchfunktionen zur Verfügung. Zusätzlich können neben reinen Punkten hier auch geometrische Objekte wie Polygone oder Linien abgelegt werden.

In unserer Lösung wird mit dem zweiten Datentyp gearbeitet, weil in den OpenStreetMap-Daten auch Polygone (wie „Konturen eines Waldes“ etc.) hinterlegt sind

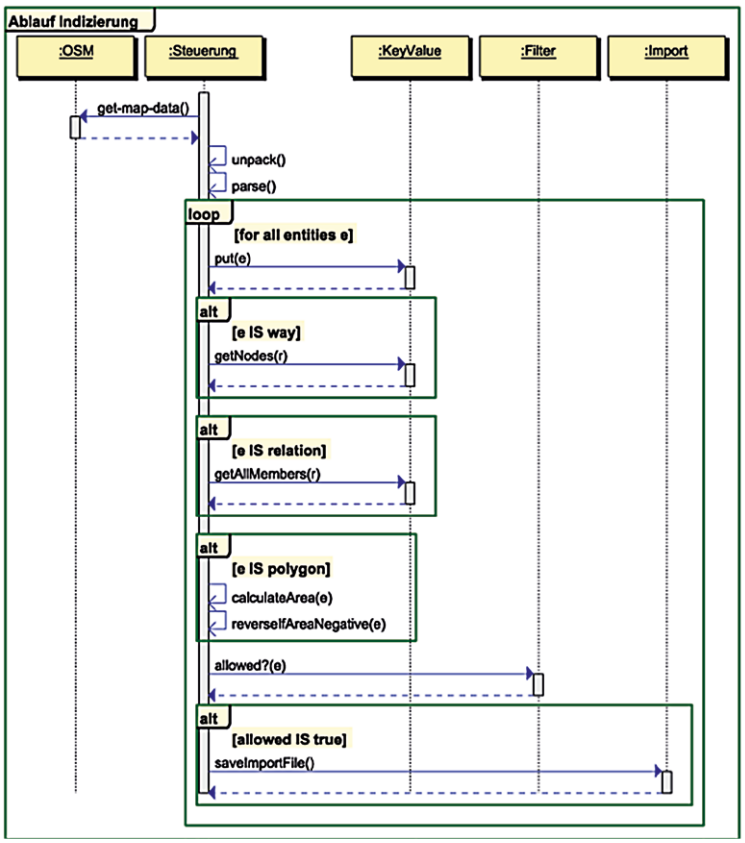


Abbildung 2: Schematischer Ablauf der Übernahme von OpenStreetMap-Daten in den Index-Server

und es sich als sehr interessant herausgestellt hat, diese in die Suche mit einzubeziehen. Listing 1 zeigt den für die Geosuche relevanten Teil des Solr-Schemas.

Hier arbeiten wir mit einem maximalen Fehler von einem Prozent in Entfernungen und einem maximalen absoluten Fehler von etwa hundert Metern, was für unsere Zwecke völlig ausreichend ist. Eine Erhöhung dieser Werte würde zu einem deutlich größeren Index und einer langsameren Suche führen.

Die schon vorhandenen OpenStreetMap-Daten werden anschließen in Solr importiert. Normalerweise würde man dafür einen „DataImportHandler" schreiben (beziehungsweise konfigurieren), der die XML-Daten interpretieren kann. Leider geht das in diesem Fall aber nicht so einfach, weil es unterschiedliche Datentypen im OpenStreetMap-XML gibt, die sich teilweise gegenseitig referenzieren. Es kommt also auf die Reihenfolge des Imports an und es muss dereferenziert werden. Außerdem wollen wir nicht alle Daten in den Solr-Index importieren, sondern vorher die Daten filtern. Listing 2 zeigt ein Fragment der OpenStreetMap-Datei.

Hier kann man bereits erkennen, dass die Nodes (also die Punkte) in sogenannte „Ways" referenziert sind. Eine solche Referenz ist in Solr allerdings nicht vorgesehen und daher wird diese Referenz aufgelöst. Die OpenStreetMap für Deutschland hat allerdings mehrere Millionen Nodes, was eine solche Dereferenzierung nicht ganz einfach macht. Wir haben dazu einen Ansatz mit einer Key-Value-Datenbank (LevelDB [7]) gewählt.

Die Darstellung von Polygonen oder Linien erfolgt in Solr im sogenannten „Well Know Text"-Format (WKT) [8]. Die entsprechenden Objekte sind also vom OpenStreetMap-Format in WKT zu überführen, was nicht einfach ist. In WKT ist nämlich der Umlaufsinn der Polygone festgelegt und entscheidet darüber, was im Innen- und im Außenbereich des Polygons liegt, während das für OpenStreetMap keine Rolle spielt. Viele Polygone haben dort den falschen Umlaufsinn. Folglich sind alle Polygone in den korrekten Umlaufsinn (umgekehrter

```
<lst name="amenity">
  <int name="recycling">22</
int>
  <int name="restaurant">22</
int>
  <int name="telephone">18</
int>
  <int name="pub">16</int>
  <int
name="kindergarten">13</int>
  <int name="post_box">13</
int>
  <int name="fast_food">9</
int>
  <int name="school">9</int>
  <int name="vending_ma-
chine">8</int>
  <int name="place_of_wor-
ship">7</int>
  <int name="biergarten">5</
int>
  <int name="cafe">5</int>
  <int name="university">4</
int>
  <int name="bank">3</int>
  <int name="fuel">3</int>
  <int name="pharmacy">3</int>
  <int name="doctors">2</int>
  <int name="hospital">2</int>
</lst>
```

Listing 3

Uhrzeiger) zu konvertieren, was sich durch eine Flächenberechnung erreichen lässt.

Noch etwas schwieriger als die oben angegebenen Wege ist die Umsetzung sogenannter „Relationen", die Wege miteinander verbinden und damit nichtzusammenhängende Polygone erzeugen. Glücklicherweise bieten WKT und Solr eine Multipolygon-Form. Zudem können Polygone sogenannte „Löcher" enthalten, die WKT auch darstellt, die für unsere Zwecke aber nicht relevant sind. So könnte zum Beispiel ein Wald ein Loch haben, in dem ein See liegt, aber dieses Loch ist für alle Entfernungsberechnungen nicht relevant.

OpenStreetMap enthält eine sehr große Menge an interessanten Daten für fast alle denkbaren Fragestellungen. Allerdings sind für die Adress-Qualifizierung oder Umgebungssuche einige Objekte vernachlässigbar (wie die Position von Hydranten oder die Farbe von Parkbänken). Um den für die Indizierung benötigten Platz so klein wie möglich zu halten, werden diese ausgefiltert. Gleiches gilt auch für einige Konturen, die nicht direkt relevant sind, wie zum Beispiel Gebäude, Wirtschaftswege etc. Die Lösung lässt sich aber für diese Attribute konfigurieren.

In der eigentlichen Implementierung des Filters sind wir zweistufig vorgegangen: Über eine Positivliste werden Objekte definiert, die betrachtet werden sollen. Manche können sich dann aber im Nachhinein als dennoch irrelevant herausstellen und werden über eine Negativliste dann wieder entfernt. Bei den Attributen selbst sind selbstverständlich auch nicht alle relevant, diese werden dann auch dynamisch auf die relevanten reduziert. Als Ergebnis dieses Schrittes stehen Daten bereit, die direkt von Solr indiziert werden können.

Im nächsten Schritt findet dann die Indizierung der relevanten Daten statt (siehe Abbildung 2). Dies erfolgt im Batch-Verfahren und lässt sich parallelisieren, da für Geodaten (und insbesondere Konturen) die CPU der limitierende Faktor ist. Auf einem Standard-PC mit 8 GB RAM und 2.5 GHz Quad-Core-Prozessor lassen sich die Daten von Deutschland in etwa einem Tag indizieren. Nach diesem Schritt steht nun eine in Solr indizierte Version der OpenStreetMap-Daten zur Verfügung und kann für Suchen genutzt werden.

Beispiele für Suche in Geodaten

Eine besonders naheliegende Suche ist die Umkreissuche, die sich in Solr problemlos realisieren lässt. Exemplarisch zeigt der Autor das anhand der Geo-Koordinaten seines Nürnberger Büros (49.447206,11.102245) und sucht alle Objekte in einer Entfernung von maximal einem Kilometer. Daraus ergibt sich eine Solr-Suche in der Form „{!geofilt sfield=geo pt=49.447206,11.102245 d=1}". Nach kurzer Zeit (deutlich weniger als eine Sekunde) erhalten wir Resultate, allerdings gibt es dafür 630 Ergebnisse, von denen nur die ersten 10 angezeigt sind (das lässt sich einfach anpassen). Die Resultate sind nicht nach der Entfernung sortiert, da das im Moment mit Solr noch sehr speicheraufwändig und für so viele Daten im Index nicht gut möglich ist. Allerdings lässt sich einfach die Entfernung reduzieren und wiederholt Suchen durchführen.

Besonders spannend ist bei der Suche die Facettierung. Hier werden weitere Suchkriterien simuliert, die die Suche weiter einschränken, und ihre Anzahl genannt. Wenn man beispielsweise über das Feld „amenity" facettiert, erhält man folgendes Ergebnis (siehe Listing 3).

Natürlich kann nicht nur nach „amenity", sondern nach jedem Feld wie beispielsweise „landuse" facettiert werden. Auch funktionale Facetten sind möglich, in denen man (zukünftig) etwa die Summe von Flächen berechnen kann. Die sich daraus ergebenden Möglichkeiten zur Daten-Qualifizierung sind sehr umfangreich und lassen sich sowohl zur Exploration als auch für statistische Zwecke nutzen. Eine Facettierung ist mit klassischen Datenbanken nicht möglich und ein weiterer Grund dafür, dass Solr zum Einsatz kam.

Wie beschrieben lässt sich ganz Deutschland mit Standard-Hardware problemlos indizieren. Damit kann man selbst ausprobieren, welche interessanten Informationen man über seinen eigenen Wohnort noch finden kann oder auch Urlaubsorte näher inspizieren. Je nach Genauigkeit sind dafür neben etwas Wartezeit nur einige GB an Festplattenplatz notwendig.

Die Informationen lassen sich aber auch ganz anders nutzen. So können Firmen die wichtigen Adressen (Apotheke, Ärzte etc.) in ihrer Umgebung zusammenstellen. Grundsätzlich lässt sich dies mit der Lizenz von OpenStreetMap auch als Service anbieten.

Für die private Nutzung gibt es noch andere Anwendungsfälle, so kann man etwa nach Sehenswürdigkeiten in der Natur suchen, aber auch die Suche nach einem Restaurant in der Umgebung lässt sich einfach abbilden. Hier ist auch wieder die Facettierung hilfreich, weil man so die unterschiedlichen Landesküchen problemlos herausfindet. Auch Einkaufsmöglichkeiten wie Supermärkte (wieder facettiert nach Anbieter) können leicht aufgelistet werden, die fleißigen Helfer von OpenStreetMap haben dankenswerterweise oft auch die Öffnungszeiten und URLs der entsprechenden Einrichtungen hinterlegt.

Die Immobiliensuche ist seit dem Entstehen der entsprechenden Portale im Internet ein schwer umkämpfter Markt. Fast alle Immobilien sind in der Zwischenzeit geocodiert, auch wenn dies (wegen der Vermittlungspauschalen) im Internet nicht sofort angezeigt wird. Über die OpenStreetMap-Daten lassen sich hier Differenzierungsmerkmale schaffen. So können Daten schon zum Indizierungszeitpunkt

angereichert werden. Dabei kann man die Entfernung zur nächsten Schule, Kindergarten, Arzt oder Supermarkt ermitteln und in den eigentlichen Such-Index einfließen lassen. Damit kann man beispielsweise Suchen formulieren, mit denen nur Immobilien gefunden werden, die nicht mehr als 1 km von einem Kindergarten und einem Supermarkt entfernt sind.

Der Suchindex selbst lässt sich natürlich auch als Datenquelle verwenden. Eine für Discounter statistisch interessante Frage könnte etwa sein, wie viele eigene Supermärkte weniger als 1 km von einem Konkurrenz-Markt entfernt sind. Das lässt sich auch auf Bundesländer und bestimmte Städte einengen.

Aber die statistischen Möglichkeiten sind noch weit größer. So kann man beispielsweise herausfinden, in welcher Stadt die Wohnungen im Mittel die geringste Entfernung zu Waldflächen oder Parks haben oder wie weit die mittlere Entfernung zu Betreuungseinrichtungen, Krankenhäusern und Ärzten ist. Ein Gastronom, der ein neues Restaurant gründen möchte, könnte ermitteln, wo für eine bestimmte Küche die Entfernung zu ähnlichen Restaurants am größten ist.

Das Verfahren lässt sich auch zur Augmentierung und Statistik von Drittdaten verwenden, die über die in letzter Zeit verstärkt verfügbaren Portale zu Open Data gewonnen werden können. Um die Ergebnisse weiterhin interpretierbar zu halten, was bei der Vielfalt und Fülle der Daten schon schwierig ist, können moderne Verfahren der künstlichen Intelligenz („Machine Learning“) eingesetzt werden. Apache Mahout [9] bietet hier viele bereits fertige Algorithmen, die über die dort verwendete Hadoop-Plattform [10] auch temporär auf Cloud-Servern genutzt werden können; diese werden nach der Berechnung einfach wieder abgeschaltet (und verursachen damit keine Kosten).

In aller Munde ist das „Internet of Things“, bei dem zukünftig viele Geräte IP-Adressen erhalten, die momentan noch keine Internet-Anbindung haben. Auch hier ergeben sich interessante Einsatzszenarien. Ist die Position eines „Dings“ bekannt, so kann es sich unter dieser Position anmelden und seine Dienste anbieten, die dann über eine entsprechende Geodaten-Plattform auffindbar sind. Auf der anderen Seite können diese „Dinge“ (wenn sie beweglich sind) aber auch eine Such-Plattform nutzen, um andere interessante Punkte in ihrer aktuellen Umgebung finden. Die nahezu unbeschränkten Möglichkeiten werden sich hier erst im Laufe der Zeit konkretisieren und zu vielen neuen Anwendungen führen, an die jetzt noch keiner denkt.

Mehr und mehr wird auch bei Nachrichten eine Geoposition hinterlegt. Damit sind Dienste machbar, bei denen nach Nachrichten in einer bestimmten Region oder in der Nähe des aktuellen Standorts gesucht wird. Besonders bei Verkehrsnachrichten, Sperrungen, Demonstrationen, aber auch bei Konzerten und anderen Ereignissen können sich hier interessante Anwendungsfälle ergeben.

Einfach realisierbar sind auch mobile Anwendungen, bei denen der Benutzer einen Bedarf anzeigt. Denkbar ist etwa, dass er einen Supermarkt sucht. Sobald das Gerät über die Geosuche erkennt, dass ein Supermarkt in einer gewissen Entfernung zur aktuellen Position liegt (und geöffnet hat), wird der Benutzer darauf hingewiesen. Dies lässt sich natürlich auch mit vielen anderen Einrichtungen (Toiletten etc.) durchführen.

Wenn Personen über eine geschlossene Community ihre eigenen Positionen melden, können sie benachrichtigt werden, wenn sich andere Personen in ihrer Nähe befinden. Das lässt sich auch mit der Nähe zu bestimmten Einrichtungen verbinden. So könnte der Golfspieler eines Vereins darauf hingewiesen werden, dass sich gerade mindestens zwei Personen in der Nähe des Golfplatzes befinden, die dann unmittelbar mit einem Spiel beginnen.

Da in OpenStreetMap auch viele Gebäude selbst modelliert sind, lässt sich auch der mittlere Abstand von Gebäuden innerhalb von Städten ausrechnen. In Zusammenhang mit Grünflächen etc. lässt sich hier möglicherweise eine Korrelation zu der Lebensqualität ermitteln.

Mit OpenLayers [11] verfügt OpenStreetMap zudem über eine leistungsfähige Möglichkeit zur Anzeige von Karten. Allerdings kann dies nicht aus dem Solr-Index geschehen, dafür ist vielmehr eine PostgreSQL-Datenbank [12, 13] erforderlich. Eine Installation kann sich lohnen, wenn Daten verifiziert werden sollen oder eine Visualisierung notwendig ist. Wenn dies nur gelegentlich erforderlich ist, ist ein einfacher Link auf die öffentliche Website von OpenStreetMap aber deutlich einfacher zu realisieren.

Fazit

Mit der gezeigten Lösung lässt sich auf gewöhnlicher Hardware unter Nutzung von offenen Quellen (OpenStreetMap) und offener Software (Apache Solr) ein Geo-Informationssystem implementieren.

Referenzen

[1] https://maps.google.de
[2] http://www.openstreetmap.org
[3] https://en.wikipedia.org/wiki/GPS_tracking_unit
[4] https://code.google.com/p/protobuf
[5] http://lucene.apache.org/solr
[6] http://en.wikipedia.org/wiki/Geohash
[7] https://code.google.com/p/leveldb
[8] http://en.wikipedia.org/wiki/Well-known_text
[9] http://mahout.apache.org
[10] http://hadoop.apache.org
[11] http://openlayers.org
[12] http://www.postgresql.org
[13] http://wiki.openstreetmap.org/wiki/PostgreSQL

Dr. Christian Winkler
christian.winkler@mgm-tp.com

Christian Winkler hat sein erstes Unternehmen gegründet, als er noch zur Universität ging und arbeitet daher seit zwanzig Jahren mit Internet-Technologien. Dabei reizen ihn besonders der Umgang mit sehr großen Datenmengen oder sehr vielen Nutzern. Heute arbeitet er für mgm technology partners GmbH als Enterprise Architect. Neben Big Data und Suche liegt dort sein Fokus auf Hochleistung und Hochverfügbarkeit von Webseiten.

http://ja.ijug.eu/14/4/8

Pimp my Jenkins – mit noch mehr Plug-ins

Sebastian Laag, adesso AG

Im ersten Teil dieser Serie wurde in der letzten Ausgabe eine Reihe nützlicher Jenkins-Plug-ins [1, 2] zur vereinfachten Administration, Erweiterung der Benutzeroberfläche oder auch zur Gestaltung komplexer Build-Pipelines vorgestellt. In diesem Artikel kommen Plug-ins zur Integration mit Tools wie Sonar oder mit dem Deployment auf unterschiedliche Application-Server zur Sprache. Zudem wird gezeigt, wie bei der Verwaltung und Erstellung ähnlicher Jobs viel Zeit gespart werden kann. Zum Abschluss werden Möglichkeiten zur grafischen Darstellung und automatischen Analyse von sich wiederholenden Fehlern in Builds vorgestellt.

Repository Connector

Der Build-Schritt „Artifact Resolver" lädt Artefakte aus dem Nexus-Repository, ohne Maven installieren zu müssen. Das Repository-Connector-Plug-in [3] nutzt dazu das Sonatype-Aether-API [4]. In der Jenkins-Verwaltung sind der Nexus-Server sowie die zugehörigen Zugangsdaten zu definieren. Es können auch mehrere Nexus-Server angegeben werden. Im Build werden das Artefakt, die jeweilige Version und das Zielverzeichnis angegeben. Der Benutzer entscheidet, ob das Herunterladen auf der Build-Konsole zu sehen sein soll und ob ein fehlgeschlagener Download eines Artefakts den Build fehlschlagen lässt. Dazu kann die Häufigkeit der Aktualisierung der Artefakte für Releases und Snapshots einzeln definiert werden.

Jenkins Maven Repository

Das Maven-Repository-Plug-in [5] geht einen anderen Weg. Damit ist es möglich, die Ergebnisse eines Jobs als Maven-Repository anzubieten, sodass nachgelagerte Jobs oder andere externe Systeme auf die erstellten Artefakte zugreifen können. Dazu müssen die Maven-Settings-Datei des Jenkins sowie der Maven-Aufruf im nachgelagerten Job um das Kommando „–Pupstream" erweitert werden. Dies ist nicht der Fall, wenn das Repository-Connector-Plug-in zum Einsatz kommt. Hiermit können in der Jenkins-Verwaltung weitere Repositories angegeben werden. Das kann dann auch eines der Repositories sein, die von Jobs, die das Maven-Repository-Plug-in verwenden, angeboten werden. Externe Systeme können über eine festgelegte URL mit dem Schema „http://jenkins-server:port/plugin/repository/project/projectName/Build/buildNumber/repository" auf die erstellten Artefakte zugreifen. Eine detaillierte Anleitung, wie mit dem Plug-in umzugehen ist, steht auf dessen Wiki-Seite.

SonarQube

Das SonarQube-Plug-in [6] verwaltet im Jenkins mehrere SonarQube-Installationen. Nach erfolgreicher Einrichtung lassen sich Jobs so konfigurieren, dass die Ergebnisse der Code-Analyse auf der im Job ausgewählten Sonar-Instanz veröffentlicht werden. Standardmäßig entstehen die Ergebnisse mittels Maven Goal „sonar:sonar".

Wenn das Projekt kein Maven verwendet, kann der Code auch über die Stand-Alone-Variante von SonarQube, den sogenannten „Runner", analysiert werden. Dabei wird in der Jenkins-Verwaltung ein eigenes Verzeichnis mit der Installation konfiguriert oder dieser über den Maven Central Server automatisch heruntergeladen und installiert. Alternativ kann auch eine direkte URL zur „zip"- oder „tar.gz"-Datei angegeben werden, die Jenkins dann entpackt.

Gradle

Das Gradle-Plug-in [7] erweitert Jenkins um die Möglichkeit zum Starten der Builds mit Gradle. Nach der Installation können die eigenen Jobs um einen Build-Schritt erweitert und mit dem Gradle-Skript aufgerufen werden. Hier besteht die Wahl zwischen dem Aufruf selbstkonfigurierter Wrapper-Skripte oder Standard-Gradle in der jeweils installierten Version. Zusätzlich können Parameter, wie der Name der Build-Datei oder die auszuführenden Switches und Tasks, angegeben werden.

Deployment

Das Deployment erstellter Artefakte auf einen Application-Server kann auf unterschiedliche Art und Weise erfolgen. Mit dem Deploy-Plug-in [8] besteht die Möglichkeit, das Deployment nach jedem Build automatisch durchzuführen. Aktuell sind Tomcat (4.x/5.x/6.x/7.x), GlassFish (2.x/3.x) und JBoss (3.x/4.x) unterstützt. Nach Auswahl des Post-Build-Schrittes „Deploy war/ear to container" müssen die zu kopierenden Artefakte, Manager-Benutzername und -Passwort sowie das Verzeichnis des Application-Servers angegeben werden.

Die unterstützten Versionen des JBoss sind leider veraltet. Zudem kann das Plug-in nicht mit Umgebungsvariablen in den Pfad-Angaben zum Application-Server umgehen. Für WebLogic [9] und WebSphere [10] existieren eigene Plug-ins, die im Falle von WebSphere nur Version 6.1 unterstützen. In produktiven Umgebungen sollte auf die Verwendung von Hot Deployment verzichtet werden, da dieses unkalkulierbare Risiken beinhaltet [11].

Job Generator

Das Job-Generator-Plug-in [12] erleichtert die Automatisierung und Verwaltung der

Jobs. Es ist nicht notwendig, eine eigene Skript-Sprache zu erlernen, um das Plug-in zu verwenden. Nach dessen Installation steht ein weiterer Job-Typ zur Verfügung, der als Vorlage für das Kopieren weiterer Jobs dient. In ihm werden die gewünschten Parameter eingerichtet. Über das Parameterized-Trigger-Plug-in wird ein weiterer Job erstellt, der den Vorlage-Job aufruft und verschiedene Parameter übergibt. Mit einem Klick werden dann Jobs für die gewünschten Anwendungsfälle generiert (siehe Abbildung 1).

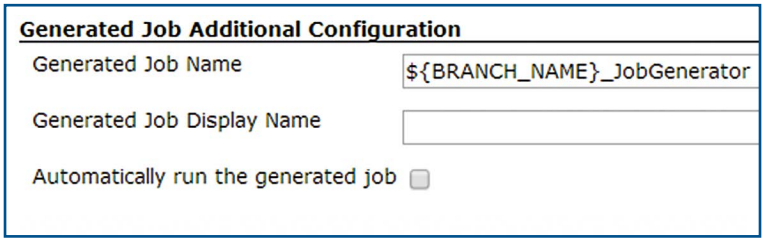


Abbildung 1: Beispiel für die Verwendung des Job-Generator-Plug-ins

Job DSL

Anders als das Job-Generator-Plug-in verfügt das Job-DSL-Plug-in [13] über ein komplexes Kommando-API zur Generierung der Jobs. Groovy-Skripte führen das Anlegen und Umkonfigurieren von Jobs durch. Zur Verwendung des Plug-ins werden ein Free-Style-Job erstellt und der Build-Schritt „Process Job DSLs“ hinzugefügt. In diesen muss ein Groovy-Skript mit Job-DSL-Kommandos [14] eingefügt werden. Es dient als Vorlage für weitere Generierungen. Nach Ausführung des Jobs werden die generierten Jobs angezeigt. Alternativ ist auch der Pfad zur Datei innerhalb des Workspace möglich, um das verwendete Vorlage-Skript zu versionieren und durch den Job auschecken zu lassen.

Subversion Merge

Eine Alternative zu Feature Toggles sind Feature Branches. Für Jenkins existiert ein Plug-in [15], um Feature Branches in Subversion komfortabel zu verwalten. Nach dessen Installation besteht die Möglichkeit, einen Job zu generieren, mit dem Feature Branches erstellt, und einen, mit dem die Änderungen per „svn merge“ in den „trunk“ synchronisiert werden können. Der Merge kann auch automatisch bei der Ausführung eines jeden Builds erfolgen. Dafür stellt das Plug-in einen eigenen Build-Schritt bereit. Der aktuelle Status der Integration, etwa die letzte Integration, ist auf der Seite „Jobs“ einsehbar, der das Plug-in verwendet.

Dashboard View

Das Dashboard-View-Plug-in [16] stellt eine neue View zur Verfügung. Damit lässt sich ein eigenes Dashboard mit verschiedenen Statistiken einrichten. In der Konfiguration muss mindestens ein Job definiert sein, um die Statistiken anzuzeigen. Zudem kann das Layout bestimmt werden. Die Konfiguration des Dashboards definiert verschiedene Portlets, die auf unterschiedlichen Positionen auf dem Bildschirm angeordnet werden. Innerhalb der Portlets sind Bilder, Iframes oder die letzten Build-Ergebnisse festgelegt. Es ist auch möglich, instabile Builds oder aggregierte Build-Statistiken wie erfolgreiche oder fehlgeschlagene Jobs anzuzeigen. Andere Plug-ins können ihre Statistiken für die Dashboard View anbieten, sodass diese vom Benutzer ebenfalls in der Konfiguration der Portlets ausgewählt werden können (siehe Abbildung 2).

Sectioned View

Eine Alternative zur Dashboard View ist die Sectioned View. Nach Installation des Sectioned-View-Plug-ins [17] kann ein neuer View-Typ angelegt werden. Dieser zeigt die verfügbaren Views und Jobs auf einer Seite gruppiert an. Zudem lassen sich Statistiken wie Test-Ergebnisse oder Kompilier-Warnungen anzeigen. Die Positionierung der einzelnen Sektionen kann im Gegensatz zur Dashboard View feiner gestaltet werden und die Auswahl der Jobs pro Sektion erfolgen, was in der Dashboard View nicht möglich ist. Dafür ist die Verfügbarkeit der Statistiken in der Sectioned View deutlich eingeschränkter.

Build Failure Analyzer

Der Build Failure Analyzer [18] definiert eine Wissensdatenbank von möglichen Build-Fehlern. Anhand dieser werden die geschriebenen Log-Ausgaben von Builds automatisch nach Abschluss untersucht. Dazu definiert der Benutzer ein Text-Pattern, anhand dessen jeder Build analysiert wird. In der Wissensdatenbank kann ein Hinweistext zur leichteren Fehlerbehebung hinter-

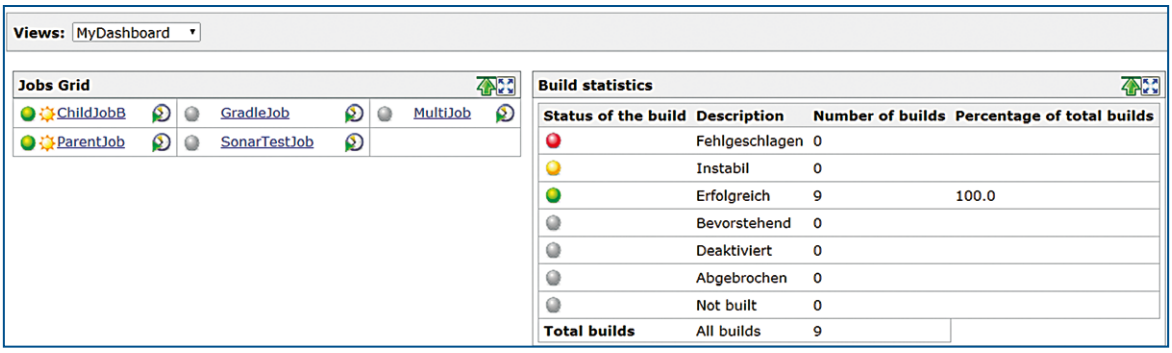


Abbildung 2: Beispielhafte Dashboard View

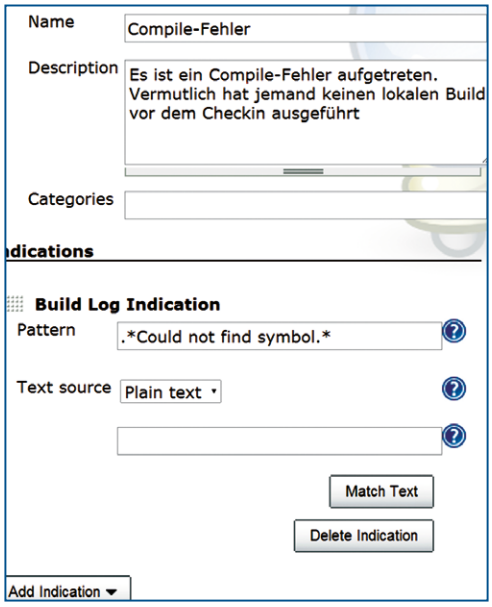


Abbildung 3: Verwendung des Build Failure Analyzer

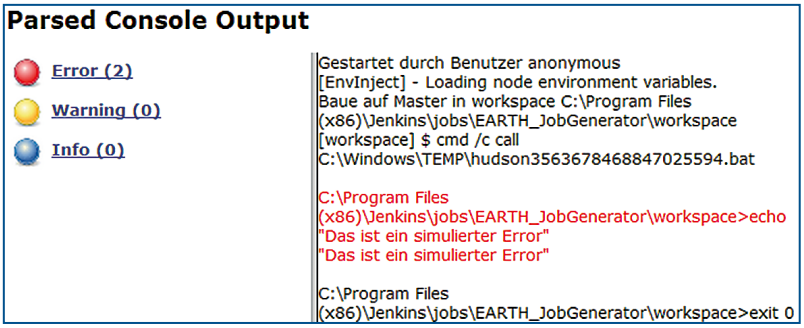


Abbildung 4: Ausgabe des Log-Parser-Plug-ins

legt sein. Wenn einer der bereits definierten Fehlertexte aufgetreten ist, wird der Build in der Übersicht markiert. Zudem wird auf der Seite des Build-Ergebnisses der gespeicherte Hinweistext angezeigt. Soll der Scan nicht mehr ausgeführt werden, so kann dieser über die Jenkins-Verwaltung deaktiviert werden. Der Benutzer kann entscheiden, in welcher Form die Statistiken der Analyse gespeichert sind. Es steht eine XML-Datei im Jenkins-Verzeichnis oder in einer MongoDB zur Auswahl. Mit der Speicherung in der MongoDB ist sogar eine Anbindung an das Graph-Tool Graphite [19] möglich (siehe Abbildung 3).

Log Parser

Das Log-Parser-Plug-in [20] ermöglicht das Überprüfen der Build-Konsole auf selbstdefinierte reguläre Ausdrücke. Wenn ein bestimmter Textabschnitt des Builds dem regulären entspricht, wird dieser markiert. Anhand der Ergebnisse kann der Build als fehlerhaft oder instabil gekennzeichnet werden. Dies ist gerade dann von Vorteil, wenn selbst geschriebene Skripte ausgeführt und der Build-Status nicht automatisch von Jenkins ermittelt werden kann (siehe Abbildung 4).

Fazit

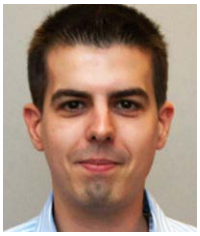
Jenkins ist ein vielseitiger CI-Server, der mit den verschiedensten Plug-ins erweitert werden kann. Zudem ergänzen sich einige Plug-ins, sodass Synergien entstehenden. Die vorgestellten Erweiterungen zur Generierung von Jobs können eine Menge Aufwand bei der Pflege und Erstellung von Jobs ersparen. Das ist dann hilfreich, wenn immer wieder die gleichen Aufgaben erledigt werden müssen, aber verschiedene Branches ausgecheckt werden sollen. Für den Umgang mit Feature Branches kann das SVN-Merge-Plug-in genutzt werden, mit dem das Erstellen und Integrieren von Branches deutlich erleichtert wird.

Die Plug-ins zur Aufbereitung von Statistiken, wie die Dashboard View, bieten eine gute Möglichkeit, die Häufigkeit von Build-Fehlschlägen transparent zu machen. So lassen sich häufige Fehlschläge der Builds schnell erkennen und Maßnahmen ergreifen, um die eigenen Prozesse zu verbessern. Dies lässt sich durch das Build-Failure-Analyzer- und das Log-Parser-Plug-in erweitern, sodass automatisiert Gründe für die Fehlschläge festgestellt und zusammengefasst werden.

Weitere Informationen

[1] http://jenkins-ci.org
[2] ttps://wiki.jenkins-ci.org/display/JENKINS/Plugins
[3] https://wiki.jenkins-ci.org/display/JENKINS/Repository+Connector+Plugin
[4] http://blog.sonatype.com/2010/08/introducing-aether/
[5] https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+Maven+Repository+Server
[6] http://docs.codehaus.org/pages/viewpage.action?pageId=116359341
[7] https://wiki.jenkins-ci.org/display/JENKINS/Gradle+Plugin
[8] https://wiki.jenkins-ci.org/display/JENKINS/Deploy+Plugin
[9] https://wiki.jenkins-ci.org/display/JENKINS/WebLogic+Deployer+Plugin
[10] https://wiki.jenkins-ci.org/display/JENKINS/Deploy+WebSphere+Plugin
[11] http://publib.boulder.ibm.com/infocenter/wsdoc400/v6r0/index.jsp?topic=/com.ibm.websphere.iseries.doc/info/ae/ae/trun_app_hotupgrade.html
[12] https://wiki.jenkins-ci.org/display/JENKINS/Job+Generator+Plugin
[13] https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin
[14] https://github.com/jenkinsci/job-dsl-plugin/wiki/Job-DSL-Commands
[15] https://wiki.jenkins-ci.org/display/JENKINS/Subversion+Merge+Plugin
[16] https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View
[17] https://wiki.jenkins-ci.org/display/JENKINS/Sectioned+View+Plugin
[18] https://wiki.jenkins-ci.org/display/JENKINS/Build+Failure+Analyzer
[19] [19] http://graphite.wikidot.com
[20] http://wiki.hudson-ci.org/display/HUDSON/Log+Parser+Plugin

Sebastian Laag
sebastian.laag@adesso.de



Sebastian Laag (Dipl. Inf., Univ.) ist als Senior Software-Engineer bei der adesso AG in Dortmund tätig und arbeitet dort als Entwickler an Java-basierten Web-Anwendungen.

http://ja.ijug.eu/14/4/9

Apache DeviceMap

Werner Keil, Creative Arts & Technologies

Die steigende Anzahl von Mobiltelefonen, Tablets und ähnlichen Geräten, die den Markt geradezu überschwemmen, erleben wir Tag für Tag. Die Spezifikation jedes einzelnen genau zu verfolgen, ist ein Knochenjob. Diese Mühe lässt sich reduzieren, wenn zur Verbesserung ein sogenanntes „Device Description Repository" (DDR) beigesteuert wird und Anwender dieses selbst verwalten können. Apache DeviceMap entstand als Kooperation von OpenDDR und anderen, um ein umfassendes Open-Source-Daten-Repository mit Geräte-Informationen, Bildern und anderen relevanten Informationen für alle Arten von mobilen Geräten wie Smartphones, Tablets, Smart-TV, Automotive etc. zu schaffen.

Mobile Geräte-Erkennung ist beinahe so alt wie die ersten mobilen Geräte mit Netz-Anbindung. Viele frühe Mobiltelefone nutzten dazu längere Zeit den Wireless Application Protocol Standard (WAP). Im Jahr 1999 wurde WAP in der Version 1.1 veröffentlicht, die insbesondere XHTML-Konventionen einbezog. Mit WAP 1.1 konnte sich der Standard auf dem Endgerätemarkt entscheidend durchsetzen und es wurde bald fast jedes Daten-fähige Mobiltelefon mit einem WAP-Browser ausgestattet.

Das nur kurz darauf vorgestellte WAP 1.2 stellte in erster Linie eine Verbesserung von WAP 1.1 dar. Es beinhaltete Erweiterungen wie etwa den Push-Service und sogenannte „User Agent Profile" für WAP-Browser, die es erlaubten, übertragene WAP-Inhalte in ihrer Formatierung komfortabel an die jeweils verwendete Browser-Software anzupassen.

In dieser Zeit ging aus dem WAP-Umfeld ein anderes Protokoll mit „W" hervor, Wireless Universal Resource FiLe (WURFL). Auch wenn das Geschwister-Protokoll WAP mit der Smart-Phone-Ära praktisch in Vergessenheit geriet, war das bis Mitte 2011 quelloffen gepflegte WURFL-Format relativ beliebt.

Kommerzielle Anbieter

Ob das berühmte Wall-Street-Zitat „Greed is Good" von Gordon Gecko die WURFL-Mitbegründer um Luca Passoni veranlasste, das System in die Versenkung zu stoßen, klingt wie der wahrscheinlichste Grund für die Zerstörung des Open-Source-Projekts. Als man die Geräte-Signaturen von unzähligen Mitgliedern der einstigen Community, aber auch kommerziellen Partnern wie der Open Mobile Alliance (OMA), lizenzierte, war dies die faktische Enteignung all jener, die davor ihren Teil zum WURFL-Projekt beigetragen hatten.

Ob PHP-Web-Frameworks wie Zend und Typo3 oder Java-Enterprise-Frameworks von Spring MVC Mobile bis Red Hat JBoss Mobile, alle namhaften Open-Source-Projekte kehrten nach dem proprietären Lizenz-Schluss von WURFL im August 2011 diesem sehr rasch den Rücken, um ihre Nutzer nicht in Lizenz-Konflikte oder gar von WURFL-Vertreiber ScientiaMobile aufgestellte Abo-Fallen zu locken.

Tabelle 1 zeigt eine Übersicht über die gängigsten kommerziellen Produkte am Markt. Während kommerzielle Mitbewerber, bis zu gewissem Grad sicher Vorbilder für WURFL, etwa noch APIs oder sogar vom Umfang her eingeschränkte Device Repositories unter fair nutzbaren Open-Source-Lizenzen von LGPL bis Mozilla anbieten, erlaubt die AGPL gewisser WURFL-APIs bestenfalls deren Nutzung gemeinsam mit dem proprietären und kostenpflichtigen WURFL-Service, also nicht mehr als eine Open-Source-Mogelpackung.

Die Tatsache, dass seit dem Jahr 2011 keine WURFL-Java-APIs mehr im zentralen Sonatype Maven Repository zu finden sind, spricht Bände und zeigt, dass abgesehen von kommerziellen Produkten oder Abos und direkt damit erwerbbaren proprietären APIs WURFL sonst so gut wie tot ist.

So gut wie jeder kommerzielle Mitbewerber von WURFL macht einen solideren und seriöseren Eindruck. Jene, die wie etwa DeviceAtlas oder Volantis jahrelang mit kommerziellen Partnern, Mobilgerätehersteller etc. kooperiert haben, können dadurch ein teils beachtliches Maß an erkennbaren Geräten bieten. Bei Volantis muss erwähnt werden, dass es bereits Anfang 2011 vor der WURFL-Abschottung an Antenna verkauft wurde. Dieses Unternehmen ging wiederum Oktober 2013 an Pegasystems. Seit der ersten Übernahme fand sich auf der Antenna Website nie ein Hinweis auf Volantis oder Produkte auf W3C-DDR-Basis. Es wird also offenbar nicht mehr vertrieben. Andere, insbesondere DeviceAtlas oder MaDDR, sind wie die Open-Source-Vertreter OpenDDR beziehungsweise Apache DeviceMap zum W3C-DDR-Standard kompatibel. Sowohl MaDDR als auch 51Degrees.mobi bieten zumindest in beschränktem Umfang auch Open-Source-Daten oder APIs, deren Nutzung nicht durch Knebelverträge, proprietäre Lizenzen auf „Non Profit" oder Ähnliches eingeschränkt ist, wie bei praktisch allen anderen kommerziellen Anbietern.

Da die Preislisten und -politik der kommerziellen Anbieter teilweise wechseln können und speziell im Enterprise-Segment sehr intransparent sind, kann hier nur sehr bedingt darüber Auskunft gegeben werden, was diese Produkte tatsächlich kosten. In zahlreichen Fällen wurde von früheren WURFL-Unterstützern und -Nutzern berichtet, dass die Vertreter der Anbieter

Projekt	Stärken	Schwächen	Lizenz
MaDDR Projekt	Kompatibel zu W3C Standard	Device Repository funktioniert nur mit kommerziellem mobileAware DDR (APIs beinhalten einfaches Beispiel DDR); das maDDR Projekt bietet keine adaptive Technologie für optimierte Geräte-erkennung	Repository: Nur kommerzielle Lizenz API: Kommerzielle Lizenz oder Simple DDR API mit LGPL Lizenz
DeviceAtlas	Kompatibel zu W3C-Standard, Daten werden von verschiedenen führenden Partnern aus der Industrie geliefert	Nur kommerzielle Lizenz	Repository: Kommerzielle Lizenz API: Kommerzielle Lizenz
Volantis (ab Februar 2011 Antenna, seit Oktober 2013 Pegasystems)	Relativ breite Geräteabdeckung	Nur kommerzielle Lizenz, unklare Zukunft nach doppelter Übernahme	Repository: Kommerzielle Lizenz API: Kommerzielle Lizenz
WURFL	Ehemals gemeinschaftliches Projekt (bis Aug 2011)	Die Lizenz erlaubt keine Nutzung des Repository ohne das kommerzielle API; das API erlaubt keine Nutzung in Projekten mit eigener Lizenz!	Repository: Nutzung ohne WURFL API unzulässig API: Kommerzielle Lizenz, „Alibi“ AGPL, kommerzielle Nutzung unzulässig
51Degrees.mobi	Vorhersehbare Produktpalette, .NET Unterstützung	Eingeschränkter Umfang und Nutzungsmöglichkeit freier Daten	Repository: MPL oder kommerziell („Pro Edition“) API: Mozilla-Public-Lizenz

Tabelle 1: Übersicht kommerzieller Systeme zur Geräte-Erkennung

praktisch wie Steuerfahnder nach Umsatz oder Gewinn fragen und man die monatlichen Abo- oder Lizenzkosten gerne danach ausrichtet, es also gerade für Start-Ups sehr gefährlich sein kann, wenn diese mit der Zeit größere Gewinne machen sollten.

Eine Stichprobe bei DeviceAtlas auf Rückfrage im Q&A-Teil eines kürzlich gehaltenen Vortrags ergab, dass der monatliche Lizenz- oder Abo-Preis dort bei ungefähr 400 bis 500 Euro für kleine Systeme beginnt. Das deckt nur ein oder zwei Entwickler ab sowie eine gewisse Anzahl von Geräte-Erkennungen pro Monat. Kommt man darüber hinaus, weil etwa der Blog oder die Website plötzlich populärer wurde, dann kostet es natürlich mehr. Auch DeviceAtlas schreibt bei größeren Sites und Unternehmenskunden nur „Kosten auf Anfrage“, dazwischen liegen Preise von eher 4.000 bis 5.000 Euro, also rund das Zehnfache der Einstiegsklasse für mittlere Systeme.

Eine frühere Stichprobe auf der WURFL-Seite von ScientiaMobile brachte grob vergleichbare Ergebnisse, der Einstiegspreis dürfte dort allerdings noch etwas höher liegen, wobei längere Benutzer am besten sagen können, ob man für so viel mehr Geld als bei angestammten kommerziellen Mitbewerbern wirklich vergleichbare oder bessere Ergebnisse bekommt.

W3C DDR

Das World-Wide-Web- oder W3-Konsortium (W3C) hat neben sehr bekannten Standards und Empfehlungen wie HTML5 oder CSS auch etwas weniger bekannte Beispiele hervorgebracht, etwa die vor ungefähr zehn Jahren gestartete Initiative zur Schaffung eines einheitlichen Device Description Repository (DDR). Es wurde im Jahr 2008 unter dem Titel „Device Description Repository Simple API“ zuletzt aktualisiert (siehe „http://www.w3.org/TR/DDR-Simple-API“).

Bevor man dort vom Open-Source-Gleis abkam, waren in einigen der W3C-DDR-Arbeitsgruppen auch Namen von Leuten zu lesen, die nun WURFL geschlossen und kommerziell zu vertreiben versuchen. Allerdings gaben diese auf Anfrage recht klar zu verstehen, dass selbst das einst noch nach Open-Source-Methoden entwickelte WURFL (pre 2011) zu keiner Zeit wirklich kompatibel zum W3C-DDR-Standard war.

OpenDDR

Als Reaktion auf die Enteignung der Open-Source-Gemeinschaft durch die WURFL-Verhüllung unterstützt der Autor eine Gruppe von Open-Source-Entwicklern und Web-Designern, die davor bei WURFL aktiv mitgeholfen hatten und sich durch dessen Mauerbau um die Früchte ihrer jahrelangen Arbeit betrogen fühlten, beim Aufbau einer offenen Alternative. „Open-DDR“ war geboren.

GitHub kam dem Take-down-Ansuchen zwar anfangs nach, doch wurde nicht zuletzt dank der Natur von Git rasch auf Basis von Forks durch andere Anwender Ersatz geschaffen. Selbst die Open-Source-Baseline der „WURFL.xml“-Datei ist bis heute an einigen wenigen Orten im Netz zu finden, speziell bei GitHub, was lizenzrechtlich völlig legitim ist. Ebenso das Angebot von OpenDDR, was man bei ScientiaMobile letztlich einsehen musste; GitHub stellte deshalb ab Q1/2012 das OpenDDR-Res-

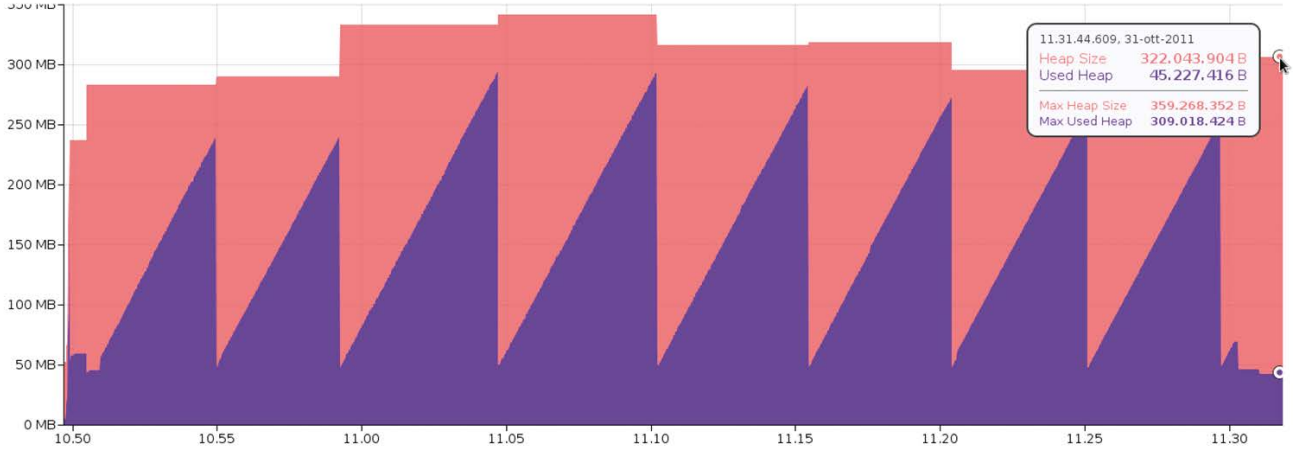


Abbildung 1: JVM-Heap-Speicher von WURFL

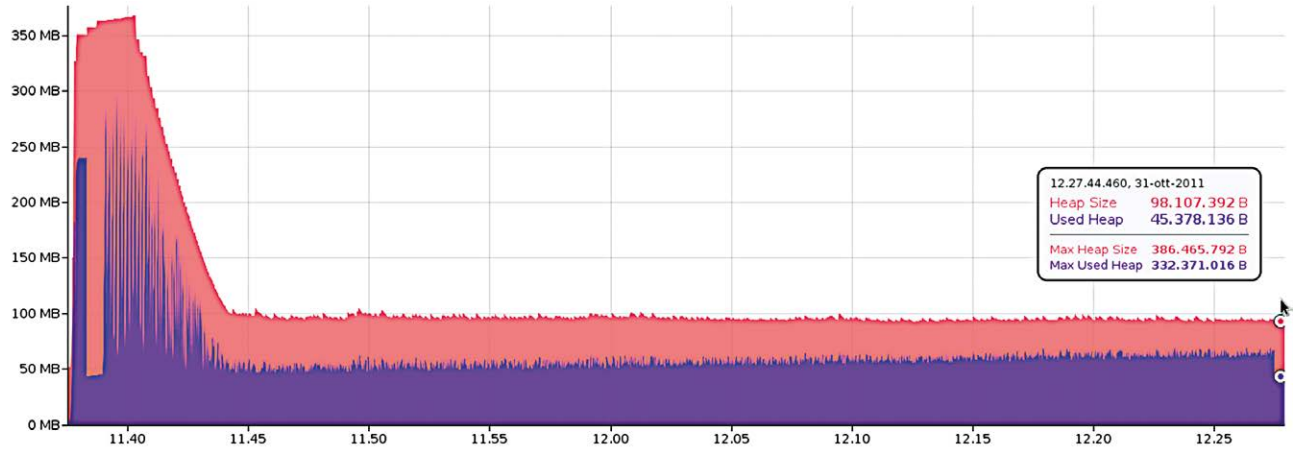


Abbildung 2: JVM-Heap-Speicher von OpenDDR, unreduziert

source-Repository erneut zur Verfügung, wo es seither in regelmäßigen Abständen weiter gepflegt wurde.

Im Herbst des Jahres 2012 fällte ein UK-Höchstgericht die Entscheidung im „Football DataCo"-Prozess (siehe „http://en.wikipedia.org/wiki/Football_DataCo"), den Lizenz-Makler im Auftrag der Britischen Premier League gegen diverse Medien von Yahoo bis hin zu kleinsten Fanclubs oder Fußfallmagazinen geführt hatten. Das Urteil befand das Erheben einer Lizenzgebühr auf Allgemeinwissen, wie es die Ergebnisse von Fußball- der ähnlichen Sportergebnissen darstellen, für unzulässig. Die Kläger mussten jegliche Gerichtskosten bezahlen und, wo gegebenenfalls Gebühren zu Unrecht bezahlt wurden, diese wohl auch rückerstatten. Sobald ein neues mobiles Endgerät auf dem Markt ist, oder manchmal auch schon durch fundierte Vorab-Tests, sind Daten wie etwa die „User Agent"-Signatur und andere Kennzahlen zu diesem Gerät nichts anderes als Fußball- oder andere Sport-ergebnisse, eben Allgemeinwissen. Es ist somit unzulässig, exklusive Gebühren dafür zu erheben. Niemand wird WURFL oder anderen kommerziellen Portalen verbieten, teilweise fünf- oder sechsstellige Beträge von Leuten zu verlangen, die es nicht besser wissen oder bereit sind, für geringfügige Mehr-Information einen vergleichsweise sehr hohen Preis zu bezahlen. Man verbietet ja auch Glücksspiel um teils horrende Summen fast nirgends auf der Welt, schon gar nicht im Internet. Aber diese Anbieter haben kein Recht, anderen Portalen oder Medien die Veröffentlichung der allgemein gültigen Informationen zu verbieten, erst recht, wenn die Geräte bereits im Handel sind.

Neben dem Kostenaspekt ist WURFL (die jeweils letzte verfügbare Version von DB und Client 2011) verglichen mit selbst dem nicht reduzierten OpenDDR-Vokabular deutlich speicherhungriger, wie die Abbildungen 1 und 2 zeigen.

Während nach anfänglicher Initialisierung und Einlesen der Daten der OpenDDR-„W3C Simple"-Client auf einem gleichmäßigen Speicherniveau bleibt, weist der Java-WURFL-Client ständige hohe Spitzen auf, die mehr oder weniger der „Max Heap Size" entsprechen. Das kann insbesondere im Cloud-Umfeld bedeuten, dass entweder ständig Swap Memory konsumiert werden muss, was die Performance empfindlich beeinträchtigt, oder man für gleiche Leistung deutlich mehr in echten Speicher in-

```
oddr.ua.device.builder.path =
   PATH_TO_FILE/BuidlerDataSource.xml
oddr.ua.device.datasource.path=
   PATH_TO_FILE/DeviceDataSource.xml
oddr.ua.device.builder.patch.paths =
   PATH_TO_FILE/BuilderDataSourcePatch.xml
oddr.ua.device.datasource.patch.paths =
   PATH_TO_FILE/DeviceDataSourcePatch.xml
oddr.ua.browser.datasource.path =
   PATH_TO_FILE/BrowserDataSource.xml
ddr.vocabulary.core.path = PATH_TO_FILE/coreVocabulary.xml
oddr.vocabulary.path = PATH_TO_FILE/oddrVocabulary.xml
oddr.limited.vocabulary.path =
   PATH_TO_FILE/oddrLimitedVocabulary.xml
oddr.vocabulary.device =
   http://www.DeviceMap.org/oddr-vocabulary
oddr.threshold = 70
```

Listing 1

```
Service identificationService = ServiceFactory.newService
("org.apache.devicemap.simpleapi.ODDRService",
ODDR_VOCABULARY_IRI, initializationProperties);
```

Listing 2

```
PropertyRef displayWidthRef;
PropertyRef vendorRef;
PropertyRef modelRef;

try {
       displayWidthRef = identificationService.
newPropertyRef("displayWidth");
       vendorRef = identificationService.newPropertyRef("vendor");
       modelRef = identificationService.newPropertyRef("model");
     } catch (NameException ex) {
       throw new RuntimeException(ex);
     }
[…]
```

Listing 3

vestieren muss. Das mag heute nicht mehr so tragisch wirken, aber gerade in hochskalierbaren Servern und Portalen muss man das mit Faktoren von Hundert oder Tausend multiplizieren – ganz zu schweigen vom bereits erwähnten finanziellen „Appetit" von WURFL und ScientiaMobile bei solchen Enterprise-Anwendungen.

Apache DeviceMap

Fast zeitgleich mit der Entstehung des OpenDDR-Projekts machte man sich in der Apache Foundation ähnliche Gedanken und evaluierte verfügbare Open-Source-Lösungen. Diese Evaluierung befand OpenDDR als das vielversprechendste Projekt und im ersten Halbjahr 2012 wurden, neben dem W3C-konformen Device Repository, Java- und .NET-Clients von OpenDDR als Initial Contribution zur Verfügung gestellt (siehe „http://incubator.apache.org/devicemap").

Anwender wollen beziehungsweise müssen manchmal die Betriebssysteme ihrer Geräte aktualisieren (auch eigene benutzerspezifische Builds) und/oder einen neuen Web-Browser installieren können. Die Identifizierung eines Gerätes durch den ursprünglichen User Agent, der von Herstellern bereitgestellt wird, ist dann oft nicht mehr ausreichend. Deshalb betrachtet Apache DeviceMap das Gerät als eine Kombination dreier wichtiger Aspekte:

- Physical Device
- Operating System
- Web-Browser

DeviceMap kann spezielle Versionen ihres Betriebssystems und Web-Browser von Drittherstellern erkennen. Falls die Version eines bestimmten Browsers oder ein Betriebssystem nicht genau bekannt ist, liefert DeviceMap die Information der nächstgelegenen Version statt gar keiner.

DeviceMap erkennt ein Gerät, einen Browser oder ein Betriebssystem mit einem gewissen Vertrauensgrad. Die Anwender können dessen gewünschte Präzision beim Erkennungsprozess selbst bestimmen. Ein größerer Vertrauensgrad kann längere Erkennungszeiten bewirken, während ein geringerer Vertrauensgrad die Erkennung beschleunigt, dabei aber das Risiko weniger präziser Erkennung birgt.

DeviceMap erlaubt auch das Patchen der Datenquelle und implementiert die W3C-Simple-API-Schnittstelle. Es unterstützt das Basisvokabular, das im Dokument zur DDR-W3C-Empfehlung festgelegt wurde. Um das DeviceMap-Simple-API zu nutzen, muss man lediglich Werte einer derartigen Property-Datei anpassen (siehe Listing 1).

Die „oddr.threshold"-Eigenschaft erlaubt dem Entwickler, den gewünschten Vertrauensgrad festzulegen. In diesem Fall wählten wir einen Vertrauensgrad von zumindest 70%. Zur Erstellung eines „Identification Service" nutzt man die „ServiceFactory" der W3C-DDR-„Simple-API.jar" (siehe Listing 2).

Das erste Argument ist die implementierende Klasse des „DDRService", das zweite das Standardvokabular zur Identifikation, falls kein Vokabular explizit angegeben wurde, und das dritte die „DeviceMap Properties"-Datei. Listing 3 zeigt ein kurzes Beispiel, um die Eigenschaften „displayWidth", „model" und „vendor"aus dem Standard-Vokabular zu ermitteln.

Im Zuge der demnächst geplanten Graduation des DeviceMap-Projekts aus dem Incubator hat sich insbesondere auf Java-Seite einiges bewegt. Aktuell noch nicht sofort als Ersatz, aber als sinnvolle Weiter-

```
    //create a client object, store this somewhere permanent
    DeviceMapClient client = new DeviceMapClient();

    //load the device data, do this only once!!!
    client.initDeviceData(LoaderOption.JAR);

    //classify a User-Agent string
    String test = „Mozilla/5.0 (Linux; U; Android 2.2; en; HTC
Aria A6380 Build/ERE27) AppleWebKit/540.13+ (KHTML, like Gecko)
Version/3.1 Mobile Safari/524.15.0“;
    Map<String, String> m = client.classify(test);

    //iterate thru the attributes
    for (String attr : m.keySet()) {
        System.out.println(attr + “: “ + m.get(attr));
    }
```

Listing 4

entwicklung der W3C-DDR-Simple-Implementation, entstand ein neuer Java-Client. Dieser lässt unter anderem das Einbeziehen unterschiedlicher Datenquellen zu, darunter neben Dateisystemen in einem Ordner auch jene direkt aus einer JAR-Datei oder von einer URL beziehungsweise einem vergleichbaren Service. Weitere denkbare Varianten sind auch NoSQL-Dateien etc. Listing 4 zeigt ein kurzes Beispiel, wie das neue DeviceMap-Client-API aufrufbar ist.

Gemeinsam mit der Graduation von Apache DeviceMap aus dem Incubator sind auch vereinfachte Wege zur Datenpflege angedacht – wo technische Infrastruktur und Nachvollziehbarkeit eingereichter Updates (der User Agent eines neuen Geräts etwa direkt vom Gerät ist zulässig, die Kopie aus einem anderen (speziell Closed-Source-)Repository dagegen nicht) es erlauben, mit einem bequemen Web-Service beziehungsweise Web-Interface, das Crowd Sourcing der Pflege neuer Geräte-Signaturen erleichtert. Auch heute ist dies bereits über den JIRA-Issue-Tracker von DeviceMap möglich (siehe „https://issues.apache.org/jira/browse/DMAP/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel“).

Werner Keil
werner@catmedia.us

Werner Keil ist Agile-Coach, Java-Embedded- und Eclipse-RCP-Berater bei einem führenden Anbieter von Embedded- und Real-time-Systemen. Er hilft Global-500-Unternehmen aus Branchen wie Mobil/Telekom, Web 2.0, Finanz, Tourismus/Logistik, Autobau, Gesundheit, Umwelt & Öffentliche Hand sowie IT-Anbietern, darunter Oracle oder IBM. Für einen Mediengiganten entwarf er Mikro-Formate für die Suchmaschine eines Online-Musik-Shops, für eine Privatuniversität ein Portal, das heute gemeinhin als Soziales Netzwerk gilt. Werner Keil entwickelt Enterprise-Systeme mithilfe von Java, JEE, Oracle, IBM oder Microsoft, betreibt Web-Entwicklung mit Adobe, Ajax/JavaScript, dynamischen oder funktionalen Sprachen. Neben seiner Tätigkeit für große Unternehmen betreibt er die eigene New Media Agentur Creative Arts & Technologies, leitet oder unterstützt Open-Source-Projekte, schreibt Songtexte, Romane, Drehbücher und technische Artikel. Werner Keil ist Committer in der Apache und Eclipse Foundation, Babel-Sprache-Champion (deutsch), UOMO Project Lead und aktives Mitglied des Java Community Process, darunter Mitgliedschaft in JSRs wie 321 (Trusted Java), 331 (CP), 333 (JCR), 342 (Java EE 7), 344 (JSF 2.2), 346 (CDI 1,1), 348/364 (JCP.next), 354 (Geld), 363 (Maßeinheiten, dort auch einer der Spec Leads) und einziges KMU-Mitglied des Executive Committee.

http://ja.ijug.eu/14/4/10

Schnell entwickeln – die Vorteile von Forms und Java vereint

René Jahn, SIB Visions GmbH

Mit Forms hat Oracle vor vielen Jahren eines der ersten Produkte für das Rapid Application Development (RAD) auf den Markt gebracht. Damit konnten innerhalb kürzester Zeit mittels PL/SQL-Syntax klassische ERP-/Datenbank-Anwendungen entwickelt werden. Das Produkt wird weltweit von unzähligen Unternehmen eingesetzt und ist nach wie vor nicht wegzudenken.

In den letzten Jahren wurde es jedoch still um Forms, es gab zwar neue Versionen, aber keine neuen Features beziehungsweise Modernisierung. Die Community half sich selbst über die Runden und versuchte mithilfe von Java-Komponenten, neuen Schwung in die Applikationen zu bringen. Doch so richtig glücklich wurde die Forms-Community damit nicht, weil Java sehr komplex wirkt und die Entwicklung dadurch eher verlangsamt als beschleunigt wird. Doch da muss der Autor widersprechen: Mit Java kann genauso, wenn nicht sogar effizienter entwickelt werden als mit Forms. Wenn man die Vorteile beider Technologien vereint, entstehen innerhalb kürzester Zeit modernste Lösungen.

Über Forms

Das Produkt „Forms" wurde erstmals Ende des Jahres 1980 veröffentlicht. Die ersten Versionen waren noch zeichenorientiert (bis Forms 3) und hatten kein GUI, wie wir es gewohnt sind. Im Laufe der Jahre erfolgten zahlreiche Versionssprünge (6/6*i*, 9*i*, 10*g*, 11*g*), Verbesserungen und Technologie-Wechsel. Die letzte wirklich große Veränderung war jedoch der Wechsel von einer Client/Server- zu einer Drei-Tier-Architektur, auch „Web Forms" genannt. Seit dieser Umstellung laufen Forms-Anwendungen als Java-Applet im Webbrowser und die Unterstützung für den Client/Server-Betrieb wurde eingestellt.

Die Beziehung zwischen Oracle Forms und Java besteht also schon seit einigen Jahren und selbst das Forms-GUI läuft in Java. Seitdem ist es auch möglich, eigene Java-Komponenten in Forms zu integrieren, sogenannte „Pluggable Java Components" (PJC) oder „Java Beans".

Die enormen Vorteile von Forms sind die Entwicklung mit PL/SQL-Syntax, die nahtlose Integration der Oracle-Datenbank und die direkte Verwendung von SQL. Somit kann mit einer einzigen Programmiersprache eine komplexe Datenbank-Lösung mit angebundenem GUI entwickelt werden. Doch das in die Jahre gekommene Forms hat auch wesentliche Nachteile: Der Forms Builder, die Entwicklungsumgebung für Forms-Applikationen, ist alles andere als zeitgemäß und wurde auch schon etliche Jahre nicht mehr weiterentwickelt. Sie bietet zwar einen WYSIWYG-Editor, doch es fehlen grundlegende Dinge wie ein Layout-Manager. Die Anbindung von REST, Web-Services oder die Interaktion mit Schnittstellen ist zwar technisch kein Problem und auch lösbar, jedoch weitaus komplexer und zeitaufwändiger, als dies mit Java der Fall wäre.

Am Ende ist Forms zwar ein effizientes und mächtiges Werkzeug, aber angestaubt, proprietär und kaum ansprechend für junge Entwickler. Der fehlende Nachwuchs an Entwicklern wird auch zunehmend zu einem Problem für Forms-Kunden.

Ergänzung durch Java

An dieser Stelle kommt Java ins Spiel, denn es bringt genau das mit, was Forms fehlt. Es ist offen, flexibel, modern und spricht die jungen Entwickler an. Doch Java ist grundsätzlich nur eine Programmiersprache, mit der man beliebige Aufgabenstellungen umsetzen kann. Um die Vorteile von Java in die Forms-Welt zu übernehmen, benötigt man Frameworks, die ähnlich wie Forms funktionieren. Andernfalls können zwar beide Technologien vereint werden, aber die Forms-Entwickler werden sich mit Java nicht anfreunden. Es ist auch eher unrealistisch, dass ein Forms-Entwickler ohne die von Forms bekannten Möglichkeiten mit Java arbeitet. Wenn man beides erfolgreich kombinieren möchte, muss auch der Funktionsumfang vereinbar sein.

ADF oder Apex

Die Vorteile von Java sind natürlich auch Oracle bekannt und so wurde schon früh begonnen, an einem möglichen Nachfolger für Forms zu arbeiten. Das Application Development Framework (ADF) wurde unter anderem auch dafür entwickelt. Es handelt sich dabei um ein auf Java-EE basierendes Framework für die Erstellung von Web-Anwendungen. So vollständig und mächtig ADF auch ist, so groß ist auch die Hürde der Einarbeitung für Forms-Entwickler. Denn mit ADF verlässt man PL/SQL und begibt man sich in die komplexe Welt von Java EE. Auch die gewohnten Entwicklungstools müssen gewechselt werden, denn ohne JDeveloper würde man auf Effizienz verzichten. Es ist auch nicht daran zu denken, die Forms-Entwicklung durch eine Java-Entwicklung zu ersetzen, denn das Applikations-Know-how steckt in den Forms-Entwicklern. Das Problem einer Integration oder gar einer Migration ist daher nicht nur technischer Natur.

Ein weiteres Produkt, das als Forms-Nachfolger infrage kommt, wäre Apex. Der große Vorteil dabei ist die kleinere

Technologie-Hürde, denn mit Apex bleibt man im gewohnten Umfeld von PL/SQL. Es können ebenfalls ansprechende Web-Anwendungen umgesetzt werden – auch komplett ohne Java. Bei komplexeren Aufgabenstellungen führt aber wohl kein Weg an JavaScript vorbei, denn interaktive Web-Anwendungen ohne JavaScript sind nicht denkbar. Die Vorteile von Apex liegen auf der Hand und dennoch ist dieses Produkt kein vollständiger Ersatz für Oracle Forms. Forms ist weitaus umfangreicher und optimierter für Geschäftsanwendungen hinsichtlich Datenmengen und Eingabeverhalten der Benutzer.

Es ist zwar immer stark abhängig von den Anforderungen an eine Geschäftsanwendung, aber in vielen Fällen führen weder ADF noch Apex zum gleichen Ergebnis wie Forms. Wenn Forms also gesetzt ist, sollte auf keinen Fall auf Java verzichtet werden, denn dadurch erhält man Ersatz für fehlende Features und gewinnt vor allem durch die Offenheit.

Forms und Java vereint

Wie bereits erwähnt, lässt sich mit Java nahezu jedes Problem lösen. Ohne geeignete Frameworks ist man jedoch nicht so effizient wie mit Forms. Zu viele Frameworks dürfen es aber auch nicht sein, denn sonst wirkt Java wieder zu kompliziert. Außerdem schafft man Abhängigkeiten, auf die man gerne verzichten würde. Daher sollte die Wahl auf ein Full-Stack-Framework fallen, das allerdings für Desktop-Applikationen geeignet ist. Ein reines Framework für die Entwicklung von Web-Anwendungen ist keine Option, um beide Welten zu vereinen.

Das Open-Source-Framework JVx [1] ist ein interessanter Kandidat, weil es ähnlich wie Forms funktioniert (Events, Trigger, LOVs), weil es ohne Abhängigkeiten auskommt und weil die Nähe zur Datenbank ein großer Vorteil ist. Es ist außerdem auch GUI-Technologie-unabhängig, da es den Single-Sourcing-Ansatz verfolgt. Damit lässt sich eine einmalig entwickelte Applikation am Desktop, im Webbrowser oder als native App am Smartphone betreiben, ohne den Sourcecode zu verändern. Das ist sicherlich auch für Oracle-Forms-Anwendungen eine neue Perspektive und ermöglicht den einfachen Einstieg in die Web- und Mobile-Entwicklung.

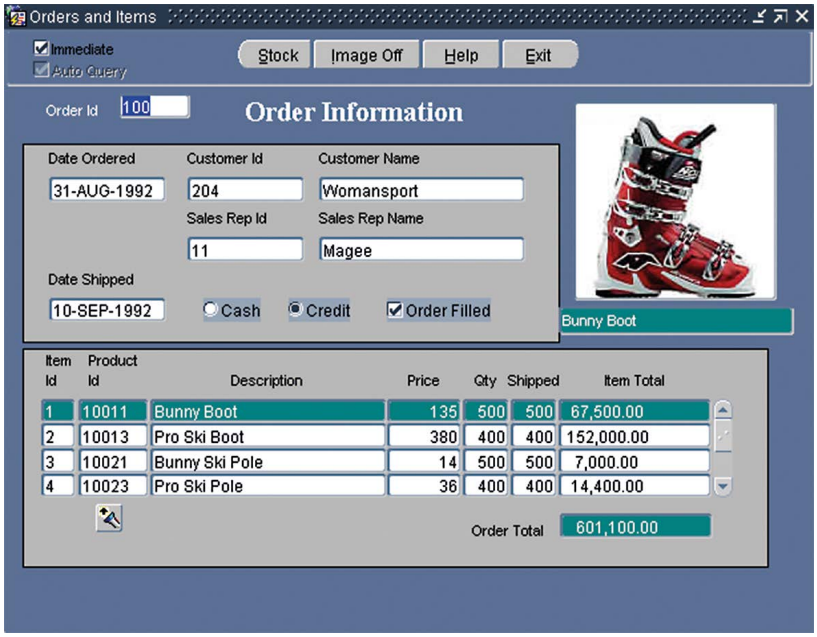


Abbildung 1: Order-Information mit Oracle Forms

Ein konkretes Beispiel: Die offizielle Summit-Demo-Applikation [2] von Oracle, die für Forms und ADF zum Download angeboten wird, verwaltet Kunden, Sportartikel und Angebote. Es existieren mehrere Bearbeitungsmasken, doch wir konzentrieren uns ausschließlich auf die Maske „Order Information" (siehe Abbildung 1). Diese zeigt Bestellungen an. Eine Bestellung enthält den jeweiligen Kunden und alle Artikel inklusive Vorschaubild. Der Zahlungsbetrag wird ausgewiesen und es können weitere Artikel zur Bestellung hinzugefügt werden.

Java-Umsetzung

Im ersten Schritt verwendet der Autor das gleiche Datenmodell und erstellt mit Java eine nahezu identische Maske. Diese kommt anschließend direkt in Oracle Forms anstatt der ursprünglichen Forms-Maske zum Einsatz, um die nahtlose Integration beider Welten zu zeigen. Abschließend wird die Maske auch im Webbrowser ohne Applet laufen.

Die Umsetzung der Java-Maske ist nicht sonderlich aufwändig, da sie keinerlei Logik beinhaltet. Die Codierung beschränkt sich auf die Datenbindung und das Layout. Auch die in Forms übliche Auswahl von Datensätzen mittels Sub-Masken kann entfallen, da Java-übliche Auswahllisten/ComboBoxen zum Einsatz kommen.

```
DBStorage dbsSOrd = new DB-
Storage();
dbsSOrd.setWritebackTable("S_
ORD");
dbsSOrd.
setDBAccess(getDBAccess());
dbsSOrd.open();

DBStorage dbsSItem = new DB-
Storage();
dbsSItem.setWritebackTable("S_
ITEM");
dbsSItem.
setDBAccess(getDBAccess());
dbsSItem.open();
```

Listing 1

Die Datenbindung für die Liste der Produkte ist mit wenigen Code-Zeilen erledigt. Zuerst definiert man den Zugriff auf die Tabelle in der Datenbank. Dafür stellt JVx die Klasse „DBStorage" zur Verfügung (siehe Listing 1). Diese übernimmt sämtliche CRUD-Vorgänge und es ist kein zusätzlicher Code erforderlich. Man liest alle Bestellungen („dbsSOrd") aus der Tabelle „S_ORD" und findet die zugehörigen Artikel („dbsSItem") in der Tabelle „S_ITEM".

Im nächsten Schritt geht es um die Anbindung der GUI-Tabelle an das Datenobjekt. Dazu bietet JVx mit der Klasse „RemoteDataBook" die Möglichkeit, auf das Datenobjekt standardisiert zuzugreifen

(siehe Listing 2). Für den Zugriff auf die Bestellungen („dbsSOrd“) kommt das DataBook „rdbSOrd“ und für den Zugriff auf die zugeordneten Artikel („dbsSItem“) das DataBook „rdbSItem“ zum Einsatz. Abschließend benötigt man noch ein GUI-Element für die Anzeige der Daten (siehe Listing 3). Dazu wird die Tabelle „navItems“ mit dem DataBook „rdbSItem“ verknüpft. Natürlich fehlt jetzt noch das Layout, aber darauf wird an dieser Stelle verzichtet, da es sich um üblichen GUI-Code mit Positionierung von Komponenten handelt. Die in Java entwickelte Maske sieht nun sehr ähnlich zum Original in Forms aus (siehe Abbildung 2) und deckt denselben Funktionsumfang ab.

Im Unterschied zu Forms bietet die Java-Variante in jeder Tabelle Sortier- und verschiebbare Spalten, Auswahllisten in- und außerhalb von Tabellen sowie Datensatz-Operationen (Einfügen, Löschen, Suche, Export) direkt bei den Daten.

Sieht aus wie Forms, ist aber Java

Bisher wurde in zwei unterschiedlichen Technologien die gleiche Anforderung umgesetzt. Das Besondere daran ist jedoch, dass die Java-Variante äußerst wenig Code benötigt und jeder Forms-Entwickler ohne große Schulung damit umgehen kann. Es sind auch keine Model-Definitionen sowie keine XML-Konfigurationen erforderlich und es ist mit zwei Java-Klassen eine saubere Drei-Tier-Lösung entstanden. Das Beste kommt natürlich zum Schluss, denn nun wird die Java-Maske in die bestehende Forms-Anwendung eingebettet, um die Vorteile von Java in Forms bereitzustellen. Es besteht dadurch die Möglichkeit, neue Anforderungen mit Java umzusetzen und dann in Forms zu integrieren, oder aber auch, die Forms-Applikation schrittweise durch Java zu modernisieren.

Die lauffähige Forms-Applikation ist voll funktionsfähig und bietet die gewohnten Möglichkeiten. Abbildung 3 zeigt die Java-Maske innerhalb einer Forms-Anwendung. Es wurde absichtlich der Forms-Applikationsrahmen belassen und das Fenster enthält einen zusätzlichen Close-Button, der direkt mit Forms hinzugefügt wurde.

Die Integration einer Java-Maske in Forms ist relativ einfach, denn Forms erlaubt die Einbindung von PJC oder Java Beans. Abbildung 4 zeigt, wie zu einem be-

```
RemoteDataBook rdbSOrd = new RemoteDataBook();
rdbSOrd.setName("order");
rdbSOrd.setDataSource(getDataSource());
rdbSOrd.open();

RemoteDataBook rdbSItem = new RemoteDataBook();
rdbSItem.setName("item");
rdbSItem.setDataSource(getDataSource());
rdbSItem.setMasterReference(
   new ReferenceDefinition(new String[] { "ORD_ID" },
                           rdbSOrd, new String[] { "ID" }));
rdbSItem.open();
```

Listing 2

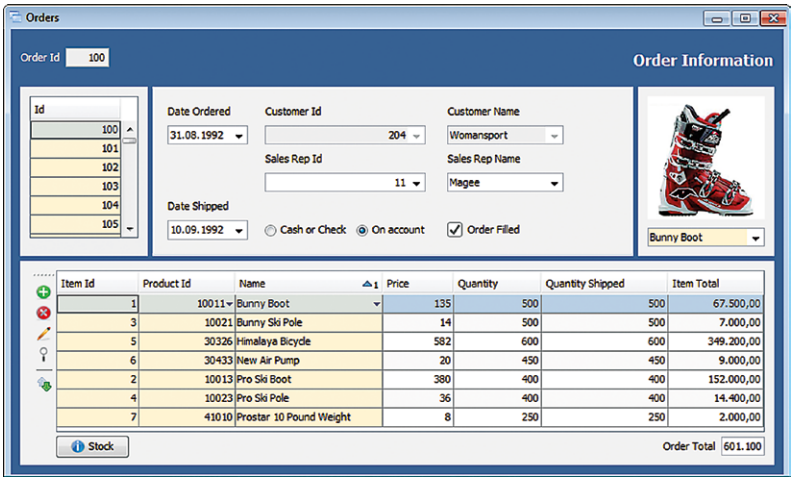


Abbildung 2: Order-Information mit Java

```
NavigationTable navSItem = new NavigationTable();
navSItem.setDataBook(rdbSItem);
```

Listing 3

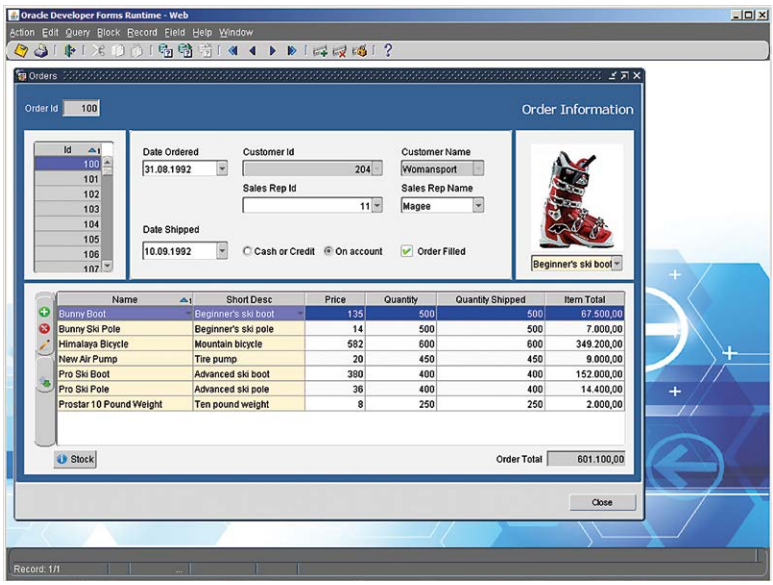


Abbildung 3: Order-Information mit Java, eingebettet in Forms

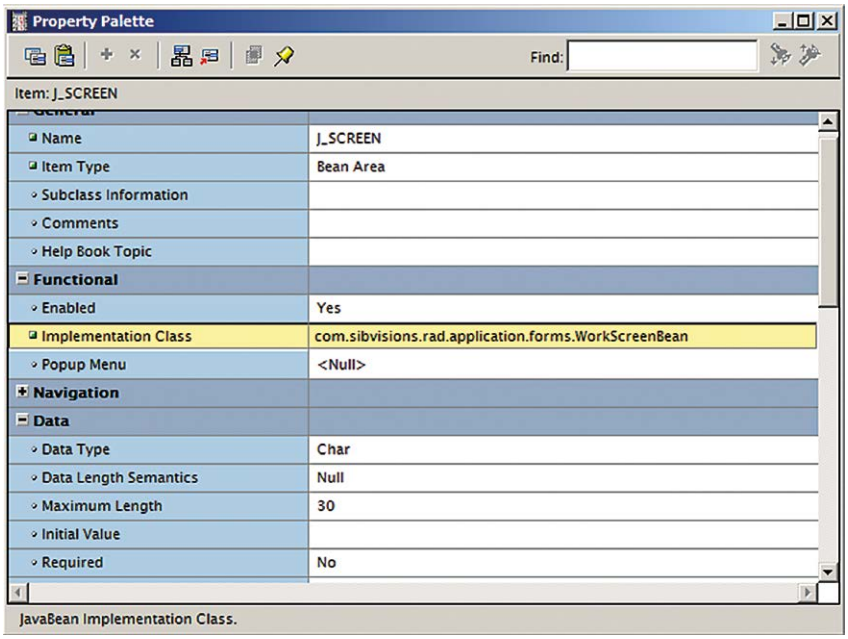


Abbildung 4: Java-Bean-Konfiguration

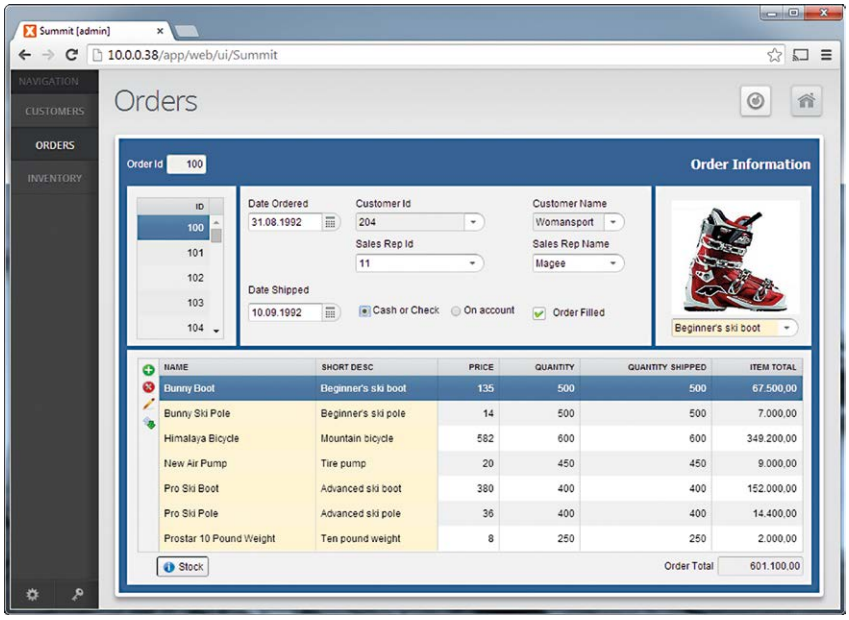


Abbildung 5: Das Ergebnis

reits vorhandenen Canvas eine Java Bean hinzugefügt und die Java-Klasse konfiguriert wurde.

Fazit

Die Integration von Java in Forms bietet vollkommen neue Möglichkeiten für die Forms-Entwicklung. Es stehen beispielsweise ansprechende „Look & Feels" zur Verfügung, um die Applikation aufzupeppen. Auch die Anzahl der GUI-Controls (Tree, TreeTable, Charts, Kalender) ist um ein Vielfaches höher.

Es gibt aber auch einen Produktivitätsgewinn durch die automatische Unterstützung von Mehrsprachigkeit, Layout-Manager für die automatische Größenanpassung von Masken, Drag & Drop, Upload- und Download von Dateien ohne Zusatzaufwand, einfache Integration von Open-Source-Reporting-Tools wie BIRT [3] oder JasperReports [4].

Durch den Einsatz von JVx sind noch weitere Anwendungsfälle möglich, die ansonsten nur mit erheblichem Mehraufwand realisierbar gewesen wären. Eine Applikation ist auf unterschiedlichen Plattformen und mit unterschiedlichen Technologien zukunftssicher einsetzbar. Als kleine Demonstration wird die zuvor entwickelte Java-Maske – ohne Sourcecode-Änderung – im Webbrowser als HTML5-Anwendung betrieben (siehe Abbildung 5). Durch die UI-Abstraktion von JVx könnte das Backend als Forms-Applikation laufen, das Frontend als HTML5-Applikation und für Benutzer, die ständig unterwegs sind, ist auch eine Smartphone-Lösung möglich. Aber unabhängig von der eingesetzten Technologie muss am Ende des Tages eine Lösung entstehen, die übersichtlich, wartbar und auch in zehn Jahren noch lauffähig ist. Genau das ist möglich, wenn Forms und Java vereint im Einsatz sind.

Links

[1] JVx: http://sourceforge.net/projects/jvx
[2] Summit Applikation: https://blogs.oracle.com/jdevotnharvest/entry/adf_summit_sample_application_a
[3] BIRT: http://www.eclipse.org/birt
[4] JasperReports: http://www.jaspersoft.com

René Jahn
rene.jahn@sibvisions.com

René Jahn ist Mitbegründer der SIB Visions GmbH und Head of Research & Development. Er verfügt über langjährige Erfahrung im Bereich der Framework- und API-Entwicklung. Sein Interessenschwerpunkt liegt auf der Integration von State-of-the-Art-Technologien in klassische Business-Applikationen. Darüber hinaus betreut er die Open-Source-Sparte bei SIB Visions und veröffentlicht regelmäßig Artikel im Unternehmensblog.

http://ja.ijug.eu/14/4/11

Oracle-ADF-Business-Service-Abstraktion: Data Controls unter der Lupe

Hendrik Gossens, OPITZ CONSULTING GmbH

Oracle bietet mit seinem Application Development Framework (ADF) die Möglichkeit, Enterprise-Applikationen auf deklarative Art und Weise zu erstellen. Dies wird unter anderem durch die ADF-Model-Schicht ermöglicht, die verschiedene Business Services abstrahiert und für diese ein einheitliches API bereitstellt. Ein Bestandteil der ADF-Model-Schicht sind die sogenannten „Data Controls", die eine Art Adapter für den Business Service darstellen. Dieser Artikel zeigt deren Rolle bei der Entwicklung mit ADF.

Betrachtet man die Architektur des Application Development Frameworks (ADF), so wird deutlich, dass ADF das Model View Controller (MVC) Design Pattern implementiert. Es erfolgt also eine klare Trennung zwischen Code für das Datenmodell, der grafischen Benutzeroberfläche und der Applikationssteuerung. Klassische MVC-Architekturen implementieren in der Regel genau diese drei Schichten. Die Model-Schicht setzt in Form von Business Services wie beispielsweise Enterprise JavaBeans (EJB) die Geschäftslogik um und zeichnet sich für die Interaktion mit der Datenquelle verantwortlich.

Auf der View-Schicht-Ebene ist die visuelle Repräsentation realisiert, während sich die Controller-Schicht um die Anwendungssteuerung und die Kommunikation zwischen View- und Model-Schicht kümmert. Alle drei Schichten sind lose gekoppelt, sodass der Grad an Wartbarkeit der Anwendung und Wiederverwendbarkeit einzelner Module möglichst hoch ist. Die MVC-Implementierung von ADF umfasst vier statt der sonst üblichen drei Schichten. Grund dafür ist, dass ADF mit dem ADF Model (ADFm) eine eigene Model-Schicht einführt, die heterogene Business Services abstrahiert und den darüberliegenden Schichten eine einheitliche Schnittstelle zu den Business Services bereitstellt. Die Business Services selbst können als eine Art „Backend-Model" betrachtet werden.

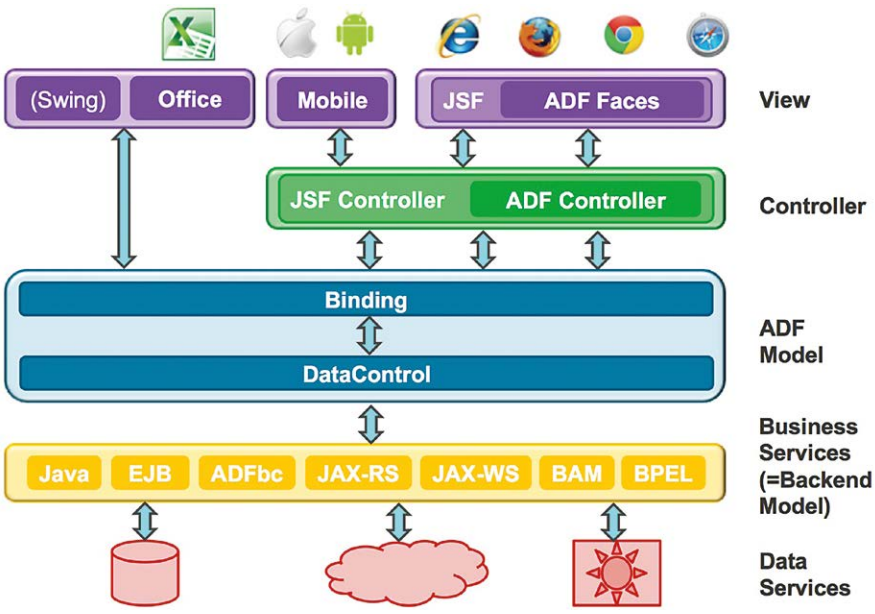


Abbildung 1: MVC-Pattern in ADF

Abbildung 1 zeigt die vier Schichten der MVC-Implementierung von ADF. Im Einzelnen werden in den Schichten folgende Aufgaben realisiert:

- *Business-Service-Schicht*
  Daten aus verschiedenen Quellen und Umsetzung von Geschäftslogik
- *Model-Schicht*
  Abstrahiert die Business-Service-Schicht und ermöglicht der Controller- und der View-Schicht, gegen ein einheitliches API zu arbeiten
- *Controller-Schicht*
  Kontrolliert den Anwendungsfluss
- *View-Schicht*
  Stellt das User-Interface bereit

Aus der Sicht des Frameworks kann man die Architektur von ADF also auch als klassisches MVC-Modell mit einer zusätzlichen Business-Service-Schicht verstehen.

Die ADF-Model-Schicht im Detail

Wie in Abbildung 1 dargestellt, umfasst die ADF-Model-Schicht zwei Kernbestandteile, zum einen Data Controls, zum anderen Data Bindings. Während Data Controls eine Abstraktion der Business Services darstellen, verknüpfen Data Bindings die aus dem Business Service erhaltenen Daten mit der UI-Schicht. Die Bindings einer Seite referenzieren Data-Control-Objekte, um Daten für die UI-Komponenten bereitzu-

stellen. Die Konfigurationen von Bindings und Data Controls werden in XML-Dateien vorgehalten.

Im Falle der Bindings wird für jede Seite, die datengebundene Komponenten enthält, ein „Page Definition File" generiert. Die Entwicklungsumgebung JDeveloper legt dieses automatisch an, sobald auf einer Seite eine datengebundene Komponente aus einem Data Control erstellt wird. Es trägt standardmäßig den Namen der Seite mit dem Suffix „pageDef". Zur Laufzeit werden die darin definierten Bindings instanziiert und in einer „Binding Container" genannten „request scoped"-Map vorgehalten. Ein Binding Container enthält also die Gruppe der Bindings, die auf der aktuellen Seite verwendet werden.

Binding Container und referenzierte Data Controls werden im sogenannten „Binding Context" verwaltet. Dieser ist zur Laufzeit eine Java Map, die Referenz-Objekte auf alle Data Controls und Page Definitions der Applikation enthält. Die eigentlichen Data-Control- oder Binding-Container-Objekte werden „on demand" aus den Referenz-Objekten erstellt und wieder freigegeben, wenn sie nicht mehr notwendig sind. Zentrale Konfigurationsdatei für den Binding Context ist standardmäßig die Datei „DataBindings.cpx".

Data-Control-Typen und Konfigurationsdateien

ADF bringt eine Vielzahl vorimplementierter Data Controls mit. Beispiele für solche vorimplementierten Data Controls sind in Abbildung 1 in der Business-Service-Schicht zu erkennen. Für jeden der dort aufgeführten Backend Services existiert eine korrespondierende Data-Control-Implementierung in ADF. Dabei fällt auf, dass Oracle für nahezu jeden bekannten Business Service Data-Control-Implementierungen mitliefert.

Neben Oracle-Fusion-Middleware-spezifischen Data Controls wie BAM, BI und ADF Business Components (ADFbc) gibt es auch Data Controls für Java EE Standard Backend Services wie EJB/JPA, WebServices etc. Das API der Data Controls ist einheitlich, jedoch teilweise im Funktionsumfang unterschiedlich. Dies ist abhängig von den Möglichkeiten des Business Service. So kann das URL Data Control beispielsweise XML-Daten liefern und diese „read only" anzeigen, während ein Data Control auf einer EJB Session Bean oder Java Service Facade auch die Persistierung von Daten ermöglicht.

Grundsätzlich unterscheidet man zwischen zwei Haupt-Kategorien der Data Controls. Die einen basieren auf einem ADFbc Application Module, die anderen umfassen die sogenannten „Adapter Data Controls". Bei den Business Components handelt es sich um das Oracle-ORM-Framework. Es integriert direkt in das ADF Model und benötigt keine weiteren Konfigurationsdateien, um die Struktur und weitere Metadaten des Data Control zu beschreiben.

Die Adapter Data Controls agieren als Adapter für Nicht-ADF-spezifische Business Services. Erstellt man Wizard-gestützt ein Adapter Data Control, wird automatisch eine Konfigurationsdatei namens „DataControls.dcx" angelegt, die eine Data-Control-Definition im XML-Format enthält. Adapter Data Controls können darüber hinaus um weitere Metadaten angereichert werden, die in sogenannten „Structure Definition Files" ebenfalls im XML-Format hinterlegt sind. Sie ermöglichen die Konfiguration von Eigenschaften, die bei Verwendung der ADFbc als Business Service bereits auf Ebene der Business Services konfiguriert werden können:

- UI Hints wie Texte für Tooltips, Labels etc.
- Standardwerte für Attribute
- Transiente Attribute
- Einschränkung der Wertemenge durch Named Criteria bei JPA-basierten Data Controls
- Validierungsregeln
- Definition von List of Values

Abbildung 2 zeigt die Konfigurationsdateien, die das Framework im gesamten Binding Context verwendet.

Betrachtet man den Data-Control-Teil des Binding Context, so wird deutlich, dass die Anzahl an Metadaten schnell recht groß werden kann. Pflegt man beispielsweise im Falle eines EJB Data Control Metadaten für die Entities, so wird pro Entity ein eigener Structure Definition File generiert. Oracle hat dies scheinbar als kleine Unschönheit im Framework identifiziert und liefert evolutionär mit neueren JDeveloper-Versionen Verbesserungen in diesem Bereich. Während bis einschließlich JDeveloper/ADF 11g R1 auch dann Structure Definition Files pro Entity generiert wurden, wenn für eine Bean gar keine Zusatzinformationen gepflegt wurden, ist mit Einführung des JDeveloper/ADF 11g R2 das Verhalten dahingehend optimiert, dass ein Structure Definition File nur dann angelegt wird, wenn auch tatsächlich Metadaten gepflegt werden, die über das Standardverhalten hinausgehen. Oracle bezeichnet dies als „Sparse Bean Data Control". Die im Juni 2014 erschienene dritte Version von ADF 12c (12.1.3.0.0) ermöglicht es sogar, komplett auf XML-basierte Strukturinformationen zu verzichten und stattdessen Annotations in der Bean-Klasse zu verwenden.

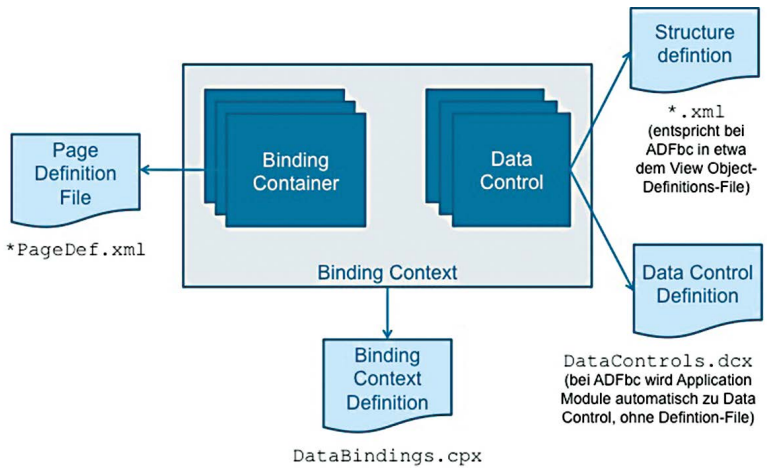


Abbildung 2: Konfigurationsdateien im Binding Context

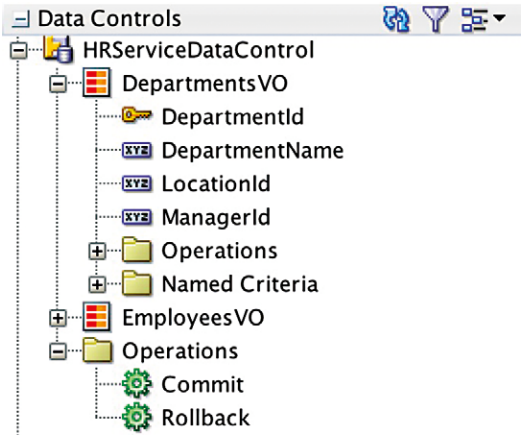


Abbildung 3: Repräsentation eines Data Control

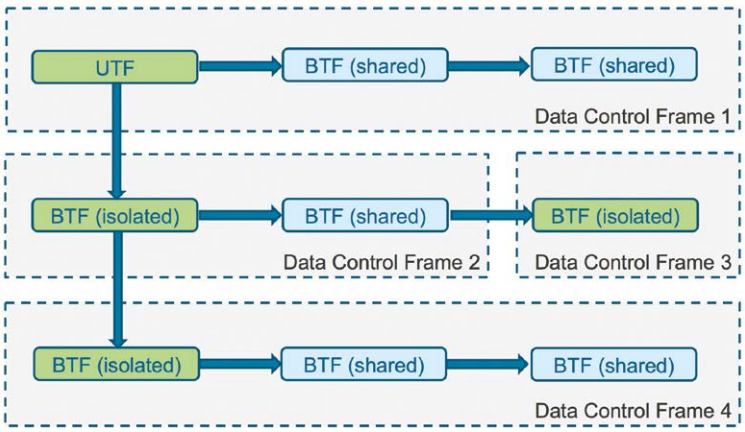


Abbildung 4: Data Control Frames

Für Entwickler mit Datenbank-Hintergrund, die ADF erlernen möchten, bietet es sich an, die ADF Business Components als Business Service zu verwenden. Die dort umgesetzten Konzepte orientieren sich an der relationalen Welt. Entwickler mit Java-EE-Hintergrund können ihre EJB-Kenntnisse in ein ADF-Projekt einbringen, in dem beispielsweise EJB als Business Service zur Anwendung kommt.

Das Data Control stellt Collections, Attribute und Operationen des Business Service bereit. Diese Informationen werden in der UI-Schicht verwendet, um Daten zu visualisieren und zu manipulieren. Die bereitgestellten Operationen können sowohl von der View- als auch von der Controller-Schicht eingebunden werden. Abbildung 3 zeigt beispielhaft die Struktur eines Data Control. Die dort gezeigten Collections, Attribute und Operationen können per „Drag & Drop" auf eine JSF-Seite gezogen und als Komponente visualisiert werden.

Managed Transactions von Data Controls

Business Services können ihre Daten aus verschiedenartigen Quellen wie Web Services, Dateien, BPEL-Engines etc. erhalten. Bei Data Controls, deren Quelldaten in einer Datenbank persistiert sind, ist das Transaktionsmanagement von großer Bedeutung. Besonders gut ausgeprägt sind die Transaktionsmöglichkeiten bei Verwendung des ADFbc Data Control. Aber auch Adapter Data Controls, die ihre Daten via Java-Persistence-API (JPA) verwalten, bieten die Möglichkeit, Transaktionen deklarativ zu steuern. Ein Beispiel dafür ist ein Data Control, das auf einer Java Service Facade basiert.

Der ADF-Controller verwaltet die Datenbank-Transaktionen. Zentrale Bausteine sind dabei die sogenannten „Taskflows". Diese sind in etwa mit dem JSF Page Flow vergleichbar, bieten aber vielfältigere Komponenten. Es gibt zwei Typen von Taskflows: „unbounded" (UTF) und „bounded" (BTF). Während es pro Anwendung einen UTF gibt, der den Einsprung in die Applikation darstellt, eignen sich BTFs für die Modularisierung des Anwendungsflusses. BTFs können als Regionen in Seiten eingebettet und von anderen Taskflows aufgerufen werden. Sie erlauben dabei die deklarative Transaktionsverwaltung durch den ADF-Controller.

Um zu verstehen, wie dieser das Transaktionsmanagement übernimmt, ist es zunächst wichtig zu wissen, dass es mit „shared" und „isolated" zwei Data Control Scopes gibt, zwischen denen der BTF wählen kann. Definiert der BTF einen Shared Data Control Scope, teilt er sich die Data-Control-Instanzen mit dem aufrufenden Task Flow. Ist der Data Control Scope als „isolated" definiert, instanziiert der Taskflow eigene Data-Control-Instanzen. Dafür stehen vier verschiedene Konfigurationsmöglichkeiten zur Verfügung:

- *No Controller Transaction*
  Der aufgerufene Taskflow ist nicht Bestandteil des Transaktionsmanagements des ADF-Controllers. Die Transaktionssteuerung wird an den Business Service delegiert.
- *Always Begin New Transaction*
  Der BTF eröffnet eine neue Transaktion, unabhängig davon, ob bereits eine existiert. Ist der Data Control Scope des BTF „shared" und ist für die geteilten Data Controls bereits eine Transaktion geöffnet, wird eine Exception geworfen.
- *Always Use Existing Transaction*
  Der BTF nutzt eine existierende Transaktion oder wirft eine Exception, falls keine existiert.
- *Use Existing Transaction If Possible*
  Der BTF nutzt eine existierende Transaktion, falls vorhanden. Ansonsten wird eine neue Transaktion eröffnet.

Durch die Wahl der Data Control Scopes der BTFs ergeben sich in Kombination mit den Transaktionsoptionen sogenannte „Data Control Frames". Diese geben an, welche Taskflow- und Data-Control-Instanzen eine Transaktion bilden. Der UTF einer Applikation sowie jeder Isolated Data Control Scope Taskflow spannen einen neuen Data Control Frame auf. Definiert ein Taskflow einen Shared Data Control Scope, so wird dieser dem Data Control Frame des aufrufenden Taskflow hinzugefügt. Zu einem bestimmten Zeitpunkt kann immer nur ein Data Control Frame aktiv sein. Abbildung 4 skizziert den Aufbau von Data Control Frames beim Aufruf von BTFs. In Klammern ist jeweils der für den BTF gewählte Data Control Scope angegeben.

Data Controls, deren darunterliegende Business Services Transaktionen erlauben, stellen in der Regel „commit"- und

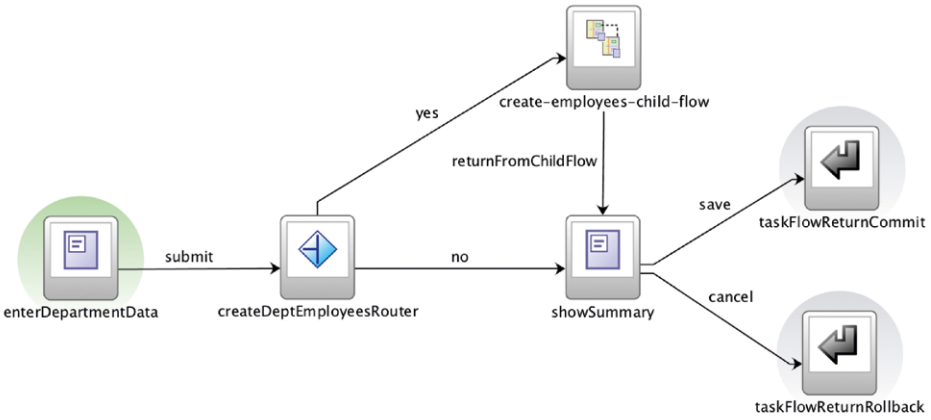


Abbildung 5: BTF mit Child-BTF und Taskflow Return Activities

„rollback"-Operationen zur Verfügung, wie in Abbildung 3 dargestellt. Neben diesen Operationen können BTFs auch Task-Flow-Return-Komponenten enthalten, die ihrerseits definieren, wie die Transaktion am Ende des Taskflow beendet werden soll. Hier stehen ebenfalls die Optionen „commit" oder „rollback" zur Verfügung. Sie rufen lediglich die Operation für die korrespondierende Data-Control-Instanz auf, während die Task-Flow-Return-Komponenten des BTF die Operationen für den gesamten Data Control Frame durchführen.

Diese Unterscheidung ist besonders wichtig bei Isolated Data Control Scope BTFs, die ihrerseits einen Shared Data Control Scope aufrufen. Werden dort die Operationen eines Data Controls verwendet, so werden die Daten der anderen Data Controls nicht gespeichert oder zurückgerollt. Abbildung 5 zeigt einen Isolated Data Control BTF, der einen Shared Data Control BTF aufruft und die Transaktion mit Taskflow Return Activities abschließt.

Im Gegensatz dazu darf ein BTF mit der Transaktionsoption „Always Use Existing Transaction" die Transaktion nicht schließen. Werden in einem BTF mit dieser Option Taskflow Return Activities verwendet, so wird dies zur Entwicklungszeit als Fehler markiert. Die Anwendung kann aber trotzdem deployt werden. Zur Laufzeit ignoriert der ADF-Controller die fälschlicherweise konfigurierten Optionen der Taskflow Return Activities und ruft diese stattdessen mit der Option „<Default> None" auf.

Fazit

Die ADF-Model-Schicht und der ADF-Controller vereinfachen die Entwicklung von Java-EE-Applikationen dahingehend, dass sie die deklarative Entwicklung im 4GL-Stil begünstigen und dabei implizit etablierte Design-Pattern einsetzen. Die Abstraktion von Business Services durch Data Controls beschleunigt die Entwicklung, da diese gegen eine „Business Service"-unabhängige, konsistente Schnittstelle erfolgen kann. Am besten integriert man ADF Business Components in dem restlichen Framework.

Bei den Nicht-ADF-spezifischen Adapter Data Controls sind teilweise zusätzliche Metadaten notwendig, um eine vergleichbare Funktionalität zu ADFbc-basierten Applikationen zu bekommen. Unter der Vielzahl der Metadaten-Dateien kann dabei bei extrem großen Applikationen die Übersichtlichkeit in der Quellcode-Struktur etwas leiden. Oracle scheint dies zumindest als kleinere Unschönheit in seinem ADF erkannt zu haben und bietet ab der Version 12.1.3.0.0 die Möglichkeit, Metadaten über Annotationen direkt in der Bean-Klasse zu pflegen.

Unabhängig davon bleibt festzuhalten, dass die in Oracle ADF umgesetzten Konzepte – zu denen auch die Business-Service-Abstraktion durch Data Controls zählt – die Entwicklung skalierbarer und hochverfügbarer Enterprise-Applikationen mit geringem Entwicklungsaufwand ermöglichen.

Quellen

- http://docs.oracle.com/cd/E15586_01/web.1111/b31974/appendixa.htm#CDJCBIGB
- http://docs.oracle.com/cd/E15586_01/web.1111/b31974/bcdcpal.htm#CEGJFBAD
- http://docs.oracle.com/cd/E15586_01/web.1111/b31974/bcdcpal.htm#CHDBFCFC
- http://download.oracle.com/otn_hosted_doc/jdeveloper/1012/developing_mvc_applications/adf_aoverviewdatacontrols.html
- http://biemond.blogspot.de/2013/07/jdeveloper-1212-ejb-java-service-facade.html
- http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adf-task-flow-trans-fund-v1-1-1864319.pdf

Hendrik Gossens
hendrik.gossens@opitz-consulting.com

Hendrik Gossens ist Consultant bei der OPITZ CONSULTING Deutschland GmbH. Sein beruflicher Schwerpunkt liegt dort in der Beratung, dem Training und der Umsetzung von Software auf der Basis von Oracle Produkten. Seit 2009 liegt sein Fokus auf Fusion Middleware und ADF-basierten Softwareprojekten.

http://ja.ijug.eu/14/4/12

Neue Features in JDeveloper und ADF 12*c*

Jürgen Menge, Oracle Deutschland B.V. & Co. KG

Um eine integrierte Entwicklungsumgebung für die Fusion Middleware 12c zur Verfügung zu stellen, bietet die neue Version von JDeveloper und ADF zahlreiche neue Features.

JDeveloper ist eine von drei Java-Entwicklungsumgebungen, die mit dem Hause Oracle verbunden sind. Im Unterschied zum Oracle Enterprise Pack for Eclipse (OEPE) und NetBeans verantwortet Oracle allein die Weiterentwicklung des JDeveloper. Grund dafür ist, dass Oracle JDeveloper für die Entwicklung der Fusion Applications und der Fusion Middleware nutzt. Er steht darüber hinaus auch Partnern und Kunden zur Verfügung, die die Fusion Middleware einsetzen beziehungsweise eigene Applikationen entwickeln wollen.

Bei der Entwicklung von Java-Anwendungen kommt das Oracle Application Development Framework (ADF) zum Einsatz. Es integriert verschiedene Standards (wie EJB/JPA, JAX-WS, JAX-RS) und ergänzt diese um eigene Implementierungen und Frameworks (wie ADF Business Components, ADF Faces). Typische Web-Anwendungen auf Basis von Oracle ADF setzen im Business Service Layer wahlweise das Framework ADF Business Components oder EJB/JPA und im View Layer ADF Faces mit dem zugehörigen ADF Controller ein (siehe Abbildung 1).

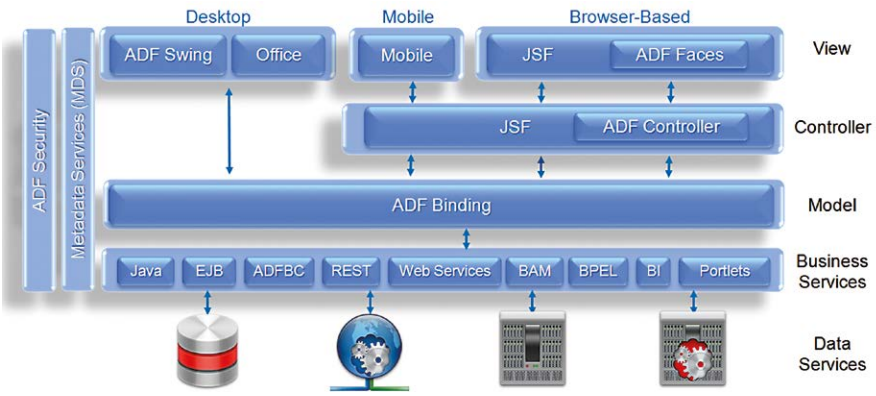


Abbildung 1: Oracle-ADF-Architektur

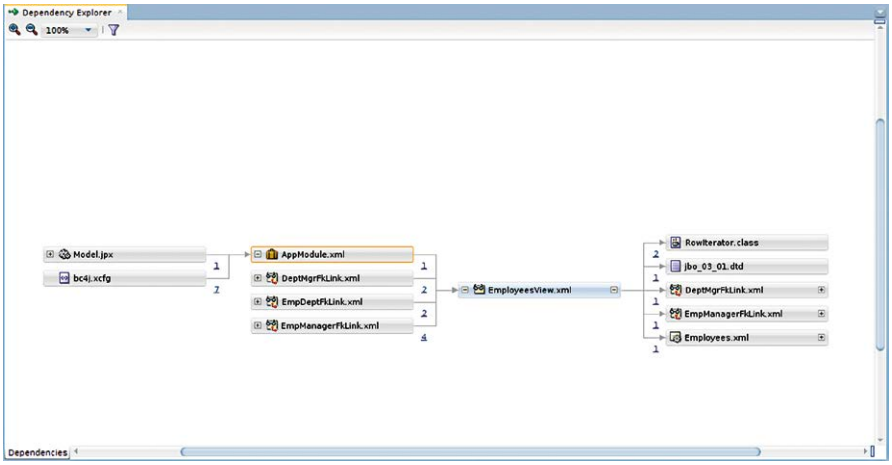


Abbildung 2: Dependency Viewer im Oracle JDeveloper

Oracle ADF muss für die Laufzeitumgebung lizenziert sein. Mit ADF Essentials gibt es allerdings eine kostenfreie Edition mit einem eingeschränkten Funktionsumfang.

JDeveloper unterstützt als Entwicklungsumgebung den vollen Funktionsumfang von ADF, wohingegen OEPE nur jene Bestandteile von ADF unterstützt, die für Java EE-Entwickler interessant sind.

Ende Juni 2014 ist die Version 12.1.3 des JDeveloper zeitgleich mit dem OEPE 12.1.3 und einer großen Zahl von Fusion Middleware-Produkten erschienen. Mit dieser Version steht auch das Mobile Application Framework (MAF) 2.0 als Extension sowohl für JDeveloper als auch für Eclipse (OEPE) zur Entwicklung von hybriden Mobil-Applikationen zur Verfügung. Der Artikel gibt einen Überblick über die im aktuellen Release 12*c* verfügbaren neuen Funktionalitäten, der aufgrund von deren Vielzahl nicht den Anspruch auf Vollständigkeit erfüllt. Für Entwickler besteht die Möglichkeit, den JDeveloper sowie OEPE kostenfrei aus dem Oracle Technology Network (siehe „http://www.oracle.com/technetwork/developer-tools“) herunterzuladen und zu installieren, um mit den neuen Möglichkeiten praktisch zu arbeiten.

Neue JDeveloper-Features

Oracle JDeveloper ist komplett in Java geschrieben und verwendet in der Version 12*c* das JDK7. Er nutzt das Modulkonzept von OSGi, sodass benötigte Funktionen erst bei Bedarf geladen werden. Für Testzwecke ist ein WebLogic Server 12*c* in die

IDE integriert. Ab JDeveloper 12.1.3 kann die Java SE 8 zum Compile und Test von Applikationen genutzt werden.

Die Teams von JDeveloper und NetBeans tauschen Entwicklungsergebnisse aus. So wird im JDeveloper 12*c* das Windowing-System von NetBeans verwendet, das viele Verbesserungen für den Benutzer bringt. Ebenfalls neu ist der Dependency Viewer, in OEPE unter dem Namen „AppXray" bekannt. Er zeigt die Abhängigkeiten zwischen Artefakten einer Applikation in grafischer Form an (siehe Abbildung 2).

Web Development

Mit dem JDeveloper 12*c* wird die Entwicklung von HTML5 und CSS 3 durch entsprechende Editoren unterstützt. Ebenfalls neu ist die Unterstützung von Java EE 6, also Standards wie EJB 3.1, Servlet 3.0, CDI, JPA 2.0, EL 2.2 (siehe Abbildung 3).

Neben der bestehenden Funktionalität für JAX-WS Services kommen in JDeveloper 12*c* folgende Funktionen für RESTful Web Services hinzu:

- Generierung von RESTful Services (1.1, 2.0) aus annotierten POJOs oder aus einer WADL-Datei (Web Application Description Language)
- Generierung eines JAX-RS Client Proxy für RESTful Web Services
- Generierung eines Web Service Data Controls für RESTful Web Services. Data Controls können vom UI-Entwickler per „Drag & Drop" verwendet werden, um auf den Service zuzugreifen.

Die Unterstützung für JAX-RS ist besonders für Applikationen wichtig, die für mobile Endgeräte entwickelt werden.

ADF Business Components

Mithilfe des Frameworks ADF Business Components (ADFbc) können Datenzugriffe auf eine relationale Datenbank und die Geschäftslogik des Business Service implementiert werden. Mit JDeveloper 12*c* sind einige interessante neue Funktionen dazugekommen:

- Für die Arbeit mit den Business Components muss keine Datenbank online im Zugriff sein; im Offline-Modus kommt ein lokales Abbild des Datenbank-Schemas in der Applikation zum Einsatz.
- Das Standard-Mapping von Oracle- auf Java-Datentypen wurde in „Java Extended for Oracle" geändert, um das Exponieren von ADFbc-Komponenten als Web Service zu erleichtern.
- Groovy kann in ADFbc benutzt werden, um Validierungen oder abgeleitete Attribute zu definieren. Mit dem aktuellen Release stehen dafür sowohl ein Editor als auch die Möglichkeit des Code-Debuggings zur Verfügung.
- ADF Business Components können als RESTful Web Services exponiert werden.

Standard EJB/JPA

Anstelle von ADFbc können Entwickler den Standard EJB/JPA für den Datenzugriff nutzen. Allerdings gab es im JDeveloper bisher keine weitgehende Unterstützung einer deklarativen Arbeitsweise. Mit JDeveloper 12*c* hat es hier eine Angleichung des Funktionsumfangs gegeben:

- Mit Sparse Bean Data Control lassen sich Modell-getriebene Wertelisten (LOV) definieren.
- Der Modus für den Datenzugriff über das Data Control ist durch Annotatio-

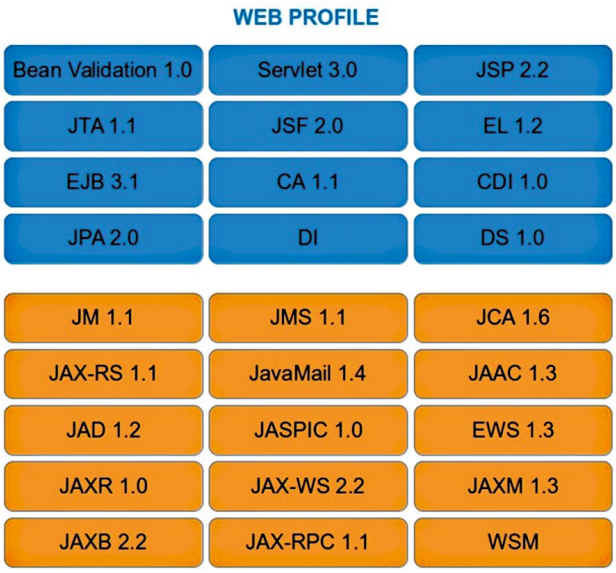


Abbildung 3: Java EE 6 mit Web Profile

```
import oracle.adf.model.adapter.bean.annotation.AttributeHint;
import oracle.adf.model.adapter.bean.annotation.ControlHintType;
import oracle.adf.model.adapter.bean.annotation.DateFormatter;
import oracle.adf.model.adapter.bean.annotation.FormatterType ;
import oracle.adf.model.adapter.bean.annotation.TimeZoneID

@AttributeHint  ( label = "Hire Date", tooltip = "Type date in the
form MM/dd/YYYY",
                  display = true, controlType = ControlHintType.
DEFAULT, width = 40, height = 20,
                  autoSubmit = true)

@DateFormatter ( type = FormatterType.SIMPLE_DATE, format = "MM/
dd/YYYY", formatter = "",
                timezoneId = TimeZoneID.DEFAULT)

public Date getHireDate() { return hireDate; }
```

Listing 1

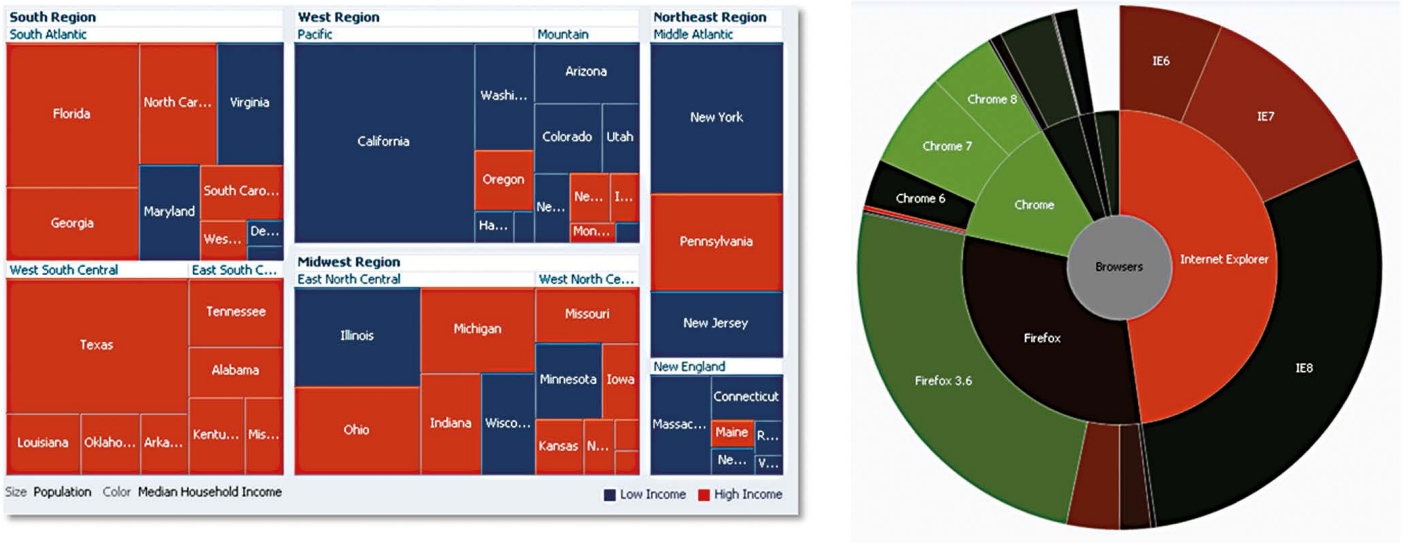


Abbildung 4: Treemap und Sunburst

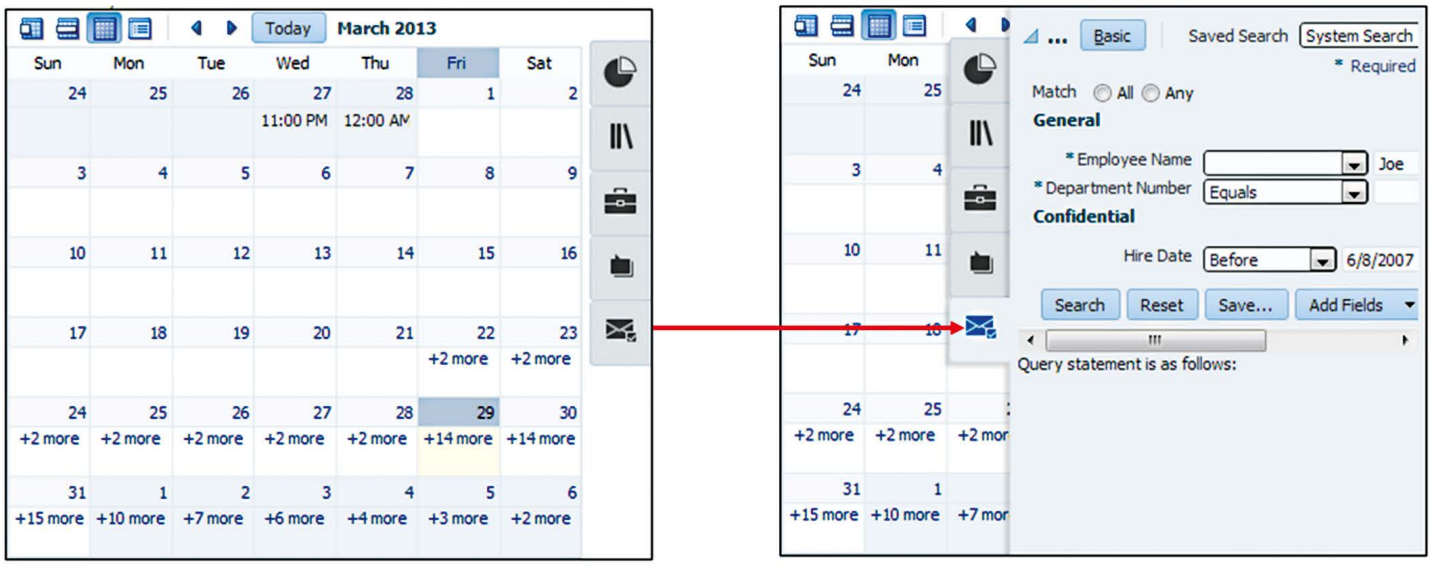


Abbildung 5: PanelDrawer

nen oder deklarativ einstellbar (Scrollable, Range Paging, NoPagination).
- Metadaten für die Darstellung im UI (Labels, Hilfe, Tooltips etc.) sind über Annotationen bereit (siehe Listing 1).

ADF Faces

Mit JDeveloper 12*c* wurde auf die Version 2 des JSF-Standards umgestellt, der als Neuerungen unter anderem Facelets und AJAX-Support mitbringt. Bei bestehenden ADF-Applikationen der Version 11g werden bestehende Pages von JSPX auf die neue JSF-Syntax automatisch migriert.

Mit ADF Faces steht dem UI-Entwickler eine große Zahl von Oberflächen-Komponenten zur Verfügung, die ein in unterschiedlichen Browsern getestetes „Look & Feel" einschließlich des Bedienkonzepts (Usability) mitbringen.

In jedem Release des JDeveloper kommen neue Oberflächen-Komponenten dazu. Dies ist auch in den Releases 12.1.2 beziehungsweise 12.1.3 der Fall. Bestehende Komponenten, die bisher in Flash implementiert waren, wurden in HTML5 umgeschrieben. Einige Beispiele für die neuen Komponenten im Release 12.1.3 sind:

- PanelGridLayout erlaubt die Definition einer Gitterstruktur im Layout und ähnelt damit den traditionellen Tabellen beim HTML Design.
- Treemap und Sunburst sind neue Chart-Typen, die die Bibliothek der DVT-Komponenten (Data Visualization Tools) erweitern (siehe Abbildung 4).
- PanelDrawer und PanelSpringboard erlauben eine komfortable Navigation und sind besonders für Applikationen auf Mobilgeräten geeignet (siehe Abbildung 5).
- Die Diagram-Komponente kann Daten in Form von Knoten und Beziehungen

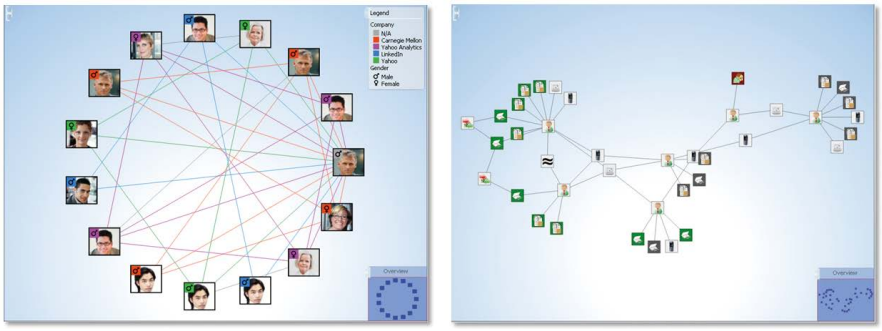


Abbildung 6: Diagram

darstellen und bietet großen Freiraum in Bezug auf das Layout (siehe Abbildung 6).

Für viele Komponenten ist die Bedienung durch Gesten hinzugekommen, um dem wachsenden Bedarf an mobilen Applikationen gerecht zu werden. Eine Live-Darstellung aller zur Verfügung stehenden Komponenten findet man unter „http://jdevadf.oracle.com".

Mit dem JDeveloper 12*c* sind neue Skins (Neuer Default Skin: „Skyros") hinzugekommen. Der grafische Skin.Editor hat einige Verbesserungen erfahren. Zudem wurde Maven 3 als Standard Build System integriert. Applikationen können bereits beim Anlegen in einem Maven Repository verwaltet oder zu einem späteren Zeitpunkt in ein solches überführt werden. Neben der Untersützung für SVN und anderen verbreiteten Systemen kommt mit dem JDeveloper 12*c* git/GitHub als weiteres populäres Versionskontrollsystem hinzu.

Weitergehende Informationen

- Oracle JDeveloper 12*c* (12.1.2.0.0.): www.oracle.com/technetwork/developer-tools/jdev/documentation/1212-nf-1964675.html
- Oracle JDeveloper 12*c* (12.1.3.0.0): www.oracle.com/technetwork/developer-tools/jdev/documentation/1213nf-2222743.html?msgid=3-10243408667

Dr. Jürgen Menge
juergen.menge@oracle.com

Dr. Jürgen Menge hat bei Oracle zunächst sechs Jahre in der Schulung als Trainer für die Entwicklungs-Werkzeuge Oracle Designer und Oracle Forms gearbeitet, um dann als technischer Berater in den Lizenzvertrieb zu wechseln. Seine fachlichen Schwerpunkte sind sowohl die klassischen Entwicklungs-Werkzeuge als auch die neuen, Standard-basierten Tools (JDeveloper/ADF, BI Publisher etc.).

http://ja.ijug.eu/14/4/13

Der (fast) perfekte Comparator

Heiner Kücker, Freiberufler

Eine Liste ganz einfach sortieren, entsprechende Methode aufrufen, Comparator übergeben. Dieser Artikel zeigt, dass auch ein einfacher Comparator Verbesserungsmöglichkeiten birgt.

Jeder Java-Programmierer hat sicher schon mal so einen Comparator geschrieben (siehe Listing 1).

Wenn nach einer anderen „Property (Member)" unserer Bean sortiert werden soll, müssen der gesamte Code kopiert und die abzufragenden Properties angepasst werden. Gibt es mehrere Beans, sind außerdem die beiden Parameter-Typen der „compare"-Methode zu ändern, also bereits vier Code-Stellen – eine langweilige und fehleranfällige Angelegenheit.

Getter ohne Reflektion

Basis dieses Alternativ-Vorschlags ist ein Property-Accessor, der ohne Reflektion auskommt, in diesem Fall nur der lesende Anteil, der „Getter" (siehe Listing 2).

Die beiden Typ-Parameter (Generics) „B-(Bean)" und „P(Property)" bestimmen die jeweils beteiligten Typen. Der Sinn der „get"-Methode sollte klar sein. Dieses Interface wird für alle benötigten Properties (Member) implementieren (siehe Listing 3).

Die Getter-Objekte sind statisch, existieren also nur einmal je JVM. Durch die Typ-Parameter sind Namensverwechslungen

und Typ-Fehler wie bei der Benutzung von Reflektion ausgeschlossen. Die Laufzeit ist sicher auch ok, wahrscheinlich besser als mit Reflektion. Das Getter-Objekt wird nun dem verbesserten wiederverwendbaren Comparator (Code der Java-Klasse mit Member und Initialisierung im Konstruktor hier weggelassen) übergeben: „new AccessorComparator(Customer.SUR_NAME_GETTER);". Jetzt ist nur noch ein Wert zu ändern. Die Getter-Objekte müssen auch vorhanden sein, man kann sie allerdings noch an anderen Codestellen verwenden. Listing 4 zeigt die „compare"-Methode des verbesserten Comparators.

Ein weiterer Wunsch ist die „null"-Sicherheit. „null"-Werte sind nicht vergleichbar, sie verstoßen gegen die totale Ordnung, in realen Applikationen tauchen sie jedoch oft auf. Hier wird davon ausgegangen, dass „null" kleiner als Nicht-„null" ist (siehe Listing 5).

Kaum ist dieses Problem gelöst, ist nach mehreren Properties in einer definierten Rangfolge zu sortieren. Man übergibt die Getter-Objekte als Array mit einem „vararg"-Parameter oder dynamisch als Liste (siehe Listing 6).

Diese Lösung ist effektiv, der Aufruf der „compareTo"-Methode und damit der Zugriff auf die Properties erfolgt nur, wenn unbedingt nötig. Dies könnte aufwändig sein und das Abfragen der Properties könnte im Hibernate-/JPA-Umfeld eine Datenbank-Abfrage auslösen. Nun kommt noch jemand daher und möchte nach bestimmten Properties absteigend sortieren. Ein Problem dabei ist, dass wir nicht wie mit „Collections.reverseOrder(Comparator)" die Sortier-Richtung für eine einzelne Property umkehren können. Aber das ist möglich (siehe Listing 7).

Die Methode „invert" ist eine statische Methode des „AccessorComparator", im obigen Code-Schnipsel per statischem Import ohne Klassen-Präfix notierbar. Sie liefert ein spezielles magisches Objekt mit dem originalen Getter als Member (siehe Listing 8, Generics aus Platzgründen weggelassen). Die Klasse „InvertGetter" ist „private" im „AccessorComparator" versteckt. Beim Sortieren wird per „instanceOf" geprüft, ob der magische Getter auftaucht (siehe Listing 9).

Das ist nicht besonders schön, bleibt aber unter dem API des „AccessorComparator" verborgen. Dynamisch kann man die Sortier-Properties festlegen, indem man sie in einer Liste sammelt und die Liste dann per „toArray"-Methode in ein Array für den „vararg"-Parameter umwandelt.

Unter [2] kann der gesamte Code mit einer Beispiel-Bean und Unit-Tests heruntergeladen werden. Für die Benutzung gibt es keine Einschränkungen, jeder kann mit dem Code alles tun, zum Beispiel in sein

```
Comparator<Customer> comparator =
        new Comparator<>() {
            @Override
            public int compare(
                    final Customer c1 ,
                    final Customer c2 ) {
                return c1.getSurName().compareTo(
                        c2.getSurName() );
            }
        };
```

Listing 1

```
interface Getter<B, P> {
    P get( B bean );
}
```

Listing 2

```
public static final Getter<Customer, String> NAME_GETTER =
        new Getter<Customer, String>() {
            @Override
            public String get( Customer bean ) {
                return bean.getName();
            }
        };
```

Listing 3

```
 @Override
  public int compare(
          final T o1 ,
          final T o2 ) {
      return this.getter.get( o1 ).compareTo(
              this.getter.get( o2 ) );
  }
```

Listing 4

```
    P p1 = getter.get( o1 );
    P p2 = getter.get( o2 );
    if ( p1 == null && p2 == null ) {
        return 0;
    } else if ( p1 == null ) {
        return -1;
    } else if ( p2 == null ) {
        return 1;
    } else
        ...
```

Listing 5

```
    for ( Getter g : getters ) {
        int compareResult =
            getter.get( o1 ).compareTo(
                getter.get( o2 );
        if ( compareResult != 0 ) {
            return compareResult;
        }
    }
    return 0;
```

Listing 6

```
  Comparator<Customer> comparator =
          new AccessorComparator<>(
                  invert(
                     Customer.SUR_NAME_GETTER ) );
```

Listing 7

```
  public static Getter invert(
          final Getter getterToInvert )
  {
      return new InvertGetter<>( getterToInvert );
  }
```

Listing 8

```
    if ( getter instanceof InvertGetter )
        // absteigende Sortierung
    {
        final InvertGetter<B, ?> invertGetter = (InvertGetter<B,
?>) getter;
        p1 = (P) invertGetter.getterToInvert.get( o1 );
        p2 = (P) invertGetter.getterToInvert.get( o2 );
    }
```

Listing 9

Projekt kopieren und das Package entsprechend anpassen. In einem weiteren Package ist ein Beispiel für einen PropertyComparator abgelegt, von dem es auch eine „NullIsLesserPropertyComparator"-Implementierung zum Behandeln von „null"-Werten gibt und der, wie beim klassischen Comparator üblich, durch die Methode „Collections.reverseOrder()" umgedreht werden kann. Schließlich ist eine andere Lösung immer nur ein paar Refactoring-Schritte entfernt.

Der fast perfekte Comparator

Der verbesserte Comparator vermeidet „Copy & Paste" und arbeitet effektiv bezüglich des Zugriffs auf die benötigten Properties. Die Anforderungen sind:

- Typsicher
- Effizient
- Multi-Property-/Field-fähig
- „null"-sicher
- Property-individuell aufsteigend/absteigend sortierbar

Die Lösung zum Umkehren der Sortier-Richtung mit „instanceOf" hat zwar den Geschmack der Unsauberkeit im Sinne der objektorientierten Programmierung, dem Autor ist jedoch keine bessere Lösung eingefallen. Wer eine bessere Idee hat, kann sich gern bei ihm melden.

Das Erstellen der Property-Accessoren ist erstmal ein höherer Aufwand, der sich allerdings bei entsprechend häufiger Benutzung auszahlt. Durch die Verwendung von Typ-Parametern („Generics") in der „Getter"-Klasse müssen primitive Properties in die jeweiligen Wrapper-Objekte verpackt werden. Bei der Arbeit mit „vararg"-Parametern wird jedes Mal ein Array erzeugt; Konstruktoren mit unterschiedlichen Parameter-Listen könnten effektiver sein. Mit den Java8-Lambdas ergeben sich eventuell neue Möglichkeiten, die der Autor hier nicht berücksichtigt hat.

Fazit

Dieser Comparator kann sicher kein Projekt retten, aber die zugrunde liegende Philosophie, eine ausdrucksstarke und der Problematik angemessene Lösung zu suchen, tut sicher jedem Projekt gut. Es ist oft zu beobachten, dass etliche Frameworks oder Tools alle möglichen Probleme lösen sollen. Das ist wie mit den Dampfmaschinen in der Anfangszeit der Industrialisierung: je größer, desto besser. Das einfache Programmierhandwerk wird nicht geschätzt und wenn, dann werden oft Prinzipien wie Clean-Code oder testgetriebene Entwicklung pauschal bis zur Umkehrung ihres Sinns betrieben.

Links

[1] http://www.nosid.org/java-compound-comparator.html
[2] http://www.heinerkuecker.de/Comparator.html

Heiner Kücker
mail@heinerkuecker.de

Heiner Kücker ist seit dem Jahr 2000 freiberuflicher Java-Entwickler und freut sich, wenn das Java-Typ-System ihn vor seinen Flüchtigkeitsfehlern (macht er ständig) schützt. Zur Rationalisierung seiner Arbeit schreibt er sich selbst kleine Main-Methoden-Tools. Außerdem interessiert er sich für fortschrittliche Programmiertechniken wie funktionale Programmierung.

http://ja.ijug.eu/14/4/14

Clientseitige Anwendungsintegration: Eine Frage der Herkunft

Sascha Zak, adesso AG

Eine clientseitige Integration von Web-Anwendungen verspricht, heterogene Systemlandschaften in Unternehmen kostengünstig und effizient miteinander zu verbinden, ohne dabei die bestehenden Systeme maßgeblich zu verändern. Clientseitige Skriptsprachen, allen voran JavaScript, sind ein mächtiges Instrument, um dies zu bewerkstelligen. In der Praxis stößt man dabei jedoch schnell auf ein scheinbar schwerwiegendes Hindernis: Die „Same Origin Policy" ist als Sicherheitskonzept des Browsers eine Art „digitaler Türsteher" für Clientskripte. Der Artikel zeigt, wie sich diese auf clientseitige Integration auswirkt und welche Mittel und Wege es gibt, diese Hürde zu nehmen.

Der Blick auf die Uhr verrät: Kurz vor elf, Zeit für einen Kaffee, den dritten an diesem Vormittag. Seit Stunden schon das gleiche Bild auf dem Monitor, genau wie gestern. Das interne Bestellsystem mit Listen von zahlungskräftigen Kunden auf der einen Seite, das zugekaufte CRM auf der anderen. Davor die Tastatur, auf der drei abgegriffene Tasten die Beschriftungen „STRG", „C" und „V" gerade noch erahnen lassen. Die Aufgabe: Datensätze von links nach rechts kopieren.

Solche Szenarien, hier zugegebenermaßen ein wenig überspitzt dargestellt und zuweilen spöttisch als „Drehstuhlschnittstelle" bezeichnet, gehören durchaus nicht in den Bereich der Fiktion. Im Unternehmen des Autors gibt es reale Projekte, in denen es sich genau um solche Problemstellungen dreht. Nicht nur, dass diese Tätigkeiten unterfordernd und demotivierend für die Mitarbeiter sind, sie sind ebenso fehlerträchtig, insbesondere über einen längeren Zeitraum. Die Integration solcher Anwendungen könnte Abhilfe schaffen, doch leider gibt es oftmals viele Argumente dagegen: hoher Entwicklungsaufwand, inkompatible Mischkulturen aus Eigenentwicklung und zugekaufter Software, Sicherheitsbedenken, Betriebs- und Supportfragen und nicht zuletzt der Wunsch, die endlich stabil laufenden Systeme aus Angst vor Seiteneffekten nicht anzutasten.

Auch in anderen Bereichen ist die Integration von Anwendungen und Inhalten gefragt. In der Werbung beispielsweise geht es um die Integration der Werbeinhalte in gut frequentierte Websites. Ein anderes Beispiel sind sogenannte „Mashups", die verschiedenste Dienste und Inhalte kombinieren, um ihren Mehrwert zu generieren. Bei diesen Anwendungen ist eine serverseitige Integration von vornherein ausgeschlossen, da sich die Fremdinhalte gar nicht im direkten Zugriff befinden.

Hier kommt die clientseitige Integration ins Spiel. JavaScript bietet über das Document Object Model (DOM) Zugriff auf den gesamten Inhalt einer Anwendung, inklusive mannigfaltiger Manipulationsmöglichkeiten, einschließlich dem Senden und Empfangen von Daten, etwa mittels asyn-

Abbildung 1: Beispiele für Herkünfte nach SOP

chronem „XMLHttpRequest“ (Ajax). Das macht clientseitige Skriptsprachen zu einem mächtigen Werkzeug, um Integration zu schaffen, ohne dabei die eigentlichen Anwendungen maßgeblich zu verändern. Für das skizzierte Szenario könnte eine Integrationsmöglichkeit darin bestehen, die beiden Anwendungen in jeweils einem eigenen (i)Frame innerhalb eines dritten Dokuments einzubinden. JavaScript-Funktionen könnten die zu kopierenden Daten aus dem DOM der Quell-Anwendung lesen und in die entsprechenden Input-Felder des Zieldokuments übertragen. Auf dem Weg dorthin könnten die Daten darüber hinaus beliebig prozessiert werden, etwa im Rahmen einer Validierung, Plausibilitätsprüfung oder Konvertierung – und das auf Knopfdruck.

Digitaler Türsteher

Dass mit JavaScript neben solch einer Integration, mehr Komfort und einzigartiger User-Experience auch eine Menge Schaden angerichtet werden kann, sollte keiner ausführlichen Erläuterung bedürfen. Ebenso wenig, dass es ein gewisses Vertrauensgefälle vom Anwender zu verschiedenen Anwendungen im Netz gibt. Um dem Rechnung zu tragen und potenziell schädlichen Skripten Einhalt zu gebieten, hat Netscape im Jahr 1996 in seinen Netscape Navigator ein Sicherheitskonzept integriert, dass seither Bestandteil aller gängigen Browser ist. Gemeint ist die sogenannte „Same Origin Policy“ (SOP) [1]. Ihre Aufgabe besteht darin, als eine Art „digitaler Türsteher“ Zugriffe von clientseitigen Skripten (dazu gehören neben JavaScript auch Flash beziehungsweise ActionScript) zu unterbinden, so lange sie nicht die gleiche Herkunft wie das Dokument haben, das sie beinhaltet. Auf diese Weise soll verhindert werden, dass fremde Web-Anwendungen in (i)Frames Daten des Elterndokuments lesen oder sie manipulieren können und umgekehrt. Gleiche Herkunft bedeutet im Sinne der SOP die Gleichheit von „Protokoll“, „Hostname“ und „Port“, wobei „Hostname“ buchstäblich zu verstehen ist. Es ist völlig egal, zu welcher IP-Adresse dieser ausgelöst wird. Abbildung 1 zeigt einige Beispiele dazu.

Es steht außer Frage, dass die SOP eine sinnvolle und notwendige Maßnahme ist. Aber wie so oft zeigt sich auch hier, dass Sicherheit und Usability zumeist komplementäre Eigenschaften sind. Für die clientseitige Integration von Anwendungen stellt die SOP ein ernst zu nehmendes Hindernis dar. Es ist leicht vorstellbar, dass Anwendungen, selbst wenn sie innerhalb des gleichen Unternehmens betrieben werden, unter verschiedenen (Sub-)Domains gehostet sind oder unterschiedliche Protokolle (wie HTTP/HTTPS) oder Ports verwenden. Im Kontext von Mashups kann nahezu ausgeschlossen werden, dass die integrierten Inhalte dieselbe Herkunft besitzen.

Allheilmittel: Proxys

Proxys sind ein einfacher wie sehr effektiver Weg, die SOP zu umgehen. Ein Proxy tritt hierbei als alleiniger Kommunikationspartner des Clients auf, nimmt alle Anfragen entgegen und leitet diese selbst an die verschiedenen zu integrierenden Anwendungen weiter. Für den Client gibt es dabei nur noch eine Herkunft – die des Proxy. Ein gängiges Beispiel hierfür sind mehrere Tomcats, die durch einen Apache-Webserver als Proxy integriert werden und damit allesamt unter Port 80 und verschiedenen Kontextpfaden erreichbar sind. Der große Vorteil dabei ist, dass sowohl verschiedene Ports als auch verschiedene Hostnamen vereinheitlicht werden können, ohne dass die zu integrierenden Anwendungen etwas davon wissen müssen. Problematisch wird es allerdings, wenn verschiedene Protokolle im Einsatz sind. Auch dürfen die zu integrierenden Anwendungen keine absoluten Pfade oder hart kodierte URLs verwenden. Darüber hinaus will ein Proxy natürlich betrieben und gewartet werden, belegt Ressourcen und führt im schlechtesten Fall zu einer Verdopplung des Netzwerk-Traffics.

Tricky: URL-Fragmentation

Ein anderer ausgeklügelter Ansatz nennt sich „URL-Fragmentation“. Er dient vor allem dazu, einen Datenaustausch zwischen (i)Frames innerhalb eines Drittdokuments zu ermöglichen, die durch die SOP eigentlich strikt voneinander getrennt sind. Ein URL-Fragment bezeichnet den Teil einer URL, der nach dem Hashmark folgt. Dieser wird üblicherweise dazu verwendet, Ankerpunkte innerhalb eines Dokuments zu adressieren.

Der Ansatz der URL-Fragmentation macht sich dabei die Tatsache zunutze, dass Änderungen eines URL-Fragments nicht zu einem neuen Request und damit dem Reload des Dokuments führen. In einem solchen Fragment lassen sich Daten verpacken und einem (i)Frame über das Attribut „frame.location“ beziehungsweise „iframe.src“ zur Verfügung stellen. In einem einfachen Beispiel könnte auf diese Weise ein Elterndokument beispielsweise die aktuelle Fenstergröße in der Form „http://host/document#width=800&height=600“ propagieren. Die Anwendung im (i)Frame kann daraufhin ihre eigene URL auslesen, die Daten auswerten und zum Beispiel ein Werbebanner oder Pop-up-Fenster in der passenden Größe anzeigen. Die einzige Schwierigkeit besteht darin, über eine Änderung der URL informiert zu werden. Hierbei gibt es im Wesentlichen zwei Möglichkeiten: Entweder das (i)Frame-Dokument liest seine eigene URL in regelmäßigen Abständen aus (polling) oder aber das Elterndokument löst beispielsweise ein Resize-Event aus, auf das das (i)Frame-Dokument über einen entsprechenden Handler reagiert.

Diese Form der Kommunikation funktioniert ebenso gut in die andere Richtung und hat den großen Vorteil, dass sie sicher ist, da hier nicht direkt manipuliert wird. Vielmehr werden Daten bewusst bereitgestellt und können bei Bedarf von anderen Anwendungen abgerufen und genutzt werden. Datenmenge und -format unterliegen hierbei allerdings den Beschränkungen, die URLs mit sich bringen, sowohl hinsichtlich der erlaubten Zeichen als auch bezüglich der maximalen Länge, die je nach Browser variieren kann.

JSON, aber bitte mit Padding (JSONP)

Unter diesem Namen verbirgt sich eine Methode, Daten über Ajax (XMLHttpRequest in JavaScript) auszutauschen, die durch die SOP direkt nicht zugreifbar sind, weil sie zum Beispiel von einem fremden Host bereitgestellt werden. Sie macht sich die Tatsache zunutze, dass das Einbinden von Bildern („<img>“), Medien („<audio>“/„<video>“), Plug-ins („<applet>“/„<object>“/„<embed>“), CSS („<link>“) sowie Skripten („<script>“) von fremden Quellen trotz SOP erlaubt ist. Ein JavaScript fremder Herkunft, das mittels

```
// Callback bereitstellen
function doIt(data) {
  // hier Daten verarbeiten
}

function getDataWithJSONP() {

  // Script-Element erzeugen
  var url = "http://adesso.de/service/jsonp?callback=doIt";
  var script = document.createElement("script");
  script.setAttribute("src", url);
  script.setAttribute("type", "text/javascript");

  // Script-Element im Dokument einfügen
  document.getElementByTagName("head")[0]
      .appendChild(script);
}
```

Listing 1: JavaScript für JSONP (Client)

```
protected void doGet(HttpServletRequest req,
  HttpServletResponse res) throws ServletException,
 IOException {

  String callback = req.getParameter("callback");
  ServletOutputStream out = res.getOutputStream();

  // Standardausgabe ohne padding
  if (callback == null) {
      out.write(this.data.getBytes()); // z. B. JSON
  }
  // Ausgabe mit Padding
  else {
      String jsonp = callback + "(\" + this.data + "\");";
      out.write(jsonp.getBytes())    ;
  }
  out.flush();
}
```

Listing 2: Servlet für JSONP (Server)

„<script>" in ein Dokument eingebunden wird, wird aus Sicht der SOP mit den gleichen Rechten ausgeführt, als hätte es die Herkunft des Dokuments selbst. Dies ermöglicht folgenden Trick, mit dem Daten via „XMLHttpRequest" auch über Herkunftsgrenzen hinweg ausgetauscht werden können: Ein Client-Skript stellt einen Request nicht direkt über „XMLHttpRequest", sondern erzeugt im DOM des Elterndokuments dynamisch ein „<script>"-Element, dessen „src"-Attribut der Request-URL entspricht. Aufrufparameter werden dabei als Query-Parameter der URL hinzugefügt. Zuletzt wird ein weiterer Parameter übergeben, der den Namen einer JavaScript-Callback-Funktion angibt. Diese Callback-Funktion muss im Dokument existieren und dient als Response-Handler (siehe Listing 1).

Der fremde Host nimmt den Request entgegen, verpackt (daher das „padding") seine Daten in einem JavaScript-Funktionsaufruf mit dem Namen der übergebenen Callback-Funktion und sendet seine Response. Der Client erhält damit ein gültiges JavaScript, das er trotz SOP laden und ausführen darf und das die angegebene Callback-Funktion mit den Daten des Servers aufruft (siehe Listing 2).

Obwohl der Name „JSONP" eine Übertragung der Daten im JSON-Format nahelegt, ist diese Einschränkung nicht zwingend erforderlich. Einfache Strings, XML oder beliebige andere Formate sind ebenso anwendbar. Der Vorteil von JSON als Datenformat beruht auf der Handhabung, die einfacher ist als beispielsweise ein aufwändigeres Parsen von XML. Der Nachteil dieser Methode ist die Beschränkung auf GET-Requests. Auch gelten hier wiederum die Beschränkungen einer URL in Bezug auf Länge und erlaubte Zeichen. Darüber hinaus muss der Service den übergebenen Callback-Parameter auswerten und das Padding erzeugen können. Für REST-basierte Services ist dies allerdings in der Regel mit minimalem Aufwand realisierbar. Für Designer von REST-Services könnte es eine Überlegung wert sein, ein Padding von vornherein zu implementieren und damit out of the box integrationsfähig zu sein. Anwendung findet diese Technik etwa in rein clientseitigen Monitoring-Frontends wie Bigdesk [2] für ElasticSearch [3].

Obwohl die bisher dargestellten Möglichkeiten zuverlässig und browserunabhängig funktionieren, haftet ihnen der Eindruck von Trickserei und Ausnutzung von Schwachstellen an. Die nachfolgenden Techniken beschäftigen sich daher mit offizielleren und eher standardisierten Wegen, um mit den Problemen durch die SOP umzugehen.

Cross Origin Resource Sharing

Das Cross Origin Resource Sharing (CORS) [4] ist eine W3C-Empfehlung zur kontrollierten Freigabe von Zugriffen fremder Herkunft. Sie besteht im Wesentlichen aus der Definition einer Reihe von Request-, vor allem aber Response-Headern, über die der Server dem Client (Browser) mitteilt, welche Cross-Origin-Zugriffe dieser zulassen darf. Sie eignet sich vor allem für Zugriffe mittels Ajax, ohne dabei auf eine bestimmte HTTP-Methode beschränkt zu sein.

Bei ausgehenden Requests auf eine Cross-Origin-Ressource fügt der Browser automatisch einen „Origin"-Request-Header ein. Wenn der Webserver diesen Zugriff gestatten möchte, fügt er der Response seinerseits einen passenden Response-Header, etwa „Access-Control-Allow-Origin", hinzu (siehe Listing 3).

Findet der Browser diesen Response-Header und lässt sich der Wert auf die eigene Herkunft abbilden, darf die Response trotz SOP passieren. Im dargestellten Beispiel entspricht der Wert exakt der Her-

```
Nach SOP verbotener Request an “http://sub.adesso.de/whatever“:
GET /whatever HTTP/1.1
Host: sub.adesso.de
…
Origin: http://adesso.de

Antwort mit CORS-Legitimation von “sub.adesso.de”:
HTTP/1.1 200 OK
Date: Mon, 19 May 2014 09:23:53 GMT
…
Access-Control-Allow-Origin: http://adesso.de
```

Listing 3: Cross-Origin-Request und Response

```
<!-- Filter-Deklaration -->
<filter>
  <filter-name>CORSFilter</filter-name>
  <filter-class>
      org.apache.catalina.filters.CorsFilter
  </filter-class>
  <init-param>
      <param-name>cors.allowed.origins</param-name>
      <param-value>http://adesso.de</param-value>
  </init-param>
</filter>

<!—Mapping auf URLs -->
<filter-mapping>
  <filter-name>CORSFilter</filter-name>
  <url-pattern>/whatever/*</url-pattern>
</filter-pattern>
```

Listing 4: CORS-Filter in web.xml (Tomcat)

kunft des Aufrufers. Hier wäre allerdings auch durch den Einsatz von Wildcards (*) eine allgemeinere Angabe möglich. Ebenfalls können Header wiederholt in die Response eingefügt und damit Listen von erlaubten Origins angegeben werden. Dies in Kombination mit den anderen Headern (etwa „Access-Control-Allow-Methods“), die CORS definiert, ermöglicht eine feingranulare Steuerung der Zugriffsberechtigungen.

Dieses Beispiel zeigt besonders deutlich, dass es sich bei der SOP um ein clientseitiges Sicherheitskonzept handelt. Dies bedeutet vor allem, dass Requests in jedem Fall zugestellt werden und der Browser mittels SOP lediglich entscheidet, ob er die Response zum Aufrufer durchlässt.

Die Nutzung von CORS ist ohne serverseitige Anpassungen nicht möglich und steht dem eingangs genannten Ziel, die Anwendungen auf dem Webserver möglichst unangetastet zu belassen, grundsätzlich entgegen. In Application-Servern und Servlet-Engines ist die Nutzung von CORS jedoch lediglich eine simple Ergänzung in der Konfiguration. Mit einem Tomcat wird seit Version 7.0.41 ein CORS-Filter (siehe „org.apache.catalina.filters.CorsFilter“) ausgeliefert, der im Deployment-Descriptor aktiviert werden kann und eine Abbildung von CORS-Headern auf URL-Patterns ermöglicht (siehe Listing 4). Damit bleibt der serverseitige Aufwand minimal und erfordert keine Änderung an den Anwendungen selbst. Allerdings wird dieser Standard nicht von allen (insbesondere älteren) Browsern unterstützt.

HTML5 und Cross-Document-Messaging

Mit HTML5 steht eine Reihe neuer clientseitiger Features zur Verfügung. Darunter ist auch ein Mechanismus, um Nachrichten über Dokumentgrenzen hinweg, einschließlich Dokumente fremder Herkunft, austauschen zu können. Hierzu stellt das Window-Objekt die Funktion „postMessage(messageText, domain)“ für das Senden von Nachrichten bereit. Über die Funktion „addEventListener(„message“, callback)“ können Callback-Funktionen für den Empfang dieser Nachrichten registriert werden. Das Besondere dabei ist die Angabe der Ziel-Domain, für die Nachrichten versendet werden. Darüber kann der Sender steuern, dass nur Dokumente mit der angegebenen Herkunft die Nachrichten vom Browser überhaupt zugestellt bekommen.

Der Empfänger der Nachricht findet hingegen im Event-Objekt das Attribut „event.origin“ und kann seinerseits prüfen, ob er der Herkunft der Nachricht vertraut und diese verarbeiten möchte. Insgesamt stellt dieser Weg eine sauber definierte und durch die Entscheidungsfreiheit im Eventhandler sichere Möglichkeit dar, trotz SOP eine Kommunikation über Herkunftsgrenzen hinweg zu etablieren. Auch wenn hierbei die zu integrierenden Anwendungen ihren Anteil an der Kommunikation implementieren müssen, bleibt dieser jedoch rein clientseitig, sodass keine Änderungen der Web-Anwendung selbst notwendig werden. Auch hier gilt es allerdings zu prüfen, ob die Ziel-Browser dieses Feature bereits unterstützen.

Das Rad nicht neu erfinden

Die vorgestellten Mechanismen geben dem Entwickler einen recht umfangreichen Maßnahmenkatalog an die Hand, um eine clientseitige Integration von Web-Anwendungen zu realisieren. Dennoch muss nicht alles selbst entwickelt werden. Es gibt einige Frameworks, die sich eines oder mehrerer dieser Mechanismen bedienen und ein komfortables API bereitstellen, dass nur noch benutzt werden muss. jQuery [5] beispielsweise bietet out of the box die Möglichkeit, „ajax()“-Aufrufe mit JSONP umzusetzen. Es kümmert sich intern um die dynamische Erzeugung der notwendigen „<script>“-Elemente und Callbacks. Daneben gibt es Tools wie EasyXDM [6] oder Porthole [7], die die Möglichkeiten des jeweiligen Browsers erkennen und entsprechend mit den jeweils geeigneten Ansätzen wie dem „postMessage()“ aus HTML5 oder der URL-Fragmentation in versteckten i-Frames arbeiten.

Fazit

Clientseitige Integration ist ein vielversprechender Ansatz, um Web-Anwendungen zu integrieren, wo dies serverseitig nicht sinnvoll realisierbar ist. Die „Same Origin Policy" ist ein sinnvoller und notwendiger Mechanismus, um sich gegen schädliche Skripte und unbefugten Zugriff auf die eigenen Daten zu schützen. Sie steht jedoch einer clientseitigen Integration grundsätzlich als größtes Hindernis im Weg.

Die aufgezeigten Mechanismen schaffen ein vielfältiges Repertoire an Möglichkeiten, mit diesem Hindernis umzugehen. Was bleibt, ist die Entscheidung, das geeignete Mittel mit Blick auf Technologie und Sicherheit für den konkreten Anwendungsfall auszuwählen und einzusetzen. Darüber hinaus kann das Wissen um diese Mechanismen helfen, bei der Konzeption von Serveranwendungen bestimmte Mechanismen (wie JSONP oder CORS) zu berücksichtigen und damit von vornherein die notwendigen Voraussetzungen für eine clientseitige Integration zu schaffen.

Links

[1] http://de.wikipedia.org/wiki/Same-Origin-Policy
[2] http://bigdesk.org
[3] http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/common-options.html#_jsonp
[4] http://www.w3.org/TR/cors
[5] http://jquery.com
[6] http://easyxdm.net/wp
[7] http://ternarylabs.github.io/porthole

Sascha Zak
sascha.zak@adesso.de

Sascha Zak ist Senior Software Engineer bei der adesso AG im Competence Center Telematik in Berlin. Schwerpunktmäßig beschäftigt er sich mit Architektur und Entwicklung von webbasierten Systemen sowie Anwendungen im Umfeld von Smartcards und der elektronischen Gesundheitskarte.

http://ja.ijug.eu/14/4/15

Unbekannte Kostbarkeiten des SDK Heute: Die Klasse „Objects"

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir stellen in dieser Reihe derartige Features des SDK vor: die unbekannten Kostbarkeiten.

Die Klasse „java.lang.Object" ist hinreichend bekannt, da sie die Oberklasse aller Java-Klassen ist. Die mit Java SE 7 eingeführte „java.util.Objects" ist hingegen vielen Entwicklern unbekannt, obwohl sie durchaus sinnvoll verwendet werden kann.

Die Klasse „Objects"

Die Klasse „java.util.Objects" ist im API-Doc [1] beschrieben als: „This class consists of static utility methods for operating on objects. These utilities include null-safe or null-tolerant methods for computing the hash code of an object, returning a string for an object, and comparing two objects."

Die hinter der Klasse steckende Motivation ist also, den Umgang mit „Null"-Referenzen zu vereinfachen. Dies hat ganz offensichtlich auch die in Java SE 8 eingeführte Klasse „java.util.Optional" zum Ziel. Wir wollen in unserer Reihe der unbekannten Kostbarkeiten allerdings keine neuen, kürzlich eingeführten Features vorstellen, sondern solche, die schon einige Zeit existieren, aber nicht genutzt werden, einer breiten Öffentlichkeit bekannt machen.

Die Klasse „Objects" kann nicht instanziiert werden und enthält in der Version Java SE 7 neun statische Methoden, die vor allem dem Vergleich von Objekten und der Berechnung des Hash-Codes beziehungsweise der String-Repräsentation eines Objekts dienen. Wir wollen uns hier nur exemplarisch die „equals()"-Methode anschauen, bitten jedoch den Leser, einen Blick auf die komplette Klasse zu werfen – es lohnt sich.

```
public final class Pair<L, R> {

  private final L left;
  private final R right;

  public Pair(L left, R right) {
    this.left = left;
    this.right = right;
  }
  ...
}
```

Listing 1

```
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  Pair<?, ?> other = (Pair<?, ?>) obj;
  if (left == null) {
    if (other.left != null)
      return false;
  } else if (!left.equals(other.left))
    return false;
  if (right == null) {
    if (other.right != null)
      return false;
  } else if (!right.equals(other.right))
    return false;
  return true;
}
```

Listing 2

```
public boolean equals(Object obj) {
  if (Objects.equals(getClass(), obj.getClass())) {
    Pair<?, ?> other = (Pair<?, ?>) obj;
    return Objects.equals(left, other.left)
        && Objects.equals(right, other.right);
  } else {
    return false;
  }
}
```

Listing 3

Bevor wir auf die Code-Ebene eines Beispiels hinabsteigen, ist hier zunächst noch eine ganz allgemeine Bemerkung zu „Null"-Referenzen angebracht. Charles Antony Richard Hoare, meist als „C.A.R. Hoare" oder „Tony Hoare" abgekürzt, ist ein Informatiker, den jeder Informatikstudent zumindest dem Namen nach als Erfinder des Quicksort-Algorithmus kennt. Er hat im Jahr 1965 die „Null"-Referenz in die Sprache „ALGOL W" eingeführt. Er gilt damit als deren geistiger Vater. Unter [2] findet man ein Video, in dem Tony Hoare seine Einführung der „Null"-Referenz als „Billion Dollar Mistake" beschreibt.

Verwendung von „Objects.equals()"

Wir demonstrieren an einem Beispiel, wie die Verwendung der „equals()"-Methode von „java.util.Objects" Code deutlich vereinfachen kann. Dazu dient die generische Klasse „Pair<L, R>", deren Instanzen gleich sein sollen, wenn der linke und rechte Teil des Paares jeweils gleich sind. Listing 1 zeigt, wie eine einfache Implementierung aussehen könnte. Die von Eclipse generierte „equals()"-Methode fällt recht umfangreich aus (siehe Listing 2). Die Überarbeitung unter Verwendung von „Objects.equals()" reduziert den Code erheblich (siehe Listing 3).

Wir haben dabei alle Fallunterscheidungen des ursprünglichen Codes übernommen, entweder explizit oder implizit über die „Objects.equals()"-Implementierung. Einzige Ausnahme ist das erste „if" des ursprünglichen Codes, auf das wir verzichtet haben, da es durch die darauffolgenden „equals()" impliziert wird. Der Leser wird sicher zustimmen, dass diese Variante deutlich kürzer und wesentlich besser zu verstehen ist.

Fazit

Die Klasse „java.util.Objects", die seit Version 7 in Java SE enthalten ist, enthält Utility-Methoden, die bei der Verwendung von Objektreferenzen diese implizit auf „Null"-Referenzen überprüfen. Dies erspart explizite Tests darauf, sodass der resultierende Code unter Verwendung der Objects-Methoden zum Teil erheblich schlanker ist als ohne deren Verwendung. Neben der verwendeten „equals()"-Methode enthält die Klasse „Objects" weitere „Null"-tolerante Methoden, wie etwa „compare()", „deepEquals()", „hash()" und „hashCode()", die man sich unbedingt anschauen sollte.

Literatur/Videos

[1] http://docs.oracle.com/javase/7/docs/api/java/util/Objects.html

[2] http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Bernd Müller
bernd.mueller@ostfalia.de

Bernd Müller ist Professor für Software-Technik an der Ostfalia. Er ist Autor des Buches „JavaServer Faces 2.0" und Mitglied in der Expertengruppe des JSR 344 (JSF 2.2).

http://ja.ijug.eu/14/4/16

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
http://www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e. V.
http://www.doag.org

Java User Group Stuttgart e.V. (JUGS)
http://www.jugs.de

Java User Group Köln
http://www.jugcologne.eu

Java User Group Darmstadt
http://jugda.wordpress.com

Java User Group München (JUGM)
http://www.jugm.de

Java User Group Metropolregion Nürnberg
http://www.source-knights.com

Java User Group Ostfalen
http://www.jug-ostfalen.de

Java User Group Saxony
http://www.jugsaxony.org

Sun User Group Deutschland e.V.
http://www.sugd.de

Swiss Oracle User Group (SOUG)
http://www.soug.ch

Berlin Expert Days e.V.
http://www.bed-con.org

Java Student User Group Wien
www.jsug.at

Java User Group Karlsruhe
http://jug-karlsruhe.mixxt.de

Java User Group Hannover
http://www.jug-h.de

Java User Group Augsburg
http://www.jug-augsburg.de

Java User Group Bremen
http://www.jugbremen.de

Java User Group Münster
http://www.jug-muenster.de

Java User Group Hessen
http://www.jugh.de

Java User Group Dortmund
http://www.jugdo.de

Java User Group Hamburg
http://www.jughh.de

Java User Group Berlin-Brandenburg
http://www.jug-berlin-brandenburg.de

Der iJUG möchte alle Java-Usergroups unter einem Dach vereinen. So können sich alle Java-Usergroups in Deutschland, Österreich und der Schweiz, die sich für den Verbund interessieren und ihm beitreten möchten, gerne unter office@ijug.eu melden.

iJUG Verbund

www.ijug.eu

Impressum

Herausgeber:
Interessenverbund der Java User Groups e.V. (iJUG)
Tempelhofer Weg 64, 12347 Berlin
Tel.: 030 6090 218-15
www.ijug.eu

Verlag:
DOAG Dienstleistungen GmbH
Fried Saacke, Geschäftsführer
info@doag-dienstleistungen.de

Chefredakteur (VisdP):
Wolfgang Taschner, redaktion@ijug.eu

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt;
Daniel van Ross, NeptuneLabs;
Dr. Jens Trapp, Google;
André Sept, InterFace AG

Titel, Gestaltung und Satz:
Fana-Lamielle Samatin,
DOAG Dienstleistungen GmbH
Foto S. 20 © Volodimir Kalina / 123rf.com

Anzeigen:
Simone Fischer
anzeigen@doag.org

Mediadaten und Preise:
http://www.doag.org/go/mediadaten

Druck:
adame Advertising and Media GmbH
www.adame.de

Inserentenverzeichnis

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG *News* und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN
0700 11 36 24 39

ODER BESTELLEN SIE ONLINE
go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr

☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:
Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.


2014
DOAG
Konferenz + Ausstellung
18. - 20. November | Nürnberg
Experience
Passion
Weil sich Perfomance-Tuning wie
das Erklimmen von Gipfeln anfühlt.
Eventpartner:
AOUG
AUSTRIAN ORACLE USER GROUP
SOUG
Swiss Oracle User Group
2014.doag.org