01-2015 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Fünf Jahre Java aktuell

Praxis
Mehr Typsicherheit mit Java 8

Programmieren
Der selbe GUI-Code für Desktop, Web und native mobile App

SAP HANA
Was für Java-Anwendungen drin ist

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

4 191978 304903 01

iJUG Verbund

Frühbucherrabatt
bis 30. Januar 2015.

Programm online

JavaLand 2015

24.-25.03.2015

WWW.JAVALAND.EU

JATUMBA!

www.JavaLand.eu

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

Community Partner: iJUG Verbund

Wolfgang Taschner
Chefredakteur Java aktuell

http://ja.ijug.eu/15/1/1

So schnell vergehen fünf Jahre

Ich erinnere mich noch genau: Am 20. April 2009 kündigt Oracle die Übernahme von Sun Microsystems an. Noch am selben Tag schreibe ich eine Online-Meldung auf www.doag.org: „Die DOAG begrüßt die Anwender aller Sun-Technologien". Genau vier Monate später genehmigt das US-Justizministerium den Firmenkauf. Die EU-Kommission teilt nach einer ersten Prüfung am 10. November 2009 mit, dass sie wettbewerbsrechtliche Probleme auf dem Markt für Datenbanken sehe und äußert deshalb Bedenken gegen die Übernahme von Sun Microsystems durch Oracle. Im Januar 2010 genehmigt sie jedoch die Übernahme ohne Auflagen.

Ende des Jahres 2009 schließen sich sechs Java-Usergroups und die DOAG Deutsche ORACLE Anwendergruppe e. V. zum Interessenverbund der Java User Groups e.V. (iJUG) zusammen. Ziel ist die umfassende Vertretung der gemeinsamen Interessen der Java-Usergroups sowie der Java-Anwender im deutschsprachigen Raum, insbesondere gegenüber Entwicklern, Herstellern, Vertriebsunternehmen sowie der Öffentlichkeit. Neben der Herausgabe von Pressemeldungen und eines monatlichen Newsletters erscheint Ende des Jahres 2010 die erste Ausgabe der Java aktuell.

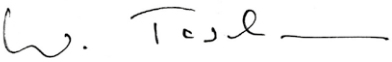
Es folgt eine schwierige Zeit. Die Zukunft von Java ist ungewiss, die Gerüchteküche brodelt. Viele Java-Anwender sind verunsichert, Oracle hält sich mit klaren Aussagen weitgehend zurück. Erste Java-Entwickler sehen sich bereits nach Alternativen um. Auch die Community ist aktiv, auf Blogs werden Möglichkeiten diskutiert, etwa Harmony und einen Fork des OpenJDK zu einer eigenständigen Sprache zusammenzulegen – unabhängig von Oracle. Oracle verhält sich anfangs sehr undiplomatisch und degradiert die JavaOne, einstige Pilgerstätte der Java-Community, zum Anhängsel der OpenWorld. Java steht auf der Kippe.

Zum Glück wendet sich alles zum Guten. Oracle geht mit der Vorstellung von Java 7 auf die Community zu und kommuniziert auch eine Roadmap für die weiteren Schritte. Es entsteht der Eindruck, dass Oracle es ernst meint mit Java, auch wenn in erster Linie kommerzielle Interessen dahinterstehen. Der Java Community Process kommt ebenfalls langsam wieder in Schwung.

Der Erfolg stellt sich ein. Mittlerweile ist Java 8 verfügbar und im iJUG sind bereits 22 Usergroups aus Deutschland, Österreich und der Schweiz vereint. JavaLand 2014, die erste gemeinsame Veranstaltung, ist auf Anhieb eine der gefragtesten Java-Events in Europa.

Was mich besonders freut: Die Java aktuell hat bereits nach einem Jahr einen führenden Platz unter den Java-Magazinen hierzulande erreicht. Heute gilt die Zeitschrift als fester Bestandteil der deutschsprachigen Java-Community. In diesem Sinne bin ich gespannt auf die kommenden fünf Jahre.

Ihr



PS: Über die QR-Codes beziehungsweise Links unter jedem Artikel können Sie mir jederzeit Feedback geben.

Happy Birthday Java aktuell

Wege zum Aufbau von modernen Web-Architekturen

Groovy-Entwickler argumentieren, dass Java in die Jahre gekommen sei

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 dritten Quartal 2014.

15. Juli 2014

JSR für „JSON Binding" soll bald starten

„JSON Binding" war das am meisten gewünschte Thema in der Community-Umfrage zu Java EE 8. Nach dem Vorbild von JAXB soll es die Konvertierung von JSON in Java-Objekte und umgekehrt standardisieren, aufbauend auf dem JSR-353 („JSON Processing"). Oracle hat jetzt den JSR in die Wege geleitet: Der designierte Spec-Lead Martin Grebac veröffentlichte einen Entwurf des JSR Proposal und bittet um Feedback. Wenn alles nach Plan läuft, wird die Expertengruppe noch im Herbst zusammengestellt und Anfang 2015 bereits einen Entwurf präsentieren. Ein Jahr später soll dann die finale Version vorliegen. Der Zeitplan ist auf eine Aufnahme in Java EE 8 abgestimmt, die Spezifikation ist aber mit Java EE 7 kompatibel.

https://blogs.oracle.com/theaquarium/entry/standardising_json_binding

JDK 8 Update 11 mit Bug-Fixes und einigen Neuerungen

Das JDK 8 Update 11 ist freigegeben und kommt neben den üblichen Bug-Fixes mit einigen Neuerungen: Zum einen das Java-Dependency-Analysis-Tool („jdeps") zur statischen Analyse von Abhängigkeiten in Java-Applikationen und Libraries, zum anderen das neue Attribut „Entry Point", mit dem für ein Jar-File die erlaubten „Einstiegspunkte" eingeschränkt werden können, falls es mehrere infrage kommende Klassen (Applets, JavaFX-Application-Klassen oder solche mit einer main()-Methode) gibt.

Eine kleine Zugabe am Rande: Die nervigen „Sponsor Offers" während der Installation von Updates lassen sich nun auch per Java-Control-Panel-Einstellung abschalten.

http://www.oracle.com/technetwork/java/javase/8u11-relnotes-2232915.html

17. Juli 2014

Context und Dependency Injection 2.0

CDI hat sich zu einem zentralen Teil von Java EE entwickelt, auch wenn es noch längst nicht alle Einzelspezifikationen unterstützt. Jetzt wurde der Start des CDI 2.0 JSR angekündigt. Spec-Lead Antoine Sabot-Durand hat seine Meinung schon vor einiger Zeit veröffentlicht. Dieser Blog ist generell empfehlenswert für interessante CDI-Einsichten. Wesentliche Themen werden wohl die Unterstützung für Java SE (ab Version 8) und eine Modularisierung inklusive „CDI Lite" sein. Von der Modularisierung verspricht man sich in Zukunft auch eine bessere Integration in Java-EE-Einzelspezifikationen. Eines der weiteren Themen ist ein „CDI Container Hot Swap". Mal sehen, wie viel am Ende drin ist, die Meilensteine sind jedenfalls für die Integration in die Java EE 8 Roadmap gesetzt.

www.nextpresso.com/2014/03/forward-cdi-2-0

22. Juli 2014

JavaServer Faces 2.3

Der alte und neue JSF-Spec-Lead Ed Burns hat einen Textvorschlag für die Einreichung von JavaServer Faces 2.3 beim Java Community Process veröffentlicht. Eines der Themen ist eine verbesserte CDI-Integration (@Inject FacesContext, Aufruf CDI-verwalteter Bean-Methoden mit Ajax), weitere Kandidaten sind Multi-Feld-Validierung, EL-Performance-Optimierung sowie Verbesserungen bezüglich „Cross-form Ajax". Eine weitere Neuerung: JSF 2.3 soll nicht nur in Java EE 8 integriert werden, sondern auch selbst EE-8-Features nutzen – in der Vergangenheit war die Grundlage immer die vorhergehende EE-Spezifikation. Ed Burns wird diesmal von Manfred Riem (ebenfalls Oracle) als „Co-Spec-Lead" unterstützt.

https://blogs.oracle.com/theaquarium/entry/javaserver_faces_2_3

24. Juli 2014

Servlet 4.0

Ein weiteres Spezifikations-Update, das für die Aufnahme in Java EE 8 geplant ist: Servlet 4.0. Wie der Sprung in der Versionsnummer verspricht, wird es größere Neuerungen geben: Hauptthema ist HTTP/2, vormals „HTTP 2.0" geschrieben (die bislang unterstützte Version 1.1 stammt noch aus dem letzten Jahrtausend). Spec-Lead Shing Wai Chan hat einige der HTTP/2-Features aufgelistet, die von Servlet 4.0 unterstützt werden sollen: Request/Response Multiplexing, Stream-Priorisierung, Server Push basierend auf Multiplexing, (zwischen Browser und Server verhandelter) Upgrade von HTTP 1.1. Die Komplexität wird mit diesen Themen deutlich steigen. Um dies zu bewältigen, wird Ed Burns als Co-Spec-Lead mit einsteigen. Ein (zeitliches) Risiko ist natürlich die Abhängigkeit vom noch unfertigen HTTP/2. Da der RFC für HTTP/2 aber bereits Anfang 2015 veröffentlicht werden soll, wird vermutlich genug Zeit vor der Finalisierung von Java EE 8 bleiben.

https://blogs.oracle.com/theaquarium/entry/servlet_4_0

31. Juli 2014

„Man in the middle"-Attacken bei Jar-Downloads? In Zukunft bitte nicht mehr

In seinem Blog hat Andreas Kull sich darüber gewundert, dass Sonatype, unter anderem Betreiber von Maven Central, keine generelle Verschlüsselung für den Zugriff auf den Service anbietet. Ein verschlüsselter Zugang mit personalisiertem Token wird nur gegen eine Spende von zehn Dollar an die Apache Foundation zur Verfügung gestellt. In einem kleinen Proof of Concept demonstriert er, wie leicht eine „Man in the middle"-Attacke möglich ist, mit der Jars beim Herunterladen

verändert werden. Die SHA1-Checksumme lässt sich auf diesem Weg natürlich ebenfalls verändern und Jar-Signaturen werden in der Regel nicht genutzt, sodass die Veränderung unentdeckt bleibt.
Sonatype hat auf seine Anfrage reagiert und bietet jetzt eine generelle Verschlüsselung an. Die Spende, so die Begründung, sollte eine minimale Hürde sein, um den tatsächlichen Bedarf an Verschlüsselung herauszufinden. Innerhalb von zwei Jahren hätten sich aber – trotz Aufklärung über die Risiken eines unverschlüsselten Zugangs – gerade mal zwölf Entwickler registriert. Inzwischen sei die Aufmerksamkeit für solche Gefahren aber wohl gestiegen, daher werde die Hürde jetzt zugunsten einer generellen Verschlüsselung abgeschafft.
Ich muss jetzt gleich los, um die ganzen Jar-Files zu überprüfen, die ich in den letzten Jahren heruntergeladen habe …
http//blog.ontoillogical.com

1. August 2014

Kleiner Kreis trifft sich zur Zukunft der Sprache Java

Der jährliche JVM Language Summit hat diese Woche auf dem Oracle Campus stattgefunden; ein Treffen, bei dem sich eine kleinere Zahl von JVM- und Sprach-Entwicklern – häufig Repräsentanten großer Java-Nutzer – über neue Anforderungen und mögliche Umsetzungen austauschen. Zwei Projekte wurden intensiv diskutiert: „Valhalla", nach Aussage des JVM-Architekten John Rose die größte Änderung am JVM-Type-System bislang, auch mit dem Ziel, die JVM für neue Hardware fit zu machen, und „Panama", ein nach außen gerichtetes Vorhaben, um die JVM auf neue Arten mit Nicht-Java-APIs zu verbinden. Ein Blick in die Zukunft des OpenJDK.
http://openjdk.java.net/projects/mlvm/jvm-langsummit

13. August 2014

Apache DeltaSpike 1.0 macht mehr aus CDI

Apache hat die Version 1.0 von DeltaSpike angekündigt, einem Satz portabler Erweiterungen für CDI. Das Projekt basiert auf JBoss Seam, Apache CODI und einigen anderen Projekten. Es soll nach Aussage von Pete Muir von RedHat/JBoss die Vielzahl der CDI-Erweiterungen wieder zu einer gemeinsamen Basis zurückführen. Es bietet eine Reihe von Features, die für den Kern von CDI erst für 2.0 (Java EE 8) auf der Liste stehen, und stellt auch einige EE-7-Features schon unter EE 6 zur Verfügung.
https://deltaspike.apache.org

18. August 2014

walkmod: Automatisiertes Prüfen und Fixen der Code-Qualität

Für geplagte Code Reviewer und ihre Entwickler: Auf der Devoxx UK hat walkmod ein gleichnamiges Open-Source-Tool vorgestellt, mit dem nicht nur Coding Conventions geprüft, sondern eine große Zahl automatisierter Fixes auch gleich erledigt werden. Das Tool lässt sich mit Plug-ins beliebig erweitern.
http://www.walkmod.com

19. August 2014

JDK 8 Update 20

Oracle JDK 8 enthält mit dem neuen Update 20 zwei für System-Administratoren interessante Features (die allerdings der kostenpflichtigen „Java SE Advanced License" unterliegen): Die „Advanced Management Console" benutzt den Java Usage Tracker und Deployment Rule Sets, um Daten zur Nutzung von Java WebStart und Applets im Unternehmen zentral zu sammeln und zu visualisieren sowie effizient Regeln für die Nutzung und die einzusetzenden Java-Versionen zu definieren und umzusetzen. Der „MSI Installer" für Windows integriert sich in Desktop-Management-Werkzeuge und sorgt für eine reibungslosere, auch interaktionsfreie Verteilung von Java auf Desktops. Darüber hinaus ist das Sicherheitslevel „Medium" abgeschafft worden; Applets, die nicht mit den aktuellen Sicherheitsregeln konform sind, können also nicht mehr gestartet werden, es sei denn, die spezifische Site, von der sie heruntergeladen werden, ist in die „Exception Site List" eingetragen.
http://www.oracle.com/technetwork/java/javase/8u20-relnotes-2257729.html

6. August 2014

JAX-RS 2.1 JSR eingereicht

Auch das Java-API für RESTful Web Services soll für Java EE aufpoliert werden. Ein paar der Themen wurden schon in anderem Zusammenhang im Tagebuch erwähnt: Integration mit JSON-B, verbesserte Integration mit CDI, Support für „Server-Sent Events" (SSE), Unterstützung von nicht-blockierendem I/O in Providern, deklarative Sicherheit und einiges mehr.
https://java.net/projects/jax-rs-spec/lists/users/archive/2014-08/message/15

26. August 2014

Noch ein MVC für Java?

Ein weiteres Resultat des „Java EE 8 Community Survey" aus dem Frühjahr ist der MVC 1.0 JSR: Es soll neben dem etablierten, Komponenten-basierten JSF eine Spezifikation für einen weiteren MVC-Ansatz geben, der Action-basiert ist und damit näher an bekannten Frameworks wie Struts und Spring MVC. Die ursprüngliche Idee, dies innerhalb der JAX-RS-Spezifikation zu machen, wurde verworfen. JAX-RS ist aber immer noch eine mögliche Grundlage für den Controller-Teil. Manfred Riem, neuer Co-Spec-Lead für JSF, wird auch für MVC 1.0 einer der beiden Spec-Leads sein, somit sollte eine weitestgehende Abstimmung zwischen den beiden Spezifikationen sichergestellt sein.
Nachtrag: Ed Burns stellt noch einmal klar, dass es nicht um einen Ersatz für JavaServer Faces geht, sondern beide Spezifikationen ihre Berechtigung haben. In einem zum Lesen empfohlenen Blog-Eintrag vergleicht er beide Ansätze sehr anschaulich.
www.oracle.com/technetwork/articles/java/mvc-2280472.html

27. August 2014

Java EE 8 startet

Java EE 8 ist offiziell als JSR gestartet – die Annahme durch den „Review Ballot" ist wohl als Formalität anzusehen. Einige der Neuerungen wurden im Tagebuch bereits erwähnt, hier eine Auswahl daraus, was uns schließlich im Q3 2016 erwartet (wenn der Zeitplan eingehalten wird): Einige Cloud-Features sind mal wieder geplant, darunter verbesserte Konfiguration mit Multi-Mandanten-Unterstützung, REST-basierte Management- und Monitoring-APIs sowie „Simplified Security". Erleichterungen für Entwickler stehen auch im Fokus, unter anderem eine Verbesserung und Vereinheitlichung des Managed-Bean-Modells. Natürlich soll Java SE 8 mit allen Features als

Basis genutzt werden (Lambda Expressions, Type Annotations etc.). Kandidaten für neue Einzelspezifikationen sind, wie bereits früher erwähnt, JCache, JSON-B und MVC.
https://jcp.org/en/jsr/detail?id=366

9. September 2014

GlassFish Server Open Source Edition 4.1 freigegeben

Die Version 4.1 von GlassFish ist da, im Gepäck die Referenz-Implementierungen zu WebSockets 1.1 (Tyrus), JAX-RS 2.0 (Jersey) und JMS 2.0 (OpenMQ). Jersey unterstützt jetzt OAuth 2 zumindest als Client, auch wenn das für einen Applikationsserver sicher der weniger interessante Teil ist.
https://glassfish.java.net/download.html

14. September 2014

Java-Configuration-Projekt muss Pläne ändern

Der Java EE Configuration JSR wird vermutlich nicht wie ursprünglich geplant starten. Hauptgrund ist die bislang mangelnde Unterstützung durch weitere JCP-Mitglieder, insbesondere was die Aufwände für das Test-Compatibility-Kit und die Referenz-Implementierung angeht. Diese wollte Credit Suisse, Arbeitgeber des designierten Spec-Lead-Initiators Anatole Tresch, nicht allein tragen. Momentan arbeitet Anatole an einem Plan B: Java-EE-Deployment-Konfiguration spielt darin keine große Rolle mehr, es geht vielmehr um die Standardisierung einer generischen Applikationskonfiguration in Java SE und die Integration mit CDI, um diese Konfiguration umzusetzen. Vielleicht taucht ja auch „Deltaspike" in diesem Zusammenhang wieder auf.
http://javaeeconfig.blogspot.nl

16. September 2014

Java 8 wird der neue Standard

Java 8 soll aufgrund der großen Akzeptanz noch vor Ende des Jahres Java 7 als Standard-Java-Version für alle Nutzer ablösen, wie Henrik Ståhl in einem Blog-Post angekündigt hat. Zum einen wird mit dieser Änderung auf vielen Systemen der automatische Update-Prozess für das installierte Java auf die neue Version gestartet, zum anderen wird es voraussichtlich ein halbes Jahr später keine weiteren Sicherheits-Updates für Java 7 ohne den Kauf einer Oracle-Lizenz mehr geben. Unternehmen, die Anwendungen verwenden, die auf der Standard-Java-Installation aufsetzen, oder Webseiten mit Java-Applets einsetzen, sollten jetzt mit Tests daraufhin beginnen, ob es trotz der Bemühungen zur Rückwärtskompatibilität zu Problemen mit dem neuen Major Release kommt. Der Interessenverbund der Java User Groups e.V. (iJUG) hat zu diesem Thema eine Pressemitteilung herausgegeben und fordert insbesondere alle Software-Hersteller, die Java-Applets nutzen, auf, wenn notwendig neue Releases für Java 8 zur Verfügung zu stellen, um die gestiegenen Sicherheitsanforderungen an Applets zu erfüllen.
http://www.ijug.eu/home-ijug/aktuelle-news/article/java-8-wird-der-neue-standard-fuer-alle.html

23. September 2014

Alle Java EE 8 JSRs angenommen

Kurz und schmerzlos: Alle bislang für Java EE 8 eingereichten Einzel-JSRs sind jeweils einstimmig angenommen worden und die Expertengruppen können nun ihre Arbeit aufnehmen.
https://jcp.org/en/jsr/detail?id=366

30. September 2014

JavaOne Kick-Off

Es ist wieder soweit: Die JavaOne 2014, höchster (Selbst-)Feiertag für alle Java-Fans, startet in San Francisco. Erwartungsgemäß verkündet Oracle in der strategischen und technischen Keynote keine großen Überraschungen. Alle Java-Technologien entwickeln sich im Wesentlichen nach Plan; Themen wie Cloud und IoT sind auch nicht mehr der große Hype, sie müssen einfach umgesetzt werden. Trotzdem ist es im Detail natürlich immer noch spannend, gerade wenn es um etwas zum Anfassen geht: Eine IoT-Demo zur ganzheitlichen Automobilsteuerung zeigt alle Java-Technologien im Client-, Gateway- und Server-Tier im Zusammenspiel. Die Vorstellung der nach Java SE 9 geplanten Features durch Brian Goetz (Chef-Architekt für die Sprache Java) am Ende der Keynote wird zur Enttäuschung der Zuhörer aus Zeitgründen abgebrochen, die Inhalte sollen aber auf der Konferenz in einzelnen Vorträgen dargestellt werden.
http://www.ijug.eu/home-ijug/aktuelle-news/article/javaone-2014-alles-im-lot.html

2. Oktober 2014

JavaOne beendet

In der abschließenden Keynote gehen Brian Goetz und Plattform-Architekt Mark Reinhold doch noch auf „Java SE 9 and beyond" ein. Sie bestätigen, dass Modularisierung (Projekt „Jigsaw") fest für SE 9 eingeplant ist. Was folgt jenseits von Java SE 9? „In der Vergangenheit habe man sich mehr darum gekümmert, Entwickler bei der Lösung immer komplexerer Probleme zu unterstützen", so Goetz. Als Nächstes sei es wichtig, Java an neue Hardware und geänderte Anforderungen an Art und Menge von Daten anzupassen, damit es nicht irrelevant wird – siehe Projekt „Valhalla". Im Weiteren beschreibt er dieses und das ebenfalls bereits erwähnte Projekt „Panama".
https://www.oracle.com/javaone/live/on-demand

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/15/1/2

JavaOne 2014: Alles im Lot

Peter Doschkinow und Wolfgang Weigend, ORACLE Deutschland B.V. und Co. KG

Erwartungsgemäß hat Oracle in der strategischen und technischen Keynote keine großen Überraschungen verkündet, da die Java-Technologien planmäßig nach den veröffentlichten Roadmaps geliefert werden. Man ist sich einig, Java ist erfolgreich, Java entwickelt sich weiter, Java ist innovativ. Als Zugabe zeigte man eine spannende, Java-basierte IoT-Demo zur ganzheitlichen Automobilsteuerung, die alle Java-Technologien im Client-, Gateway- und Server-Tier veranschaulichte. Die Vorstellung der nach Java SE 9 geplanten Features fiel zur Enttäuschung der Zuhörer aus Zeitgründen aus. Mark Reinhold, Chief Architect der Java Platform Group, betonte aber, dass diese Inhalte auf der Konferenz in einzelnen Vorträgen dargestellt werden. Die Zusammenfassung in den einzelnen Bereichen:

Java-Community

- Weltweit 80 neue JUGs im letzten Jahr
- 2014 Duke's Choice Award winners: https://blogs.oracle.com/java/entry/2014_duke_s_choice_award
- Steigende Beteiligung der Java Community:
  - Neue Konferenzen: JavaLand, organisiert von 22 JUGs aus Deutschland, Österreich und der Schweiz
  - Ausbildungsinitiativen: London JUG
  - Mitarbeit bei der Standardisierung: Adopt-a-JSR
  - JEP-Beteiligung: JEP-104, JEP-150, JEP-155, JEP-171, JEP-175
  - JCP feiert 15-jähriges Bestehen

Java SE

- Schwerpunkt auf Lambda und Streams sowie ihrer Anwendung in Kundenprojekten
- Security-Verbesserungen für Entwickler und Administratoren in Bezug auf das Java-Plug-in, WebStart und diverse Sicherheitsstandards
- JDK 9 Roadmap
  - Modularity mit Jigsaw
  - Unterstützung für HTTP 2.0
  - Lightweight JSON
  - Cloud-optimierte JVM

Abbildung 1: JavaOne-Keynote, Peter Utzschneider, Vice President Product Management für Java bei Oracle

  - Fortsetzung der Java SE Advanced Features
  - Ahead-of-Time Compilation
  - JDK9 kann bereits heruntergeladen und getestet werden

Java ME

- Steigende Verbreitung: 500.000 Downloads im letzten Jahr
- Mehr als 20 neue Hardware-Plattform-Portierungen
- Ankündigung von Java ME 8.1 Early Access mit nur 190 KB Footprint auf FRDM-K64F Hardware http://tinyurl.com/mlctzyj

Java EE

- Anerkennung der Beiträge von Herstellern, JUGs und Einzelpersonen
- Einstimmige Annahme der Java-EE-8-Spezifikation und der folgenden Teilspezifikationen: Servlet 4.0, JSON-Binding, JAX-RS 2.1, JSF 2.3, MVC 1.0, JMS 2.1, CDI 2.0
- GlassFish 4.1 ist freigegeben mit diversen Java-EE-7-API-Updates und Java-SE-8-Support
- WebLogic 12.1.3 ist ab sofort für Java SE 8 zertifiziert und somit der erste kommerzielle AppServer, der Java SE 8 unterstützt

Peter Doschkinow
peter.doschkinow@oracle.com

Peter Doschkinow arbeitet als Senior Java Architekt bei Oracle Deutschland. Er beschäftigt sich mit serverseitigen Java-Technologien und Frameworks, Web Services und Business Integration, die er in verschiedenen Kundenprojekten erfolgreich eingesetzt hat. Vor seiner Tätigkeit bei Oracle hat er wertvolle Erfahrungen als Java Architect and Consultant bei Sun Microsystems gesammelt.

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/15/1/3

Fünf Jahre Java aktuell

Vor dem Hintergrund der Sun-Übernahme durch Oracle haben sich Ende des Jahres 2009 sechs Java-Usergroups und die DOAG Deutsche ORACLE Anwendergruppe e. V. zum Interessenverbund der Java User Groups e.V. (iJUG) zusammengeschlossen. Mittlerweile sind es bereits 22 Usergroups aus Deutschland, Österreich und der Schweiz. Ende des Jahres 2010 erschien die erste Ausgabe der Java aktuell. Aus Anlass des fünfjährigen Jubiläums blicken Autoren der Zeitschrift und Mitglieder des Interessenverbunds der Java User Groups e.V. (iJUG) auf die vergangenen fünf Jahre zurück und wagen einen Ausblick, wie das Java-Biotop in fünf Jahre aussehen könnte.

Per Anhalter durch die Java-Galaxie

Java SE und seine Schwester Java EE sind von frechen Jugendlichen zu reifen Erwachsenen herangereift. Ihr Bruder Java ME ist ihnen dabei abhandengekommen. Die Zeiten, in denen Applikations-Server die Flaggschiffe für Java EE waren, sind vorbei. Die eingetretene Markt-Konsolidierung hinterließ neben einer kurzen Unsicherheit auch eine neue Übersichtlichkeit. Sie ist letztendlich Ausdruck eines gereiften Markts.

Inzwischen läuft Java fast überall – von kleinen bis zu großen Geräten. Nach dreißig Jahren wird die Sun-Vision „The Network Is The Computer" mit dem Internet der Dinge und Cloud Computing Wirklichkeit. Endlich stehen mit Milliarden über das Internet kommunizierenden Geräten und im Netz nahezu unbegrenzt und jederzeit verfügbaren Ressourcen Kapazitäten bereit, die unseren Alltag verändert haben und noch verändern werden.

Die Stärken von Java sind nach wie vor Stabilität, Flexibilität und Offenheit. Was ursprünglich von einer Handvoll Entwickler entworfen wurde, wird inzwischen milliardenfach in verschiedenen Anwendungen auf unterschiedlichsten Geräten eingesetzt und von Millionen von Programmierern verwendet.

Auch wenn Java unser Leben nicht wie in der Vision des Java-Tutorials verändert hat, so hat es die Software- und Programmier-Landschaft doch sehr bereichert. Letztendlich bleibt Java mehr als die Summe seiner Teile.

Java wird auch als „Cobol des 21. Jahrhunderts" bezeichnet – zumindest Java-Programmierer brauchen sich keine Zukunftssorgen zu machen; sie führen immer noch viele Job- und Projekt-Statistiken an. Die Zukunft der Java-Plattform ist nach stürmischen Zeiten bei Oracle und dem dazu gehörigen Java Community Process (JCP) in guten Händen.

Frank Pientka
frank.pientka@materna.de

Frank Pientka ist Senior Software Architect bei der MATERNA GmbH in Dortmund. Er ist zertifizierter SCJP und Gründungsmitglied des iSAQB. Als Autor eines Buches zu Apache Geronimo beschäftigt er sich intensiv mit dem Einsatz von Java-Open-Source-Software, insbesondere mit Applikationsservern.

http://ja.ijug.eu/15/1/4

2010

2011

2012

2013

2014

Java – gestern, heute, morgen

Java in den letzten fünf Jahren: Eine gesetzte Person, die ihre Midlife-Crisis überwunden hat und mit neuem Schwung die nächsten Jahre angeht. Während Java im Jahr 2009 noch eine feste Größe war und es wenig Fortschritt gab, konnte Oracle nach einigen Unruhen in der Community aufgrund der Sun-Übernahme dann doch zeigen, dass es auf die Community hört.

Trends waren: JVM als Plattform für unterschiedliche Sprachen; Öffnung für dynamische Sprachen wie JRuby und Groovy, aber auch funktionale wie Clojure und Scala. Nicht zuletzt die funktionale Erweiterung durch Lambdas in Java selber.

Java EE ist leichtgewichtiger geworden. In der Zukunft wird sich hier noch einiges tun: Von der Vereinheitlichung der unterschiedlichen Komponenten-Modelle zu einem einzigen über die stärkere Unterstützung moderner HTTP-2.0-Möglichkeiten bis hin zu asynchroner Kommunikation, angetrieben durch das Internet der Dinge.

Polyglot Programming und Polyglot Persistence werden noch wichtiger werden und Anforderungen aus dem Bereich „Continuous Delivery" dazu führen, dass leichtgewichtiges und entwicklerfreundliches Arbeiten weiter an Bedeutung gewinnt, wofür heute ja schon DropWizard oder Spring Boot sorgen. Für Architekten und Entwickler wird es eine Herausforderung sein, die Breite des Java-Ökosystems noch zu überblicken, sodass ich hier eine vermehrte Spezialisierung erwarte. Es liegt also eine spannende Zeit vor uns!

Richard Attermeyer
richard.attermeyer@opitz-consulting.com

Richard Attermeyer ist Senior Solution Architects bei der OPITZ CONSULTING Deutschland GmbH. Er beschäftigt sich seit vielen Jahren als Entwickler, Architekt und Coach mit den Themen „Enterprise Applikationen" und „Agile Projekte".

http://ja.ijug.eu/15/1/5

Java 2014 ± 5

Die Entwicklung von Java sollte nicht nur unter technischen Gesichtspunkten betrachtet werden. Je erfolgreicher sich eine Programmiersprache etabliert, desto stärker gewinnen übergreifende Faktoren an Einfluss. Diese kommen in der öffentlichen Diskussion häufig zu kurz.

Ein wichtiger Punkt, der gemeinhin mit einem Achselzucken abgetan wird, ist der, dass sich der Abstand zwischen den beiden wesentlichen Gruppen der Java-Anwender weiter vergrößert hat. Während auf der einen Seite aktiv und engagiert an der Fortentwicklung der Sprache und ihrer Umgebung gearbeitet wird, gibt es auf der anderen Seite eine große Gruppe von Anwendern, die ihre Aufgaben mit den lange etablierten Mitteln lösen müssen.

In gewisser Weise lässt sich dieser Abstand sogar messen, wenn man die jeweils aktuelle Java-Version mit der in den Projekten eingesetzten vergleicht. Allerdings ist das nur die halbe Wahrheit, weil die Verwendung einer bestimmten Java-Version noch nicht bedeutet, dass die für diese adäquaten Techniken auch verwendet werden. Java ist mit jedem Release gewachsen und komplexer geworden. Dementsprechend müsste sich zum Beispiel die Dauer von Schulungen seit dem Jahr 2000 mindestens verdoppelt haben, um den gleichen Überblick über die gesamte Java-Plattform zu vermitteln. Da das jedoch (gewöhnlich) nicht erfolgt ist, bleibt es oft der Initiative des Einzelnen überlassen, sich diese Kenntnisse anzueignen – oder auch nicht.

Dabei ist es ganz natürlich, dass unter wirtschaftlichen Gesichtspunkten die Entwicklung großer und aufeinander aufbauender Anwendungssysteme nicht dem relativ kurzen Release-Zyklus von drei Jahren folgen kann. Wie in anderen Bereichen (siehe Linux) wird wohl auch für Java das Schlagwort „Long Term Support" (LTS) an Bedeutung gewinnen. Anwendungen werden auch in Zukunft zwanzig oder dreißig Jahre laufen und über die Jahre wachsen. Das ist keine Innovationsfeindlichkeit oder Trägheit – dafür gibt es objektive Gründe.

Wie schwierig es im Allgemeinen schon ist, die extrem wachsende Komplexität zu beherrschen, zeigen die immer häufiger werdenden Rückruf-Aktionen. Tendenziell werden sich die Entwicklungszeiten deshalb eher verlängern als verkürzen. Aufwändig entwickelte Produkte brauchen länger, um sich zu rentieren. Kostenverursachende Release-Wechsel ohne großen Nutzen sind dabei unerwünscht. Gleichzeitig muss aber der zuverlässige Betrieb gewährleistet sein.

Bei einem alten Mainframe-Programm lässt sich die notwendige Sicherheit vielleicht noch durch die Abschottung des Rechenzentrums erreichen. Vernetzte Anwendungen bleiben jedoch über den gesamten Lebenszyklus auf Sicherheits-Updates angewiesen. Das stellt auch für Open-Source-Software eine erhebliche Herausforderung dar.

Die Community hat bei ihren Aktivitäten zu oft die Erstellung neuer Software im Fokus. Das ist verständlich, weil es einfach mehr Spaß macht, Neues zu entwickeln. In der Praxis wird jedoch viel mehr Zeit damit verbracht, Code zu lesen, zu verstehen und zu ändern. Allzu oft wird Objektorientierung fälschlich mit leichter Wartbarkeit gleichgesetzt. Die „Clean Code"-Bewegung ist eine Reaktion auf die mittlerweile herangewachsenen Probleme. Unabhängig davon, wie man zu einzelnen ihrer Thesen steht, die Behauptung, dass Code ohne intensives Gegenarbeiten (= „Refactoring") verkommt, wird wohl niemand widerlegen können.

Ein ursprünglicher Vorteil von Java war, dass dem „Klasse durch Masse" von C++ durch ein „klein aber fein" entgegengetreten wurde. Wenn man in C++ ganz unterschiedliche syntaktische Strukturen hatte, um etwas auszudrücken, gab es in Java meist nur einen vernünftigen Weg. Diese Tatsache erleichterte das Erlernen der Sprache und das Lesen des Programmcodes. Die zwischenzeitlichen Erweiterungen haben diesen Vorsprung

schrumpfen lassen. Aus meiner Sicht ist es überfällig, bei der Integration neuer Features eine Pause einzulegen, den erreichten Stand kritisch zu analysieren und zu konsolidieren – also ein Refactoring zu starten.

Bei all den Weiterentwicklungen sind beispielsweise viele der System-Bibliotheken nur angepasst worden. Da findet sich noch manches To-do aus den Anfangstagen oder Code (auch vermeintliche Optimierungen aus Vor-Hotspot-Zeiten), den kein erfahrener Entwickler mehr so schreiben würde. Alternativ oder parallel zur Überarbeitung der System-Bibliotheken sollte über deren Neugestaltung nachgedacht werden, und zwar derart, dass die alternativen Neu-Implementierungen langfristig ihre Vorgänger ersetzen können. Je länger eine solche Konsolidierung hinausgezögert wird, desto größer und schwieriger wird sie. Die Altlasten, die teilweise dem Zeitdruck bei der Veröffentlichung anzulasten sind, verseuchen mangels Alternativen noch heute jede Neuentwicklung.

Ungelöst ist immer noch die Modularisierung. Das Projekt „Jigsaw" wird von einer Version zur nächsten verschoben. Dabei ist es zweifelhaft, ob dem enormen Aufwand ein entsprechender Nutzen gegenüberstehen wird. Wenn man sich daran erinnert, dass in technischen Übersichten fast immer eine zweidimensionale Struktur aus Funktionssäulen und Implementierungsschichten auftaucht, ist die Frage, ob Baumstrukturen für die Modularisierung angemessen sind, nicht völlig abwegig.

Für die Zukunft sieht sich die Java-Community verschiedenen Herausforderungen gegenüber. Ein großes Problem ist – wie in vielen anderen Bereichen auch –, dass langfristig erfolgversprechende Strategien auf kurze Sicht oft unattraktiv sind und gleichzeitig die hohe Volatilität der Entwicklung Risiken fast unkalkulierbar macht. Ich sehe die folgenden wichtigen Tendenzen, die sich gegenseitig nicht völlig ausschließen müssen:

- *Die Entwicklung verläuft weiter wie bisher*
  Das ist (wie beim Wetter) die wahrscheinlichste Variante. Java wird dabei immer komplexer, der Abstand zwischen den oben erwähnten Gruppen der Anwender wächst weiter. Über kurz oder lang wird eine andere, neue Programmiersprache Java den Rang ablaufen. Letzteres ist auch deshalb wahrscheinlich, weil nach zwanzig bis fünfundzwanzig Jahren die Welt für ein neues Heilsversprechen reif ist, selbst wenn insgeheim jeder weiß, dass es nur ein kleiner Schritt vorwärts sein wird.
- *Die Entwicklung spaltet sich auf*
  Etwa so, wie das in gewissen Maß beim Firefox zu sehen ist, in einen erneuerungsfreudigen und einen eher bestandswahrenden (LTS-)Zweig. In Ansätzen praktizieren das bereits einige Anbieter von eigenen Java-Implementierungen. Zusammen mit der vorherigen Tendenz kann das bedeuten, dass auf die Dauer nur der (eventuell rein kommerzielle) LTS-Zweig überlebt. Für die IT-Industrie muss das nicht schlecht sein, das zeigt ein Blick auf Sprachen wie Ada oder Cobol, für Entwickler ist das eher weniger cool.
- *Java erneuert sich*
  Das ist das wünschenswerte, aber mit Abstand unwahrscheinlichste Szenario. Auf der Basis einer umfassenden Entsorgung der Altlasten in Syntax und Bibliotheken erfolgt ein evolutionärer Übergang zu einem echten „Java2". Die Default-Implementierungen von Interface-Methoden in Java 8 sind ein ganz kleiner Schritt in diese Richtung. Die vielen interoperablen Implementierungen anderer Programmiersprachen auf der JVM beweisen, dass dieser Weg technisch realisierbar ist.

Ganz gleich, wie es kommt – es wird interessant bleiben. Und hoffentlich wird wenigstens ein Teil der vielen bisher gewonnenen Erfahrungen in die Weiterentwicklung einfließen.

Java lebt!

Fünf Jahre Java aktuell. Ich gratuliere recht herzlich und möchte meine Eindrücke aus den letzten fünf Jahren und meine Vision, wohin sich Java und die Community entwickeln könnten, mit den Glückwünschen verbinden.

„Java ist tot" waren meine ersten Gedanken zur Übernahme von Sun durch Oracle. Ich denke, vielen von Euch ging es ähnlich. Natürlich sah ich auch Chancen, aber im Grunde war dort nur ein schwarzes Loch von Ahnungslosigkeit, wie es mit Java weitergehen sollte. Und genau dort kam die Gemeinschaft ins Spiel. Sie versuchte einen Software-Riesen zu steuern und zu lenken. Stellte sich ihm in den Weg bei radikalen Einschnitten und unterstützte ihn bei guten Ideen und Weiterentwicklungen.

Natürlich konnte nicht jedes Projekt oder jede Idee gerettet werden. Auch Oracle hat nichts zu verschenken und will strategisch am Markt agieren. Aber schlussendlich hat die Gemeinschaft in meinen Augen gut funktioniert und ich möchte mich bei allen bedanken, die mitgewirkt haben, noch immer wirken und auch bald ein Teil dieser Gemeinschaft sind. Ich denke, mit dem aktuellen Release von Java hat Oracle die Weichen in Richtung Zukunft gestellt.

Java lebt! Alles Gute, Java aktuell! Ich freue mich, die nächsten fünf Jahre mit euch zu verbringen.

Jürgen Lampe
juergen.lampe@agons-solutions.de

Dr. Jürgen Lampe ist IT-Berater bei der A:gon Solutions GmbH in Frankfurt. Seit mehr als fünfzehn Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Banken-Umfeld. In dieser Zeit hat er sich immer wieder intensiv mit Performance-Problemen beschäftigt. An Fachsprachen (DSL) und Werkzeugen für deren Implementierung ist er seit seiner Studienzeit interessiert.

http://ja.ijug.eu/15/1/6

Gunther Petzsch
gunther.petzsch@saxsys.de

Gunther Petzsch ist Sun-zertifizierter Java-Entwickler. Er lebt und arbeitet in Dresden. Dort studierte er auch erfolgreich Wirtschaftsinformatik an der Hochschule für Technik und Wirtschaft Dresden. Danach arbeitete er unter anderem für das Bundeskriminalamt in Wiesbaden. Aktuell ist Gunther Petzsch als Senior Consultant für die Saxonia Systems AG tätig.

http://ja.ijug.eu/15/1/7

Java ist aktuell ein sicherer Hafen

Java hat in den letzten Jahren in der Tat einen größeren Aufschwung erlebt und wird in allen Bereichen der IT, von Kleinst-Systemen bis hin zum Großrechner, von kleinen Tools bis hin zu geschäftskritischen 24x7-Anwendungen, eingesetzt. Die hohe Geschwindigkeit, mit der Java-Anwendungen entwickelt werden können, haben jedoch auch Spuren hinterlassen. Im Artikel „Estimating the Principal of an Application's Technical Debt" von Bill Curtis, Jay Sappidi und Alexandra Szynkarski wird Java-EE-Anwendungen ein um 40 Prozent höherer Nachbesserungsbedarf bescheinigt als durchschnittlichen Anwendungen. Wir müssen hier aufpassen, dass wir die nächsten Jahre nicht damit beschäftigt sind, die Nachlässigkeiten der Vergangenheit nachzupflegen.

Dennoch, die Tatsache, dass viele große Unternehmen die Ablösung ihrer Altsysteme von Cobol und PL/1 nach Java planen, spricht für sich. Java ist aktuell der sichere Hafen für einen zukunftssicheren Betrieb einer Anwendungsplattform. Entwickler und Betriebspersonal sind in ausreichendem Maß vorhanden.

http://ja.ijug.eu/15/1/8

Marc Bauer
marc.bauer@de.ibm.com

Marc Bauer arbeitet im Software Service der IBM Deutschland GmbH. Er berät und unterstützt Kunden bei Modernisierungen von Mainframeanwendungen mit Hilfe von WebSphere und Java-Applikationen

Unbekannte Kostbarkeiten des SDK Heute: Informationen über die Virtual Machine und die Programmausführung

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.

Application-Server stellen in Management-Seiten vielerlei Informationen über die Virtual Machine (Executable, Version etc.) sowie die Programmausführung (durchgeführte Garbage Collections, Anzahl Threads etc.) bereit. Diese Informationen können jedoch auch in eigenen Anwendungen über einfach zu verwendende APIs erfragt und verwendet werden. Der Artikel stellt einige davon vor.

System-Properties

Die Methode „System.getproperties()" liefert die System-Properties der aktuellen Programmausführung. Im API-Dokument der Methode sind die Schlüssel aufgeführt, die standardmäßig vorhanden sind. Sie werden aber durch weitere Schlüssel ergänzt. Bei Java 8 sind dies im Augenblick 52, von denen einige in *Tabelle 1* mit ihren Werten beispielhaft dargestellt sind.

Exkurs: Abwärts- und Aufwärtskompatibilität

Java ist generell nicht aufwärtskompatibel. Eine Klasse, die mit Java 8 kompiliert wurde, ist unter Java 7 und kleiner nicht lauffähig. Bei der Ausführung wird die Exception „UnsupportedClassVersionError" geworfen und die nicht unterstützte Version angegeben. Im Falle von Java 8 ist dies der Fehlertext „Unsupported major.minor version 52.0", wobei 52.0 der Wert des System-Property „java.class.version" ist.

Java ist jedoch prinzipiell abwärtskompatibel. Klassen, die mit früheren Java-Versionen kompiliert wurden, sind auf neueren JVMs ausführbar. Kürzlich gab es ein damit zusammenhängendes Problem *[1]*. In den JVMs für 8u11 und 7u65 wurde ein bestimmter Punkt der Byte-Code-Verifikation erstmalig der Sprachdefinition entsprechend durchgeführt. Java verlangt, dass die erste Anweisung in einem Konstruktor der Aufruf des Konstruktors der Oberklasse ist. Dies wird selbstverständlich von allen Java-Compilern überprüft und entsprechender Byte-Code erzeugt.

Bei der nachträglichen Instrumentierung durch Byte-Code-Manipulationswerkzeuge wie „Javass" ist wird dies jedoch scheinbar von einigen Frameworks missachtet, sodass bei derartigen Klassen die Byte-Code-Verifikation

diesen Umstand erkennt und diese Klassen nicht ausgeführt werden können. Das Interessante an diesem Beispiel ist die Tatsache, dass rund zwanzig Jahre nach der ersten Java-Version in der JVM nun tatsächlich etwas verifiziert wird, was schon immer erfüllt sein musste.

Soll verhindert werden, dass ein Java-Programm mit einer höheren Java-Version ausgeführt wird, muss die Version der verwendeten Klassen mit der JVM-Version aus obiger Tabelle verglichen werden. Dies für die Praxis wenig relevante Problem ist so einfach zu lösen, dass wir es nicht vorenthalten wollen. *Listing 1* zeigt den Zugriff auf die ersten Bytes einer Java-Klasse, die die Java-Identifizierung („CAFEBABE") sowie die Minor- und Major-Version enthalten. *Listing 2* zeigt die Ausgabe, wenn die Klasse mit Java 8 kompiliert wird.

Javas Management-Interface

Das Package „java.lang.management" enthält eine Reihe von Schnittstellen zu MXBeans und eine Fabrikklasse zur Erzeugung von Instanzen dieser MXBeans. Es existieren MXBeans, die Informationen zum Laufzeitsystem, zu Speicherbereichen und zur Speicherverwaltung, zur Thread-Verwaltung und weiteren Themen bereitstellen.

Betrachten wir zunächst die Speicherbereiche. Die Fabrikmethode „ManagementFactory.getMemoryPoolMXBeans()" liefert eine Liste von MemoryPoolMXBeans, die die verschiedenen Speicherbereiche einer JVM repräsentieren. *Tabelle 2* zeigt diese Speicherbereiche in verschiedenen JVM-Versionen. Die Namen der Speicherbereiche wurden durch das folgende Code-Segment erzeugt *(siehe Listing 3)*.

Die ersten beiden Spalten der Tabelle wurden durch Aufruf der JVM ohne weitere Option, die dritte Spalte mit der Option „-XX:+UseG1GC" erzeugt. Man erkennt, dass in Java 8 der Permanent-Generation-Speicherbereich durch den Metaspace-Speicherbereich ersetzt sowie ein zusätzlicher Speicherbereich für die Repräsentation von Klassen eingeführt wurde.

Eine weitere wichtige Information ist die Größe einzelner Speicherbereiche. Das Interface „MemoryPoolMXBeans" stellt hierzu die Methoden „getUsage()", „getPeakUsage()" und „getCollectionUsage()" mit der offensichtlichen Semantik bereit, die jeweils eine Instanz von „MemoryUsage" zurückgeben. Diese Instanz enthält die vier Werte „init", „used", „committed" und „max", die den initialen, aktuell benutzten, verfügbaren und maximalen Speicherbereich in Bytes angeben.

Die JVM in ihrer heutigen Form erlaubt die Verwendung mehrerer Hundert Kommandozeilen-Optionen, von denen wir bereits „-XX:+UseG1GC" eingesetzt haben. Der Garbage Collector ist, wie im Beispiel, indirekt über die „MemoryPoolMXBeans" in Erfahrung zu bringen. Die Fabrikmethode „ManagementFactory.getGarbageCollectorMXBeans()" liefert eine Liste von „GarbageCollectorMXBeans" und erlaubt so den direkten Zugriff. Bei vielen weiteren Optionen gestaltet sich dies jedoch nicht so einfach. Abhilfe schafft hier die Methode „ManagementFactory.getRuntimeMXBean()", die Laufzeit-Informationen in Form des Interface „RuntimeMXBean" zurückgibt. Deren Methode „getInputArguments()" liefert die Liste aller Kommandozeilen-Optionen, also neben Standard-, „-X"- und „-XX"-Optionen auch Property-Definitionen mit „-D".

Fazit

Java und die JVM stellen eine Reihe von Schnittstellen zur Verfügung, um Informationen über die virtuelle Maschine und die Programmausführung zu erlangen. Es wurden die beiden großen Alternativen über Properties und MXBeans eingeführt und mit kleinen Beispielen vorgestellt.

Literatur/Videos

[1] http://www.infoq.com/news/2014/08/Java8-U11-Broke-Tools

Schlüssel	Wert
java.vm.version	25.11-b03
java.runtime.version	1.8.0_11-b12
java.class.version	52.0
java.specification.version	1.8
java.vm.specification.version	1.8
java.home	/java/jdk/jdk1.8.0_11/jre
os.name	Linux
user.home	/home/bernd
file.separator	/

Tabelle 1

Java 5/6/7	Java 8 (Default)	Java 8 (-XX:+UseG1GC)
PS Eden Space	PS Eden Space	G1 Eden Space
PS Survivor Space	PS Survivor Space	G1 Survivor Space
PS Old Gen	PS Old Gen	G1 Old Gen
Code Cache	Code Cache	Code Cache
PS Perm Gen	Metaspace	Metaspace
	Compressed Class Space	Compressed Class Space

Tabelle 2

Alle Listings finden Sie online unter:
http://www.doag.org/go/javaaktuell/link2

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/15/1/9


<Superheld/>
-Web-Applikationen
mit AngularJS
Joachim Weinbrenner, jsolutions

Dank HTML5, CSS3 und JavaScript wird der Browser mehr und mehr zum Betriebssystem für Anwendungen. So schießen zahlreiche Frameworks aus dem Boden, die der Erstellung solcher clientseitigen, im Browser laufenden Anwendungen dienen. Ein Kandidat hat sich in jüngster Zeit besonders hervorgetan: AngularJS aus dem Hause Google [1]. Dieses JavaScript-Framework bietet einen optimalen Weg zur Trennung von Design und Anwendungslogik sowie zur Anbindung an Backends. Der Artikel gibt eine grundlegende Einführung in die Implementierung von AngularJS-Anwendungen und zeigt Wege zum Aufbau von modernen Web-Architekturen.

AngularJS gehört zur Kategorie der „JavaScript MV*"-Frameworks, mit der sich Single Page Applications (SPA) erstellen lassen. In Anlehnung an Model View Controller (MVC) kommt das Model-View-ViewModel Pattern (MVVM-Pattern) zum Einsatz, wobei Google mittlerweile selbst von „Model View Whatever" (MVW) spricht und damit zum Ausdruck bringen will, dass es gewisse Interpretations-Spielräume gibt. AngularJS steht in Konkurrenz zu ähnlichen Frameworks wie „ember.js", „knockout.js" oder „backbone.js", ein Vergleich mit diesen soll hier aber außen vor bleiben. Zu den Hauptvorteilen von AngularJS zählen die Modularität, der lesbare Code, die strikte Trennung von Design und Logik, die Erweiterbarkeit und die vorzügliche Testbarkeit.

Aus einer einfachen HTML-Seite eine Angular-Anwendung zu machen, ist in zwei Schritten erledigt: Einbinden des Angular-Skripts, entweder eine heruntergeladene Version oder per Google CDN *[2]*, und zweitens Definieren der Anwendung via „ng-app". Die Angabe von „ng-app" kann in einem beliebigen HTML-Element erfolgen, typischerweise gleich ganz oben in „<html>" oder „<body>". Der Gültigkeitsbereich der Anwendung lebt dann in dem durch „ng-app" geklammerten Bereich.

Darüber hinaus kennt AngluarJS die Möglichkeit, Ausdrücke auszuwerten. Ein Ausdruck wird durch doppelte geschweifte Klammern angegeben, analog zum „<% %>" aus JSP. *Listing 1* zeigt eine minimale AngularJS-Anwendung, im Header wird das Skript eingebunden, im „<body>"-Tag die App definiert und in Zeile 6 der Ausdruck „1+1" ausgewertet. Im Browser erscheint dann an dieser Stelle die berechnete Zahl „2".

Hinweis: Alle Beispiele lassen sich in der Präsentation des Autors anlässlich der diesjährigen JavaLand 2014 nachvollziehen *[3]*. Die Sourcen stehen auf GitHub bereit *[4]*. Es sollen nun allerdings keine trivialen Rechenaufgaben gelöst, sondern Daten verarbeitet werden. AngularJS repräsentiert Daten in Models und so kommen wir zur „Hello Angular"-App, die einen eingegebenen Namen in Begrüßungsform wieder ausgibt *(siehe Listing 2)*.

Im „input"-Feld ist durch „ng-model" angegeben, in welchem Model die Eingabe verwaltet werden soll. Und in der Zeile darunter wird dieses Model wieder ausgelesen. So kann man beim Eintippen des Namens beobachten, wie dieser zeitgleich unten stets aktualisiert wird.

```
<html>
  <head>
     <script src="js/angular.
js"></script>
  </head>
  <body ng-app>
     <p>Ein einfacher Angular
Ausdruck: {{ 1 + 1 }}</p>
  </body>
</html>
```

Listing 1: Minimale AngularJS-Anwendung

```
Dein Name: <input type="text"
ng-model="name">
<p>Hello {{name}}!</p>
```

Listing 2: Hello Angular

```
HTML:
<div ng-controller="DatumCtrl">
  <p>{{datum}}</p>
</div>

JAVASCRIPT:
var DatumCtrl = function($scope)
{
  $scope.datum = new Date();
};
```

Listing 3: Einfacher Controller

Controller, Direktiven, Filter und Services

Für ernsthaftere Anwendungen genügt es natürlich nicht, das Model nur als Zwischenspeicher zu nutzen. AngularJS bietet noch weitere Konstrukte: „Controller" dienen zur Verwaltung und Manipulation von Models und „Direktiven" erweitern HTML um eigene Elemente, die eine gewisse Logik implementieren. „Filter" formatieren die Ausgabe und eignen sich außerdem zum Sortieren und Filtern von Datenstrukturen. „Services" bilden wiederverwendbare Module für häufig benötigte Funktionen innerhalb der Anwendung oder über mehrere Anwendungen hinweg. Sie lassen sich beispielsweise in einem Controller, einer Direktive oder einem anderen Service via Dependency Injection injizieren.

Alles unter Kontrolle

Listing 3 zeigt die Verwendung eines einfachen Controllers. In einem umspannenden HTML-Element (hier „ ") wird mittels

```
HTML:
<div ng-controller="SqrCtrl">
  Eine Zahl: <input type="text" ng-
model="zahl"></input>
  Quadrat: {{getSqr()}}
</div>

JAVASCRIPT:
var SqrCtrl = function($scope) {
  $scope.zahl = 1;

  $scope.getSqr = function() {
     return $scope.zahl * $scope.zahl;
  };
};
```

Listing 4: Berechnung von Quadraten

„ng-controller" der Gültigkeitsbereich des Controllers abgesteckt. Jeder Controller hat einen Scope, in dem seine Models sind; im Beispiel ist das der Datumswert, der beim Laden der Seite mit dem aktuellen Zeitstempel vorbelegt wird. In HTML wird das Model einfach via Ausdruck ausgewertet und ausgegeben.

Controller können auch Funktionen bereitstellen, *Listing 4* zeigt einen Controller mit einer Funktion zum Quadrieren von Werten. Neben der Definition von Controllern als „function" bietet AngularJS auch eine Factory-Methode zum Erzeugen eines Controllers *(siehe Listing 5)*.

Zudem ist es möglich, die Gültigkeitsbereiche von Controllern zu verschachteln, sodass ein Controller seinerseits einen oder mehrere „Kind"-Controller enthält. Jeder Controller hat dabei seinen Scope; auf oberster Ebene stellt die AngularJS-Anwendung den Root-Scope zur Verfügung. So entsteht ein Baum aus Scopes, bei dem auch Möglichkeiten zum Zugreifen auf den Parent-Scope bereitgestellt werden (etwa über „$parent"). Beim Aufbau einer Anwendung ist jedoch darauf zu achten, mit diesen Möglichkeiten sparsam umzugehen, damit die Übersichtlichkeit erhalten bleibt.

```
JAVASCRIPT:
var app = angular.module("MyApp", []);
app.controller("MyCtrl", function($scope) {
  $scope.name = "Tom";
});

HTML:
<body ng-app="MyApp">
...
<div ng-controller="MyCtrl">
...
```

Listing 5: App und Controller

```
<label>
  <input type="checkbox" ng-model="checked"/>
  Ich stimme zu!
</label>
<button ng-disabled="!checked">Jetzt bestellen</button>
```

Listing 6: „ngDisabled"-Direktive

HTML erweitern

Jeder, der schon einmal mit jQuery gearbeitet hat, kennt das: In HTML ist nicht ersichtlich, warum an einer Stelle eine gewisse Funktionalität stattfindet, beispielsweise ein „Slider" – nur in JavaScript kann man sehen, dass hier jQuery mittels Selektoren manipulativ eingreift. Viel schöner wäre es, wenn direkt in HTML einfach ein „<slider>...</slider>" stünde. Genau dies bieten die Direktiven in AngularJS. Sie können dabei sowohl als HTML-Tags als auch als Attribute in Tags implementiert sein. AngularJS liefert bereits ein großes Sammelsurium an Direktiven, deren Bezeichner mit „ng" beginnen (ngClick, ngShow, ngDisabled etc.), siehe *Listing 6*; der Button wird erst aktiviert, wenn die Checkbox ausgewählt ist. Außerdem implementieren inzwischen unzählige Drittanbieter Direktiven für jeden erdenklichen Anwendungsfall.

Eine besonders nützliche Direktive ist „ngRepeat" zum Iterieren über Listen. *Listing 7* zeigt einen Controller, der ein Array aus Entwicklern vorhält, und eine Ausgabe desselben mittels „ng-repeat". Die Angabe „dev in devs" bewirkt, dass für jeden Eintrag in „devs" ein Schleifendurchlauf erfolgt und der aktuelle Eintrag in „dev" landet. Ausgegeben wird also eine Aufzählungsliste mit den Namen der Entwickler und deren Spezialgebieten in Klammern.

```
JAVASCRIPT:
app.controller("DevsCtrl", function($scope) {
          $scope.devs = [
              { name: "Lisa", speciality: "HTML/JS" },
              { name: "Kim", speciality: ".Net" },
              { name: "Berta", speciality: "Java" },
              { name: "Xaver", speciality: "AngularJS" }
          ];
});

HTML:
<div ng-controller="DevsCtrl">
  <ul>
      <li ng-repeat="dev in devs">
          {{dev.name}} ({{dev.speciality}})
      </li>
  </ul>
</div>
```

Listing 7: Über Entwickler iterieren

Eine eigene Direktive ist auch kein Hexenwerk. Im Beispiel implementieren wir eine „Superheld"-Direktive, in Anlehnung an Googles AngularJS-Slogan „A Superheroic JavaScript MVW Framework". Sie stellt ein HTML-Tag „<superheld>" bereit, dessen Inhalt das Wort „SUPER-" vorangestellt wird und an dessen Ende drei Ausrufezeichen gesetzt werden, also eine Art Decorator. Auch für Direktiven bietet AngularJS eine Factory-Funktion *(siehe Listing 8)*.

In diesem Fall wird gerendert, also „SUPER- toll !!!". Die Angabe von „restrict" schränkt die Direktive auf bestimmte Anwendungsbereiche ein, in unserem Fall „E" auf Element. Das heißt, man kann diese Direktive nur als HTML-Element verwenden. Weitere Restriktionen sind „A" für Tag-Attribut, „C" für CSS-Klassen und „M" für die Ausgabe in HTML-Kommentaren. Auch Kombinationen sind erlaubt, etwa die Angabe von „EA", dann könnte man diese Direktive als HTML-Element verwenden, aber auch als Attribut in beispielsweise einem . Die Angabe von „template" bestimmt, was gerendert werden soll, und „transclude" bewirkt, dass Kind-Elemente im Template platziert werden. Zahlreiche weitere Optionen und Möglichkeiten stehen zur Verfügung.

```
JAVASCRIPT:
app.directive('superheld', function() {
  return {
      restrict: 'E',
      template:
          '<div>SUPER-<span ng-transclude></span>!!!</div>',
      transclude: true
  };
});

HTML:
<superheld>
  toll
</superheld>
```

Listing 8: Die Direktive „Superheld"

```
<div ng-controller="DatumCtrl">
  <p>Datum: {{datum | date:"dd.MM.yyyy"}}</p>
</div>
```

Listing 9: Der „date"-Filter

```
<div ng-controller="DevsCtrl" ng-init="reverse='false'">
  <a href="" ng-click="reverse=!reverse">asc/dsc</a>
  <ul>
      <li ng-repeat="dev in devs | orderBy:'name':reverse">
          {{dev.name}} ({{dev.speciality}})
      </li>
  </ul>
</div>
```

Listing 10: Sortierung

```
app.factory("OSService", function() {
  var os = [ "Linux", "MacOS", "Windows" ];
  return {
      all : function() {
          return os;
      },
      first : function() {
          return os[0];
      }
  };
});

app.controller("OSCtrl", function($scope, OSService) {
  $scope.oss = OSService.all();
});
```

Listing 11: Ein einfacher Service

Direktiven können beliebig komplex werden, aber wie im Beispiel zu sehen ist, ergibt sich für den Anwender (also den HTML-Entwickler) ein einfacher und lesbarer Code. Es ist sinnvoll, zusammengehörige Direktiven in Module zu packen und so Sets aus wiederverwendbaren Komponenten zu schaffen.

Formatiert und gefiltert

Auch was die Filter betrifft, stellt uns Google schon eine Sammlung davon bereit. Beispiele sind „currency" (Formatieren von Geldbeträgen), „date" (Formatieren von Datum-/Zeitangaben), „json" (JSON formatiert ausgeben), „upper-/lowercase" oder „orderBy". In *Listing 9* wird der „date"-Filter für eine wunschgemäße Ausgabe verwendet (zum Beispiel „03.07.2014", vergleiche *Listing 3*, dort wird „2014-07-03T06:20:37.758Z" angezeigt). Es ist zu sehen, dass Filter einfach „Pipe"-Zeichen (analog zur Pipe-Verkettung in Linux/UNIX) an den zu manipulierenden Wert anhängen.

Mit „orderBy" wird das „Entwickler"-Array aus *Listing 7* sortierbar. Dazu wird die Liste um einen Link erweitert, der die Reihenfolge alphabetisch auf-/absteigend wechselt – das Model „reverse" wird einfach invertiert und oben zunächst auf „false" gesetzt. Der „orderBy"-Filter in der „ngRepeat"-Direktive erwartet zwei Parameter, die erstens angeben, welches Feld für die Sortierung dienen soll, und zweitens, ob auf- oder absteigend zu sortieren ist *(siehe Listing 10)*. Am Anfang wird die Liste also „Berta", „Kim", „Lisa" und dann „Xaver" nennen, nach Klick auf „asc/dsc" dreht sich die Reihenfolge um.

Auch das Implementieren eigener Filter ist möglich; mit „app.filter" definiert man eine Funktion, die den Input (also das, was vor dem Pipe-Zeichen steht) und die Parameter injiziert bekommt. Filter lassen sich auch verketten, sodass mehrere Manipulationen hintereinandergeschaltet werden.

Stets zu Diensten

Services runden die Angebotspalette von AngularJS ab. Auch hier sind schon einige mitgeliefert, etwa „$http" (siehe unten), „$locale" (Lokalisierung), „$log" (für Logging-Ausgaben) oder „$window" als Wrapper für das JavaScript-Objekt „document.window". Dazu ein Beispiel, wie man einen eigenen Service implementieren kann. Der „OSService" aus *Listing 11* ist ein trivialer Service, der zwei Methoden anbietet: Eine liefert ein komplettes Array der Betriebssysteme („all") und eine zweite das erste Betriebssystem („first"). Das Array mit den Betriebssystemen ist dabei gekapselt und für Verwender des Service nicht von Belang. Darunter ist ein Controller zu sehen, der sich den Service injiziert und eine Methode daraus aufruft.

Der „$http2"-Service, mit dem asynchron HTTP-Requests gestartet werden können, leistet ein bisschen mehr. Er kennt entsprechend Methoden wie „get", „put", „post", „delete", „head" und „jsonp". *Listing 12* zeigt ein Beispiel mit „$http.get()". Im HTML-Anteil ist nicht ersichtlich, dass die anzuzeigenden Bücher asynchron via HTTP nachgeladen werden, es wird einfach mittels „ngrepeat" über das „buecher"-Array aus dem „BuecherCtrl" iteriert. Erst der Blick in die

Controller-Implementierung legt dies offen: „$http.get“ lädt eine (in diesem Beispiel lokal liegende) „buecher.json“. Die „get“-Methode liefert einen sogenannten „Promise“ zurück, der entweder erfolgreich sein kann oder einen Fehler liefert. Die Auswertung erfolgt einfach durch Anhängen der „success“- beziehungsweise „error“-Methoden an den „get“-Aufruf. So wird im Erfolgsfall „buecher“ mit den geladenen Daten belegt, im Fehlerfall könnte ein Logging erfolgen.

Jetzt könnte man auf die Idee kommen, sich mittels „$http2“ seine REST-Anbindung ans Backend zu implementieren, aber das geht viel einfacher. AngularJS bietet das „ngResource“-Modul, um REST-Consumer im Handumdrehen zu erstellen. Das Modul enthält dazu einen Service namens „$resource“. Wenngleich sich sehr viel parametrisieren lässt, beschränken wir uns auf ein einfaches Beispiel. In *Listing 13* ist alles enthalten, um eine REST-Anbindung zu realisieren. Es wird ein Service „Buecher“ erstellt, der sich per REST an die URL „http://apidomain.xy/buecher“ anbindet und dabei noch die „buchId“ auf „id“ mappt. *Tabelle 1* gibt einen Überblick darüber, welche Möglichkeiten sich für uns mit der Buecher-REST-Anbindung ergeben. In *Listing 14* sind beispielhaft weitere Aufrufe dargestellt und kommentiert.

Wo bin ich?

Zum Schluss stellt sich noch die Frage, wie eine Navigation innerhalb der AngularJS-Anwendung realisiert werden kann. Wir sind schließlich in einer SPA und haben entsprechend nur eine Seite. Die Lösung bietet das „ngRoute“-Modul. Beim Initialisieren der Angular-App kann ein „config“-Block angegeben werden, in dem sich das Routing einstellen lässt. Dies ist eine einfache Verkettung von Bedingungen (URL-Bestandteil) und der Angabe des jeweiligen Templates. AngularJS nennt die eingebundenen Seiten „Partials“, wobei das aktuelle an der mit „ng-view“ markierten Stelle eingebunden wird. Dadurch gliedert sich das HTML automatisch in kleine Dateien, die je ein Partial realisieren *(siehe Listing 15)*.

Arbeiten in der Praxis

Ohne Tests geht es nicht. AngularJS-Anwendungen lassen sich sehr gut testen, beispielsweise mit Jasmine. Dort können etwa für einen Controller künstlich ein Scope erzeugt und darin dann die Tests aufrufen werden. Wer von Java her gewohnt ist, mit vernünftigen Tools

```
HTML:
<div ng-controller="BuecherCtrl">
  <ul>
      <li ng-repeat="buch in buecher">{{buch.titel}}</li>
  </ul>
</div>

JAVASCRIPT:
app.controller("BuecherCtrl", function($scope, $http) {
  $http.get('exampledata/buecher.json').
  success(function(data, status, headers, config) {
      $scope.buecher = data;
  }).
  error(function(data, status, headers, config) {
      // log error
  });
});
```

Listing 12: Daten laden mit „$http.get()“

```
app.factory('Buecher', ['$resource', function($resource){
  return $resource('http://apidomain.xy/buecher/:buchId',
      {buchId: '@id'});
}]);
```

Listing 13: REST-Anbindung an „Bücher-Api“

```
// alle Bücher laden:
var buecher = Buecher.query();

// Ein Buch laden und direkt im Callback arbeiten:
Buecher.get({id: 123}, function(buch) {
  buch.autor = "Bugs Bunny";
  // non-GET Methoden werden auf die Instanzen gemapped:
  buch.$save();
});

// neues Buch anlegen:
var buch = new Buecher();
buch.autor = "Speedy Gonzales";
buch.titel = "Fiesta Fiasco";
buch.$save();
// Erzeugt POST: /buecher mit
// {autor:'Speedy Gonzales', titel:'Fiesta Fiasco'}
```

Listing 14: Arbeiten mit Büchern

```
var app = angular.module("JLApp", [ngRoute]).
  config(function($routeProvider, $locationProvider) {
      $locationProvider.hashPrefix('!');
      $routeProvider.
          when("/buecher",
              { templateUrl: "partials/buecherlist.html" }).
          when("/buecher/:id",
              { templateUrl: "partials/buchdetails.html",
              controller: "ShowCtrl" }).
          otherwise( { redirectTo: "/buecher" });
});

// in index.html: Rahmen mit Menü, Footer, ...
// ...und Partial anzeigen:
<div ng-view></div>
```

Listing 15: Routing innerhalb der Angular-App

Call	Methode	URL (ohne Domain)	Return
Buecher.query()	GET	/buecher	JSON Array
Buecher.get({id: 47})	GET	/buecher/47	Single JSON
Buecher.save({}, buch)	POST	/buecher mit post data "buch"	Single JSON
Buecher.save({id: 48}, buch)	POST	/buecher/48 mit post data "buch"	Single JSON
Buecher.remove({id: 47})	DELETE	/buecher/47	Single JSON
Buecher.delete({id: 47})	DELETE	/buecher/47	Single JSON

Tabelle 1: Überblick Bücher-Service

für das Generieren, Managen, Bauen und Testen zu arbeiten, wird auch in der JavaScript-Welt nicht allein gelassen. Yeoman bietet einen ganzen Stack aus Tools, die sich leicht erlernen lassen *[5]*. Wer nun selbst eine erste kleine Angular-Anwendung erstellen möchte, dem seien die diversen Seed-Applikationen *[6, 7]* ans Herz gelegt, mit denen man blitzschnell ein Code-Gerüst erzeugt, das einen guten Ausgangspunkt zum Loslegen bietet.

Quellen und Links

[1] https://angularjs.org
[2] https://developers.google.com/speed/libraries/devguide#angularjs
[3] http://joachim.weinbrenner.name/vortrag/jl14/angularjs
[4] https://github.com/weinbrenner/angularjs-superheld
[5] http://joachim.weinbrenner.name/2013/10/15/aufbau-einer-enterprise-angularjs-anwendung-mit-yeoman-in-5-minuten
[6] https://github.com/angular/angular-seed
[7] https://github.com/thedigitalself/angular-sprout

Joachim Weinbrenner
jw@jsolutions.de

Joachim Weinbrenner (Inhaber von jsolutions.de) ist Software-Architekt und Entwickler im Java-EE-Umfeld. Er unterstützt JEE-Projekte mit Schwerpunkt auf Web-basierten, verteilten Anwendungen. Darüber hinaus vermittelt er seine Kenntnisse in verschiedenen Schulungen.

http://ja.ijug.eu/15/1/10

Login und Benutzerverwaltung: Abgesichert und doch frei verfügbar

Martin Ley, tarent solutions GmbH

Bei der Anwendungsentwicklung – insbesondere bei Web-Anwendungen – gelangt man irgendwann an den Punkt, an dem eine Benutzerverwaltung ansteht. Diese reicht von der Anmeldung der Benutzer an der Anwendung über die Speicherung der E-Mail-Adresse für einen Newsletter bis hin zur Verwaltung von sensiblen Daten wie Namen, Adressen und Kreditkartendaten. Zur Absicherung dieser Daten gibt es Standards wie OAuth2. Um die Daten leicht zu importieren oder anderen Anwendungen bereitzustellen, bieten sich REST-Schnittstellen und der SCIMv2-Standard für Benutzerdaten an.

Möchte man eine Benutzerverwaltung selbst implementieren, gehen je nach Umfang mindestens einige Tage, wenn nicht sogar Wochen für die Entwicklung dieser Komponente ins Land. Dieser Aufwand reduziert sich deutlich, wenn man das Rad nicht neu erfindet, sondern eine fertige Benutzerverwaltung wie OSIAM einsetzt.

Open Source, offene Standards

OSIAM steht für „Open Source Identity and Access Management" und ist ein leichtgewichtiges, skalierbares Identity-Management mit dem Fokus auf Sicherheit und einfacher Integration. Die Java-Software steht unter der MIT-Lizenz und lässt sich damit auch in kommerziellen und Closed-Source-Produkten einsetzen. OSIAM setzt außerdem auf etablierte und offene Standards, um maximale Interoperabilität und Flexibilität zu gewährleisten und die Abhängigkeit von einem Hersteller zu vermeiden.

So wird zur Authentisierung und Autorisierung das OAuth2-Protokoll verwendet, das zum Beispiel auch Amazon, Google, Facebook und viele andere einsetzen. Die Daten speichert OSIAM nach dem System-for-Cross-domain-Identity-Management-Standard (SCIMv2) und stellt diese über REST-Schnittstellen zur Verfügung. Das SCIM-Datenmodell genügt für die meisten Anwendungsfälle bereits und konzentriert sich auf die Verwaltung von Benutzern und Gruppen mit gängigen Attributen. Reichen diese nicht aus, lässt OSIAM es über die SCIM Extensions zu, Informationen gemäß den eigenen Bedürfnisse zu verwalten.

OAuth2

OSIAM unterstützt mehrere Authorization-Flows des OAuth2-Standards, die in Abhängigkeit der Vertrauensbeziehung zwischen Benutzer, Anwendung und der OSIAM-

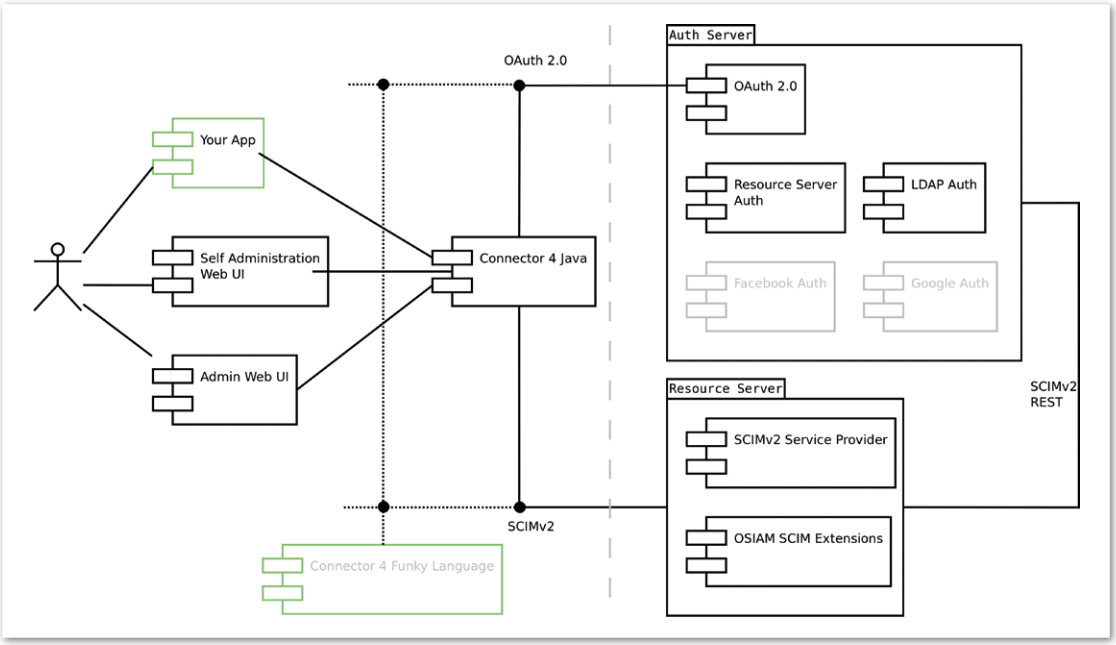


Abbildung 1: OSIAM-Architektur

Installation genutzt werden können. Mit dem Authorization-Code-Grant prüft die Anwendung, ob der Benutzer angemeldet werden muss. Falls ja, wird der Browser des Benutzers auf die Login-Seite des OAuth-Providers weitergeleitet. Dort meldet sich der Benutzer mit seinem Benutzernamen und Passwort an und gibt den Zugriff durch die Anwendung auf seine Daten frei. Danach wird der Browser auf den Redirect-URI der Anwendung zusammen mit dem Auth-Code kommuniziert.

Mit dem Auth-Code kann die Anwendung dann ein Access-Token vom Authentication-Server beziehen. Mit dem Access-Token wiederum ist die Anwendung imstande, auf die Daten des Resource-Servers zuzugreifen. Der Vorteil bei diesem Flow ist, dass der Benutzer sein Passwort nie der Anwendung direkt gibt. Vertraut der Benutzer der Anwendung, kann der Resource-Owner-Password-Credentials-Grant eingesetzt werden, bei dem der Anwender seine Zugangsdaten direkt der Anwendung zur Anmeldung übermittelt. Zudem kann sich die Anwendung selbst mit dem Client-Credentials-Grant ohne den Kontext eines bestimmten Benutzers anmelden.

OSIAM kann auch als Service-Provider für Facebook Connect dienen. Facebook Connect basiert ebenfalls auf OAuth2, verhält sich jedoch geringfügig anders. Damit ist es möglich, Anwendungen, die eine Facebook-Anmeldung verwenden, mit wenig Aufwand auf eine Anmeldung gegen OSIAM umzustellen.

SCIMv2

SCIM wurde entwickelt, um die Benutzerverwaltung in Cloud-basierten Systemen einfacher zu machen und zu vereinheitlichen. Im SCIM-Standard sind dazu die Basisdaten-Typen „User" und „Group" definiert. SCIM schreibt als Schnittstelle zur Manipulation der Daten ein REST-API vor, das starken Gebrauch von den verschiedenen HTTP-Methoden macht. Über diese Schnittstelle können Objekte angelegt, gelesen, ersetzt, gelöscht, aktualisiert und gesucht werden. Darüber hinaus gibt es Möglichkeiten zur Massenverarbeitung ganzer Datensätze.

Sicherheit und Flexibilität

OSIAM wurde mit dem Ziel entwickelt, eine sichere Identitätsverwaltung zu sein. Erreicht wird dies durch einen sicheren Entwicklungsprozess. Dabei ist das CI-System dafür konfiguriert, potenzielle Sicherheitslücken zu erkennen und den Build in einem solchen Fall fehlschlagen zu lassen. Weitere Maßnahmen sind der Einsatz starker Kryptographie, die Minimierung der Angriffsflächen, das Least-Privilege-Prinzip, die sichere Default-Konfiguration und viele weitere. Jede Änderung durchläuft ebenfalls einen Review-Prozess durch mindestens einen zweiten Entwickler. Durch die offene Lizenzierung kann auch jeder selbst den Code einsehen und sich ein Bild davon machen.

Ein weiterer Schwerpunkt der Entwicklung ist die Flexibilität. Der Aufwand, OSIAM in seine Software zu integrieren, sollte möglichst klein sein. Dieses Ziel wird erreicht durch eine geringe Basisfunktionalität, die einfach zu erfassen und einzusetzen ist, sowie durch den Einsatz der erwähnten Standards. OSIAM erfordert insgesamt nur geringe Installationsaufwände und ist ebenfalls ressourcenschonend zu betreiben.

Architektur

Der OSIAM-Server ist aufgeteilt in zwei Komponenten: den Authentication- und den Resource-Server *(siehe Abbildung 1)*. Der Authentication-Server dient zur Identifikation und Authentisierung der Benutzer nach dem OAuth2-Standard. Ebenfalls hat der Authentication-Server eine Schnittstelle zur Validierung der Access-Tokens. Diese Validierung wird unter anderem vom Resource-Server verwendet. Er implementiert die SCIM-Endpunkte, über die die Benutzer- und Gruppen-Daten verwaltet werden. Der Austausch der Daten geschieht über JSON-Objekte.

Die Anwendungen, die OSIAM benutzen, werden aus OSIAM-Sicht „Clients" genannt. Jeder Client muss in OSIAM registriert sein und hat eigene Anmeldedaten sowie weitere Einstellungen zur Gültigkeitsdauer der Anmeldung und des Client-Redirect-URI. Dadurch kann OSIAM als Stand-Alone-Anwendung für einen Client sowie als zentrale Benutzerverwaltung für mehrere Anwendungen genutzt werden. Es ist möglich, die Komponenten sowohl auf einem Server mit einer gemeinsamen Datenbank als auch auf verschiedenen Servern und mit unterschiedlichen Datenbanken zu betreiben. Der getrennte Betrieb von Authentication- und Resource-Server eignet sich vor allem für Systeme, bei denen die Performance besonders relevant ist oder der Resource-Server besonders abgesichert werden soll. Die Datenbank wird mit Hibernate angesprochen, wodurch eine große Menge unterschiedlicher Datenbanken genutzt werden kann. Aktuell wird OSIAM mit PostgreSQL und mit H2 betrieben.

Entwicklungsinstanz von OSIAM mit Docker

Um direkt in die Materie einzusteigen und gegen OSIAM zu entwickeln, gibt es ein Docker-Image mit einem kompletten, lauffähigen System. OSIAM selbst läuft in einem Tomcat 7 und verwendet eine PostgreSQL-Datenbank. Zum jetzigen Zeitpunkt ist das Docker-Image noch nicht in der Docker-Hub-Registry aufgenommen, woran jedoch schon gearbeitet wird.

Das Docker-Image lässt sich auch mit wenigen Handgriffen selbst bauen. Auf einem Linux-System werden dazu Java7, Maven3, Git und Docker benötigt. Die Docker-Installation sollte so konfiguriert

```
$ git clone https://github.com/osiam/docker-image.git
$ cd docker-image
$ mvn clean initialize docker:package
$ docker tag osiam-docker_osiam-dev osiam/dev
$ sudo docker run -it -p 8080:8080 -p 5432:5432 osiam-docker_osiam-
dev
```

Listing 1

```
<dependency>
  <groupId>org.osiam</groupId>
  <artifactId>connector4java</artifactId>
  <version>1.0</version>
</dependency>
```

Listing 2

sein, dass der Docker-Dienst auf dem Unix-Socker „/var/run/docker.sock“ und dem TCP-Port 4243 lauscht. Dazu muss gegebenenfalls die Docker-Konfigurationsdatei „/etc/defaults/docker“ angepasst und der Dienst danach mit „DOCKER_OPTS=»-H tcp://127.0.0.1:4243 -H unix:///var/run/docker.sock»“ neu gestartet werden. Danach werden das OSIAM Docker Git geklont und das Docker-Image gebaut *(siehe Listing 1)*.

Docker bindet den Port des Tomcat an Port 8080 und den der PostgreSQL an Port 5432 des Hosts. Falls auf den Ports andere Dienste laufen, sollten diese während des Testens gestoppt werden.

OAuth-Login mit dem Java-Connector

Um von Java auf OSIAM zuzugreifen, kann der vorgefertigte OSIAM-Java-Connector verwendet werden. Dafür wird ist die Maven-Dependency vom „connector4java“ erforderlich *(siehe Listing 2)*. Die aktuelle Version steht unter *„http://search.maven.org/#search|ga|1|osiam“*.

Das Beispiel-Projekt benutzt darüber hinaus noch Jetty und Jersey, um einen REST-Service zu implementieren. Als Anwendungsfall wird der OAuth2-Login mit Authorization-Code-Grant gezeigt. Die zentrale Klasse des OSIAM-Java-Connector ist „OSIAMConnector“. Die Instanziierung erfolgt über ihren Builder *(siehe Listing 3)*.

Der Endpoint legt fest, wo der OSIAM-Service zu erreichen ist. Der Client-Redirect-URI gibt einen Endpunkt an, den unsere Anwendung anbietet und auf den OSIAM nach erfolgreichem Login zurückleitet. Client-Id und Client-Secret sind die Zugangsdaten unserer Anwendung zum Zugriff auf OSIAM. Die hier verwendeten Daten sind die Beispiel-Daten, die auch im OSIAM-Docker-Image vorliegen. Als Nächstes wird die „Einstiegsseite“ der Anwendung implementiert *(siehe Listing 4)*.

Hier wird geprüft, ob es ein Access-Token in der HTTP-Session gibt. Falls nicht, wird der Link zur Login-Seite gezeigt, andernfalls ein Link zur Ressource „/secured“. Den URI zur Login-Seite stellt der OSIAM-Connector zur Verfügung. Der Scope gibt an, welche HTTP-Methoden (GET, POST, PUT, PATCH, DELETE) auf dem Resource-Server verwendet werden sollen. So kann mit Scope.GET nur ein lesender Zugriff erfolgen. *Listing 5* zeigt die Implementierung des Redirect-URI-Endpunkts. Der übergebene Auth-Code liefert ein Access-Token, das dann in der

```
private OsiamConnector getOsiamConnector() {
  return new OsiamConnector.Builder()
      .setEndpoint(“http://localhost:8080”)
      .setClientRedirectUri(“http://localhost:5000/oauth2”)
      .setClientId(“example-client”)
      .setClientSecret(“secret”)
      .build();
}
```

Listing 3

```
@GET
@Produces(MediaType.TEXT_HTML)
@Path(“”)
public Response login(@Context HttpServletRequest req) {
  AccessToken accessToken = (AccessToken)
      req.getSession().getAttribute(“accessToken”);

  if (accessToken == null) {

      OsiamConnector oc = getOsiamConnector();
      URI uri = oc.getAuthorizationUri(Scope.ALL);
      return “<html><body><a href=‘“ + uri.toString() +
      “’>Login</a></body></html>”;

  } else {
      return “<html><body><a href=>/secured>>secured”+
      “</a></body></html>”;
  }
}
```

Listing 4

```
@GET
@Path(“oauth2”)
public Response oauth2(@Context HttpServletRequest req,
@QueryParam(“code”) String authCode) throws URISyntaxException{
  OsiamConnector oc = getOsiamConnector();
  AccessToken at = oc.retrieveAccessToken(authCode);
  req.getSession().setAttribute(“accessToken”, at);
  return Response.temporaryRedirect(
      new URI(“http://localhost:5000/”)).build();
}
```

Listing 5

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path(“secured”)
public User getSelf(@Context HttpServletRequest req) {
  AccessToken accessToken = (AccessToken)
  req.getSession().getAttribute(“accessToken”);

  return getOsiamConnector().
  getCurrentUser(accessToken);
}
```

Listing 6

HTTP-Session gespeichert ist. Dann wird der Browser wieder auf die Hauptseite umgeleitet *(siehe Listing 6)*.

Mit dem Access-Token aus der HTTP-Session wird der eingeloggte Benutzer abgefragt und zurückgegeben. Im Browser sieht man nun die Daten des Benutzers im JSON-Format. Ist das Access-Token „null" oder ungültig, kommt es hier zu einer entsprechenden Exception. Der komplette Code dieses Beispiels ist auf GitHub verfügbar *(siehe „https://github.com/mley/osiam-demo-ja")*.

Das Benutzerschema mit SCIM-Extensions erweitern

Falls neben den Standard-SCIM-Benutzer-Feldern noch weitere Felder erforderlich sind, können diese in Form von SCIM-2.0-kompatiblen Extensions abgelegt werden. Hierzu muss ein eindeutiger Extension-Name (etwa ein kompletter URL) in der Datenbank des Resource-Servers abgelegt sein. Daneben ist noch zu spezifizieren, welche Felder die Extension besitzt und von welchem Typ diese sind. Als Typen werden aktuell String, BigDecimal, BigInteger, DateTime, Boolean, URI und Binary unterstützt. Mithilfe des SCIM-Schemas lässt sich nun sehr einfach eine Extension erzeugen und dem Benutzer übergeben *(siehe Listing 7)*. Ebenso einfach ist es nun, mit „osiamConnector.createUser(user, accessToken);" den erzeugten Benutzer in der Datenbank zu speichern.

Benutzer und Gruppen verwalten

Die Klasse „OSIAMConnector" bietet für die Verwaltung von Benutzern und Gruppen eine Reihe intuitiver Methoden nach dem CRUD-Schema. Das Ändern einzelner Felder von Entitäten erfolgt mithilfe von Update-Objekten, die nur die geänderten Daten enthalten *(siehe Listing 8)*.

Selbstverwaltung und Administrations-Oberfläche

Im Rahmen verschiedener Projekte sind bereits die ersten Erweiterungen entstanden. Mit der Selbstverwaltung können sich Benutzer selbst neue Accounts anlegen und erhalten darauf eine E-Mail mit einem Aktivierungslink zugeschickt. Für die Registrierung können die auszufüllenden Felder frei konfiguriert werden. Auch die Nutzung von SCIM-Extensions ist möglich. Bei Verlust des Passworts kann der Benutzer sein Passwort selbst zurücksetzen. Mit dem Administrations-Add-on können Benutzer und Gruppen einfach über den Browser verwaltet werden. Beide Erweiterungen sind erst nach dem Long-Term-Support-Release von OSIAM 1.0 entstanden und nicht im oben erwähnten Docker-Image vorhanden.

Ausblick

Für die Zukunft sind Erweiterungen geplant, um OSIAM noch sicherer und leistungsfähiger zu machen, wie zum Beispiel eine Mehr-Faktor-Authentisierung und die Föderation der Benutzerdaten mit Dritt-Systemen. Ebenfalls soll OSIAM in noch mehr Paketierungen verfügbar sein. Zurzeit gibt es ein Debian-Paket und es wird angestrebt, OSIAM in die Paket-Repositories verschiedener Linux-Distributionen zu bringen.

Weiterführende Links

1. http://osiam.org
2. https://github.com/osiam
3. https://blog.tarent.de/tag/osiam

```
Extension extension = new Extension.
            Builder("urn:scim:schemas:osiam:2.0:Example")
            .setField("age", BigInteger.valueOf(28))
            .setField("newsletter", false)
            .build();

User user = new User.Builder("userName")
            .setPassword("password")
            .addExtension(extension).build();
```

Listing 7

```
Email deleteEmail = new Email.Builder()
            .setValue("user@example.com")
            .setType(Email.Type.HOME).build();
Email oldEmail = new Email.Builder()
            .setValue("myEmail@example.com")
            .setType(new Email.Type("private")).build();
Email newEmail = new Email.Builder(oldEmail)
            .setPrimary(true).build();

UpdateUser updateUser = new UpdateUser.Builder()
            .deleteEmail(deleteEmail)
            .updateEmail(oldEmail, newEmail)
            .addPhoto(photo)
            .deleteAddresses().build();

osiamConnector.updateUser("userId", updateUser, accessToken);
```

Listing 8

Martin Ley
m.ley@tarent.de

Martin Ley arbeitet seit 2006 als Senior-Entwickler bei der tarent solutions Gmbh und verwendet dort OSIAM bei seinen Projekten.

http://ja.ijug.eu/15/1/11

Software-Erosion gezielt vermeiden

Kai Spichale, adesso AG

Anzeichen für Software-Erosion sind Komplexitätszunahme und schleichender Architektur-Zerfall, hervorgerufen durch Änderungen und Erweiterungen der Codebasis. Software-Erosion äußert sich nicht selten auch durch eine Lücke zwischen geplanter Architektur und tatsächlicher Implementierung. Mit gezielten Gegenmaßnahmen wie kontinuierlicher Architektur-Validierung, Software-Metriken und Coding-Rules können diese Probleme vermieden werden. Wichtige Ansätze und Werkzeuge dazu sind in diesem Artikel vorgestellt.

Die Organisation des Quellcodes gehört zu den klassischen Aufgaben der Software-Entwicklung. Bereits 1972 schlug David Parnas *[1]* das Geheimnis-Prinzip und die Daten-Kapselung als Modularisierungskriterien vor, um Codebasen flexibler und leichter verständlich zu machen. Diese Ideen wurden zu wichtigen Konzepten der objektorientierten Programmierung. Eine andere Denkschule der Quellcode-Organisation stammt von Larry Constantine *[2]*, der mit Kopplung und Kohäsion alternative Entwürfe objektiv bewerten wollte. Er verstand unter „Kopplung" den Grad der intermodularen Abhängigkeit und unter „Kohäsion" den Grad der intramodularen funktionalen Beziehungen. Lose Kopplung und hohe Kohäsion definierte er als Kennzeichen guter Strukturierung. Kohäsion ist jedoch schwer messbar.

Die „Lack of Cohesion of Methods"-Metriken (LCOM) *[3]* messen die Kohäsion nur indirekt, indem sie den Kohäsionsmangel errechnen. Es gibt verschiedene Varianten der LCOM-Metriken, für die sich leicht Gegenbeispiele finden lassen, in denen der errechnete Kohäsionsmangel unbegründet ist. Daher konzentrieren wir uns auf andere Aspekte, um Software-Erosion pragmatisch zu bekämpfen. Nichtsdestotrotz ist Kohäsion die Basis des Single-Responsibility-Prinzips. Dieses Grundprinzip objektorientierter Programmierung wird später noch einmal aufgegriffen.

Im Vergleich zur Kohäsion ist Kopplung vergleichsweise einfach zu bestimmen. Kopplung, etwa in Form von „Message Passing Coupling", ist in einem objektorientierten System nicht nur unvermeidbar, sondern sogar als Indiz für gute Strukturierung erwünscht. Denn eine Klasse mit hohem Fan-in wird häufig wiederverwendet. Doch nicht jede Form von Kopplung ist wünschenswert und schon gar nicht zwischen beliebigen Elementen.

Vermeidbare Formen starker Kopplung

Eine starke Form von Kopplung ist Vererbung, denn eine erbende Klasse ist im hohen Maße von der Implementierung ihrer Basisklasse abhängig. Umgekehrt sollte eine Basisklasse nicht ohne Berücksichtigung ihrer Subklassen verändert werden.

Während Polymorphie im Allgemeinen sehr elegante Lösungen erlaubt, kann Vererbung auch dazu verleiten, undurchschaubare Konstrukte zu bauen. Aufgrund der hohen Komplexität, die Vererbung mit sich bringen kann, und eines statistischen Zusammenhangs zwischen Fehlerhäufigkeit und der Tiefe von Vererbungsbäumen (engl. „depth

```
String outputDir = user .getClient()
                        .getProperties()
                        .getReportDirectory()
                        .getAbsolutePath();
```

Listing 1: Lange Aufrufketten führen zu unnötiger Kopplung

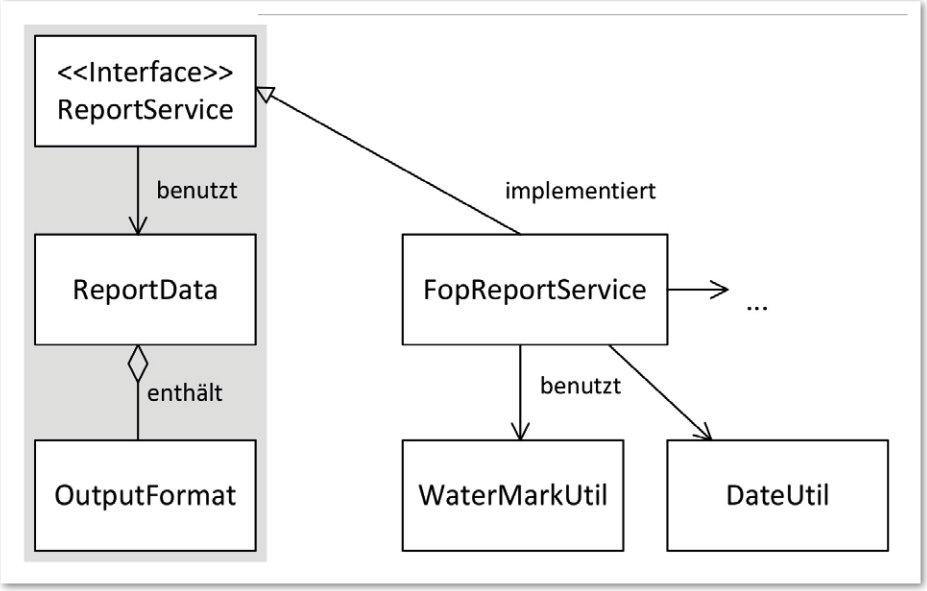


Abbildung 1: Aufbau des Report-Service-Moduls

```
<subpackage name="reportservice">
  <subpackage name="api">
     <allow pkg="com.foo.reportservice.api"/>
   </subpackage>
   <subpackage name="impl">
     <allow pkg="com.foo.reportservice.api"/>
     <allow pkg="com.foo.reportservice.impl"/>
     <allow pkg="com.lowagie"/>
     <allow pkg="org.apache.fop"/>
  </subpackage>
</subpackage>
```

Listing 2: Import-Regeln für das Report-Service-Modul unterteilt in „api" und „impl"

of inheritance tree") sollten beispielsweise alternative Kompositionen [4] in Betracht gezogen werden. Die maximale Tiefe von Vererbungsbäumen lässt sich unter anderem mit SonarQube [5] messen.

Durch ein API mit nicht offensichtlicher Reihenfolge-Erwartung entsteht eine andere Form von unnötig starker Kopplung: zeitliche Kopplung (engl. „temporal coupling"). Ein Problem ist, dass das API falsch verwendet werden könnte. Ein anderes Problem besteht darin, das Wissen über die Eigenheiten des API beziehungsweise dessen Implementierung in den Client-Code einzubringen. Eine Konsequenz daraus ist, dass der Client-Code stärker als notwendig an das API gebunden wird. Zeitliche Kopplung führt zu fragilem Code, der schwer zu warten ist.

Ebenfalls unnötig viel Wissen wird in der Codebasis verteilt. Dies widerspricht dem Modulkonzept nach Parnas, wenn Objekte nicht nur mit denen in ihrer unmittelbaren Umgebung kommunizieren. Regeln, wie sie vom Gesetz von Demeter [6] vorgegeben werden, sind zwar praktisch schwer umsetzbar, aber lange Aufrufketten, wie sie *Listing 1* als Beispiel zeigt, sollten definitiv vermieden werden.

Der Code wird von einem Report Service verwendet, um zu bestimmen, wo die erzeugten Berichte gespeichert werden sollen. Der Report Service sollte jedoch die mandantenspezifischen Konfigurationen nicht kennen und auch nicht wissen, wie man an diese gelangt. Einfacher wäre es, dem Report Service nur die Informationen beziehungsweise Objekte zu geben, die tatsächlich für die Erfüllung seiner Funktion notwendig sind. Entsprechend des Single-Responsibility-Prinzips sollte das Abhängigkeitsnetz zwischen diesen Objekten von einer anderen zentralen Komponente durch Dependency Injection aufgebaut werden.

Zyklische Abhängigkeiten sind ein weiteres Beispiel für starke Kopplung, die vermieden werden kann. Das Acyclic-Dependencies-Prinzip [7] verbietet zyklische Abhängigkeiten zwischen Packages. Eine Aussage über zyklische Abhängigkeiten zwischen Klassen – obwohl diese ebenfalls Weiterentwicklung, Test und Verständlichkeit erschweren – wird nicht getroffen. Trotzdem sollten auch zyklische Klassenabhängigkeiten überwacht und zeitnah durch Refactoring-Maßnahmen ausgebaut werden. Denn schnell können diese zyklischen Netze wuchern und sich in der gesamten Codebasis ausbreiten. Ein Refactoring ist

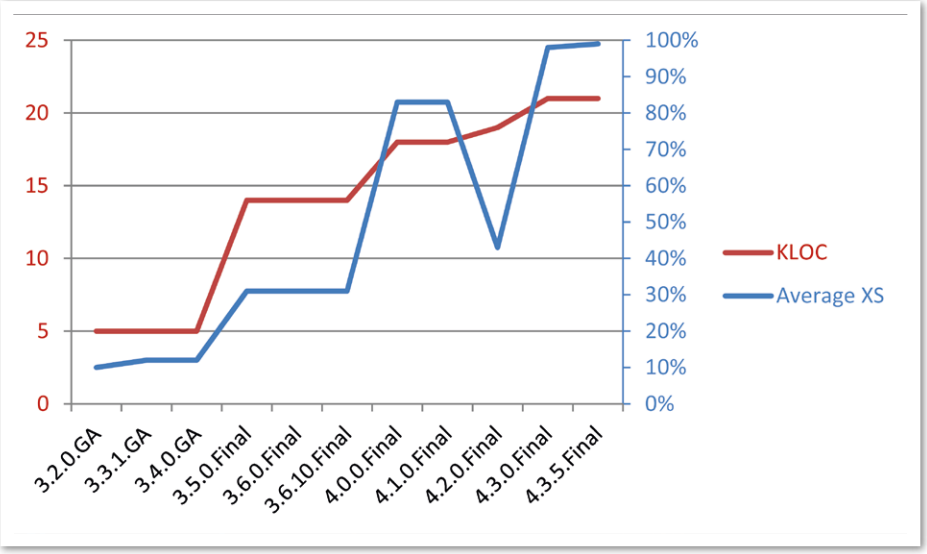


Abbildung 2: Entwicklung von Größe und Komplexität des „hibernate-entitymanager.jar"

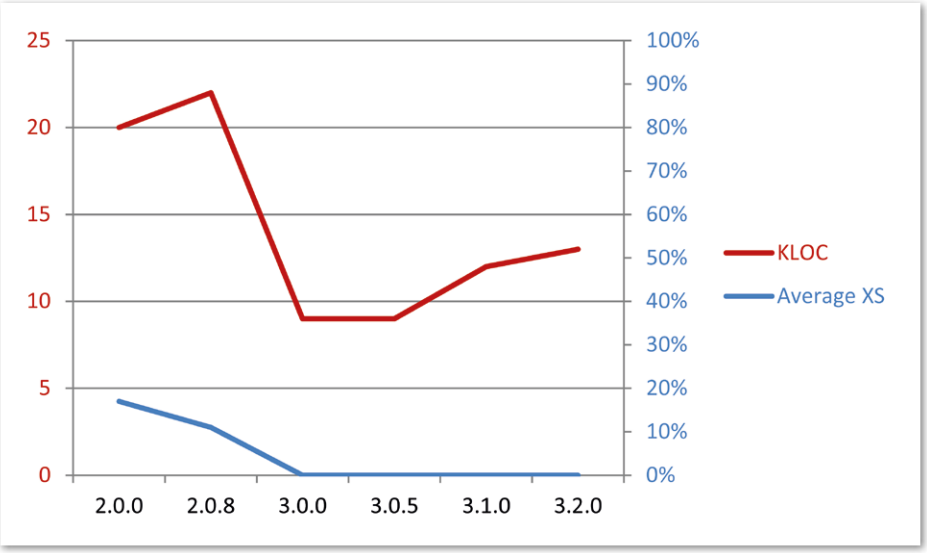


Abbildung 3: Entwicklung von Größe und Komplexität des „spring-security-core.jar"

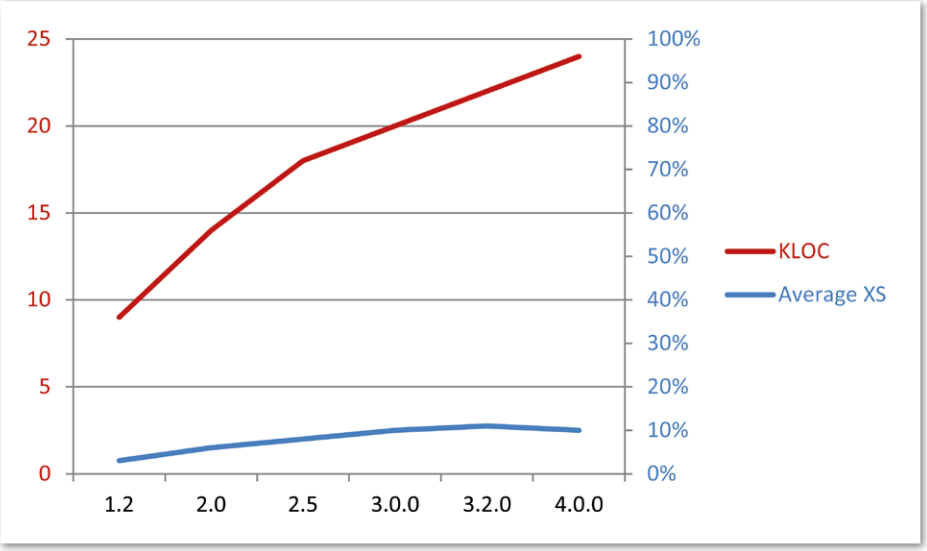


Abbildung 4: Entwicklung von Größe und Komplexität des „spring-beans.jar"

dann nur noch mit sehr großem Aufwand möglich. Zyklische Abhängigkeiten können leicht mit Werkzeugen wie JDepend *[8]* erkannt werden.

Potenzielle Kopplung minimieren

Ein bekannter Ansatz zur Komplexitätsreduktion ist „teile und herrsche". Die Codebasis wird dabei in kleinere Module unterteilt, die insgesamt weniger komplex und leichter zu warten sind als eine monolithische Alternative. Doch im Laufe der Zeit können durch Imports neue Abhängigkeiten hinzukommen, die diese Module immer stärker zusammenziehen.

Das Beispiel in *Abbildung 1* zeigt das Problem. Der Report Service ist Teil einer größeren Codebasis und hat die Aufgabe, Berichte als PDF und in anderen Formaten zu erzeugen. Der Client-Code soll nur das API des Moduls, das grau hinterlegt ist, benutzen können. Die eigentliche Implementierung soll im Inneren des Moduls verborgen bleiben.

Das OSGi-Komponentenmodell bietet die Möglichkeit, zwischen exportierten und internen Packages zu unterscheiden. Die Klassen und Ressourcen in den internen Packages können nicht von anderen Bundles benutzt werden. Die exportierten Packages repräsentieren die Schnittstelle des Bundles. Genau diese Situation soll auch ohne den Einsatz von OSGi erreicht werden. Doch eine Klasse kann, abgesehen von Ausnahmen wie privaten inneren Klassen, potenziell jede andere Klasse im Klassenpfad importieren. Es besteht also die Gefahr, dass sich neue Abhängigkeiten einschleichen. Diese Abhängigkeiten sind nachfolgend als „potenzielle Kopplung" bezeichnet.

Im Beispiel haben die Klassen „FopReportService" eine Abhängigkeit auf Apache FOP zur PDF-Generierung und „WaterMarkUtil" auf iText zur PDF-Signierung. Das Problem der potenziellen Kopplung besteht auch für diese Drittthersteller-Bibliotheken, denn Klassen außerhalb des Report Service könnten ebenfalls Apache FOP und iText verwenden. In der Projekt-Konfiguration, in der die Abhängigkeiten auf die Drittthersteller-Bibliotheken definiert sind, ist nicht klar erkennbar, warum und für welche Teile der Codebasis die jeweiligen Abhängigkeiten gedacht sind.

Eine mögliche Lösung für dieses Problem ist das Verschieben des Report Service in ein eigenes Projekt („JAR"-Datei). Die Zugehörigkeit der Drittthersteller-Bibliotheken ist dann eindeutig. Die Unterscheidung zwischen Kompilier- und Laufzeit-Abhängigkeiten stellt sicher, dass andere Klassen außerhalb des Report Service diese Bibliotheken nicht direkt verwenden können. Das Problem der potenziellen Kopplung auf die interne Implementierung ist jedoch damit nicht gelöst. Dafür sind konkrete Import-Regeln notwendig, wie Checkstyle *(siehe „http://checkstyle.sourceforge.net")* sie standardmäßig unterstützt. Die Import-Regeln können direkt in der IDE, im Build-Prozess und in SonarQube verwendet werden.

„ReportService", „ReportData" und „OutputFormat" liegen im Package „com.foo.reportservice.api". Das Basis-Package „com.foo.reportservice" hätte auch genügen können, aber der Name „api" dokumentiert besser die Intention dieses Package und macht die Import-Regel einfacher.

Die Implementierung des Report Service kommt in das Package „com.foo.reportservice.impl". *Listing 2* zeigt die konkreten Import-Regeln. Das Subpackage „impl" darf Klassen und Interfaces aus „com.lowagie" und „org.apache.fop" importieren, um die Bibliotheken iText und Apache FOP zu nutzen. Mit dem Einsatz der Import-Regeln wäre das Verschieben des Report Service in eine eigene JAR-Datei „Over Engineering", es sei denn, es wird auch an anderer Stelle wiederverwendet.

Kontinuierliche Architektur-Validierung

Die Überprüfung der potenziellen Kopplung in der Codebasis ist gleichzeitig eine einfache, aber effektive Form der Architektur-Validierung auf technischer Ebene. Hierbei wird die beabsichtigte Software-Architektur mit der tatsächlich umgesetzten verglichen. Neben Checkstyle eignen sich beispielsweise auch Sonargraph-Architect *[9]*, Dependometer *[10]* und Structure101 *[11]* für diese Aufgabe. Das letztgenannte Werkzeug berechnet zudem auch Komplexitätsmetriken, die beispielhaft für „hibernate-entitymanager.jar" in den Versionen 3.2.0.GA bis 4.3.5.Final berechnet wurden *(siehe Abbildung 2)*. Die Größe hat die Einheit Kilo Lines of Code (KLOC) und ist in Rot dargestellt. Die Codebasis wuchs von etwa 5 KLOC auf 21 KLOC, denn mit Release 3.6 wurde beispielsweise JPA 2.0 umgesetzt und mit Release 4.3.0 folgte JPA 2.1. Gleichzeitig nahm die strukturelle Komplexität stark zu. Die Metrik „Average XS" ist in Blau dargestellt. Ihre Werte wuchsen von 10 auf 99 Prozent.

Jedes Element der Codebasis (Package, Klasse oder Methode) hat eine gewisse Komplexität, die bis zu einem definierten Schwellenwert akzeptabel ist. Die Metrik „XS" *[12]* misst den Grad der Überschreitung dieses Schwellenwertes. Deshalb kann man „XS" auch als eine Metrik für Überkomplexität (engl. „over-complexity") bezeichnen. Sie berücksichtigt auf Methoden-Ebene die zyklomatische Komplexität nach McCabe und die Anzahl der Abhängigkeiten auf Klassen- und Package-Ebene. Die zyklischen Abhängigkeiten zwischen Packages sind eine weitere wichtige Komponente dieser Metrik. Der sehr hohe Wert „99 Prozent" lässt die Vermutung zu, dass die Codebasis nur mit sehr hohem Aufwand gewartet beziehungsweise weiterentwickelt werden kann.

Die Codebasis des „spring-security-core.jar" zeigte ähnliche Probleme. Auch diese Codebasis wuchs rasch durch das Hinzufügen neuer Features. Die Abhängigkeiten innerhalb der Bibliothek und auch von anderen Bibliotheken ließen sich nicht klar einzelnen Features zuordnen. Die Package-Struktur und die Klassennamen reichten historisch zurück bis auf Acegi Security aus dem Jahr 2004. Zudem erschwerten zyklische Abhängigkeiten die Weiterentwicklung.

Aus diesen Gründen beschloss man, für das Release Spring Security 3.0 die Quellcode-Organisation grundlegend zu verbessern *[13]*. Die Core-Bibliothek wurde in mehrere JAR-Dateien aufgeteilt, um unter anderem Web-spezifischen Code und Code für LDAP-Unterstützung samt ihrer Abhängigkeiten in jeweils separate JAR-Dateien zu platzieren. Auf diese Weise wurde das Single-Responsibility-Prinzip auf JAR-Ebene befolgt.

Abbildung 3 zeigt den Erfolg dieser Maßnahmen. Nach der umfassenden Restrukturierung in Version 3.0.0 blieb die Überkomplexität konstant bei null, obwohl die Codebasis erneut wuchs. Unterstützt wurde die Restrukturierung durch Structure101.

Ein anderes Beispiel ist Spring Data Commons. In diesem Projekt wird Sonargraph eingesetzt, um die Struktur zu überprüfen. Die Definition der erlaubten Abhängigkeiten ist vergleichbar mit den Import-Regeln von Checkstyle. Der große Unterschied ist jedoch, dass Checkstyle die Regeln für Packages definiert und Sonargraph diejenigen für Module. Diese Module sind eine weitere Abstraktionsschicht und bestehen entweder aus Schichten oder Subsystemen. Beide können auch flexibel miteinander kombiniert und ineinander geschachtelt werden.

Die Java-Packages werden anhand ihrer Namen auf die Module abgebildet.

Die bisher genannten Werkzeuge stellen nicht erlaubte Abhängigkeiten fest; fehlende Abhängigkeiten werden jedoch nicht erkannt. Ein vollständiges Reflexionsmodell, das Divergenzen und Absenzen feststellt, kann mit jQAssistant *[14]* umgesetzt werden, denn mit diesem Werkzeug auf Basis der Graphen-Datenbank Neo4j lassen sich sehr flexibel neue Konzepte und Bedingungen definieren *[15]*.

Ein ebenfalls positives Beispiel ist das „spring-beans.jar" der Versionen 1.2 bis 4.0.0 *(siehe Abbildung 4)*. Die Codebasis wuchs von 9 auf 24 KLOC, aber die Komplexität lediglich von 3 auf 10. Es liegt die Vermutung nahe, dass auch hier schon zu Beginn der Entwicklung Maßnahmen wie kontinuierliche Architektur-Validierung und Software-Metriken eingesetzt wurden. Auf Nachfrage wurden jedoch die herausragenden Fähigkeiten des Architekten als Hauptgrund genannt. Weiche Faktoren, wie Größe und Stabilität des Entwicklungsteams, Motivation und individuelle Fähigkeiten, spielen eine ebenso große Rolle für die innere Qualität der Software und für den Erfolg eines Projekts insgesamt.

Gesetz von Conway

Die Organisation und Kommunikation der Entwickler(-teams) haben ebenfalls Einfluss auf die Architektur eines Software-Systems. Zwei Teams, die relativ unabhängig voneinander an der Umsetzung ihrer Anforderungen arbeiten, bauen in der Regel zwei lose gekoppelte Module oder Subsysteme. Auch die Schnittstellen zwischen beiden Subsystemen korrelieren mit der Qualität und Art der zwischenmenschlichen Kommunikation *[16]*. Dieses Gesetz kann gezielt eingesetzt werden, um lose gekoppelte Module zu realisieren. Ein Extrembeispiel sind Micro-Services, die von unterschiedlichen Teams mit eventuell unterschiedlichen Technologie-Stacks entwickelt und eingesetzt werden können.

Fazit

Dem Entwurf einer modularen Architektur und dem Aufbauen einer effektiven Quellcode-Organisation folgt der Kampf gegen deren Erosion. Die Beispiele zeigen, wie sich die innere Qualität allmählich verschlechtert, wenn keine Maßnahmen zur gezielten Komplexitäts-Reduktion getroffen werden. Zum einen sollte unnötig starke Kopplung vermieden werden, um die Komplexität zu reduzieren. Zum anderen sollte potenzielle Kopplung durch Architektur-Validierung kontrolliert werden, denn die ursprüngliche Struktur verändert sich im Verlauf eines Entwicklungsprojekts. Bei diesen Änderungen besteht immer die Möglichkeit, dass sich die Modularität unbemerkt zum Nachteil verändert.

Weiterführende Links

[1] http://tinyurl.com/2cf9lx6
[2] http://tinyurl.com/mcnmocn
[3] http://tinyurl.com/lyt5864
[4] http://tinyurl.com/3arsaw4
[5] http://www.sonarqube.org
[6] http://tinyurl.com/6yl5le
[7] http://tinyurl.com/cybaa4a
[8] http://tinyurl.com/3ew2ct
[9] http://tinyurl.com/buu9sh3
[10] http://tinyurl.com/l982vcn
[11] http://structure101.com
[12] http://tinyurl.com/kkn6n3b
[13] http://tinyurl.com/pkkqv4c
[14] http://jqassistant.org
[15] http://tinyurl.com/n2srw5l
[16] http://tinyurl.com/l5hkpzx

Kai Spichale
Kai.Spichale@adesso.de
http://spichale.blogspot.de

Kai Spichale ist Senior Software Engineer bei der adesso AG. Sein Tätigkeitsschwerpunkt liegt in der Konzeption und Implementierung von Java-basierten Software-Systemen. Er ist Autor verschiedener Fachartikel und hält regelmäßig Vorträge auf Konferenzen.

http://ja.ijug.eu/15/1/12

Alta heißt das Zauberwort für moderne, elegante Benutzeroberflächen

Die Design-Prinzipien, mit denen Oracle seine Cloud-Applikationen gestaltet hat, sind unter dem Begriff „Alta UI Design" zusammengefasst. Sie sorgen für ein konsistentes und elegantes Design von Benutzeroberflächen für Desktop-, Tablet- und Mobile-Anwendungen und stehen nun auch Designern und Entwicklern zur Verfügung.

Die Richtilinien und Muster basieren auf vier Prinzipien: Applikationen werden zuerst für Mobile konzipiert, das Layout wird einfach und übersichtlich gehalten. Darüber hinaus bietet es eine klare Informationshierarchie sowie mehr Möglichkeiten der Visualisierung von Inhalten. Weitere Informationen erhalten Interessierte auf einer mit Alta entwickelten Website *(siehe „http://www.oracle.com/webfolder/ux/middleware/alta/index.html")*. Eine Live-Demo ist unter *„http://jdevadf.oracle.com/workbetter/faces/index.jsf"* zu sehen. Derzeit sind Alta-Komponentensets für Desktop in Oracle Application Development Framework (ADF) ab der Version 12.1.3 sowie in seinem mobilen Pendant Oracle Mobile Application Framework (MAF) ab der Version 2.0 verfügbar.

Das System wurde bereits für die Oracle Cloud Services genutzt, sowie für die letzte Version der Oracle Fusion Applications und für mobile Applikationen im Bereich E-Business und JD Edwards Enterprise One.

Desktop, Web und native mobile App mit demselben GUI-Code

René Jahn, SIB Visions GmbH

Die Anforderungen an die Entwicklung von Applikationen haben sich in den letzten Jahren stark verändert. Es fing an mit Desktop-Applikationen, den ersten Web-Formularen bis hin zu kompletten Web-Anwendungen und zu guter Letzt kamen die mobilen Applikationen. Wenn vor einigen Jahren eine Desktop-Anwendung noch ausreichend war, muss heute zumindest ein modernes HTML5-Frontend geboten werden.

Der Wunsch nach mobilen Lösungen, egal ob nativ oder Browser-basiert, wird ebenfalls immer größer. Alles in allem sollte eine Applikation immer und überall zur Verfügung stehen, egal ob am Desktop, Smartphone oder Tablet – im Büro, zu Hause oder von unterwegs. Um diesem Wunsch gerecht zu werden, sind unzählige Frameworks und Technologien vorhanden. Doch nicht jedes Unternehmen hat auch ebenso viele Spezialisten, um Geräte- und Plattform-unabhängig zu entwickeln. Der Artikel zeigt, wie es dennoch funktionieren kann.

Der Vorspann lässt vermuten, dass hier auf die Vorteile und Möglichkeiten von HTML5 eingegangen und am Ende eine Beispiel-Anwendung präsentiert wird. Doch mit Single-Sourcing kommt ein anderer Ansatz zur Sprache. Im Grunde wird dabei eine Anwendung nur ein einziges Mal codiert und ohne Sourcecode-Änderungen auf verschiedenen Plattformen eingesetzt. So kann eine Anwendung beispielsweise klassisch am Desktop, mit HTML5 im Browser oder als native mobile Anwendung betrieben werden.

Da dieses Thema äußerst komplex und Technologie-intensiv ist, gibt es nur wenige Entwickler, die diesen Ansatz verfolgen. Ein Weg ist das Open-Source-Java-Framework JVx *[1]*. Es handelt sich dabei um ein sogenanntes „Full Stack Application Framework", das auch als reine Bibliothek eingesetzt werden kann. Damit werden normalerweise klassische Datenbank-/ERP-Anwendungen umgesetzt.

Hinter dem Begriff „Full Stack" verbirgt sich nichts anderes als die Möglichkeit, mit einem einzigen Framework und Technologie-Stack eine vollständige Applikation umzusetzen. Das GUI, die Business-Logik und die Persistierung können mit demselben Framework entwickelt werden. Für Entwickler ergibt sich dadurch natürlich ein gewisser Komfort, da nicht ständig zwischen verschiedenen Frameworks gewechselt werden muss. Aber auch die Vereinfachung der Wartung sollte nicht außer Acht gelassen werden. Denn letztendlich laufen Applikationen schon mal bis zu zehn Jahre.

Obwohl JVx ein vollständiges Full-Stack-Framework ist, wurde es so konzipiert, dass einzelne Teile ersetzbar sind. Es wäre unter anderem möglich, die enthaltene Persistierung durch Hibernate *[2]* zu ersetzen oder die GUI anstatt der standardmäßig enthaltenen Swing-Umsetzung komplett mit JavaFX *[3]* zu lösen. Auf der anderen Seite kann JVx als Bibliothek zu einer bestehenden Applikation hinzugefügt werden, um eventuell die enthaltene Persistierung oder auch nur einzelne GUI-Controls wie den SplitButton *[4]* zu verwenden.

Unabhängig von der GUI-Technologie

Die größte Stärke von JVx ist sicherlich die Unabhängigkeit der GUI-Technologie. Man kann eine Applikation entwickeln, die aktuell mit Swing funktioniert und, wenn es nötig ist, einfach auf JavaFX wechseln. Die Applikation selbst muss bei der Umstellung nicht geändert werden, man verwendet lediglich JavaFX für die Darstellung anstelle von Swing. Aus der Sicht eines Applikationsentwicklers ist das eine feine Sache, denn es bleibt alles beim Alten und die Applikation ist auf einen Schlag wieder modern. Man weiß ja auch noch nicht, was nach JavaFX kommt. Mit JVx ist das auch egal, denn es muss lediglich die neue GUI-Technologie angebunden werden und die Applikation läuft dann damit. Es spielt dabei auch keine Rolle, ob eine Desktop- oder eine Web-Technologie eingesetzt werden soll. *Abbildung 1* zeigt, wie eine Applikation am Desktop mit Swing aussehen kann. Die gleiche Applikation in der HTML5-Variante ist in *Abbildung 2* zu sehen.

Viel spannender ist noch die Frage, wie die Applikation auf mobilen Geräten wie Smartphone oder Tablet aussieht. Die mobilen Geräte sind zwar mittlerweile sehr leistungsfähig, eine entscheidende Rolle spielt jedoch die Bildschirmgröße. Obwohl die Auflösungen immer höher werden, ergibt es wenig Sinn, eine Desktop-Anwendung „1:1" am mobilen Gerät darzustellen. Dies wäre zwar technisch möglich, würde den Anwendern aber keine Freude bereiten.

Zudem darf man nicht außer Acht lassen, dass auf mobilen Geräten ganz unterschiedliche Bedienkonzepte zum Einsatz kommen. Die Eingabemöglichkeiten mit der Software-Tastatur sind eher begrenzt und der Durchschnittsanwender würde am liebsten darauf verzichten. *Abbildung 3* zeigt, wie die mobile Variante einer JVx-Applikation aussehen kann. Es handelt sich dabei um dieselbe Anwendung wie zuvor bei der Desktop- und HTML5-Variante, sie wird lediglich am Tablet eingesetzt.

Desktop vs. Web vs. Mobile

So ansprechend die Darstellungen auch aussehen, so unterschiedlich sind sie letztendlich. Nicht jede Plattform bietet dieselben Möglichkeiten. Der Desktop ist sicherlich die entwicklerfreundlichste Variante mit Zugriff auf angeschlossene Geräte, im besten Fall mit ausreichend Hauptspeicher; große Datenmengen sind handhabbar und der Umsetzung von klassischen SDI- bis hin zu MDI-Applikationen steht nichts im Wege. Es ist zwar nicht immer zwingend erforderlich, die Applikation vor der Verwendung zu installieren, doch die Möglichkeit besteht zumindest.

Im Web hat sich der Single-Page-Ansatz letztendlich durchgesetzt und mehrere Fenster sind eher unerwünscht. Wenn etwa ein eingebettetes modales Fenster für Detaildaten angezeigt wird, ist das gerade noch in Ordnung, aber klassische Pop-ups sind zu vermeiden. Auch mit der Datenmenge sollte man es nicht übertreiben, denn je mehr angezeigt wird, desto träger wird das Handling.

Obwohl nahezu alles im Web standardisiert ist, gibt es noch immer Unterschiede zwischen den Browsern und die Applikation kann abhängig davon ganz unterschiedlich reagieren. Das kann je nach unterstützten Browsern und Versionen ganz schön viel Aufwand verursachen. Äußerst problematisch ist auch die Kurzlebigkeit von Bibliotheken und Frameworks, da gern mal alles umgekrempelt wird, um neue Standards zu etablieren. Das wirklich Praktische an Web-Applikationen sind aber unangefochten die Unabhängigkeit von der Zielplattform und somit die Installationsfreiheit sowie die Styling-Möglichkeiten aufgrund von CSS und diversen Browser-Tools wie Firebug *[5]*.

Die mobile Welt hat nicht viel mit Desktop und Web gemeinsam, denn die Bildschirmgröße ist im Vergleich sehr gering; die Leistung ist zwar theoretisch hoch, aber praktisch mit Vorsicht zu genießen. Das Bedienkonzept ist Finger-orientiert und es wird versucht, mit so wenigen Daten als möglich an den Anwender heranzutreten. Außerdem wird häufig mit Drill-Down gearbeitet, um die Daten-Navigation zu ermöglichen.

Auf mobilen Geräten ist das Eingabeverhalten auch eher zurückhaltend und es sollte auf Formulare mit zwanzig Eingabefeldern verzichtet werden. Der entscheidende Vorteil im Vergleich zu den anderen Plattformen ist die Verfügbarkeit der Geräte, denn ein Smartphone ist in den meisten Fällen immer dabei, und somit auch eine Kamera und der Zugang zum Internet.

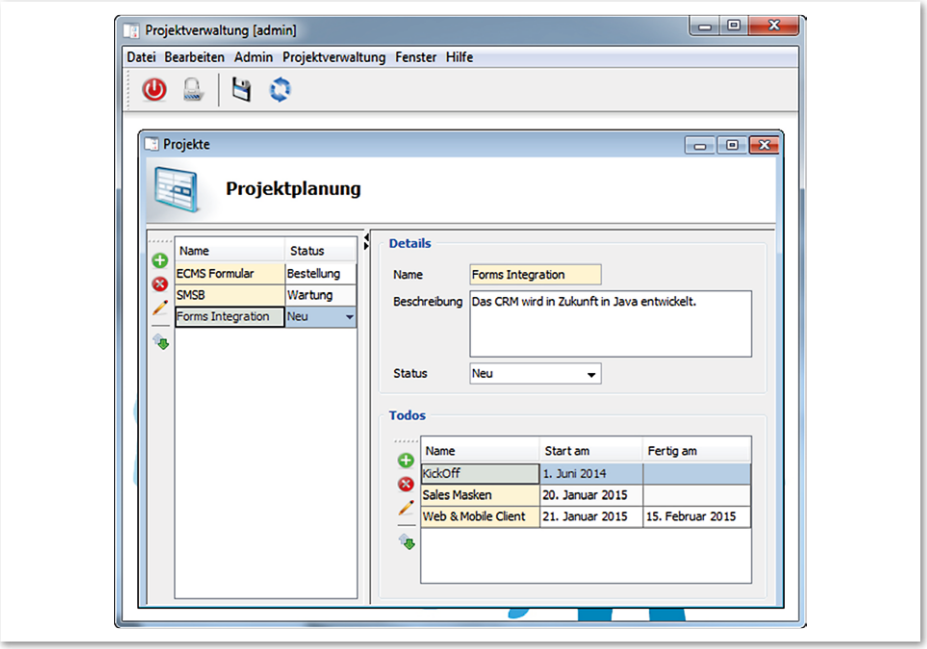


Abbildung 1: Swing-Desktop-Applikation

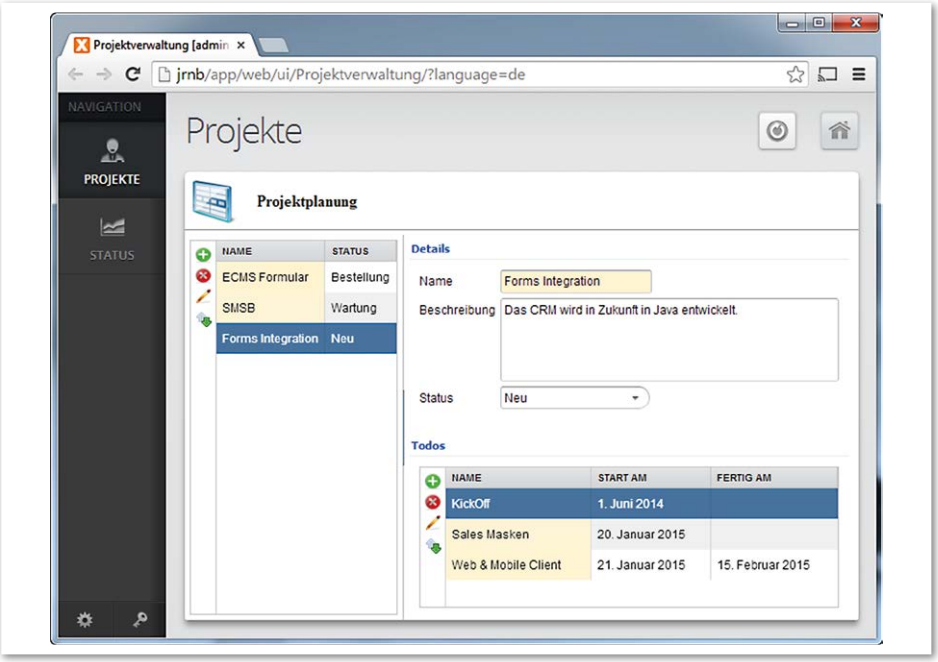


Abbildung 2: HTML5-Anwendung im Browser

Es ist unschwer zu erkennen, dass jede Plattform Stärken und Schwächen hat und die Anforderungen an die Applikationsentwicklung auch dementsprechend unterschiedlich sind. Die Bereitstellung einer Applikation auf allen Plattformen ist daher kein leichtes Unterfangen und erfordert eine Menge Ressourcen. Viele Entwickler sehen die Lösung in HTML5, denn damit lassen sich doch alle Plattformen und Geräte versorgen. Das mag in vielen Fällen so sein, aber im Regelfall braucht man auch damit wieder eine Menge Ressourcen, um alle Browser abzudecken und mit allen eingesetzten Frameworks und Bibliotheken umgehen zu können. Wenn nicht gerade die Ressourcen von Google, Amazon, Microsoft etc. zur Verfügung stehen, kann es sehr lange dauern, bis man das Ziel erreicht.

Der Single-Sourcing-Ansatz hingegen ermöglicht kleinen und natürlich auch größeren Entwicklungsteams die Bereitstellung von Applikationen auf allen Plattformen. Der Aufwand ändert sich dabei kaum und im Falle von JVx kommt nur ein Framework zum Einsatz.

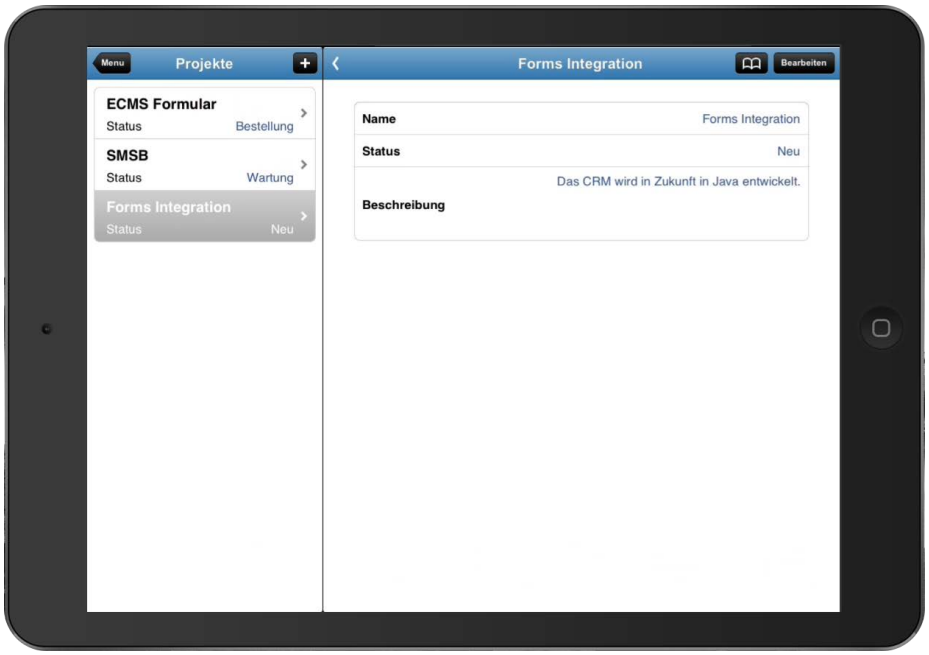


Abbildung 3: App auf dem Tablet

```
public interface IButton extends IComponent
{
   public String getText();
   public void setText(String text);

   public void setImage(IImage image);
   public IImage getImage();
}
```

Listing 1: Die Button-Definition

Vor allem bei der Schulung neuer Mitarbeiter, der Urlaubsvertretung oder bei der Übergabe von Applikationen an die Wartung ist ein einheitliches Tool-Set ein enormer Mehrwert.

Damit eine Applikation nur einmalig codiert werden muss, um dann in unterschiedlichen GUI-Technologien dargestellt zu werden, bedarf es einer sogenannten „UI-Abstraktion". Genau das übernimmt JVx, indem es ein GUI anhand von Java-Interfaces definiert. *Listing 1* zeigt das vereinfacht am Beispiel eines simplen Buttons.

Der Button hat die Eigenschaften für ein Bild und beliebigen Text. Das Interface definiert noch weitere Eigenschaften, auf diese wird hier aber nicht näher eingegangen. Analog zum Button gibt es weitere Interfaces für Label, Image, Textfeld, Farbe, Tabelle, Tree etc. Ein vollständiges Klassendiagramm für die GUI-Klassen ist im Entwickler-Blog *[6]* zu finden.

Die Definition des GUI ist bereits die halbe Miete. Was noch fehlt, um eine Applikation mit verschiedenen Technologien zu verwenden, sind ein sogenannter „Launcher" sowie die Implementierung einer Factory und aller GUI-Interfaces pro Technologie. Der Launcher ist die Startklasse für eine Applikation in der jeweiligen Technologie. Für jede Technologie muss es einen eigenen Launcher geben. Dieser legt die Technologie fest und alle Komponenten laufen dann darin. Damit die Komponenten der eingesetzten Technologie verwendet werden, müssen die GUI-Interfaces von JVx implementiert werden, wie *Listing 2* anhand des Buttons zeigt.

Die Implementierung beschränkt sich auf das Wesentliche. Im Konstruktor wird ein JButton instanziiert, der in weiterer Folge als „resource" zur Verfügung steht. Alle Methoden des IButton-Interface werden dorthin weitergeleitet. Streng genommen wurde ein Wrapper für den Zugriff auf einen JButton geschaffen.

Das letzte Puzzlestück ist nun die Factory. Das Interface „IFactory" definiert „create"-Methoden für alle Komponenten, die JVx unterstützt, beispielsweise „createButton" *(siehe Listing 3)*. In JVx ist unter anderem das Factory-Pattern *[7]* umgesetzt.

Der Launcher legt vor dem Start der Applikation die passende Factory fest und diese erzeugt ausschließlich die Komponenten. Somit bleibt die Applikation beim Tausch des Launchers unbeeinflusst.

Wie bereits erwähnt gibt es große Unterschiede zwischen den Plattformen und Technologien. Der Launcher für eine Desktop-Applikation kann direkt beim Programmstart aufgerufen werden, beispielsweise in einer „main"-Methode. Im Web-Umfeld übernimmt jedoch der Browser beziehungsweise der erste Request den Start. So ergibt sich die Frage, wie der Start in diesem Fall funktioniert.

HTML5

Das Open-Source-Java-Framework Vaadin ermöglicht die HTML5-Darstellung. Da es ein ähnliches Programmiermodell wie Swing besitzt, ist die Umsetzung vergleichbar. Mit Vaadin wird eine Applikation mit vorgefertigten Java-Komponenten entwickelt und diese läuft immer auf einem Java-Applikationsserver. Damit sie im Browser dargestellt wird, gibt es einen eigenen JavaScript-Client, der mit den Server-Komponenten kommuniziert und jegliche Änderungen im Browser nachzieht.

Um eine JVx-Applikation mit Vaadin darzustellen, sind ein Vaadin Launcher, eine Vaadin Factory sowie die Wrapper für die von Vaadin angebotenen Komponenten erforderlich. Der Vaadin Launcher ist eine Ableitung der Vaadin-Applikationsklasse „UI". Diese wird im Deployment-Deskriptor (siehe „web.xml") eingetragen und schon läuft die JVx-Applikation im Browser ganz ohne Plug-ins.

Vaadin übernimmt die Kommunikation und den Datenabgleich zwischen Browser und Server. Obwohl der Desktop und das Web nur wenige Gemeinsamkeiten haben, fällt dank der Möglichkeiten von Vaadin der Unterschied nicht auf. Es konnte auch auf die unterschiedlichen Bedienkonzepte („SID/MDI" vs. „Single Page") eingegangen werden. Aufgrund der Features von Vaadin laufen JVx-Applikationen auch ohne Weiteres als JSR-286-Portlets *[8]* in Liferay *[9]*.

Mobile

Die mobile Welt ist sehr spannend und beginnt mit einer grundlegenden Entscheidung: die Wahl zwischen „nativ" oder Browser-basierter App-Entwicklung. Mit „nativ" hat man den vollen Komfort der jeweiligen Plattform hinsichtlich des Zugriffs auf die Features und auch das volle Styling-Paket. Die Hürde „App Store" ist nicht zu vernachlässigen, hilft aber unter Umständen auch bei der Verbreitung.

```
public class SwingButton extends SwingComponent<JButton>
                      implements IButton
{
   public SwingButton(){
       super(new JButton());
   }

   public String getText(){
       return resource.getText();
   }

   public void setText(String pText){
       resource.setText(pText);
   }

   public IImage getImage(){
       return image;
   }

   public void setImage(IImage pImage)
   {
       if (pImage == null){
           resource.setIcon(null);
       }

       else{
           resource.setIcon((ImageIcon)pImage.getResource());
       }

       image = pImage;
   }
}
```

Listing 2: Der Swing-Button

```
public IButton createButton()
{
   SwingButton result = new SwingButton();
   result.setFactory(this);
   return result;
}
```

Listing 3: SwingFactory

Bei „Browser-basiert" besteht die Abhängigkeit von der Qualität der eingesetzten Frameworks; man hat eventuell Mühe mit der Umsetzung der Offline-Verfügbarkeit und muss bei System-Updates hoffen, dass der Browser noch kompatibel ist. Die Türen zum App Store sind jedoch nicht unbedingt verschlossen. Für JVx wurde der native Weg gewählt, um möglichst flexibel und nur von der Programmiersprache abhängig zu sein. Dafür wurde in Kauf genommen, dass es pro Plattform einen Client geben muss. Im Moment existieren diese für iOS und Android.

Damit die Clients recht einfach bleiben und ohne jegliche Applikationslogik auskommen, wurde ein spezieller Mobile Server umgesetzt. Dieser ermöglicht die Kommunikation mit REST und bereitet eine Applikation für die Anzeige auf mobilen Geräten auf. Wie erwähnt ist es wenig sinnvoll, eine Desktop- oder Web-Anwendung „1:1" auf das mobile Gerät zu bringen. Aus diesem Grund werden beispielsweise keine Toolbars oder Fenster übermittelt. Der Fokus liegt auf Tabellen, Buttons und Eingabefeldern.

Zum Start einer JVx-Applikation muss natürlich ein Mobile Launcher implementiert sein. Diese Aufgabe übernimmt der Mobile Server, denn der mobile Client baut beim Start in jedem Fall eine Verbindung auf. Die Factory- und Interface-Implementierungen sind natürlich auch nötig. Die Umsetzung ist etwas trickreicher als für Desktop und Web, der Mechanismus bleibt jedoch gleich. Der mobile Client wird für typische Anwendungsfälle programmiert, ist allerdings so modular, dass er an die eigenen Anforderungen angepasst werden kann, etwa um ein Dashboard zu integrieren oder das Styling an das CI anzupassen.

Fazit

Der große Vorteil von Single Sourcing besteht darin, dass man sich voll und ganz auf die Applikations-Entwicklung konzentrieren kann. Die technologischen Probleme sind bereits im Framework gelöst. Beim Wechsel auf eine neue Technologie muss die Applikation nicht komplett neu entwickelt werden. Ist die Technologie ausgelagert, vereinfacht das ebenfalls die Wartung, da nur die Applikation gewartet wird und nicht auch noch alles andere. Auf der anderen Seite gibt es aber auch Nachteile, denn die Abstraktion schränkt die Flexibilität ein. Wenn nicht alle notwendigen Aspekte abstrahiert wurden, muss wieder in die Technologie eingegriffen werden. In der Praxis kommt das allerdings selten und nur punktuell vor. Mit JVx besteht zum Glück immer die Möglichkeit, direkt auf die tatsächlich verwendete Technologie zuzugreifen, die Auswirkungen müssen jedoch berücksichtigt werden. Aus Sicht des Autors überwiegen die Vorteile, denn mit einem einzigen Werkzeug alle Plattformen bedienen zu können, ist genial.

Weiterführende Links

[1] JVx: http://sourceforge.net/projects/jvx
[2] Hibernate ORM: http://hibernate.org
[3] JavaFX: http://j.mp/javafx2
[4] SplitButton: http://j.mp/splitbutton
[5] FireBug: http://getfirebug.com
[6] GUI-Klassendiagramm: http://j.mp/jvxuidiagram
[7] Factory Pattern: http://de.wikipedia.org/wiki/Fabrikmethode
[8] JSR 286: http://jcp.org/en/jsr/detail?id=286
[9] Liferay: http://www.liferay.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/15/1/13

Das FeatureToggle Pattern mit Togglz

Niko Köbler, Qualitects Group

Das FeatureToggle Pattern ist nicht neu, aber eines der am kontroversesten diskutierten Vorgehen im Umfeld von Continuous Delivery. Das Thema polarisiert sehr stark und meistens findet man entweder glühende Anhänger oder strikte Gegner; eine Grauzone scheint es nicht zu geben. Togglz ist eine Java-Implementierung, die den Einsatz von FeatureToggles im agilen Umfeld von Continuous Deployment und Delivery vereinfacht und viele Möglichkeiten der Konfiguration bietet.

Vielleicht sollte man FeatureToggles *(nach Martin Fowler [1])* nicht so schwarz-weiß sehen und den Einsatz des Pattern mehr differenzieren. So gibt es durchaus Konstellationen, in denen FeatureToggles fehl am Platz sind oder durch falschen Einsatz eine unnötige Komplexitätsstufe ins Projekt bringen. Es geht um den richtigen Anwendungsfall und den passenden Einsatz dieses Tools, denn ein Werkzeug zum Erreichen eines bestimmten Zustands unserer zu erstellenden Software kann ein Projekt durchaus voranbringen und Kundennutzen erzeugen.

Was ist ein FeatureToggle eigentlich? Im Grunde eine ganz einfache If-Abfrage, die in Abhängigkeit von einem Zustand (State) eines Schalters (Toggle) den Feature-Code ausführt oder auch nicht *(siehe Listing 1)*.

In Zeiten von Git & Co. fragt sich jeder natürlich zunächst grundsätzlich, warum man FeatureToggles einsetzen und „unfertigen" Code mit fertig getestetem und in Produktion befindlichem Code vermischen sollte, wenn Branches nichts kosten, schnell erzeugt und einfach wieder mit dem Masterbranch zusammengeführt werden können? Das ist sicherlich richtig. Doch es gibt Szenarien, in denen so etwas durchaus anders aussehen kann.

Die Entwicklung eines bestimmten Features dauert länger als ein Release-Zyklus (oder Sprint) im Projekt. Die Puristen mögen jetzt wieder aufschreien, dass dann das Feature nicht klein genug spezifiziert und der Detailgrad falsch sei. Mag sein – Lehrbuch hin oder her – in der Realität steht man in einem Projekt oft genug vor solch einem Szenario.

Einen Feature-Branch lange laufen zu lassen und erst am Ende wieder mit dem Master zusammenzuführen, bringt die Gefahr von Merge-Konflikten beziehungsweise der Nacharbeit oder des Refactoring mit sich. Große Teile im Master können sich in der Zwischenzeit ebenfalls geändert haben – genau die großen Teile, von denen das neue Feature abhängt. Eben noch war das Feature also fertig, dann erfolgte der Merge und nun funktionieren weder das Feature noch die restliche Software.

Ausreichende Unit-Tests können dieses fehlerhafte Verhalten bereits früh genug aufdecken. Ob man in solch einem Falle regelmäßig seinen Feature-Branch „rebasen" möchte beziehungsweise dies überhaupt

```
if (schalter.ist_an()) {
  // führe Feature-Code aus
}
```

Listing 1

macht, sei dahingestellt. Hinsichtlich der Rebase-Funktionalität in Git könnte man einen eigenen Artikel schreiben – wenn man überhaupt Git einsetzen kann. In vielen konservativen Unternehmen ist heute immer noch Subversion im Einsatz; mit SVN ist ein Rebase so nicht möglich und ein umfangreiches Mergen eine, sagen wir mal, „eigene Herausforderung".

Was wäre also, wenn der Feature-Code direkt im Master entwickelt werden könnte, dieser die anderen Entwickler aber nicht tangiert und nur das Team, das das Feature entwickelt, mit der Auswirkung des neuen Codes in Berührung kommt? Natürlich nicht nur mit dem funktionalen Code, sondern auch mit Unit-Tests, das versteht sich von selbst.

Dark Testing und Dark Launching

Ein anderes Szenario könnte so aussehen, dass ein neues Feature direkt in die Produktion gebracht werden soll, der Kunde dennoch die Kontrolle darüber haben möchte, damit er, falls das Feature von den Nutzern der Software nicht wie gewünscht angenommen wird, wieder auf die alte Funktionalität zurückschalten kann („Dark Launching") oder im Fehlerfalle die Möglichkeit hat, das neue Feature wieder abzuschalten. Fehler können trotz ausreichender Tests immer auftreten. In einer solchen Fehlersituation ein gesamtes Deployment erneut durchzuführen, ist aufwändig und gegebenenfalls auch teuer. Einfacher wäre es, nur einen Schalter umlegen zu müssen.

Vielleicht soll ein neues Feature zunächst auch nur einer begrenzten Nutzer- oder Servergruppe zugänglich gemacht werden, ohne dass davon bereits alle Nutzer betroffen sind, da so Erfahrungen gesammelt und die oben angesprochenen Fehler minimiert werden können („Dark Testing"). Auch während eines A/B-Tests sind ständig zwei Features in Produktion, werden getestet, gemessen und ausgewertet, bevor entschieden wird, welches Feature weiterhin in der Software verbleibt und weiterentwickelt und welches Feature aus der Codebasis entfernt wird. Irgendwie muss aber gesteuert werden, welcher Nutzer das neue und welcher das alte Feature sehen darf.

Zu guter Letzt sei noch für alle, die gerne aus Lehrbüchern zitieren, gesagt, dass Continuous Delivery keine (Feature-)Branches kennt. Der gesamte Code wird nur in den Master-Branch eingecheckt und von dort aus in Produktion gebracht. Sicherlich ist das eine Art „Totschlag-Argument", das gerne rausgeholt wird, wenn man keine anderen Argumente mehr zur Hand hat, aber es ist Fakt und es gibt nicht wenige Unternehmen, die genau danach ihre Deployment-Pipeline aufgebaut haben und bei denen keine offiziellen Feature-Branches existieren. Facebook, Flickr und Co. setzen hier erfolgreich auf das FeatureToggle Pattern.

Fazit: In all diesen Situationen und in vielen weiteren Use-Cases können FeatureToggles helfen, die bestehenden Anforderungen zu lösen.

```
<!-- Togglz for Servlet environments (mandatory) -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-servlet</artifactId> </dependency>

<!-- The web-based admin console -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-console</artifactId> </dependency>

<!-- CDI integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-cdi</artifactId> </dependency>

<!-- Spring integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-spring</artifactId> </dependency>

<!-- JSF integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-jsf</artifactId> </dependency>

<!-- SLF4J logging bridge -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-slf4j</artifactId> </dependency>

<!-- Seam Security integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-seam-security</artifactId> </dependency>

<!-- Spring Security integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-spring-security</artifactId> </dependency>

<!-- Apache Shiro integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-shiro</artifactId> </dependency>

<!-- Apache DeltaSpike integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-deltaspike</artifactId></dependency>

<!-- Testing support -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-testing</artifactId>
<scope>test</scope> </dependency>

<!-- JUnit integration -->
<dependency> <groupId>org.togglz</groupId>
<artifactId>togglz-junit</artifactId>
<scope>test</scope> </dependency>
```

Listing 2

Hebel umlegen

Zustände von Schaltern können entweder direkt im Code gesetzt oder auch von außen gesteuert werden. Per Default sollte ein Schalter immer ausgeschaltet sein, um ungewollte Nebeneffekte zu vermeiden. Während der Entwicklung eines Features kommen so unbeteiligte Teammitglieder auch nicht direkt mit dem neuen Code in Berührung.

Kann oder will man auf ausgeschalteten beziehungsweise noch nicht aktivierten Code in der Produktion verzichten, könnte man die FeatureToggles nicht erst zur Laufzeit auswerten, sondern bereits zum Build-

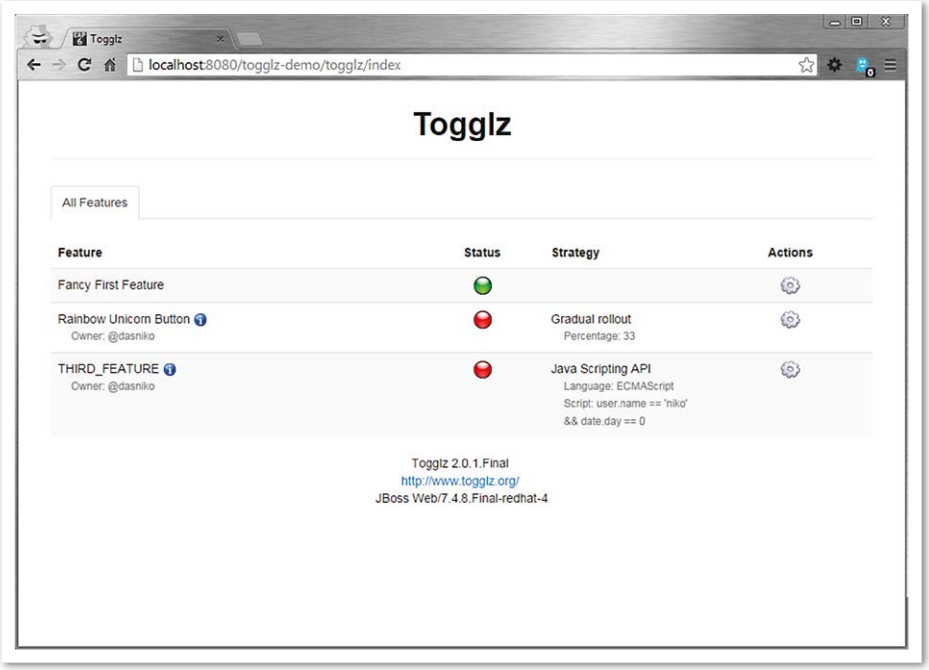


Abbildung 1: Togglz-Administrations-Webkonsole

Zeitpunkt. So käme deaktivierter Code gar nicht erst in das zu bauende Artefakt und der Code kann so auch in der Produktion keinen Schaden anrichten. Die Möglichkeiten des angesprochenen Dark Testing oder Dark Launching sind dann allerdings nicht mehr vorhanden.

Wer jetzt Angst hat, gleichzeitig neuen und alten Code testen zu müssen, und eine kombinatorische Explosion von Testfällen befürchtet, der kann beruhigt werden. Testfälle für das bestehende Feature sind im Allgemeinen bereits vorhanden und für ein neues Feature müssen ganz regulär auch Testfälle geschrieben werden, das wäre in einem Feature-Branch nicht anders. Ein intelligenter Steuerungsmechanismus des Toggle sorgt dann sogar dafür, dass gleichzeitig (beziehungsweise nacheinander) beide Zustände getestet werden können und so eine gute Vergleichsmöglichkeit der Testabdeckung zwischen altem und neuem Feature möglich ist.

Nachdem genügend getestet wurde, das neue Feature stabil in Produktion läuft und dort auch verbleiben soll, sind die gesetzten Toggles natürlich auch wieder auszubauen, denn sie sind nur eine temporäre Lösung, die mit der Einführung des neuen Features ihre Daseinsberechtigung verlieren. Nichts ist schlimmer, als alten Code mit sich herumzuschleppen, von dem in einem halben Jahr niemand mehr weiß, warum er in der Codebasis enthalten ist und welchen Zweck er hatte (oder vielleicht immer noch hat).

Togglz für Java

Da es sich bei einem FeatureToggle nur um eine If-Abfrage handelt, könnte ein solches Hilfs-Framework auch mal eben schnell selbst im Projekt entwickelt werden. Aber man möchte ja das Rad nicht jedes Mal neu erfinden und greift gerne auf bewährte Produkte zurück. Im Java-Umfeld ist dies mit Togglz *[2]* möglich – eine schlanke und leichtgewichtige Bibliothek, die sich um alle bereits angesprochenen Belange kümmert und einem hier sehr viel Arbeit abnehmen kann. Zudem arbeitet Togglz vollständig typsicher, was zur Vermeidung von Schreibfehlern bei Feature-Namen beiträgt und beim Ausbauen von FeatureToggles bereits zur Compile-Zeit mit Fehlern auf sich aufmerksam macht, falls man doch einmal eine Stelle im Code vergessen hat.

Im eigenen Projekt kann man Togglz sofort einsetzen, wenn man die Abhängigkeiten des Frameworks, etwa über Maven, hinzugefügt hat *(siehe Listing 2)*. Togglz ist hier in verschiedene Module aufgeteilt, die je nach gewünschter Funktion beziehungsweise Anwendungsfall ausgewählt werden können. Es stehen Module für Servlet-, CDI- und JSF-Integrationen zur Verfügung, außerdem für die Integration in das Spring-Framework und verschiedene Authentifizierungs-Provider (Spring Security, Seam Security, Apache Deltaspike und Apache Shiro).

Für die Administration der einzelnen Features (oder auch Feature-Groups) und deren Zustände gibt es ein Modul mit einer eigenen Web-Anwendung, die sich in der eigenen Applikation unterhalb des Kontextes mit „/<context-path/togglz" einklinkt (ab Servlet-Version 3.0 automatisch, davor manuell über den web.xml Deployment Descriptor) und registriert *(siehe Abbildung 1)*. Weitere Abhängigkeiten zu dritten Bibliotheken sind nicht notwendig, lediglich das Togglz-Core-Modul wird in allen Fällen benötigt.

Die Features werden in einer Enum-Klasse definiert, die lediglich das Interface „Feature" implementieren muss *(siehe Listing 3)*. Jedes Feature wird durch eine Konstante in der Klasse repräsentiert, dadurch erhalten die Features ihre Typsicherheit. Seit Version 2.0 muss die Methode „isActive()" nicht mehr implementiert sein; falls sie es ist, muss sie zu jedem Feature den jeweiligen booleschen Zustand zurückliefern, indem dieser im „FeatureManager" erfragt wird. Der „FeatureManager" wiederum wird über eine zu erstellende Implementierung des Interface „TogglzConfig" konfiguriert *(siehe Listing 4)*.

```
public enum MyFeaturesEnum implements Feature {

  @EnabledByDefault
  @Label(„Fancy First Feature")
  MY_FANCY_FEATURE,

  @Label(„Rainbow Unicorn Button")
  MY_FANCY_BUTTON;

  @Override
  public boolean isActive() {
      return FeatureContext.getFeatureManager().isActive(this);
  }
}
```

Listing 3

Das Interface liefert drei zu implementierende Methoden:

- *getFeatureClass()*
  Gibt die zuvor erstellte Feature-Enum-Klasse zurück. Somit kennt der Feature-Manager die gültigen Toggles.
- *getStateRepository()*
  Liefert das Repository zurück, in dem die Schalter-Zustände gespeichert werden. Im einfachsten Fall ist das mit „InMemoryStateRepository" der Arbeitsspeicher der Anwendung. Mit einem Neustart der Anwendung gehen dann jedoch auch alle Schalterzustände verloren. Möchte man diese über einen Neustart hinweg behalten, kann man auf das „FileBasedStateRepository" oder das „JDBCStateRepository" zurückgreifen. Wie die Namen schon sagen, werden im ersten Fall die Schalter-Zustände in einer (Properties-) Datei im Filesystem gespeichert, im zweiten Fall in einer Datenbank. Benötigt man ein anderes, nicht bereits vorhandenes Repository, erstellt man sich einfach eine eigene StateRepository-Implementierung und verwendet diese.
- *getUserProvider()*
  Sollen FeatureToggles in Abhängigkeit von einem bestimmten Users oder einer Rolle aktiv oder inaktiv sein, so kann ein „UserProvider" zum Einsatz kommen – entweder einer der bereits oben angeführten Security-Provider oder man implementiert sich selbst einen „UserProvider". Ist keiner notwendig, gibt man einfach „new NoOpUserProvider()" zurück, dann gelten die Einstellungen für alle Nutzer gleichermaßen.

Im Falle einer Java-EE-Anwendung mit CDI muss man nichts weiter tun, Togglz ist bereits so konfiguriert, dass es sich im Container selbst registriert. In einer Spring-Umgebung annotiert man die TogglzConfig-Implementierung einfach mit „@Component" und für Servlet-Anwendungen wird nur ein Context-Parameter in der Datei „web.xml" auf die eigene TogglzConfig-Klasse gesetzt. Ein Demo-Projekt über die Konfiguration von Togglz ist unter *[3]* zu finden.

An – Aus

Togglz ist nun einsatzbereit und die Schalter können an jeder beliebigen Stelle im Code verwendet werden *(siehe Listing 5)*. Im Falle einer JSF-Anwendung kann über das „rendered"-Attribut einer Komponente gesteuert werden, ob diese angezeigt (gerendert) wird oder nicht *(siehe Listing 6)*.

Eine Bean namens „feature" wird durch das Togglz-JSF-Modul automatisch im Kontext zur Verfügung gestellt. Einziger Nachteil: In JSF-Dateien ist keine Compiler-Typsicherheit vorhanden, sodass hier auf Schreibfehler in Feature-Namen beziehungsweise Fehler beim Entfernen von Features besonderes Augenmerk gelegt werden muss.

Soll das neu zu implementierende Feature ein bereits bestehendes Feature ablösen und möchte man die gesamte Klasse austauschen, so bietet die „FeatureProxyFactoryBean" eine elegante Möglichkeit, in einem Spring-Kontext je nach Schalter-Zustand entweder die eine oder andere Implementierung durch die Proxy-Klasse (ein gemeinsames Interface natürlich vorausgesetzt) ausliefern zu lassen *(siehe Listing 7)*.

Das Umschalten der Feature-Zustände kann entweder manuell in den gewählten „StateRepository" (in der Properties-Datei

```
public class MyTogglzConfiguration implements TogglzConfig {

  @Override
  public Class<? extends Feature> getFeatureClass() {
      return MyFeaturesEnum.class;
  }

  @Override
  public StateRepository getStateRepository() {
      return new FileBasedStateRepository(
          new File("/path/to/features.properties")
      );
  }

  @Override
  public UserProvider getUserProvider() {
      return new NoOpUserProvider();
  }
}
```

Listing 4

```
if (MyFeatures.MY_FANCY_FEATURE.isActive()) {
  // your code goes here…
}
```

Listing 5

```
<h:panelGroup rendered="#{features['MY_FANCY_BUTTON']}">
  <!-- my new fancy rainbow unicorn button -->
</h:panelGroup>
```

Listing 6

```
<bean id="fancyService" class="myapp.services.NewFancyService" />
<bean id="boringService" class="myapp.services.BoringService" />
<bean id="myService" class="org.togglz.spring.proxy.FeatureProxyFac-
toryBean">
  <property name="feature" value="MY_FANCY_FEATURE" />
  <property name="active" ref="fancyService" />
  <property name="inactive" ref="boringService" />
</bean>
```

Listing 7

oder in der Datenbank) durchgeführt werden, man verwendet dafür die Administrationskonsole oder man lässt Togglz mithilfe der „Activation Strategies" selbst entscheiden, ob ein Feature greift oder nicht. Die Möglichkeiten reichen hier von einfachen Username-Vergleichen über Client- und Server-IP-Adressen, Zeit- und Datumsgrenzen bis hin zu einem anteiligen Rollout, bei dem nur jeder „n"-te Benutzer, in Abhängigkeit vom angegebenen Prozentwert, das Feature geschaltet bekommt. Auch eine Scripting-Möglichkeit mittels ECMAScript (JavaScript) steht zur Verfügung. Zu guter Letzt, falls das alles nicht ausreicht, kann auch hier, wie bei allen Komponenten, eine eigene Implementierung geschrieben werden.

Togglz bietet noch eine große Fülle weiterer Möglichkeiten, wie „FeatureGroups" und erweiterte Konfigurationen für verschiedene Umgebungen, auf die hier im Rahmen dieses Artikels leider nicht mehr eingegangen werden kann. Details hierzu und weitere Optionen können aber auf der Projekt-Website *[3]* nachgelesen werden. Die Dokumentation des Projekts ist sehr umfangreich und verständlich.

Fazit

Togglz bietet eine einfach anzuwendende und zu integrierende, aber leistungsfähige Möglichkeit, FeatureToggles in Java-Projekten umzusetzen. Vielleicht werden sogar einige Kritiker dadurch überzeugt, sich Togglz zumindest einmal anzuschauen.

Weiterführende Links

[1] http://martinfowler.com/bliki/FeatureToggle.html
[2] http://www.togglz.org
[3] https://github.com/dasniko/togglz-demo

http://ja.ijug.eu/15/1/14

Niko Köbler
niko@n-k.de

Niko Köbler ist freiberuflicher Software-Architekt, Developer und Coach für (Java-)Enterprise-Lösungen, Integrationen und Web-Development. Er ist Mitbegründer der Qualitects Group, berät und unterstützt Kunden verschiedenster Branchen, hält Workshops und Trainings und führt Architektur-Reviews durch. Neben seiner Arbeit bei der Java User Group Darmstadt (JUG DA) schreibt er Artikel für Fachzeitschriften und ist regelmäßig als Sprecher auf Fachkonferenzen anzutreffen.

Mehr Typsicherheit mit Java 8

Róbert Bräutigam, MATHEMA Software GmbH

Das JDK 8 bietet viel mehr als nur API-Ergänzungen und kleine syntaktische Verbesserungen. Die Lambda-Ausdrücke öffnen mit dem funktionalen Paradigma eine neue Welt für Java-Entwickler, die aufregend ist, aber auch Unsicherheit bringt.

Natürlich werden sich Verfechter der funktionalen Programmierung mit den Änderungen in JDK 8 nicht zufriedengeben und die Typ-Inferenz oder sogar das fehlende Pattern-Matching kritisieren – vielleicht zu Recht. Manche Java-Entwickler werden sich wiederum beklagen, dass der Code sich mehr und mehr Java-fremd und unlesbar anfühlt – vielleicht ebenfalls zu Recht. Es bleibt also für den pragmatischen Java-Entwickler die Frage: Helfen Lambda-Ausdrücke wirklich? Wenn ja: Wo und wie kann man sie einsetzen, sodass der Code lesbar bleibt?

Die neuen Lambda-Ausdrücke

Kurz und knapp gesagt bieten Lambda-Ausdrücke in Java eine Möglichkeit, anonyme Klassen zu implementieren *(siehe Listing 1)*. Dieses Interface kann man anonym mit „normalem" Java implementieren *(siehe Listing 2)*. Mit Lambdas wird die Methode jedoch wesentlich lesbarer *(siehe Listing 3)*.

Natürlich ist das für einen Java-7-Entwickler erst einmal total unverständlich. Wo ist „message" definiert? Welchen Typ hat „message"? Was bedeutet „Minus-Größer"?

Der Pfeil-Operator („Minus-Größer") signalisiert Java, dass es sich um einen Lambda-Ausdruck handelt und dieser ist ein Pfeil, weil er eine „Abbildung" darstellt. Sie findet zwischen Parameter und Rückgabewert der „say"-Methode statt. Java erkennt sowohl, dass der Rückgabewert hier ein „Speaker" sein soll, als auch, dass Speaker nur eine einzige Methode zum Implementieren anbietet. Es nimmt daher an, dass wir diese Methode implementieren wollen, und schaut auf die Parameter sowie den Rückgabewert dieser Methode.

Links vom Pfeil definieren wir also, wie wir die Parameter benennen wollen (hier müssen wir den Typ nicht angeben, es wird „inferiert"), und rechts kann ein beliebiger Ausdruck stehen (oder auch Block), der den richtigen Rückgabewert-Typ hat (nämlich String). Die Inferenz funktioniert auch, falls das Interface „Generics" benutzt. Wie kann man so ein Werkzeug in der Praxis einsetzen?

Typsicherheit

Typsicherheit ist eine Eigenschaft des Programms, um durch die Kategorisierung von Werten Fehler zu vermeiden. Diese Definition ist nicht von Wikipedia, sie ist ein bisschen praxisnäher und etwas breiter aufgesetzt. Normalerweise versteht man unter Typsicherheit die Eigenschaft, dass man beispielsweise String-Werten keine Integer-Werte zuordnen kann, sei es der Parameter einer Methode, ein Rückgabewert oder eine Variable. Jede dieser Entitäten hat also einen Typ (eine Kategorie von Werten), der nicht verletzt werden darf. Hier endet auch leider die herkömmliche Definition. Was ist jedoch mit „public String getConfig(String key);"?

Streng betrachtet findet hier keine Typ-Verletzung statt, es ist typsicher. Die Methode nimmt einen Schlüssel (einen String) und liefert immer einen String-Konfigurationswert zurück. Falls der Schlüssel in der Konfigurationsdatei nicht existiert, liefert sie „null" – was in Ordnung ist, weil „null" zur String-Kategorie gehört.

Diese Methode schließt jedoch die Möglichkeit nicht aus, dass mit einem falschen Schlüssel aufgerufen wird, der immer „null" liefert, und man also ein semantisches (Verständnis-) Problem einbaut. Das lässt sich mit einer anderen Signatur beseitigen: „public String getConfig(ConfigKey key);".

Diese Definition verhindert einen Fehler durch besseres Kategorisieren, also durch Eingrenzen des Parameters mithilfe des Typ-Systems. Daher fällt sie unter die Definition, die oben dargestellt wurde, und damit unter „Typsicherheit". Der Compiler kann aber leider nicht registrieren, ob der Schlüssel per Design „String" oder „ConfigKey" sein sollte. Das können nur Entwickler erkennen.

Manchmal wird dieses Problem („String-Schlüsselwert") auch mit Konstanten („static final"-Variablen) gelöst. Diese Lösung ist jedoch nach obiger Definition nicht typsicher, weil sie nicht durch Kategorisierung der Werte erfolgt, sondern über Konventionen.

Defensives Entwickeln

Das Problem ist also, dass Entwickler erkennen müssen, wann eine bessere Kategorisierung möglich ist. In anderen Worten: Um typsicheren Code zu schreiben, muss man defensiv entwickeln. „Defensiv" heißt, Fehler nicht nur zu signalisieren, sondern gar nicht zuzulassen, dass sie idealerweise zu Kompilierfehlern führen. Wenn nämlich ein Fehler möglich ist, wird er früher oder später garantiert auch gemacht.

In der funktionalen Welt hört man oft: „Wenn es kompiliert, funktioniert es auch". Auch wenn das vielleicht nicht ganz realistisch klingt und sicher nicht immer sinnvoll ist, geht es doch in die richtige Richtung. Der Rechner benötigt nicht unbedingt Typen, er kann genauso Code ohne diese ausführen. Entwickler sind es, die Kategorien brauchen, um den Code besser zu verstehen – um ihre Abstraktionen für sich selbst und für andere Entwickler besser darzustellen, weil sie auch diejenigen sind, die Fehler einbauen. Für Fehler im Code sind meistens mindestens zwei Personen verantwortlich. Erstens derjenige, der den Fehler zugelassen hat, der also zum Beispiel die Schnittstelle oder Klasse nicht lesbar oder nicht verständlich genug definiert hat, und zweitens derjenige, der den Code nicht richtig verstanden und deswegen den Fehler eingebaut hat.

Es ist also unsere Aufgabe, so zu entwickeln, Code und Schnittstellen so zu schrei-

```
public interface Speaker {
   String say(String message);
}
```

Listing 1

```
public Speaker getParrot() {
   return new Speaker() {
       public String say(String message) {
           return message;
       }
   };
}
```

Listing 2

```
public Speaker getParrot() {
   return message -> message;
}
```

Listing 3

ben, dass sie mögliche Fehler erst gar nicht zulassen. Die Frage ist: Helfen Lambda-Ausdrücke, dieses Ziel zu erreichen – und wenn ja, wie?

Das Beispielproblem

Angenommen, man möchte eine „Middleware"-Library entwickeln, die zwischen Backend und Frontend sitzt und Business-Logik beispielsweise aus dem Bankwesen implementiert. Das Backend liefert bei jeder Abfrage potenziell Nachrichten an den End User zurück: „Überweisung erfolgreich durchgeführt", „Lastschrift angelegt", etc. Eine der nicht-funktionalen Anforderungen ist es, diese Nachrichten immer an das Frontend weiterzuleiten. Das bedeutet, dass alle internen Library-Schichten diese Nachrichten immer weiter kopieren müssen. *Listing 4* zeigt ein einfaches Beispiel ohne diese Funktionalität.

```
public TransactionId createCashTransfer(LocalAccount source, Account destination, Amount amount) {
   CashTransferResponse response = callCashTransfer(...);
   return new TransactionId(response.getTid());
   // TODO: was ist mit response.getMessages() ?
}
```

Listing 4

Die Frage ist, was diese Methode zurückgeben soll, sodass die Nachrichten – in „response" – auch zurückgegeben werden, das API jedoch trotzdem lesbar bleibt. Eine Lösung wäre, „TransactionId" mit Nachrichten zu ergänzen, aber die beiden haben ja miteinander nichts zu tun. Außerdem müsste man dann alle potenziellen Rückgabewerte genauso ergänzen (inklusive „primitiv"-Typen), was ja aber bekanntermaßen nicht möglich ist.

```
public TransactionId createCashTransfer(LocalAccount source, Account destination, Amount amount) {
   CashTransferResponse response = callWebservice(...);
   TransactionId txId = new TransactionId(response.getTid());
   txId.addAll(response.getMessages());
   return txId;
}
```

Listing 5

Eine ähnliche Variante wäre es, eine gemeinsame Oberklasse zu schreiben und per Abstammung alle Rückgabewerte mit einer Liste von Nachrichten zu ergänzen. Natürlich kann man die gleichen Einwände wie oben auch hier geltend machen, dass nämlich „TransactionId" und andere Rückgabewerte nicht unbedingt etwas mit Nachrichten zu tun haben etc., aber es gibt noch größere Probleme. Die „createCashTransfer"-Methode würde noch einigermaßen funktionieren *(siehe Listing 5)*. Aber dann wird es eine Schicht höher problematisch, wenn man diese und andere Methoden verwenden möchte *(siehe Listing 6)*.

```
public TransactionId createCashTransfer(LocalAccount source, String destinationName, Amount amount) {
   Account destination = lookupContact(destinationName);
   return createCashTransfer(source, destination, amount);
}
```

Listing 6

In dieser Methode wird der Begünstigte per Name gesucht und dann erst die Überweisung angelegt. Die Methode ist ganz verständlich, aber leider nicht korrekt, weil die Nachrichten von „lookupContact()" implizit weggeworfen werden. Vielleicht ist es auch absichtlich so programmiert worden, vielleicht braucht man die Nachrichten von „lookupContact()" überhaupt nicht. Leider kann man das nicht feststellen und daher ist der Fehler, die Nachrichten nicht weiter zu kopieren, hier zugelassen. Der Code ist nicht sicher und nicht aussagekräftig genug.

```
public TransactionId createCashTransfer(LocalAccount source, Account destination, Amount amount, Messages messages) {
   CashTransferResponse response = callWebservice(...);
   messages.addAll(response.getMessages());
   return response.getTransactionId();
}

public TransactionId createCashTransfer(LocalAccount source, String destinationName, Amount amount, Messages messages) {
   Account destination = lookupContact(destinationName, messages);
   return createCashTransfer(source, destination, amount, messages);
}
```

Listing 7

Eine andere Lösung wäre, noch ein Objekt als Parameter durchzuschleusen, und alle Methoden müssten dann die Nachrichten vom Backend immer in dieses Objekt kopieren *(siehe Listing 7)*. Funktionieren würde es in einfachen Fällen schon, einmal abgesehen von der Verschmutzung der Parameter-Liste, aber was geschieht, wenn

```
public TransactionId createCashTransfer(LocalAccount source, String destinationName, Amount amount, Message message) {
   Account destination = lookupContact(destinationName, message);
   if (destination.isEmpty()) {
       destination = lookupGlobal(destinationName, message);
   }
   return createCashTransfer(source, destination, amount, message);
}
```

Listing 8

```
public class MessageContainer<T> {
   private T value; // Der beinhaltete Wert
   private List<Message> messages;

   public MessageContainer(T value, List<Message> messages) {
       this.value = value;
       this.messages = messages;
   }
}
```

Listing 9

```
public MessageContainer<TransactionId> createCashTransfer(LocalAccount
source, Account destination, Amount amount) {
   CashTransferResponse response = callCashTransfer(...);
   return new MessageContainer<>(new TransactionId(response.getTid()),
response.getMessages());
}
```

Listing 10

```
public MessageContainer<TransactionId>
createCashTransfer(LocalAccount source, String destinationName,
Amount amount) {
   Account destination = lookupContact(destinationName); // Compile
Fehler!
   return createCashTransfer(source, destination, amount);
}
```

Listing 11

```
...
Account destination = lookupContact(destinationName).discardMessag-
es();
...
```

Listing 12

```
...
public T discardMessages() {
   return value;
}
...
```

Listing 13

```
public MessageContainer<String> getAddress(String name) {
   MessageContainer<Person> person = lookupPerson(name);
   return ???;
}
```

Listing 14

bestimmte Methoden einen „Rollback" erleben und die Logik einen anderen Pfad nimmt *(siehe Listing 8)*?

Wenn die Abfrage von „lookupContact()" nicht funktioniert, muss man in dem globalen Adressbuch, was immer das auch ist, nachsehen, und falls der Name dort gefunden wird, ist die potenzielle Fehlernachricht von „lookupContact()" unerwünscht. Eine richtige Lösung, zum Beispiel durch neue Message-Objekte und häufiges Kopieren, würde die Logik hier unlesbar machen. Die gleichen Anmerkungen kann man auch bei verschiedenen Tricksereien mit Dependency Injection und Ähnlichem machen, bei denen man das Message-Objekt nicht durch Parameter, sondern auf anderem Wege gewinnt.

Der Nachrichten-Container

Gut lesbar wäre es natürlich, wenn wir die Nachrichten als Rückgabewert bekommen würden, schließlich sind es ja Rückgabewerte. Aber wir können Vererbung nicht benutzen (siehe oben). Die Alternative zur Vererbung ist die Komposition. Es muss also eine neue Klasse geben und diese Klasse muss in der Lage sein, alle Arten von anderen Werten zu beinhalten und noch dazu mit Nachrichten zu ergänzen. Das geht mit Generics noch einfach *(siehe Listing 9)*. *Listing 10* zeigt, wie die einfache Überweisung auf unterster Ebene in unserer Library aussehen würde.

Die Methode liefert „MessageContainer" von „TransactionId" zurück, mit anderen Worten eine mit Nachrichten angereicherte „TransaktionsId". Das ist so weit lesbar. Aber funktioniert es und ist es sicherer als die Alternativen? *Listing 11* zeigt, wie man das Beispiel mit dem einfachen „lookup" implementieren würde.

Das würde jedoch nicht kompilieren, weil „lookupContact" ein „MessageContainer <Account>" zurückliefert. Das ist auch gut so, da man hier wieder vergessen hat, die Nachrichten von „lookupContact" zu bearbeiten. Was wäre aber, wenn die Meldungen hier tatsächlich nicht gebraucht werden? Wie kann man das am besten lesbar und verständlich aufschreiben *(siehe Listing 12)*?

Die Methode „discardMessages()" beschreibt eindeutig die Absicht des Entwicklers und führt zum Kompilierfehler, wenn sie vergessen wird. Sie vermeidet eindeutig den Fehlerfall von zuvor, daher ist sie sicherer. Die Methode selbst ist leicht in der Klasse „MessageContainer" zu implementieren *(siehe Listing 13)*.

Der Autor weist darauf hin, dass es ein Fehler wäre, „getValue()“ anstelle von „discardMessages()“ zu verwenden, wie man es bei einem POJO machen würde. Funktional würde es zwar keinen Unterschied machen, semantisch wäre es allerdings nicht das Gleiche; es würde den Geschäftsfall nicht präzise beschreiben und wäre daher auch weniger gut lesbar.

Der Nachrichten-Container und Lambdas

Da der Nachrichten-Container den eigentlichen Wert versteckt, gibt es eine Reihe neuer und ungewöhnlicher Herausforderungen *(siehe Listing 14)*.

Der einzige Weg bisher, das Person-Objekt zu bekommen, war „discardMessages()“, das hier nicht anwendbar ist, da die Nachrichten erhalten bleiben müssen. Um an das Person-Objekt zu kommen, müsste man normalerweise „MessageContainer“ auspacken, die Adresse der Person abfragen und dann die Adresse mit den Nachrichten wieder einpacken. Diese Lösung würde jedoch lang und fehleranfällig sein (und überall in den Zwischenschichten auftauchen). Was wäre aber, wenn „MessageContainer“ nicht ausgepackt werden müsste, sondern die Person innerhalb des Containers „transformierbar“ wäre *(siehe Listing 15)*?

Die „Mapper“-Klasse repräsentiert also eine Abbildung von „T“ (Person) auf „R“ (String) und „MessageContainer“ kann mit seiner „map()“-Methode diese Abbildung an sich selbst anwenden. Damit lässt sich die „getAddress()“-Methode bereits implementieren *(siehe Listing 16)*.

Das ist allerdings ganz schön viel Code, nur um „person.getAddress()“ schreiben zu können. Wenn das jetzt schon alles wäre, könnte man diese Idee hier gleich verwerfen. Mit herkömmlichem Java kommt man hier leider auch nicht weiter. Mit Java 8 kann diese Abbildung allerdings ganz anders aussehen *(siehe Listing 17)*. Ohne etwas zu kopieren und ohne überhaupt die „Mapper“-Klasse zu kennen, kann man die Transformation einfach und lesbar darstellen.

Lambda auf Quadrat

Das „lookupContact()“-Beispiel war ein bisschen gemogelt. Wir haben gesagt, dass die Nachrichten verworfen werden. Was passiert hingegen, wenn sie doch erhalten bleiben müssen? Jetzt, da wir die „map()“-Methode haben, können wir es vielleicht damit implementieren *(siehe Listing 18)*.

Also wird „lookupContact()“ ausgeführt, was ein „MessageContainer<Destination>“ zurückgibt. Mit „map()“ bildet es die „Destination“ in „MessageContainer“ auf die „createCashTransfer()“-Methode ab, dadurch kommt „MessageContainer<TransactionId>“ zurück. Kein Wunder, dass es nicht kompiliert, denn zusammen wird ja „MessageContainer<MessageContainer<TransactionId>>“ zurückgegeben. Eigentlich müsste es

```
...
public <R> MessageContainer<R> map(Mapper<T, R> mapper) {
  return new MessageContainer<R>(mapper.map(value), messages);
}
...
public interface Mapper<T, R> {
  R map(T value);
}
```

Listing 15

```
public MessageContainer<String> getAddress(String name) {
  MessageContainer<Person> person = lookupPerson(name);
  return person.map(new Mapper<Person, String>() {
      @Override
      public String map(Person person) {
          return person.getAddress();
      }
  });
}
```

Listing 16

```
public MessageContainer<String> getAddress(String name) {
  return lookupPerson(name).map(person -> person.getAddress());
}
```

Listing 17

```
public MessageContainer<TransactionId> createCashTransfer(LocalAccount
source, String destinationName, Amount amount) {
  return lookupContact(destinationName).map(destination ->
      createCashTransfer(source, destination, amount));
  // Compile Fehler!
}
```

Listing 18

```
...
public <R> MessageContainer<R> flatMap(Mapper<T, MessageContainer<R>>
mapper) {
  MessageContainer<R> result = mapper.map(value);
  return new MessageContainer<R>(result.value, merge(this.messages,
result.messages));
}
...
```

Listing 19

ja irgendwo zwei Container geben, weil es ja auch zwei Aufrufe zum Backend gab. Es ist schon fast in Ordnung so; was wir allerdings haben wollen, ist ein Container, der Nachrichten von beiden Aufrufen bekommt. Wir brauchen also eine neue Methode, die fast wie „map()" funktioniert, aber den Rückgabewert nicht verschachtelt, sondern „flach" macht *(siehe Listing 19)*.

Mit dieser Methode anstatt des üblichen „map()" erhalten wir also alle Nachrichten in einer flachen Struktur. Dieses Muster, die Logik zu übergeben, anstatt sie direkt auszuführen, kann man verwenden, um „MessageContainer" mit allen notwendigen Funktionalitäten zu ergänzen. Damit ist unser Problem gelöst – und zwar so, dass man keinen Gedanken mehr an die Nachrichten verschwenden muss. Wenn der Code kompiliert, können wir sicher sein, dass die Nachrichten richtig bearbeitet beziehungsweise weiterkopiert wurden.

Fazit

Java 8 und Lambda-Ausdrücke bieten Lösungsansätze, um sicheren Code zu schreiben, die vorher Java-Entwicklern nicht zur Verfügung standen. Obwohl diese am Anfang fremd erscheinen mögen, sind sie durchaus existenzfähig und auch praktikabel, wie es hier und natürlich in funktionalen Sprachen schon bewiesen wurde. Java ist jetzt also eine Multi-Paradigma-Sprache geworden und wir Java-Entwickler können uns freuen, neue und aufregende Muster zu entdecken und einzusetzen, um sicherer und einfacher Code zu schreiben.

http://ja.ijug.eu/15/1/15

Róbert Bräutigam
robert@mathema.de

Róbert Bräutigam arbeitet für die MATHEMA GmbH als Berater, Trainer, Architekt und Software-Entwickler mit Schwerpunkt „Java Enterprise" und interessiert sich für leichtgewichtige Lösungsansätze in der Software-Entwicklung sowie im Projekt-Management. Seine berufliche Laufbahn beinhaltet verschiedene Funktionen und Verantwortlichkeiten wie Lead Architekt an großen internationalen Projekten, Software-Qualitäts-Beauftragter, Scrum-Master in Agile-Teams und Java-Entwickler.

Alles klar? Von wegen!

Der faule Kontrolleur und die Assoziationsmaschine

Dr. Karl Kollischan, kobaXX Cosultants

Unser Leben besteht aus Entscheidungen – im Beruf wie auch im Privatleben. Häufig hängt unser Geld, unsere Karriere, unser Glück davon ab. Wem sollen wir unser Vertrauen schenken? Wen halten wir für kompetent? Welcher Mitarbeiter ist für eine bestimmte Aufgabe am besten geeignet? Sollen wir ein aus dem Ruder gelaufenes Projekt, in das schon viel Geld investiert wurde, retten oder abbrechen? Was macht uns in Zukunft glücklich? Für viele unserer wichtigen Entscheidungen kennen wir die Gründe. Wirklich?

Die moderne Kognitions-Psychologie liefert verblüffende Antworten darauf, was in unserem Gehirn passiert, wenn wir Entscheidungen treffen. Wir haben zwei Denksysteme: ein langsames, rationales, das wir bewusst erfassen, und ein schnelles, intuitives, das ständig präsent ist und uns automatisch in Bruchteilen von Sekunden mit Einschätzungen versorgt. Diese sind meistens richtig und sinnvoll, führen jedoch in bestimmten Situationen zu eklatanten Fehlentscheidungen. Dieser Artikel betrachtet die beiden Systeme, ihre spezifischen Charakteristiken, Stärken, Schwächen und wie sie bei unseren Entscheidungen zusammenwirken. In der nächsten Ausgabe gehen wir tiefer auf einige spezielle Situationen ein und lernen mentale Muster kennen, die uns immer wieder in fatale Denkfallen tappen lassen.

Schiefgelaufene Projekte

Software-Entwicklung ist eine komplexe Angelegenheit. Sie ist eingebettet in ein organisatorisches Umfeld – das System –, das Strukturen, Prozesse, Menschen, Werkzeuge etc. umfasst. Diese Bestandteile wechselwirken und beeinflussen sich gegenseitig; das Verständnis einzelner Teile genügt nicht, um ein Verständnis für das Ganze zu entwickeln. Dies hat Jerry Weinberg in seinem Buch Quality Software Management: Systems Thinking [1] als „Weinberg-Brooks' Law" auf den Punkt gebracht: „Mehr Software-Projekte sind schiefgelaufen aufgrund von Maßnahmen des Managements, die auf inkorrekten System-Modellen basierten, als durch alle anderen Gründe zusammen."

Ob wir nun als Führungskraft, Entwickler, Projektmanager oder Product Owner agieren, ein gewisses Verständnis für das Gesamtsystem bringt Vorteile beziehungsweise ist notwendig. Dieses Verständnis entwickelt man nicht nur für sich im stillen Kämmerlein, sondern insbesondere durch Diskussion und Austausch mit anderen. Für beide Aspekte ist es wichtig zu wissen, warum wir über bestimmte Dinge in einer bestimmten Art und Weise denken, also unser eigenes mentales Modell zu kennen oder zumindest unsere blinden Flecken, wie wir später sehen werden.

Schnelles Denken, langsames Denken

In seinem im Jahr 2011 erschienenen Buch „Schnelles Denken, langsames Denken" [2] fasst Daniel Kahnemann die Ergebnisse seiner langjährigen Forschungsarbeit zusammen. Er liefert darin eine völlig neue Sichtweise darüber, was in unserem Gehirn passiert, wenn wir andere Menschen und Dinge beurteilen. Folgen wir eher der Intuition oder der Vernunft? Sein Fazit: Es lassen sich zwei Denksysteme unterscheiden, das sogenannte „Schnelle Denken" und das „Langsame Denken", das er auch „System 1" beziehungsweise „System 2" nennt. In seinem Buch schildert er die vielen Illusionen, auf die wir hereinfallen, beim Versuch, mit dem Verstand die Welt zu erfassen.

Daniel Kahnemann wurde im Jahr 1934 geboren, hat Psychologie und Mathematik studiert und ist bekannt geworden durch seine Arbeiten über Urteils-Heuristiken und kognitive Verzerrungen. Im Jahr 2002 hat er für seine „Neue Erwartungstheorie" den Nobelpreis für Wirtschaft erhalten. Kahnemann räumt darin mit der Annahme auf, dass wir rational agierende Individuen seien. Die Bezeichnungen „System 1" und „System 2" sind dabei als Metaphern beziehungsweise Abkürzungen zu verstehen für „Denkprozesse, die automatisch und mühelos ablaufen und zum größten Teil unterhalb der Bewusstseinsschwelle stattfinden" und „bewusstes Denken, das Konzentration und Anstrengung erfordert". Beim Betrachten

der *Abbildung 1* kann man das eigene Denken im automatischen Modus erleben.

Das schnelle Denken beim Betrachten des Gesichts verknüpft automatisch „Sehen" und „Intuitives". So sicher, wie man sieht, dass der Mann einen Bart hat, so sicher weiß man, dass er wütend und vermutlich kurz davor ist, ein paar sehr unfreundliche Worte zu äußern. Automatisch stellt sich eine Vorahnung dessen ein, was er als Nächstes tun wird. Das alles geschieht mühelos und automatisch. Man hatte nicht die Absicht, seinen Gemütszustand einzuschätzen oder vorauszuahnen, was er als Nächstes tun wird. Man hat nichts bewusst getan – es passierte einfach. Das ist ein Beispiel für schnelles Denken.

Wir betrachten nun folgende Rechenaufgabe: „14 x 27 = ?". Man sieht sofort, dass es sich um eine Multiplikations-Aufgabe handelt, und vermutlich weiß man auch sofort, dass man die Aufgabe lösen kann. Gleichzeitig hat man ein intuitives Wissen über den Bereich, in dem die Lösung liegt. „85" erscheint einem als ebenso absurd wie „11.023". Aber ohne eine gewisse Mühe darauf zu verwenden, weiß man nicht, ob das Ergebnis „428" ist.

Wer die Aufgabe noch nicht gelöst hat, tut es jetzt. Beim Durchführen der Rechenschritte erlebt man langsames Denken. Zunächst wird ein kognitives Programm zum Lösen von Multiplikationsaufgaben, das man vermutlich von der Schule her kennt, aus dem Gedächtnis abgerufen. Die Umsetzung ist mühsam, man muss sich konzentrieren, weil man nicht den Überblick verlieren darf und gleichzeitig Zwischenergebnisse im Kopf behalten muss. Aber es ist nicht nur ein geistiger Prozess, auch der Körper ist daran beteiligt: Die Muskelspannung erhöht sich, Blutdruck und Herzschlag steigen, die Pupillen weiten sich. Sobald man das Ergebnis hat, das übrigens „378" lautet, gehen alle Körperfunktionen wieder auf normal zurück.

Wenn wir an uns selbst denken, identifizieren wir uns mit System 2, unserem bewussten, logisch denkenden Selbst, das Überzeugungen hat, Entscheidungen trifft und unser Denken und Handeln kontrolliert. Wären System 1 und System 2 Schauspieler in einem Film, wäre System 1 der Hauptdarsteller, obwohl System 2 von sich selbst glaubt, im Zentrum des Geschehens zu stehen. System 1 ist spontan, es produziert laufend Eindrücke und Gefühle, die den Input für die expliziten Überzeugungen und bewussten Entscheidungen von System 2 darstellen. Aber nur das langsame System 2 kann in einer geordneten Folge von Schritten das Denken kontrollieren. Beide Systeme gleichen Akteuren, die jeweils eigene Fähigkeiten, Funktionen und Beschränkungen aufweisen. Folgende Beispiele charakterisieren das schnelle Denken:

- Erkennen, dass ein Gegenstand weiter entfernt ist als ein anderer
- Sich der Quelle eines plötzlichen Geräusches zuwenden
- Den Ausdruck „Brot und …" vervollständigen
- Die Feindseligkeit aus einer Stimme heraushören
- Beantworten von „2 + 2 = ?"
- Wörter auf großen Reklameflächen lesen
- Mit dem Auto über eine leere Straße fahren
- Einfache Sätze verstehen

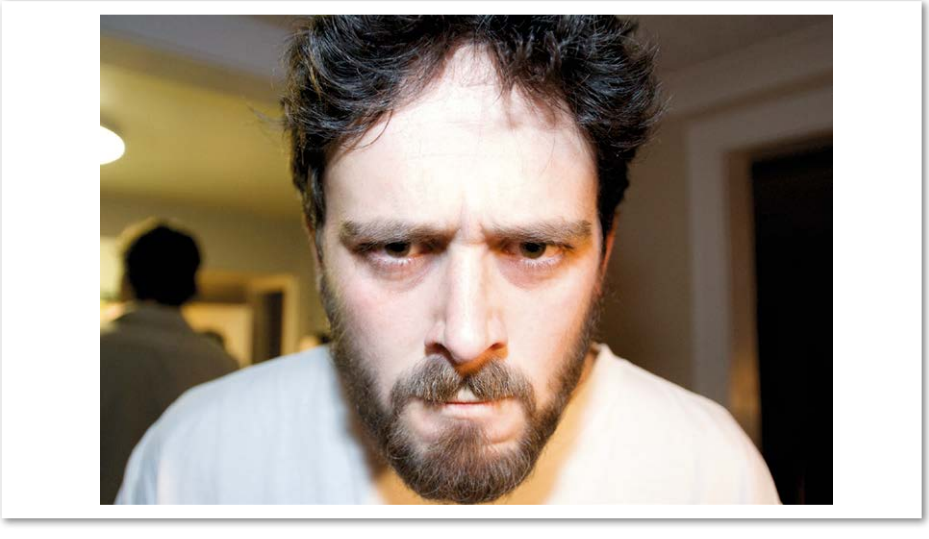


Abbildung 1: Denken im automatischen Modus (System 1)

Alle diese Beispiele gehören in die gleiche Kategorie wie der wütende Mann: Sie geschehen automatisch und mühelos. Zu den Funktionen von System 1 gehören angeborene Fähigkeiten, die wir mit Tieren gemeinsam haben, wie unsere Umwelt wahrnehmen, Gegenstände erkennen, Aufmerksamkeit steuern, Verluste vermeiden, Angst vor Spinnen haben. Andere Funktionen des schnellen Denkens haben wir durch langes Üben erworben, etwa Assoziationen (Hauptstadt von Frankreich), lesen, Verstehen von Nuancen sozialer Situationen.

Beide Systeme sind an der Aufmerksamkeitssteuerung beteiligt. Wenn wir uns automatisch einem lauten Geräusch zuwenden, etwa einer Bemerkung auf einer Party, ist das eine Funktion von System 1. Wir können unsere Aufmerksamkeit aber auch wieder davon abziehen, indem wir uns bewusst auf etwas anderes fokussieren. Dies ist eine Funktion von System 2, sie erfordert Konzentration und wird gestört, wenn die Konzentration nachlässt. Beispiele für Funktionen von System 2 beziehungsweise des langsamen Denkens sind:

- Sich auf die Stimme einer Person in einem überfüllten und lauten Raum konzentrieren
- Nach einer Frau mit weißem Haar Ausschau halten
- Schneller als normalerweise gehen
- Die Angemessenheit seines Verhaltens in einer sozialen Situation überwachen
- Zählen, wie oft der Buchstabe „a" auf einer Textseite vorkommt
- Eine Steuererklärung anfertigen
- Die Gültigkeit einer komplexen logischen Beweisführung überprüfen

All diese Beispiele erfordern bewusste Aufmerksamkeit. System 2 besitzt die Fähigkeit, die Funktionsweise von System 1 in bestimmtem Umfang zu verändern. Wenn man am Bahnhof nach einer bestimmten Person Ausschau hält, kann man sich willentlich darauf konzentrieren. Dies ist jedoch mit Anstrengung verbunden, und unser Aufmerksamkeitsbudget ist begrenzt. Wir können beispielsweise nicht im dichten Verkehr links abbiegen und gleichzeitig „14 x 27" im Kopf berechnen. Wie sehr unsere

Aufmerksamkeit begrenzt ist, illustriert zum Beispiel eindrucksvoll die bekannte „Monkey Business Illusion“ [3]. Darin werden zwei wichtige Tatsachen über mentale Prozesse verdeutlicht: „Wir können gegenüber dem Offensichtlichen blind sein, und wir sind darüber hinaus blind für unsere Blindheit.“

Arbeitsteilung zwischen System 1 und System 2

System 1 und System 2 sind immer aktiv, wenn wir wach sind. System 1 läuft automatisch, System 2 befindet sich in einem angenehmen, entspannten Modus, der nur einen Teil der mentalen Kapazität beansprucht. System 1 generiert kontinuierlich Eindrücke, Intuitionen, Gefühle und Absichten. Wenn System 2 diese unterstützt, werden sie zu bewussten Überzeugungen und Entscheidungen. Im Normalfall vertraut System 2 den Impulsen von System 1 und es liegt dabei meist richtig. Unerwartete Reize werden nur dann wahrgenommen, wenn eine gewisse Aufmerksamkeit darauf gerichtet wird, also System 2 aktiviert ist.

System 2 ist außerdem für die fortlaufende Überwachung unseres Verhaltens zuständig. Es sorgt dafür, dass wir in bestimmten Situationen höflich bleiben, wenn wir wütend sind, oder dass wir beim Autofahren nicht einschlafen, wenn wir müde sind. System 2 wird dann aktiv, wenn es einen drohenden Fehler bemerkt, etwa dass uns eine unangemessene Bemerkung über die Lippen kommt.

Die Arbeitsteilung ist äußerst effizient, in vertrauten Situationen optimiert sie die Leistung und minimiert den Aufwand. Anders würden wir in der komplexen Welt gar nicht zurechtkommen. Es würde viel zu viel Energie kosten und wäre viel zu langwierig, würden wir alle unsere Eindrücke und Vorstellungen mit dem bewussten System 2 überprüfen.

Doch gibt es bestimmte Situationen, in denen die Leistungsfähigkeit des Systems 1 durch systematische Fehler beeinträchtigt wird. Wie wir sehen werden, vereinfacht System 1 komplexe Situationen zu sehr, indem es – ohne dass wir davon etwas mitbekommen – eine einfachere Frage beantwortet als die eigentlich gestellte. Darüber hinaus versteht es kaum etwas von Logik und Statistik und es kann nicht abgeschaltet werden.

Abbildung 2 ist ein Experiment: Zuerst deckt man die rechte Seite ab und benennt laut so schnell wie möglich die Farben der Wörter – nicht die Wörter selbst. Dann deckt man die linke Seite zu und wiederholt das Experiment mit den Wörtern auf der rechten Seite.

Abbildung 2: Müller-Lyer-Illusion

Vermutlich wird man feststellen, dass die Aufgabe für die Wörter auf der linken Seite schnell und mühelos gelingt, während sie für die Wörter auf der rechten Seite um einiges schwerer zu bewerkstelligen ist. Es handelt sich dabei um die sogenannte „Stroop-Interferenz“: Die Benennung einer Farbe eines visuell dargebotenen Worts ist verlangsamt, wenn dessen Inhalt der Farbe widerspricht. Stimmt der Inhalt des Worts hingegen mit der Farbe überein, ist die Benennung der Farbe sogar schneller möglich.

Lesen ist die viel stärker automatisierte Tätigkeit (System 1) als das Benennen der Farben, wofür man sich konzentrieren muss (System 2). System 1 lässt sich nicht abschalten und funkt einem bei der Erledigung der Aufgabe dazwischen. System 1 lässt sich auch nicht überzeugen. Dazu betrachtet man die bekannte optische Täuschung in *Abbildung 3*.

Vermutlich weiß man, dass beide Linien gleich lang sind. Aber sieht man die Linien auch als gleich lang? Auch wenn man nachmisst – sobald man das Lineal wieder beiseitelegt und das Bild betrachtet, wird einem die obere Linie länger erscheinen als die untere.

So wie es optische Täuschungen gibt, gibt es auch kognitive Täuschungen beziehungsweise kognitive Verzerrungen. Die Frage ist, ob wir diese vermeiden können. Die Antwort ist: „Eher nein“. System 2 kann nicht ständig aktiv sein, es wäre viel zu mühsam und anstrengend, unser Denken ständig zu hinterfragen. Aber wir können lernen, Situationen zu erkennen, in denen Fehler wahrscheinlich sind, und dann besonders wachsam sein, wenn viel auf dem Spiel steht.

Der überforderte und faule Kontrolleur

Alle Aktivitäten des Systems 2 schöpfen aus einem gemeinsamen Pool mentaler Energie, egal ob es sich um physische, emotionale oder kognitive Anstrengungen handelt. Konflikte und die Notwendigkeit, eine natürliche Neigung zu unterdrücken, erschöpfen unsere Selbstkontrolle. Beispiele für Selbstkontrolle sind etwa, in einem Gespräch einen guten Eindruck hinterlassen zu wollen oder so viele Kniebeugen wie möglich zu machen.

Versuchsteilnehmer bekamen die Anweisung, ihre emotionale Reaktion auf einen emotional aufgeladenen Film zu unterdrücken und gleichzeitig einen Kraftmesser fest im Griff zu halten. Andere mussten nur den Kraftmesser halten. Erstere schnitten dabei signifikant schlechter ab. In einem anderen Versuch durfte die Hälfte der Teilnehmer nur Nahrungsmittel wie Sellerie und Rettich essen und musste der Versuchung nach Schokolade und Keksen widerstehen. Die andere Hälfte durfte essen, was sie wollte. Anschließend sollten alle eine schwere kognitive Aufgabe erledigen. Die erste Gruppe hat dabei deutlich eher aufgegeben. Abweichungen von der geplanten Ernährung sind Hinweise auf erschöpfte Selbstkontrolle, genauso wie aggressives Verhalten oder schlechtes Abschneiden bei Denkaufgaben.

Als Beispiel eine Aufgabe: „Ein Schläger und ein Ball kosten 1,10 Euro. Der Schläger kostet einen Euro mehr als der Ball. Wie viel kostet der Ball?" Vermutlich fällt einem sofort eine Zahl ein – 10 Cent. Diese Zahl ist intuitiv verlockend – und falsch! Ist man bei dieser naheliegenden Lösung geblieben?

Mit dieser Schläger-Ball-Aufgabe kann man überprüfen, wie intensiv System 2 die Vorschläge von System 1 überwacht. Über jeden, der 10 Cent sagt, wissen wir eine wichtige Tatsache: Sein System 2 hat die Richtigkeit der Antwort von System 1 nicht überprüft, obwohl dies mit geringer Anstrengung möglich gewesen wäre. Diese Aufgabe wurde vielen Tausend Studenten in Harvard, Princeton und am MIT vorgelegt. Mehr als 50 Prozent gaben die intuitive – falsche – Antwort. An anderen Unis waren es sogar 80 Prozent (Lösung siehe „*http://www.doag.org/go/javaaktuell/link1*").

Alle diese Studenten sind in der Lage, viel schwierigere Probleme zu lösen – aber sie haben sich mit der erstbesten Antwort, die ihnen gerade einfiel, zufriedengegeben. Außerdem haben sie einen wichtigen Hinweisreiz übersehen: Warum wird eine so einfache Frage gestellt? Dies alles legt nahe, dass System 2 nicht nur überfordert ist, sondern oft auch faul. Es zeigt, dass wir oft unserer Intuition allzu sehr Glauben schenken. Bei machen Hinweisreizen lohnt es sich, einen Gang zurückzuschalten und das von System 1 gelieferte Ergebnis zu überprüfen.

Die Assoziationsmaschine

Ganz anders arbeitet unsere Assoziationsmaschine, das System 1: Mühelos und schnell erstellt es Verknüpfungen von Vorstellungen. Unser Denken ist nicht eine Folge assoziativer Vorstellungen, sondern es laufen viele Prozesse gleichzeitig ab. Vorstellungen sind Knoten in einem riesigen Netzwerk, dem assoziativen Gedächtnis. Dabei kann man diese drei Typen von Assoziationen unterscheiden:

- Ursache – Wirkung
  (zum Beispiel Virus – Erkältung)
- Objekt – Eigenschaft
  (zum Beispiel Linde – grün)
- Objekt – Kategorie
  (zum Beispiel Banane – Frucht)

Eine Vorstellung aktiviert viele andere Vorstellungen, die wiederum viele andere Vorstellungen aktivieren. Der größte Teil des assoziativen Denkens geschieht unterhalb der Bewusstseinsschwelle. Auch wenn es schwer zu akzeptieren ist, weil es nicht unserem eigenen Erleben entspricht: Wir wissen viel weniger über uns, als wir zu wissen glauben.

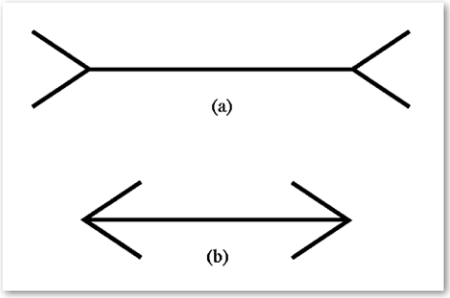


Abbildung 3: Stroop-Interferenz

Wer vor Kurzem das Wort „eat" gehört oder gesehen hat, wird das Wort-Fragment „so_p" eher zu „soup" ergänzen als zu „soap". Letzteres wäre der Fall, wenn man gerade „wash" oder „bath" gesehen hätte. Dieser Effekt wird als „Priming" bezeichnet. Die Vorstellung „essen" bahnt beziehungsweise „primt" die Vorstellung von „Suppe". Wem gerade die Vorstellung „essen" durch den Kopf geht, egal ob ihm das bewusst ist oder nicht, wird das Wort „Suppe" (beziehungsweise alles, was mit Essen zu tun hat) leichter und schneller erkennen.

Priming ist jedoch nicht auf Konzepte oder Wörter beschränkt. Einer der beeindruckendsten Priming-Effekte ist der sogenannte „Florida-Effekt": In einem Experiment wurden Studenten, alles junge Leute, gebeten, aus einer Menge von fünf Wörtern (zum Beispiel „findet", „er", „es", „gelb" und „sofort") Vier-Wort-Sätze zu bilden. Bei einer Gruppe von Teilnehmern enthielt die Hälfte der Wortmengen Wörter, die (in den USA) mit älteren Menschen assoziiert werden, wie „Florida", „vergesslich", „grau", „Falte" und „glatzköpfig".

Nachdem sie die Aufgabe erledigt hatten, wurden die Versuchsteilnehmer für ein weiteres Experiment ans andere Ende des Flurs geschickt. Dieser Spaziergang war der entscheidende Teil des Experiments. Es wurde nämlich unauffällig die Zeit gemessen, die die Probanden benötigten, um ans andere Ende des Flurs zu gelangen. Diejenigen, bei denen altersbezogene Wörter eingestreut waren, gingen erheblich langsamer als die anderen. Dabei war den Studenten die Vorstellung „Alter" nicht bewusst geworden, sie haben noch nicht einmal bemerkt, dass die Wortmengen ein gemeinsames Thema hatten.

Wenn das bei Ihnen auf Unglauben stößt, ist das normal. Unser System 2, mit dem wir uns identifizieren, glaubt, die Kontrolle zu haben und dass es die Gründe für seine Entscheidungen kennt. Auch wenn es uns schwerfällt, das zu glauben: Die Ergebnisse sind nicht erfunden und auch keine statistischen Zufallsergebnisse. Wir müssen uns damit abfinden, dass die zentralen Schlussfolgerungen aus diesen Studien wahr sind – wichtiger noch: dass sie wahr in Bezug auf uns selbst sind.

Hinweis: In der nächsten Ausgabe werden wir tiefer auf einige spezielle Situationen eingehen und mentale Muster kennenlernen, die uns immer wieder in fatale Denkfallen tappen lassen.

Literatur und Links

[1] Gerald Weinberg, „Quality Software Management: Systems Thinking", Dorset House, 1992, und „Systemdenken und Softwarequalität", Hanser Fachbuchverlag, 1994.

[2] Daniel Kahneman: „Thinking, Fast and Slow", Farrar, Straus and Giroux, 2011, und „Schnelles Denken, langsames Denken", Siedler Verlag, 2012.

[3] http://www.theinvisiblegorilla.com/videos.html

Dr. Karl Kollischan
karl.kollischan@kobaxx.com

Dr. Karl Kollischan ist selbstständiger Berater, Trainer und Coach. Seine Schwerpunkte sind agiles Denken und Arbeiten, Projektmanagement und Business-Analyse. Aktuell berät er eine große Organisation bei der Einführung agiler Software-Entwicklung.

http://ja.ijug.eu/15/1/16

Apps entwickeln mit Android Studio

Gesehen von Björn Martin

Das Angebot, eine Rezension zu einer Android-Video-Schulung zu schreiben, war ein guter Anlass für den Autor, sich Android endlich von der Code-Seite her anzusehen und die ersten Schritte als App-Entwickler zu gehen.

Schon die Verpackung macht einen sehr aufgeräumten Eindruck. Man bekommt gleich einen Überblick, worum es in der Schulung geht, selbst wenn man diese noch nicht geöffnet hat – ein dickes Plus für den Verkauf im Buchhandel. Die Packung enthält die DVD, moderiert von Sebastian Witt, und ein Begleitheft, das den Namen auch verdient: Neben einem Vorwort findet man erste Schritte zur Verwendung der Schulung unter Windows, Mac, Linux und auf mobilen Plattformen, eine Übersicht über die Android-Studio-Oberfläche sowie eine über die einzelnen Kapitel. Kleine Ironie am Rande: Wie man die Schulungsvideos auf iOS-Geräte überträgt, ist beschrieben, für Android fehlt die Beschreibung.

Nach dem Einlegen der DVD geht es dann auch direkt los. Die Schulung auf einem Windows-System funktioniert tadellos. Erfreulich: Man muss nichts installieren, die Schulungssoftware läuft komplett von der DVD. Einzig im Benutzerverzeichnis wird eine Konfigurationsdatei angelegt, die den aktuellen Fortschritt und Präferenzen speichert.

Es beginnt mit der Installation und einem ersten Überblick zu Android Studio. Die Version 0.8 liegt auf der DVD vor. Sie ist zum Zeitpunkt der Erstellung dieser Rezension auch die aktuelle Version. Die meisten Kapitel sind zwar mit Version 0.5 erstellt, beim Durcharbeiten waren aber nur angenehme Verbesserungen in der Bedienbarkeit festzustellen. Das Unterkapitel zur Installation wurde neu aufgezeichnet, damit es mit der aktuellen Version stimmig ist.

Titel:	Apps entwickeln mit Android Studio
Autor:	Sebastian Witt
Verlag:	Galileo Press
Umfang:	8 Stunden Video-Training
Preis:	39,90 Euro
ISBN:	978-3-8362-3037-7

Die Schulung selbst ist sehr ausführlich gehalten, man muss also kein Java-Guru sein, um hier mitzukommen. Sogar auf einige objektorientierte Konzepte wird eingegangen. Der allgemeine Programmierstil wirkt sehr aufgeräumt, was Neulingen in der Java-Welt auch zugutekommt – man sieht gleich zu Beginn, wie es richtig gemacht wird. Auch wird durch die einzelnen Kapitel nicht durchgehastet, Sebastian Witt bleibt für jeden Schritt bei uns – keine Aufforderung der Art „und jetzt legen Sie die restlichen Strings selbst an". Falls doch einmal ein Thema aufkommt, zu dem sich Fragen ergeben, die erst später behandelt werden, dann wird darauf hingewiesen – sehr wichtig, um die Ungeduldigen nicht an die nächste Suchmaschine zu verlieren.

Die ersten Gehversuche im Code sind von praxisnahen Tipps begleitet, die deutlich machen, dass Sebastian Witt schon eine Weile mit Android arbeitet und es sich zum Ziel gesetzt hat, dem Leser einige Stolpersteine zu ersparen. Wo nötig, wird auf Android-spezifische Code-Eigenheiten (vor allem im generierten Code) eingegangen oder schon früh darauf hingewiesen, dass geschachtelte Layouts zu einer trägen Oberfläche führen können. Auch gibt es gleich den ersten Exkurs zum Google Android Style Guide, um zu lernen, wie man Layouts richtig einsetzt. Diese Exkurse passieren im späteren Verlauf noch öfter und helfen zu verinnerlichen, dass ein Abstecher auf die Android-Entwickler-Seiten hin und wieder keine schlechte Idee ist – besonders dann, wenn man sich eines neuen Themengebietes annimmt und lieber gleich wissen möchte, wie Google sich das vorgestellt hat.

Das Durcharbeiten der Schulung ist nicht anstrengend, im Gegenteil: Man wird in die einzelnen Themenbereiche mitgenommen und ist am Ende in der Lage, die ersten Gehversuche selbst vorzunehmen. Was man von einer Acht-Stunden-Schulung allerdings nicht verlangen sollte, ist die Fähigkeit, anschließend Angry Birds nachbauen zu können. Die Schulung bietet ganz klar einen Einstieg in Android, aber das solide.

Fazit

Die DVD lohnt sich für Android-Einsteiger auf jeden Fall. Wer mit Android-Konzepten bereits vertraut ist, wird sich stellenweise langweilen. Auch die Geschwindigkeit des Vortrags könnte einigen zu langsam sein. Aber wer die acht Stunden plus Wartezeiten (meist auf den Emulator) investieren kann und will, der wird sehr gut bedient. Für den Rezensenten hat es sich gelohnt; er würde sich über einen Teil 2 freuen.

Björn Martin
jug@bjoern-martin.de

Björn Martin ist leidenschaftlicher Entwickler und seit seinem Studium an der FH Karlsruhe der Sprache Java verschrieben. Er ist seit einigen Jahren als Senior Software Engineer bei Citrix tätig und entwickelt an der Plattform der GoTo-Produktreihe mit. Zudem ist er in der Java User Group Karlsruhe als Co-Organisator aktiv.

http://ja.ijug.eu/15/1/17

SAP HANA – was für Java-Anwendungen drin ist

Holger Seubert, SAP

Neben einer In-Memory-SQL-Datenbank für transaktionale und analytische Aufgabenstellungen bietet SAP HANA weitere Möglichkeiten, die bei der Umsetzung Java-basierter Lösungsarchitekturen interessant sind. Der Artikel stellt sie vor und zeigt anhand von Beispielen ihre Anwendung.

Traditionell kommen für die Umsetzung unterschiedlicher Anforderungen auch unterschiedliche Technologie-Komponenten zum Einsatz. Das gilt besonders für die Informationsverarbeitung. Möchte man in der Anwendung transaktionale (OLTP)-Use-Cases implementieren, die möglichst schnell und transaktionssicher auf einem konsistenten Datenbestand arbeiten, wird ein relationales Datenbank-System mit entsprechend normalisiertem Datenmodell eingesetzt. Besteht gleichzeitig die Anforderung, auch unterschiedliche Auswertungen auf den operativen Daten auszuführen, kommt ein zweites relationales Datenbank-System zum Einsatz, das OLAP-Anfragen beantwortet und üblicherweise ein Star- oder Snow-Flake-Schema als Datenmodell implementiert hat.

Um beide Systeme zu koppeln, setzt man Extract-Transform-Load-Strecken (ETL) als dritte Software-Komponente ein, um die transaktionalen Daten periodisch in das auswertende System zu kopieren. Neben der Duplikation und dem daraus resultierenden, bewussten Datenverlust durch Verdichtungen (etwa Aggregationen) ist der Zeitfaktor ebenfalls ein Aspekt bei der Implementierung von Use-Cases: Analytische Ergebnisse und daraus abgeleitete Entscheidungen können erst nach einer gewissen Zeit getroffen werden.

Auch für weitere Wertschöpfungsprozesse auf den Daten sind potenziell zusätzliche Technologie-Komponenten nötig, etwa für Text-Mining-Anforderungen oder Predictive-Analysis-Aufgaben sowie für die Verarbeitung lokationsbasierter Daten. Im Umfeld wachsender Komplexität setzt HANA als In-Memory-Plattform an, um die Lösungsarchitektur von Anwendungen zu vereinfachen. Das Ziel hinter der geringeren Komplexität sind bessere Lösungen, die schneller entwickelt und die gleichzeitig den Anforderungen der heutigen Informationsverarbeitung gerecht werden. Zentral ist dabei die Betrachtung und Lösung von Anforderungen in der End-to-End-Informationsverarbeitung. Dazu finden sich unterschiedliche funktionale Komponenten in der HANA-Plattform, die in Abhängigkeit von den Use-Cases aus einer Java-Anwendung angesprochen und integriert werden.

Schnittstellen und Werkzeuge

Die unterschiedlichen Features der In-Memory-Plattform werden über Standardschnittstellen angesprochen. Für Java-Anwendungen stellt der HANA-Database-Client einen JDBC-Treiber zur Verfügung. Darauf aufbauend lässt sich mit EclipseLink und Hibernate eine JPA-Implementierung nutzen *(siehe Abbildung 1)*.

Kommt JPA zum Einsatz, ist zur Interaktion mit HANA natives SQL zu empfehlen. Grund dafür ist die einfachere Nutzung einiger Plattform-Funktionen. So werden beispielsweise In-Memory-optimierte Funktionsbibliotheken zum schnellen Data-Mining über gespeicherte Prozeduren angesprochen oder objektorientierte Datentypen mit eigenen Konstruktoren zur Verarbeitung lokationsbasierter Daten genutzt.

Mit entsprechenden Eclipse-Plug-ins für das Kepler- und Luna-Release *[3]* fällt die Interaktion mit der HANA-Plattform leicht. Entsprechende Perspektiven bieten kontextbasiert unterschiedliche Editoren und Wizards zur Nutzung aller Plattformfunktionen an. Neben der Administration und einer SQL-Konsole für datenbanknahe Entwickler werden analytische Modelle oder Data-Mining-Workflows über eigene Editoren gebaut.

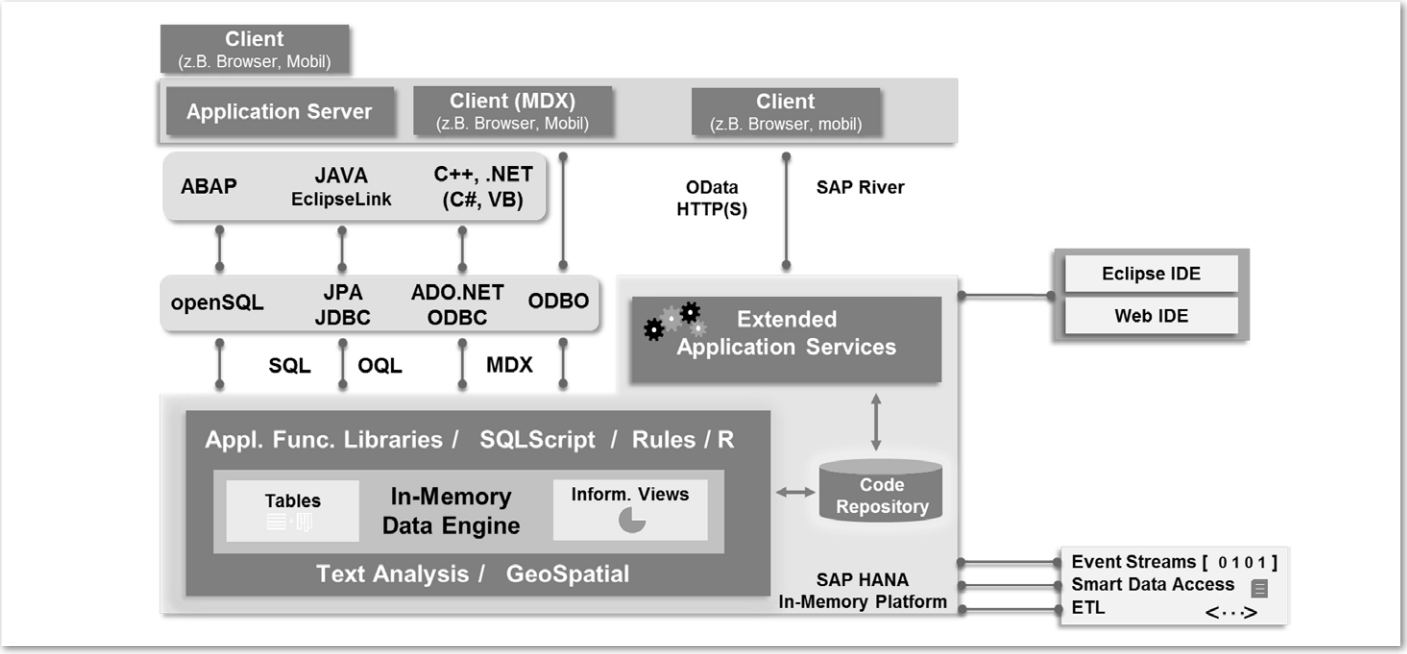


Abbildung 1: Die Anwendung spricht über Standard-Schnittstellen die unterschiedlichen Features der HANA-Plattform an

Echtzeit-Auswertung und Mixed Workloads

Ein zentrales Merkmal der HANA-Plattform ist die Hardware-optimierte, relationale Datenbank, die über Standard-Schnittstellen wie JDBC oder JPA aus einer Java-Anwendung angesprochen wird. Gerade durch die massive Parallelisierung von Datenbankabfragen auf der zur Verfügung stehenden Hardware und eine konsequente Speicherung aller Daten im Hauptspeicher können auf Basis von HANA Java-Anwendungen entstehen, die einerseits hohe transaktionale Lasten tragen und auf der anderen Seite die operativ anfallenden Daten direkt auswerten und analysieren.

Da für die Analyse der Daten keine separate Datenhaltung oder vorbereitende Aggregationen notwendig sind, vereinfacht sich nicht nur die Lösungs-Architektur, die Daten können auch in voller Detailschärfe und in Echtzeit ausgewertet werden. Ein Anwendungsbeispiel ist die HANA-Race-Analytics-Anwendung des McLaren-Teams in der Formel 1 *[5]*. Sie erfasst kontinuierlich den Zustand des Formel-1-Autos durch 120 Telemetrie-Sensoren, die in unterschiedlichen Fahrzeug-Komponenten verbaut sind. Die Sensordaten werden direkt in einem normalisierten HANA-Datenmodell gespeichert.

Pro Runde liefern die Sensoren rund zwei Gigabyte Daten, was sich pro Rennen auf etwa drei Terabyte summiert. Interessant ist bei der Anwendung nicht nur die Erfassung des Zustands wichtiger Fahrzeug-Komponenten, sondern auch die Möglichkeit, diese in Echtzeit, also während des Rennverlaufs, zu analysieren, zu korrelieren und auf der Boxenwand unter Berücksichtigung der Fahrzeugposition zu visualisieren. So wird eine Rennstrategie entworfen, die eine punktgenaue Analyse der Fahrzeugeigenschaften einbezieht.

Um eine solche Echtzeitauswertung auf Basis der sich ständig aktualisierenden, transaktionalen Daten zu realisieren, kommen HANA-Information-Views zum Einsatz *(siehe Abbildung 1)*. Diese ermöglichen eine kennzahlenbasierte Auswertung direkt auf dem normalisierten Datenmodell. Dabei werden die transaktionalen Daten vorher nicht aufbereitet oder aggregiert. Die Auswertung erfolgt in Echtzeit auf den aktuellen Daten. Aus der Java-Anwendung wird per Standard-SQL-Ausdruck „SELECT" auf die Information View zugegriffen und die Ergebnisse wie gewohnt verarbeitet. Modelliert werden Information Views mit einem grafischen Editor, der über die HANA-Development-Eclipse-Plug-ins *[3]* bereitsteht.

Da das McLaren-F1-Team auf eine zehnjährige Daten-Historie zurückgreifen kann, kommen bei der HANA-Race-Analytics-Anwendung auch Vorhersagemodelle zum Einsatz, um Wahrscheinlichkeiten für den Ausfall von Fahrzeug-Komponenten zu ermitteln. Die Berechnung erfolgt dabei nicht nur durch Algorithmen, die in der Anwendungslogik implementiert sind; stattdessen wird auf Advanced-Function-Libraries (AFL) zurückgegriffen, die von der HANA-Plattform bereitgestellt sind.

Delegieren bitte – die HANA-Funktionsbibliotheken

Die HANA Advanced Function Libraries (AFL) werden aus der Java-Anwendung wie gespeicherte Datenbank-Prozeduren mit SQL angesprochen. Es gibt aktuell die Predictive Analysis Library (PAL) und die Business Function Library (BFL). Während die BFL über mehr als fünfzig Algorithmen aus dem finanzmathematischen Bereich verfügt, stellt die PAL rund dreißig Data-Mining/Predictive-Analysis-Algorithmen bereit. So können zur

```
CREATE FULLTEXT INDEX "SENTIMENT_ANALYSIS" ON "COMMENT"("TEXT") FUZZY SEARCH INDEX ON CONFIGURATION 'EX-
TRACTION_CORE_VOICEOFCUSTOMER' TEXT ANALYSIS ON
```

Listing 1

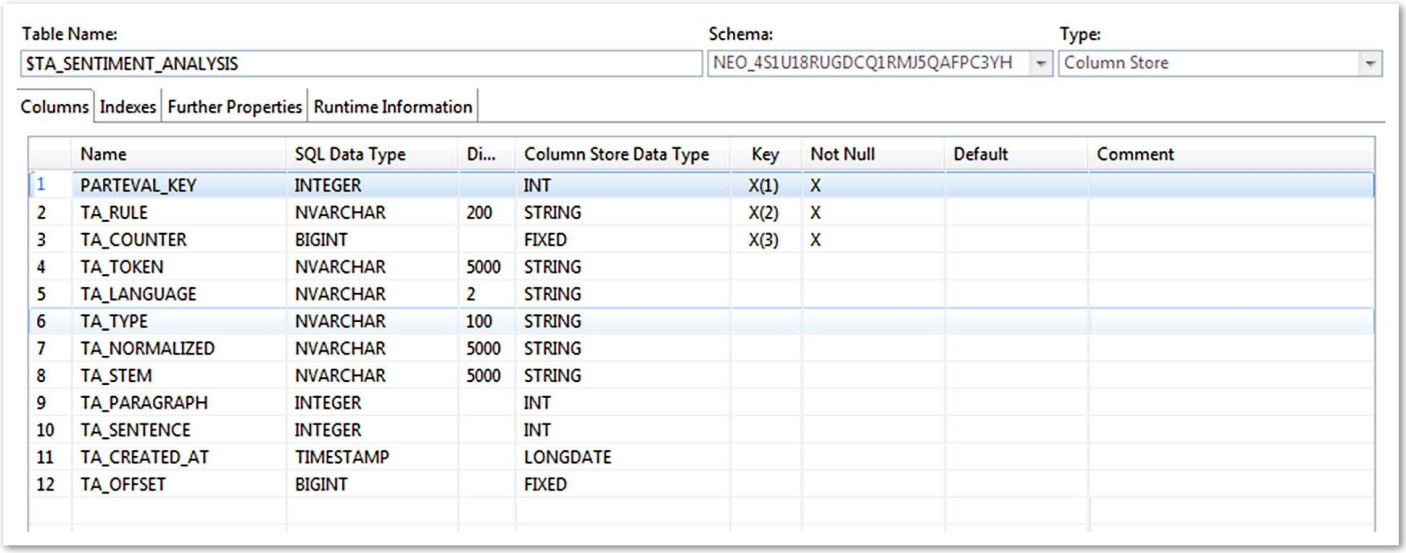


Table Name: STA_SENTIMENT_ANALYSIS | Schema: NEO_4S1U18RUGDCQ1RMJ5QAFPC3YH | Type: Column Store

Columns | Indexes | Further Properties | Runtime Information

	Name	SQL Data Type	Di...	Column Store Data Type	Key	Not Null	Default	Comment
1	PARTEVAL_KEY	INTEGER		INT	X(1)	X		
2	TA_RULE	NVARCHAR	200	STRING	X(2)	X		
3	TA_COUNTER	BIGINT		FIXED	X(3)	X		
4	TA_TOKEN	NVARCHAR	5000	STRING				
5	TA_LANGUAGE	NVARCHAR	2	STRING				
6	TA_TYPE	NVARCHAR	100	STRING				
7	TA_NORMALIZED	NVARCHAR	5000	STRING				
8	TA_STEM	NVARCHAR	5000	STRING				
9	TA_PARAGRAPH	INTEGER		INT				
10	TA_SENTENCE	INTEGER		INT				
11	TA_CREATED_AT	TIMESTAMP		LONGDATE				
12	TA_OFFSET	BIGINT		FIXED				

Abbildung 2: Unterschiedliche Informationen der Text-Analyse wie die Klassifikation (Spalte „TA_TYPE") sind in einer Metadaten-Tabelle zur weiteren Verarbeitung aus der Anwendung gespeichert

weiteren Wertschöpfung der Daten mit der PAL beispielsweise Clustering Algorithms (wie K-Means), Classification Algorithms (wie C4.5 Decision Tree, KNN), Regression Algorithms (wie Polynomial), Association Algorithms (wie Apriori), Time Series Algorithms (wie Arima), Statistics Algorithms (wie Multivariate) und Social Network Analysis Algorithms (wie Link Prediction) genutzt werden. Für ergänzende statistische Betrachtungen ist die Programmiersprache R eingebunden.

Damit übernimmt HANA nicht nur die Aufgabe einer Persistenz-Schicht, es lassen sich auch datenintensive Berechnungen direkt In-Memory optimiert auf der Plattform ausführen. So entfällt auch die Notwendigkeit, die Daten für eine weitere Wertschöpfung aus der Datenbank in die Anwendung zu kopieren, und Teile der Anwendungsschicht lassen sich auf die In-Memory-Plattform für eine optimale Informationsgewinnung auslagern.

Mehr Information durch Text-Analyse

Neben einer kennzahlenbasierten Analyse in Echtzeit sowie einer weiteren Informationsgewinnung durch Data Mining und Predictive Analysis unterstützt HANA auch bei der Verarbeitung unstrukturierter Daten. Zum Einsatz kommen dabei mitgelieferte Wörterbücher, auf deren Basis semantische Informationen erkannt und Entitäten klassifiziert werden. So ermittelt HANA neben Personen und Orten auch Stimmungen, abgestuft nach negativen, neutralen und positiven Äußerungen. Dies kommt häufig bei der Verarbeitung sozialer Medien zum Einsatz. Um HANA domänenspezifisches Vokabular anzulernen, können eigene Wörterbücher auf Basis von XML zum Einsatz kommen.

Definiert wird die Text-Analyse beim Anlegen eines Volltext-Index durch einen geeigneten SQL-Ausdruck. Sind die Textdaten beispielsweise in einer Spalte „TEXT" der Tabelle „COMMENT" gespeichert, wird mit einem SQL-Ausdruck das Standardvokabular „VOICEOFCUSTOMER" für die Ermittlung von Stimmungen verwendet *(siehe Listing 1)*.

HANA erkennt die Sprache sowie unterschiedliche Dokumentenformate automatisch, um entsprechende Algorithmen zur linguistischen Analyse zu verwenden. Die Ergebnisse der Textanalyse sind in der Metadaten-Tabelle „$TA_SENTIMENT_ANALYSIS" mit festgelegter Struktur gespeichert *(siehe Abbildung 2)* und werden von der Java-Anwendung mittels Standard-SQL ausgewertet. So liefert ein „SELECT"-Statement alle Kommentare, die sich stark negativ über ein Produkt äußern *(siehe Listing 2)*.

Lokationsbasierte Verarbeitung

Zur Verwaltung und Verarbeitung geografischer Informationen und Vektordaten unterstützt HANA mit eigenen Datentypen, Rechenoperationen, Zugriffsmethoden sowie räumlichen Koordinatensystemen. Neben der Position des Objekts werden auch die Form und Ausrichtung sowie Routen-Informationen gespeichert. Verwendet wird die HANA GeoSpatial Engine über Standard-SQL (SQL/MM-Erweiterung). Sie ist aus der Java-Anwendung transparent nutzbar, so zum Bei-

```
SELECT * FROM "$TA_SENTIMENT_ANALYSIS"
  INNER JOIN "COMMENT"
  ON "$TA_SENTIMENT_ANALYSIS"."PARTEVAL_KEY" = "COMMENT"."PARTEVAL_KEY"
  WHERE "TA_TYPE" = 'StrongNegativeSentiment';
```

Listing 2

```
SELECT point.ST_X(), point.ST_Y() FROM geo_points;

SELECT point.ST_X(), point.ST_Y() FROM geo_points WHERE
       point.ST_Within( 'POLYGON((0.0 0.0,
                                  2.0 0.0,
                                  2.0 2.0,
                                  0.0 2.0,
                                  0.0 0.0))' ) = 1
```

Listing 3

spiel bei der Definition einer Tabelle mit dem Datentyp „ST_Point", der Längen- und Breitengrad-Informationen eines Objekts speichert: „CREATE COLUMN TABLE geo_points(point ST_POINT);". Über einen Typkonstruktor wird die Koordinate in HANA gespeichert und kann über weitere Standardmethoden abgefragt werden: „INSERT INTO geo_points VALUES (new ST_POINT(0.0, 0.0));".

Der Zugriff kann explizit auf die Längen- und Breitengradinformationen erfolgen oder durch Nutzung einer der Geo-Funktionen („distance()", „surface()", „perimeter()", „intersection()", „within()", „adjacent()", „touches()" etc.) in lokationsbezogene Berechnungen einbezogen werden *(siehe Listing 3)*.

Letzterer „SELECT"-Ausdruck überprüft mithilfe der „ST_Within()"-Funktion, ob sich eine Koordinate innerhalb einer Fläche (= Polygon mit fünf Ecken) befindet. Unterstützt sind auch unterschiedliche Austauschformate wie ESRI, GeoJSON oder WKT, um eine standardisierte Anbindung an Geo-Informations-Systeme (GIS) zu realisieren.

Über den Speicherrand geschaut

Für die Gesamt-Architektur einer Java-Anwendung ist das HANA Smart Data Access (SDA) Feature interessant. Auf Basis von definierten JDBC/ODBC-Verbindungen zu anderen Datenquellen kann HANA über den eigenen Speicherrand schauen und Daten aus anderen relationalen Datenbank-Systemen sowie Hadoop anbinden. Repräsentiert werden die Daten aus den anderen Systemen über virtuelle Tabellen („Proxy-Tabellen"), die bei der Datenmodellierung ohne Einschränkung wie physische HANA-Tabellen verwendet werden.

Neben der Anbindungsmöglichkeit von Hadoop und entsprechenden Map/Reduce-Rechenergebnissen (M/R) lässt sich mit SDA ein mehrschichtiges Speicher- oder Archivierungskonzept implementieren. Die Daten der angebundenen Datenquellen werden nicht in HANA kopiert, nur die Abfrage-Ergebnisse zur Ergebniskonsolidierung werden weitergereicht.

Der Zugriff auf die angebundenen Datenquellen ist für die Java-Anwendung transparent. Lediglich in der Antwortzeit sind Unterschiede spürbar. HANA passt den SQL-Ausdruck über ein Adapter-Framework zwar immer dem jeweiligen Datenbank-System an, dennoch kosten die Verarbeitungsschritte in der anderen Datenquelle zusätzliche Zeit. Aus diesem Grund wird ein Caching der Ergebnisse unterstützt, was gerade bei der Nutzung von Rechenergebnissen der M/R-Algorithmen im Kontext von Hadoop sinnvoll ist.

Weiteres Vereinfachungspotenzial der Anwendungslogik

Abgesehen von einer Beschleunigung bei der Datenbereitstellung und -verarbeitung durch die konsequente In-Memory-Datenhaltung, die massiv-parallel arbeitende Datenbank-Engine sowie die In-Memory-optimierten, integrierten Funktionen zur Wertschöpfung von Informationen besteht weiteres Potenzial zur Vereinfachung der Java-Anwendung.

Durch die spaltenorientierte Speicherung der Daten kann auf die Definition von Sekundär-Indizes weitestgehend verzichtet werden. Dies erleichtert das Performance-Tuning der Datenbank-Schnittstelle und bietet generell mehr Flexibilität beim Zugriff auf die Daten aus der Anwendung. Auch die Implementierung von Caching-Logik in der Anwendungsschicht entfällt durch die In-Memory-Speicherung der Daten, sodass die Anwendung an dieser Stelle einfacher zu implementieren ist.

Da ist noch etwas

Im Kontext der Vereinfachung kann ein HANA-Feature für Java-Anwendungen noch ergänzt werden. Um die Möglichkeiten der HANA-Plattform im Rahmen einer eigenen Anwendung zu nutzen, bestehen prinzipiell zwei Möglichkeiten, die Anwendungsarchitektur zu gestalten. Einerseits kann die Lösung in einer klassischen Drei-Schichten-Architektur gebaut werden. HANA wird dabei über Standard-Mechanismen an einen Anwendungsserver angebunden und per SQL angesprochen. Andererseits kann die eigene Anwendung direkt auf HANA laufen, indem die HANA Extended Application Services (XS) genutzt werden *(siehe Abbildung 1)*. Dies entspricht einer Zwei-Schichten-Architektur. Teil der HANA Extended Application Services ist ein Web- und Applikationsserver zur Entwicklung mit JavaScript und HTML5 sowie zur Nutzung der HANA-In-Memory-Features. Angesprochen wird HANA dabei über OData-RESTful-Services. So lassen sich auch hybride Anwendungen bauen, die die Leichtigkeit der HANA Extended Application Services mit den Möglichkeiten klassischer J2EE-Architekturen kombinieren.

Fazit

Neben den Standard-Schnittstellen zur Nutzung von HANA als Persistenzschicht für unterschiedliche Daten gibt es weitere Aspekte, die in Abhängigkeit von den Anforderungen und Use-Cases Vorteile bei der Implementierung einer Java-Anwendung bringen. Neben der Fähigkeit einer kennzahlenbasierten Analyse direkt auf dem transaktionalen Datenmodell kann die Wertschöpfungskette der Daten durch weitere In-Memory-Funktionen wie Data-Mining, Predicitve-Analysis oder Text-Mining gestaltet werden. Im Zentrum steht dabei immer der Gedanke einer einfachen Lösungs-Architektur.

Um erste Erfahrungen mit den In-Memory-Funktionen und den Werkzeugen zu sammeln, bietet die HANA Cloud Platform (HCP) *[1]* einen kostenfreien Entwickler-Account *[2]*, der zeitlich unbegrenzt für Test- und Entwicklungsaufgaben verwendet werden kann. Mit einer 1-GB-HANA-In-Memory-Cloud-Datenbank-Instanz und einem J2EE-Anwendungsserver können sowohl JDBC- als auch JPA-basierte Java-Anwendungen programmiert werden. Ein Deployment auf der HANA Cloud Platform erfolgt direkt aus Eclipse. Neben HANA und einem Anwendungsserver bietet die HCP weitere In-Memory-PaaS-Funktionen wie ein Content-Repository, das über den CMIS-Standard angesprochen wird.

Literatur und Links

[1] Allgemeine Informationen zur HANA Cloud Platform (HCP): http://hcp.sap.com

[2] Anmeldung zum kostenfreien Entwickler-Account: https://account.hanatrial.ondemand.com

[3] HANA-Development-Tools für Eclipse: https://tools.hana.ondemand.com/kepler

[4] HANA-Platform-Dokumentation: http://help.sap.com/hana_platform

[5] McLaren F1 HANA Use-Case: http://youtu.be/5jOzUGvD_X8

Holger Seubert

h.seubert@sap.com

Holger Seubert arbeitet im Bereich „Datenbanken und Technologie" bei SAP als Senior Solution Architect mit einem Schwerpunkt auf SAP HANA. Vor seiner Tätigkeit bei SAP war er als Software-Entwickler und Technical Sales Specialist bei IBM beschäftigt und arbeitete bei einem Start-Up-Unternehmen als Solution Architect und technischer Berater. Holger Seubert hält zudem Vorlesungen über Datenbank-Grundlagen an der Dualen Hochschule Baden-Württemberg.

http://ja.ijug.eu/15/1/18

Entwicklung mobiler Anwendungen für Blinde

Mandy Goram, TREvisto GmbH

Mobile Endgeräte wie Smartphones und Tablets sind im Laufe der letzten Jahre zu einem nahezu unverzichtbaren und nützlichen Begleiter im Alltag geworden – schnell einen Begriff recherchieren und sich mithilfe einer App oder eines Online-Dienstes in einer unbekannten Stadt orientieren. Die Großzahl der Apps ist jedoch nicht barrierefrei gestaltet und somit nahezu unzugänglich für Blinde und stark Sehbeeinträchtigte. Dieser Artikel richtet sich an Designer sowie Entwickler und erläutert die Accessibility-Funktionalitäten von Android.

Für Blinde und stark Sehbeeinträchtigte sind die visuelle Wahrnehmung und die Aufnahme visueller Informationen nicht gegeben beziehungsweise sehr stark eingeschränkt. Daher können Darstellungen und Inhalte einer Anwendung nicht in der klassischen Art und Weise vermittelt werden. Man benötigt alternative Mechanismen in der Mensch-Computer-Interaktion und muss diese in geeigneter Weise implementieren.

Bei mobilen Endgeräten bieten sich meist die Sprach- und Tonausgabe sowie Vibrationsmechanismen an. Für die Aufbereitung und Wiedergabe von inhaltlichen Informationen stehen in diesem Fall auditives, haptisches und taktiles Feedback zur Verfügung. Bei auditivem Feedback handelt es sich im Wesentlichen um Sprach- und Soundwiedergabe. Unter Haptik versteht man die Berührung und aktive Erkundung von Oberflächen. Dadurch können Formen und oberflächliche Strukturen erkannt werden, etwa die Dimensionen und die Form eines mobilen Endgerätes. Die taktile Wahrnehmung ist unter anderem durch die Empfindung von Temperatur, Festigkeit, Schmerz oder Vibration geprägt, beispielsweise bei einer Auswahlbestätigung durch eine kurze Vibration.

Das haptische Feedback, im Englischen auch „Forced Feedback" genannt, spielt bei der Entwicklung virtueller Realitäten eine bedeutende Rolle und wird zur aktiven Kraftrückkopplung verwendet. Bei Smartphones und Tablets steht hingegen das taktile Feedback im Vordergrund. Daher haben

Version	Accessibility-Funktionalitäten
Donut (1.6), Eclair (2.1), Froyo (2.2), Gingerbread (2.3)	Bereitstellung von Screenreadern („TalkBack", „Spiel", „MobileAccessibility")
Honeycomb (3.0)	Verbesserte Web-Zugänglichkeit
Ice Cream Sandwich (4.0)	Explore by touch (Sprachwiedergabe der Komponenten-Beschreibung) Aktivierung „TalkBack" bei erstem Systemstart möglich Systemweite Schriftgrößen-Anpassung
Jelly Bean (4.1)	Überwachung und Steuerung komplexer Gesten Unterstützung von Braille-Services via „BrailleBack" Zoom von Bildschirm-Ausschnitten Traversieren von Texten Anpassung der Navigation (Fokussieren von Objekten) Ausstattung von Widgets mit logischen Modellen
Jelly Bean (4.3)	Verarbeitung von Tastatureingaben „Copy & Paste"-Funktion Verbesserte Web-Zugänglichkeit

Tabelle 1: Accessibility-Funktionalitäten der Android-Versionen

entsprechende Mechanismen ihren Weg in das Android-API gefunden.

Accessibility in Android

Das Android-API bietet mittlerweile umfangreiche, weit entwickelte Schnittstellen und Konzepte zur Entwicklung barrierefreier Anwendungen an. Da diese Funktionalitäten erst nach und nach über das API bereitgestellt wurden, war die barrierefreie Entwicklung in den Anfangsjahren sehr aufwändig und aufgrund der unterschiedlichen Geräte sehr unzuverlässig. Während bis zur Version 2.3. („Gingerbread") lediglich die Screenreader „TalkBack", „Spiel" und „MobileAccessibility" zur Verfügung standen und mit der Version 3.1. („Honeycomb") nur die Web-Zugänglichkeit verbessert wurde, gibt es seit 4.0 („Ice Cream Sandwich") wesentliche Verbesserungen und Neuerungen im Accessibility-API. Es steht beispielsweise der „Explore by Touch"-Modus zur Verfügung, der View-Komponenten gezielt selektieren kann und eine sprachliche Beschreibung wiedergibt. Die Funktionen stehen in Verbindung mit dem mittlerweile zentralen Screenreader „TalkBack". Dazu später mehr.

Mit der Version 4.1 („Jelly Bean") kamen durch das Accessibility-API weitere sehr nützliche Funktionen, die eine einfache, barrierefreie Entwicklung ermöglichen. So stehen Funktionen zur Gesten-Überwachung und -Steuerung bereit, um auf einfachem Wege neue Gesten zu implementieren. Zudem wird erstmals die Verwendung von Braille-Tastaturen durch das Betriebssystem direkt ermöglicht. Dazu sind Braille-Services via „BrailleBack" in den „TalkBack"-Modus integriert. Dies ermöglicht die Ein- und Ausgabe von Zeichen ohne aufwändige Implementierung und ohne Transformation der Informationen. *Tabelle 1* gibt einen Überblick zu den wichtigsten bereitgestellten Funktionen und Verbesserungen in Android.

Wie eine zusammengefasste Statistik vom August 2014 zur Verbreitung der Android-Versionen zeigt, laufen mehr als 75 Prozent aller Android-Geräte mit einer Version 4.0 oder höher *(siehe Abbildung 1)*. Somit können die meisten Geräte mit barrierefreien Apps arbeiten, die die Standard-Funktionalitäten des API verwenden.

Bedienhilfen „Out of the box"

Es ist nicht viel Aufwand erforderlich, um eine App für die Screenreader oder die Sound- beziehungsweise Vibrationswiedergabe zugänglich zu machen. Werden zur Erstellung der App Standard-Komponenten verwendet, können die von Android bereitgestellten Mechanismen einfach darauf zugreifen. „TalkBack", „SoundBack", „KickBack" und „BrailleBack" sind robuste Schnittstellen zwischen dem Gerät und dem Anwender, wodurch dem Entwickler eine aufwändige Eigenentwicklung erspart bleibt. Bei der Verwendung eigener Nicht-Standard-Komponenten in der View muss aber gegebenenfalls ein eigener Feedback-Mechanismus entwickelt werden. Konkret kann es sich dabei um spezielle Gesten oder Sprachausgaben handeln.

Die zentrale Accessibility-Komponente ist „TalkBack". Die Funktionen sind über das Menü „Bedienhilfen" aktivierbar. Ab der Version 4.0 kann bereits beim ersten Start eines Geräts dessen Aktivierung erfolgen; dadurch verändern sich die Steuerung und der Auswahlprozess des Geräts grundsätzlich. Der Modus sowie dessen spezielle Erweiterungen ermög-

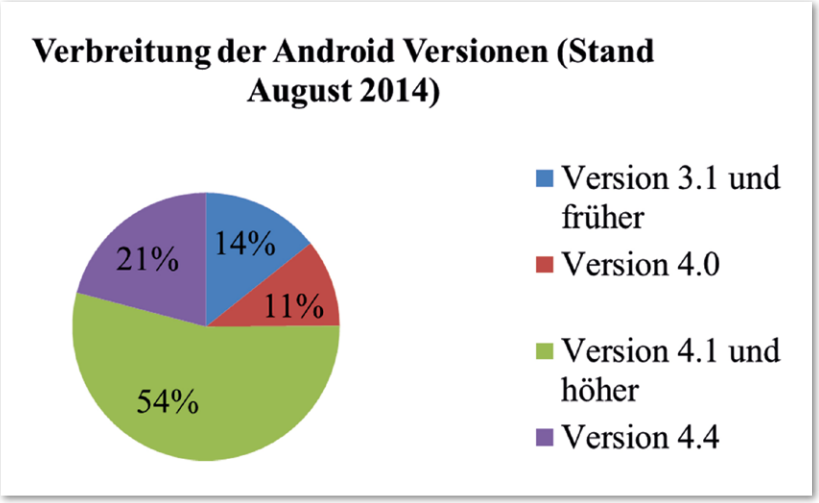


Abbildung 1: Verbreitung der Android-Versionen

```
<Button
  android:id="@+id/play_button"
  android:contentDescription="@string/play"/>
```

Listing 1: „contentDescription" eines View-Elements

```
<EditText
  android:id="@+id/search_criteria"
  android:hint="@string/search_hint"
  android:inputType="text" />
```

Listing 2: „android:hint" eines Textfelds

```
<Button android:id="@+id/focus_button"
  android:focusable="true"
  android:nextFocusUp="@id/edit"
  ... />
```

Listing 3: Setzen von „focusable" im XML-File

```
import android.speech.tts.TextToSpeech;
import android.speech.tts.TextToSpeech.OnInitListener;
…
textToSpeech = new TextToSpeech(context, this);
  textToSpeech.setLanguage(Locale.GERMANY);
textToSpeech.speak("Hallo!", TextToSpeech.QUEUE_FLUSH, null);
…
```

Listing 4: Wiedergabe mittels TTS

lichen Blinden eine selbstständige Bedienung des Android-Geräts über permanentes Feedback des Betriebssystems an den Anwender. Dadurch können verschiedene Einstellungen vorgenommen und einzelne Apps ausgewählt werden. Wie sich eine ausgewählte App nach dem Start verhält, hängt aber im Wesentlichen von deren Implementierung ab. Wurden barrierefreie Elemente nur unzureichend oder gar nicht berücksichtigt, ist die Bedienung durch einen Blinden nicht möglich und die inhaltliche Erschließung sehr schwer bis unmöglich.

Zugänglichkeit von Oberflächenelementen

TalkBack verwendet die „contentDescription" eines View-Elements, um Informationen zum Inhalt und zum Aufbau einer View wiederzugeben *(siehe Listing 1)*. Ist diese für ein Element gesetzt, kann der Screenreader sie auslesen. Das Setzen kann sowohl für Standard- als auch für eigene Komponenten erfolgen. Bei rein optischen View-Elementen sollte diese allerdings nicht gesetzt werden. Dadurch wird verhindert, dass inhaltlich irrelevante Elemente angesprochen werden. Dies könnte bei einem Blinden zu Irritationen und Verwirrung führen, da er selbst die Bedeutung nicht ohne entsprechende Rückmeldungen erfassen kann.

Bei EditBoxen muss für eine entsprechende Funktionalität anstelle der „contentDescription" ein „android:hint" verwendet werden. Dadurch erfolgt bei Auswahl eines Eingabefelds die Wiedergabe von Feldbezeichnung und -inhalt *(siehe Listing 2)*.

Für den „Explore by Touch"-Modus ist es wichtig, dass die View zugängliche Komponenten besitzt. Hierfür muss jede anwählbare Komponente „focusable = true" deklariert haben. Dies kann im XML-File gesetzt sein *(siehe Listing 3)* oder nachträglich im Code geändert werden. Dazu können die Methoden „setFocus", „isFocusable" und „requestFocus" zum Einsatz kommen.

Zur Anwahl eines Elements durch den Anwender verfügt der „Explore by Touch"-Modus über eine spezielle Gesten-Steuerung. Mit einer einfachen Wischgeste wird das nächste fokussierbare Objekt in der View ausgewählt. Die Werte „nextFocusRight", „nextFocusDown", „nextFocusLeft" und „nextFocusUp" verfeinern die Ansteuerung einzelner Elemente.

Sprachausgabe und Sprachsteuerung

Man ist jedoch nicht ausschließlich auf den Screenreader angewiesen, um Komponenten und Inhalte vorlesen zu lassen. Die Text-to-Speech-Schnittstelle (TTS) integriert individuelle Sprachausgaben in die Anwendung. Dazu ist ein Objekt mit einem vorzulesenden Text zu erzeugen und wiederzugeben; die Verwendung von „TalkBack" entfällt. Je nach Zweck kann die Wiedergabe von Text durch eine aktive Interaktion des Anwenders mit der Anwendung erfolgen oder zeitgesteuert ablaufen. Diese Feinheiten sind ganz dem Entwickler überlassen *(siehe Listing 4)*.

Es geht aber auch umgekehrt mit Speech-to-Text (STT). Der SpeechRecognizer wandelt Sprache in Text um. Im Zuge dessen lassen sich unter anderem Sprachkommandos integrieren, die eine Sprachsteuerung der Anwendung erlauben. Der SpeechRecognizer hat jedoch einen entscheidenden Nachteil, er benötigt eine Internet-Verbindung zum Server. Das API stellt damit lediglich eine Service-Schnittstelle zur Verfügung. Ist das mobile Endgerät offline, kann die Funktion nicht verwendet werden.

Eine Alternative dazu ist das Projekt „pocketsphinx". Es wurde für leichtgewichtige mobile Anwendungen entwickelt und ist eines der wenigen Projekte zur Spracherkennung und -steuerung, das auch offline verwendet werden kann. Die Programm-Bibliothek lässt sich in ein Android-Projekt integrieren und ohne Internet-Anbindung ausführen. Die Integration in die App gestaltet sich zunächst etwas aufwändig. Erst mal muss man die Bibliothek überhaupt zum Laufen bekommen, dann sind Grammatiken und das Wörterbuch selbst zu erstellen. Ein Tool unterstützt beim Anlegen des Wortschatzes, indem es die Informationen für „pocketsphinx" lesbar in eine Datei übersetzt.

Allgemeines Oberflächen-Design

Neben der eigentlichen Implementierung und Ausgestaltung des Codes spielt bei der Entwicklung barrierefreier Anwendungen auch das Design der Oberfläche eine bedeutende Rolle. Dabei gilt es, die Verwendung von Freitext-Feldern weitestgehend zu vermeiden, sofern diese nicht zwingend erforderlich sind. Bei dis-

kreten Werten sollten vor allem Checkboxen oder Radiobuttons zum Einsatz kommen. Das unterstützt auch einen normalen Anwender bei der Bedienung der App und vermeidet fehlerhafte Eingaben und Systemzustände.

Generell sollte die Anwendung nicht mit Informationen überladen werden. Zudem ist eine einfache und übersichtliche Navigationsstruktur sehr wichtig. Jede View sollte genau eine Aufgabe erfüllen, etwa die Eingabe eines Suchkriteriums oder das Anlegen eines Kontakts. Um auch Sehbeeinträchtigte bei der Bedienung zu unterstützen, sollten Schriftgrößen individuell vergrößerbar sein und wichtige Informationen zumindest eine angemessene Textgröße besitzen. Darüber hinaus ist die Verwendung von Kontrasten sehr wichtig, um auch Farbfehlsichtigen die inhaltliche Zuordnung einfach zu ermöglichen. Diese Gestaltungsgrundsätze beruhen auf den allgemeinen Forderungen zur ergonomischen und benutzerfreundlichen Gestaltung von Anwendungssoftware.

Fazit

Das Android-API bietet mittlerweile sehr umfangreiche Möglichkeiten zur barrierefreien Entwicklung von Apps. Nur wenige beziehungsweise gar keine Funktionen müssen dafür eigens entwickelt werden. Es ist also nicht notwendig, eine App speziell nur für Blinde zu implementieren.

Bestehende Anwendungen können durch kleine Anpassungen der View-Elemente barrierefrei gestaltet werden, ohne die bisherigen Funktionen einzuschränken. Lediglich bei der Abwärtskompatibilität zu älteren Android-Versionen sind gegebenenfalls Mechanismen eigenständig zu integrieren. Wie die aktuelle Statistik zeigt, kann jedoch der Großteil der Endgeräte die meisten Accessibility-Funktionalitäten verwenden.

Es gibt also keinen Grund, die Barrierefreiheit bei der Entwicklung von Android-Apps außer Acht zu lassen. Das API kommt dem Entwickler sehr weit entgegen und bietet auch dem Anwender für den normalen Einsatz der Anwendung einen echten Mehrwert, wie die Sprachausgabe, die Vorlesefunktionen und die Sprachsteuerung. Ein einfaches und intuitives Oberflächen-Design trägt zudem zur Benutzerfreundlichkeit und Bedienbarkeit bei, was letztendlich allen Anwendern zugutekommt.

Weiterführende Links

1. http://developer.android.com/guide/topics/ui/accessibility/apps.html
2. https://developer.android.com/tools/testing/testing_accessibility.html
3. https://support.google.com/accessibility/android/#topic=6007234
4. http://cmusphinx.sourceforge.net/wiki/tutorialandroid

Mandy Goram
mandy.goram@trevisto.de

Mandy Goram ist eine erfahrene Front- und Back-End-Entwicklerin. Sie arbeitet als IT-Beraterin für die TREvisto GmbH in Nürnberg. Ihre aktuellen Schwerpunkte sind Daten-Analyse und Stammdaten-Management (MDM). Privat beschäftigt sie sich unter anderem mit der Entwicklung von Android Apps und der Thematik "Usability".

http://ja.ijug.eu/15/1/19

Enterprise Apex: Die Neuerungen von Apex 5.0

Application Express (Apex) ist als Teil der Oracle-Datenbank seit mehr als zehn Jahren verfügbar und wird in etlichen IT-Landschaften in großem Stil eingesetzt. Einige Neuerungen von Apex 5.0 eignen sich besonders für „Enterprise Apex".

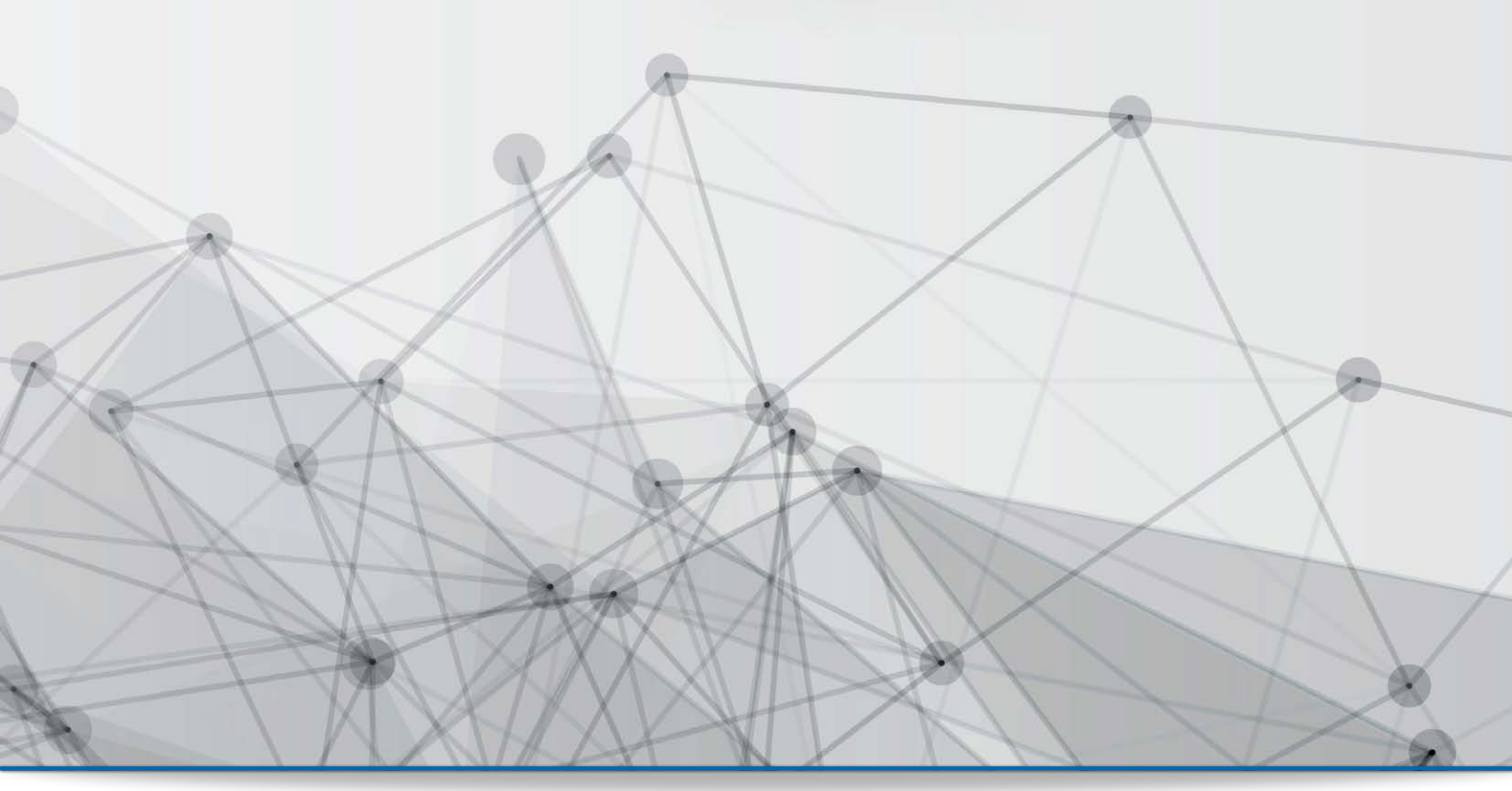
Der Page Designer ist mit Abstand die auffälligste Änderung in der neuen Version und betrifft ausnahmslos alle Apex-Entwickler. Über diese große Neuerung hinaus bringt Apex 5.0 noch eine Menge neuer Funktionen für den Unternehmenseinsatz.

Das Universal Theme und die Theme-Styles erlauben es, einer Apex-Anwendung unterschiedliche „Look & Feels" zuzuweisen, ohne das Theme austauschen zu müssen. Der HTML-Code der Templates wird nicht geändert. Vielmehr wird dem Theme ein Style zugewiesen, der im Wesentlichen aus CSS-Angaben besteht. Mit diesen CSS-Angaben lassen sich Farbe, Schriftart, Linienbreite und mehr ändern. Die Templates bleiben unberührt.

Bisher waren die statischen Dateien eines Apex-Workspaces in Bilder, CSS und sonstige statische Dateien unterteilt. Nun sind sie zusammengefasst. Entwickler können zudem zu jeder Datei einen Verzeichnispfad hinterlegen: Das ist sehr nützlich, wenn mehrere Dateien hochgeladen werden, die sich gegenseitig mit relativen URL-Pfaden referenzieren. Auch können statische Dateien als „.zip"-Dateien hochgeladen und innerhalb von Apex automatisch entpackt werden. Dafür nutzt das Entwickler-Tool das PL/SQL-Paket „Apex_ZIP", das auch das Ein- und Auspacken von ZIP-Archiven in der Datenbank ermöglicht.

Ein weiteres PL/SQL-Paket dürfte nicht nur für Apex-Entwickler hochinteressant sein: „Apex_JSON" gibt dem Anwender eine PL/SQL Schnittstelle zur Arbeit mit JSON-Daten. Diese kann auf 12c-Datenbanken mit der neuen JSON-Funktionalität auf SQL-Ebene kombiniert werden.

Einen umfassenden Einblick in die neuen Funktionen von Apex 5.0 gibt die vom Apex-Entwicklerteam gepflegte Liste der neuen Funktionen *(siehe „https://apex.oracle.com/pls/apex/f?p=65339")* sowie das Statement of Direction *(siehe „http://www.oracle.com/technetwork/developer-tools/apex/application-express/apex-sod-087560.html")*.



Eventzentrische Architekturen

Raimo Radczewski und Andreas Simon, Quagilis

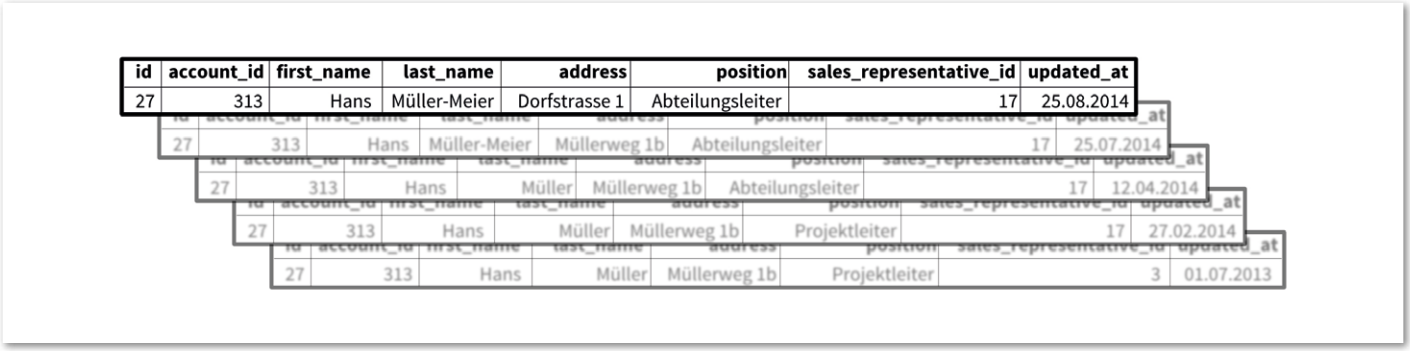
Wertvolle Informationen werden regelmäßig überschrieben und somit vernichtet, wenn wir in unserer Datenhaltung nur den aktuellen Zustand unserer Entitäten speichern. Beim Event Sourcing protokollieren wir die Zustandsübergänge und leiten den Zustand aus der Historie ab. Ergänzt mit Command-Query-Responsibility-Segregation ergibt sich eine gut skalierbare und somit performante Architektur, um flexibel auf den stetigen Wandel von Anforderungen reagieren zu können.

Buchhalter oder Banken notieren seit jeher Änderungen eines Kontos in Buchungssätzen und leiten daraus nur bei Bedarf den Saldo ab. Der Vorteil ist klar: Statt einer Zahl, deren Wert lediglich von Transaktion zu Transaktion geändert wird, erhält der Kontobesitzer darüber Aufschluss, welche Einnahmen und Ausgaben wann auf seinem Konto verbucht wurden.

Diese Technik auf ein Anwendungssystem zu übertragen, bedeutet für die Datenbank eine Radikalkur: Sie wird zu einer Tabelle, in der sämtliche Ereignisse in chronologischer Reihenfolge gespeichert sind. Das System wird darauf ausgelegt, allein aus diesen Ereignissen seinen aktuellen Zustand abzuleiten. Fehler können nun bis zu ihrem Ursprung – nämlich einer eigentlich ungültigen Folge von Ereignissen in der Domäne – zurückverfolgt werden. Das Entwicklungsteam kann historische Zustände erkunden und feststellen, unter welchen Bedingungen ein Fehler aufgetreten ist.

Für das Geschäft birgt das Speichern von Änderungen anstelle von Zuständen neue Möglichkeiten: Wie bei einem Konto die einzelnen Buchungen, so können Ereignisse innerhalb eines Domain Model die Intention oder den Grund der Änderung erfassen. Wenn beispielsweise die Adressdaten eines Kunden geändert werden, kann es sich um die Korrektur eines Tippfehlers handeln oder der Kunde ist umgezogen. Diese Ursachen dokumentieren wir mit unterschiedlichen Domain Events („CustomerHasMoved“ und „CustomersAddressWasCorrected“). Auf die verschiedenen Ursachen lässt sich nun unterschiedlich reagieren.

Zur Illustration modellieren wir nachfolgend ein CRM-System. Typischerweise werden diese Applikationen als CRUD-Anwendungen implementiert. *Abbildung 1* zeigt eine typische Kontakt-Tabelle in einem solchen CRM-System. Durch den Einsatz von Domain Events speichern wir, wie in *Abbildung 2* gezeigt, nur die geänderten Daten sowie die Ursache und den Zeitpunkt. Diese Zusatz-Informationen können später für interessante Analysen oder neue Funktionalitäten verwendet werden. So sind Reporting und eine ausführliche Historie typische Funktionalitäten von CRM-Systemen, die wir mit CQRS und Event Sourcing von vornherein im Kern unserer Anwendung un-



id	account_id	first_name	last_name	address	position	sales_representative_id	updated_at
27	313	Hans	Müller-Meier	Dorfstrasse 1	Abteilungsleiter	17	25.08.2014
27	313	Hans	Müller-Meier	Müllerweg 1b	Abteilungsleiter	17	25.07.2014
27	313	Hans	Müller	Müllerweg 1b	Abteilungsleiter	17	12.04.2014
27	313	Hans	Müller	Müllerweg 1b	Projektleiter	17	27.02.2014
27	313	Hans	Müller	Müllerweg 1b	Projektleiter	3	01.07.2013

Abbildung 1: Die Änderungen, die ein Datensatz im Laufe der Zeit durchläuft, können wertvolle Informationen liefern

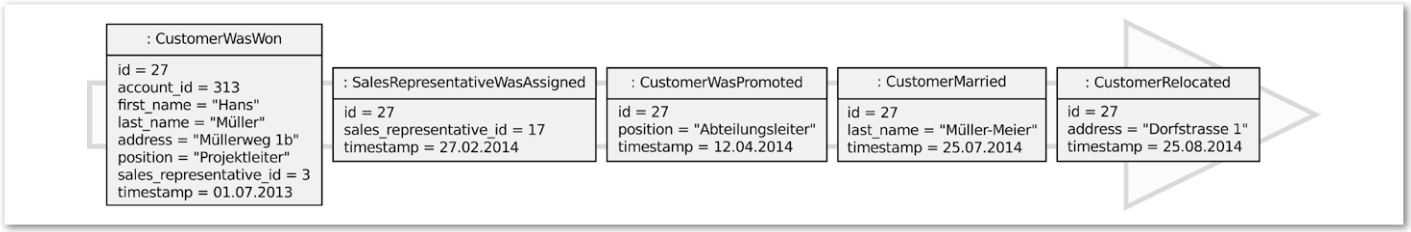


Abbildung 2: Durch Domain Events können wir Ursachen für eine Zustandsänderung explizieren

terstützen und dem Anwender später auf einfache Weise bereitstellen können.

Buchhaltung für Objekte – Event Sourcing (ES)

Wenn wir ein Domain Model mit Event Sourcing implementieren, muss jede Zustandsänderung durch ein „Domain Event" ausgelöst werden. Beispielsweise wird die Adresse eines „Customer" nicht direkt manipuliert, sondern ein „AddressWasCorrected"-Event emittiert *(siehe Listing 1)*. Dadurch wird das neue Event persistiert und anschließend der Event Handler „apply(AddressWasCorrected event)" des Aggregats aufgerufen. Erst hier wird dann der Zustand des Customer mittels „setAddress(event.getNewAddress())" geändert.

Um ein Aggregat wiederherzustellen, wird die Liste aller von dem Aggregat in der Vergangenheit erstellten Domain Events geladen. Dann werden die Domain Events in ihrer chronologischen Reihenfolge angewendet. So durchlebt das Aggregat noch einmal seine Entstehungsgeschichte. Dabei können wir auch an einem früheren Zeitpunkt anhalten und so einen historischen Zustand rekonstruieren.

Zur Adressänderung verfügt „Customer" über zwei Methoden, „correctMistakeInAddress(...)" und „noteRelocationToNewAddress(...)". Indem wir unterschiedliche Domain Events („AddressWasCorrected" beziehungsweise „CustomerRelocatedToNewAddress") erzeugen, können wir später die fachlichen Ursachen für eine Adressänderung unterscheiden. Wenn ein Kunde umzieht, können wir ihm nun angemessene Unterstützung anbieten. Das wollen wir vermeiden, wenn die Adresse nur korrigiert wurde.

Die Domain Events sind in einem sogenannten „Event Store" gespeichert. Dieser enthält für jedes Domain Event zumindest drei Informationen: „AggregatId", „SequenceNumber" und das serialisierte Ereignis als „Payload". Über „AggregatId" ist jedes Ereignis eindeutig einem Aggregat zugeordnet. „SequenceNumber" ist fortlaufend für „AggregatId". Da die Kombination aus beiden eindeutig ist, wird sichergestellt, dass ein Ereignis kein anderes, in der Zwischenzeit persistiertes Ereignis überschreibt oder überschattet („Lost Update").

Die Schnittstelle eines Event Stores ist genauso simpel wie die Datenstruktur der Events: Da Ereignisse lediglich angehängt werden, genügt zum Speichern „appendEvent(...)" und zum Abfragen die Methode „getEventsOf(AggregatId id)", die die Domain Events einer Aggregatsinstanz lädt *(siehe Listing 2)*.

Bei jedem Zugriff sind alle Ereignisse des Aggregats zu laden und zu verarbeiten. Deshalb wird mit steigender Anzahl von Ereignissen je Aggregat das Laden zwangsläufig langsamer. Abhilfe schaffen (flüchtige) Snapshots, die den Zustand eines Aggregats zu einem durch die „SequenceNumber" eindeutig bestimmten Zeitpunkt zwischenspeichern. Zur weiteren Optimierung lassen sich die Ereignisse anhand „AggregatId" partitionieren, da in der Regel nur die Ereignisse zu einem einzelnen Aggregat geladen werden müssen.

Teile und herrsche – Command-Query-Responsibility-Segregation (CQRS)

Üblicherweise kommt in Applikationen ein objektorientiertes Domain Model zum Einsatz, das in mehr oder weniger großem Umfang Geschäftsregeln implementiert. Die Entitäten aus dem Modell sind in einer relationalen Datenbank gespeichert; die Übersetzung in beide Richtungen übernimmt ein objektrelationaler Mapper (ORM) wie zum Beispiel Hibernate. Ein solcher Entwurf führt zu Problemen, wie ein kleines Beispiel zeigt: In unserem fiktiven CRM-System ist jedem Kunden („Customer") genau ein Außendienstmitarbeiter („SalesRepresentative") zugeordnet *(siehe Abbildung 3)*.

Bei der Konfiguration des ORM müssen wir uns entscheiden, wie die Assoziation aufzulösen ist. Wenn der Name des „Customer" geändert werden soll, ist kein Zugriff auf den „SalesRepresentative erforderlich". Um möglichst wenige Daten aus der Datenbank zu lesen, empfiehlt es sich, die Assoziation nur bei Bedarf („lazy") aufzulösen. Beim Anzeigen einer Liste aller „Customer" mit den zugeordneten „SalesRepresentatives" muss die Assoziation aufgelöst wer-

```
public class Customer extends Aggregate<String> {
  private String address;
  private String assignedSalesRepresentative;

  public void correctMistakeInAddress(String newAddress) {
      emit(new AddressWasCorrected(this.getId(), newAddress, new
Date()));
  }

  public void noteRelocationToNewAddress(String newAddress) {
      emit(new CustomerRelocatedToNewAddress(this.getId(), newAd-
dress, new Date()));
  }

  public void wonByRepresentative(String salesRepresentativeId) {
      emit(new CustomerWasWon(this.getId(), salesRepresentativeId,
          new Date()));
  }

  private void apply(AddressWasCorrected event) {
      setAddress(event.getNewAddress());
  }

  private void apply(CustomerRelocatedToNewAddress event) {
      setAddress(event.getNewAddress());
  }

  private void apply(CustomerWasWon event) {
      setAssignedSalesRepresentative(event.salesRepresentativeId);
  }
}
```

Listing 1

```
public interface EventStore {
  void appendEvent(DomainEvent e);
  List<DomainEvent> getEventsOf(AggregatId id);
}
```

Listing 2

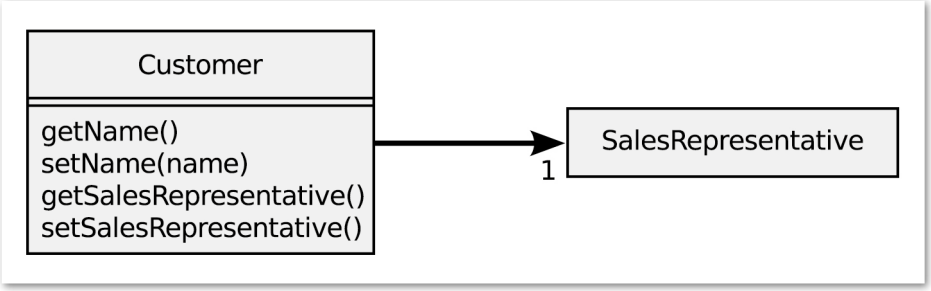


Abbildung 3: „lazy" oder „eager" Fetching?

den. Um die Anzahl der Datenbank-Abfragen zu minimieren, empfiehlt es sich nun, die Assoziation beim Laden des Customer immer („eager") aufzulösen.

Schon bei diesem einfachen Fall gibt es also keine optimale Lösung für alle Anwendungsfälle. Dieses Problem wird im täglichen Einsatz noch verschärft. Domain Models enthalten hier deutlich mehr als zwei Entitäten, es gibt eine Reihe von Assoziationen mit „m:n"-Kardinalitäten und eine Vielzahl von Funktionalitäten, die lesend oder schreibend mit den Daten arbeiten. Grundsätzlich können wir mit nur einem Modell keine optimale Lösung für Transaktionsverarbeitung, Berichterstellung, Suche und Analyse finden. Aufgrund dieser Einsicht unterteilt „Command-Query-Responsibility-Segregation" (CQRS) eine Anwendung in zwei Verantwortungsbereiche *(siehe Abbildung 4)*. Es gibt einen Command-Teil, der sich nur um die Transaktionsverarbeitung kümmert, und einen Query-Teil, der nur für lesende Zugriffe wie Berichte und Suche zuständig ist.

Das Command-API startet die Transaktionen. Dazu werden der relevante Teil des Domain Model (auch als „Command Model" oder „Write Model" bezeichnet) aus der Datenbank geladen, entsprechende Methoden des Domain Model aufgerufen und so dessen Zustand geändert. Schließlich wird dieser Zustand wieder in die Datenbank geschrieben. Das Domain Model lässt sich gezielt dahingehend entwerfen, dass mit geringstmöglichem Datentransfer die Einhaltung der Geschäftsregeln sichergestellt ist. Zu diesem Zweck sind üblicherweise nur kleine Objekt-Graphen erforderlich.

Im Anschluss an jede erfolgreiche Transaktion wird ein Domain Event veröffentlicht. Ein „Event Bus" verteilt die Domain Events im System. Je nach dessen Ausgestaltung kann dies synchron (für unverzügliche Konsistenz) oder asynchron (für höhere Performance) geschehen. Empfangen werden die Domain Events von Event Listeners, die die sogenannten „Query Models" aktualisieren.

Query Models (auch als „Read Models" bezeichnet) sind Datenstrukturen, die auf die Anforderungen der UI zugeschnitten sind. Für jede einzelne Ansicht erstellen wir ein neues Query Model. Soll eine Masteransicht aller „Customer" mit den zugeordneten „SalesRepresentatives" erstellt werden, wird eine Datenbank-Tabelle mit den erforderlichen Feldern angelegt und befüllt *(siehe Abbildung 5)*. Auch bei der Detailansicht eines „Customer" wird eine eigene Datenbank-Tabelle angelegt und befüllt. Diese Tabellen sind denormalisiert und enthalten in jeder Zeile alle notwendigen Daten. Nun sind keine Joins oder Transformationen notwendig, Lesevorgänge sind sehr schnell.

Zudem ist in den Query Models der Kontext berücksichtigt. Besteht die Anforderung, dass Nutzer mit unterschiedlichen Berechtigungsstufen unterschiedlich viel Informationen einsehen können, so legen wir für jede Berechtigungsstufe ein Query Model an, das die gewünschten Informationen enthält. Auch ein differenzierter Umgang mit Löschmarkierungen ist möglich. Während ein normaler Nutzer gelöschte Entitäten nicht mehr einsehen kann, ist dies für einen Administrator noch möglich.

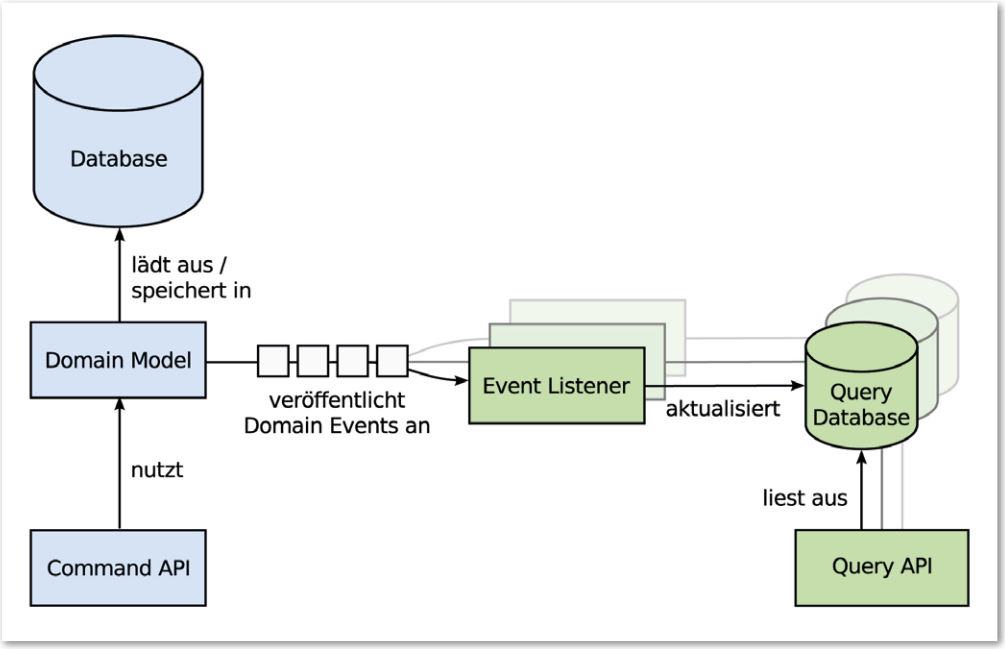


Abbildung 4: Command-Query-Responsibility-Segregation unterscheidet zwischen dem Command Model (blau) und den Query Models (grün)

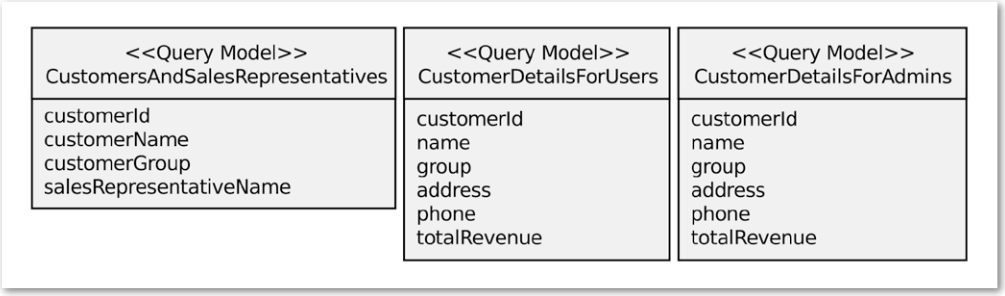


Abbildung 5: Für jeden Bericht entsteht ein maßgeschneidertes Query Model

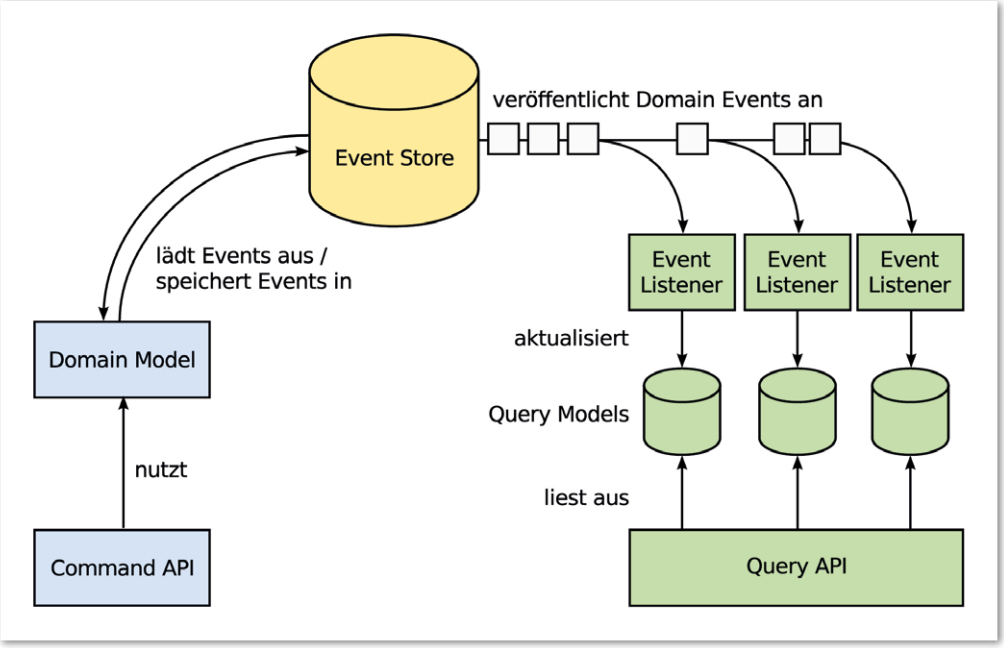


Abbildung 6: Der Event Store ist Bindeglied bei der Kombination von Event Sourcing und CQRS

Die Entkopplung durch Domain Events bietet Freiheiten in der Wahl der Persistenz-Technologie. Während für einen Großteil der Anwendungsfälle eine relationale Datenbank eine gute Wahl darstellt, lässt sich in einem Query Model für organisatorisch bedingte Autorisierungsstrukturen eine Graphen-Datenbank verwenden. Für Suchfunktionen wird in einem weiteren Query Model ein Suchserver wie Solr oder Elastic Search eingebunden. Bei sehr hohen Performance-Anforderungen können Teile des GUI (etwa HTML-Bausteine oder dynamische Grafiken) vorab gerendert und in einem Key-Value-Store abgelegt werden. So stehen die Daten mit geringer Latenz bereit.

Gemeinsam sind wir stark: ES + CQRS

Obwohl Event Sourcing und CQRS eigenständig implementiert werden können, ist es sinnvoll, beide Patterns zu kombinieren *(siehe Abbildung 6)*. Dadurch lassen sich die Vorteile beider Ansätze nutzen, während ihre Einschränkungen durch das jeweils andere Pattern aufgehoben werden.

In einer Gesamtarchitektur ist der Event Store das zentrale Synchronisationselement. Das Domain Model wird auf Basis von Event Sourcing implementiert, die Domain Events, die die Historie der Entitäten ausmachen, werden aus dem Event Store geladen und auch wieder hier persistiert. Die Publikation der Domain Events übernimmt nun nicht mehr das Domain Model. Stattdessen veröffentlicht der Event Store neue Domain Events, nachdem diese persistiert wurden.

Da dieselben Domain Events nun sowohl die Persistierung des Domain Model als auch die Aktualisierung der Query Models übernehmen, ist die Konsistenz des Domain Model und der Query Models sichergestellt. Der Aufbau neuer Query Models ist enorm vereinfacht, da nur die Verarbeitung der Events implementiert werden muss. Zum initialen Befüllen werden alle Events aus dem Event Store abgespielt und im neuen Query Model verarbeitet. Jedes Query Model stellt nun eine Projektion des Event Stores dar.

Die Aktualisierung der Query Models erfolgt voneinander unabhängig. Daher lassen sich die Query Models beliebig verteilen und somit horizontal skalieren. Man kann jedes Query Model eigenständig auf einem oder parallel auf mehreren Servern betreiben. Die Vermittlung der Domain Events erfolgt effizient über einen Message Bus.

Diese Architektur erleichtert auch die Arbeitsteilung innerhalb eines Teams oder sogar zwischen mehreren Teams. Die Entwicklung des Domain Model kann von der Erstellung der Query Models getrennt erfolgen. Schnittstelle zwischen den beiden Bereichen sind die definierten Domain Events. Auch die Erstellung der Query Models ist unabhängig davon.

Fazit

Die vorgestellten Ansätze erfordern eine eingehende Analyse und Definition der Domain Events. Einmal definierte Domain Events müssen über die gesamte Lebenszeit der Anwendung verarbeitet werden können. Diese Bedingungen erhöhen im Vergleich zu einer CRUD-Applikation den Entwicklungsaufwand. Besonders in komplexen Domänen fördern Event Sourcing und CQRS allerdings die Entwicklung eines Domain Model, das die fachlichen Anforderungen bestmöglich umsetzt. Durch die Domain Events bleiben außerdem Informationen erhalten, die in CRUD-Anwendungen verloren gehen.

Die Trennung in Domain Model und Query Models bietet große Freiheiten in der Umsetzung neuer Anforderungen auf Basis bestehender Daten. Bei deren geschickter Nutzung entstehen Vorteile für Performance, Skalierbarkeit und UI-Design.

Referenzen

1. https://github.com/andreassimon/eventzentrische-architekturen
2. Vaughn Vernon: Implementing Domain-Driven Design
3. http://cqrs.wordpress.com/documents
4. http://www.axonframework.org
5. http://www.infoq.com/presentations/greg-young-unshackle-qcon08
6. http://www.heise.de/developer/artikel/CQRS-neues-Architekturprinzip-zur-Trennung-von-Befehlen-und-Abfragen-1797489.html
7. http://www.heise.de/developer/artikel/Persistenz-mit-Event-Sourcing-1974051
8. http://martinfowler.com/bliki/CQRS.html
9. http://martinfowler.com/eaaDev/DomainEvent.html
10. http://martinfowler.com/eaaDev/EventSourcing.html
11. http://www.udidahan.com/2009/12/09/clarified-cqrs
12. http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing

Raimo Radczewski
raimoradczewski@googlemail.com

Andreas Simon
a.simon@quagilis.de

Raimo Radczewski ist Wirtschaftsinformatiker und agiler Software-Entwickler. Andreas Simon unterstützt Teams und Unternehmen als freiberuflicher Scrum-Master und Coach für Software-Qualität.

http://ja.ijug.eu/15/1/20

Community-Veranstaltungen im Trend

Mit einer Rekordbeteiligung von rund 1.600 Teilnehmern fand am 20. und 21. Juli 2014 zum 17. Mal das Java Forum in Stuttgart statt. Die Java User Group Stuttgart hatte 49 Vorträge in sieben parallelen Tracks organisiert. Zudem waren 40 Aussteller vor Ort, darunter auch der Interessenverbund der Java User Groups e.V. (iJUG). Abends gab es die Gelegenheit, sich bei verschiedenen BoF-Sessions (Birds of a Feather) mit Gleichgesinnten zu treffen, um über ein bestimmtes Thema zu diskutieren und sich auszutauschen.

„Der jedes Jahr wiederkehrende Event bietet eine hervorragende Gelegenheit, um Kollegen zu treffen und seine Kontakte in der Community zu pflegen", sagt Dr. Michael Paus, 1. Vorsitzender der Java User Group Stuttgart. „Durch den lokalen Bezug kennen sich viele Teilnehmer seit Jahren."

Das Motto „Klein, aber fein" galt für die Source Talk Tage 2014 am 26. und 27. August im Mathematischen Institut der Universität Göttingen. Studierende aus ganz Deutschland nahmen an den angebotenen Trainings teil, die Entwickler kamen vor allem aus der Region. Der Social Event am Abend des ersten Tages entwickelt sich immer mehr zum Klassentreffen der Java-Entwickler in der Gegend, an dem dann auch diejenigen teilnehmen, die wegen Projektzwängen nicht zu den Vorträgen kommen können.

„Die Source Talk Tage schaffen ein Forum für Studierende aus ganz Deutschland und potenzielle Arbeitgeber, um sie auf einer technischen Ebene jenseits von Jobmessen zusammenzufinden. Die Vortragenden zeigen, mit welch interessanten technischen Projekten sich ihre Firma beschäftigt und welche Lösungsmöglichkeiten sie gefunden haben. Die Techniker werden damit zu Botschaftern ihres Unternehmens – ein einmaliges Konzept", erläutert Stefan Koospal, Vorsitzender der Sun User Group Deutschland e. V.

„Von User Groups organisierte Fachkonferenzen sind voll im Trend und erfreuen sich stark steigender Besucherzahlen", ergänzt Fried Saacke, Vorstandsvorsitzender des iJUG. „Mit Java Forum Stuttgart, JavaLand, Source Talk Tage und Berlin Expert Days sind die führenden deutschen Java-Konferenzen von der Community organisiert."

„Spiel, Spaß, Spannung und ab und zu auch Arbeit …“

Usergroups bieten vielfältige Möglichkeiten zum Erfahrungsaustausch und zur Wissensvermittlung unter den Java-Entwicklern. Sie sind aber auch ein wichtiges Sprachrohr in der Community und gegenüber Oracle. Wolfgang Taschner, Chefredakteur Java aktuell, sprach darüber mit Jochen Stricker, dem Leiter der Java User Group Augsburg.

Was hat dich motiviert, die Java User Group Augsburg zu gründen?
Stricker: Ich war von dem Konzept, das hinter einer Java User Group steckt, schon immer überzeugt: „Wissen, Ideen und Gedanken auszutauschen, Gleichgesinnte kennenzulernen …“ Schon vor rund zehn Jahren habe ich mit den ersten Gedanken gespielt, eine lokale Java User Group zu gründen, habe aber leider in meinem damaligen Umfeld keine Unterstützung gefunden. Mitte Oktober 2009 startete ich dann zusammen mit Oliver Becherer einen zweiten Versuch und wir haben dann die Java User Group Augsburg gegründet. Durch ein gezieltes Schreiben an lokale Firmen und Entwickler konnten wir nach nur einem Monat schon fünfzig Mitglieder verzeichnen.

Wie ist die Java User Group Augsburg organisiert?
Stricker: Die JUG Augsburg ist kein eingetragener Verein und somit gibt es bei uns auch keine Mitgliedschaft. Einfach gesagt, wir sind ein kleiner Haufen, der sich regelmäßig zu Vorträgen und zum Erfahrungsaustausch trifft. Jeder, der Lust und Zeit hat, kann bei uns mitmachen, egal ob Profi oder Anfänger. Organisiert und geleitet wird die JUG Augsburg inzwischen von einem dreiköpfigen Team.

Welche Ziele hat die Java User Group Augsburg?
Stricker: Wir wollen in Augsburg und Umgebung ein Anlaufpunkt für alle Java-Interessierten sein, die sich regelmäßig treffen und Erfahrungen austauschen möchten. Wichtig ist auf jeden Fall, dass die JUG Augsburg nicht nur eine Eintagsfliege ist, sondern längerfristig bestehen bleibt.

Wie viele Veranstaltungen gibt es pro Jahr?
Stricker: Die JUG Augsburg veranstaltet im Jahr etwa zehn Vortragsabende, die in der Regel am Donnerstag stattfinden. Neben den Vorträgen versuchen wir auch immer wieder, kostenlose eintägige Workshops zu organisieren.

Was bedeutet Java für dich?
Stricker: Java begleitet mich seit 14 Jahren, und das nicht nur beruflich, sondern auch als Hobby im privaten Umfeld. Kurz gesagt: Spiel, Spaß, Spannung und ab und zu auch Arbeit.

Welchen Stellenwert besitzt die Java-Community für dich?
Stricker: Von der Java-Community im deutschsprachigen Raum bin ich sehr begeistert. Wenn ich mir so die letzten Jahre anschaue, dann hat sich sehr viel getan. Es sind viele neue User Groups entstanden, auch besonders sehr aktive Gruppen. Dank der Community gibt es die Möglichkeit, sich direkt mit Gleichgesinnten auszutauschen und von den Erfahrungen anderer zu profitieren.

Wie sollte sich Java weiterentwickeln?
Stricker: Dank Java 8 ist die Weiterentwicklung auf einem guten Weg. Dennoch besitzt Java immer noch ein großes Potenzial, egal ob im Desktop-, Server- oder Mobile-Bereich.

Wie sollte Oracle deiner Meinung nach mit Java umgehen?
Stricker: Java ist durch die Community groß geworden. Sie ist also ein wesentlicher Bestandteil des Erfolgs und dem sollte Oracle Rechnung tragen. Deshalb sollte Oracle noch offener und transparenter mit Java umgehen, also frühzeitig die Strategien kommunizieren.

Wie sollte sich die Community gegenüber Oracle verhalten?
Stricker: Die Community sollte immer wieder das Gespräch mit Oracle suchen. Eine bessere Möglichkeit, an Java mitzuarbeiten, gibt es nicht. Dafür existieren verschiedene Kanäle wie der Interessenverbund der Java User Groups e.V. (iJUG).

Jochen Stricker
jochen.stricker@jug-augsburg.de
www.jug-augsburg.de

Jochen Stricker ist „Staatlich geprüfter Wirtschaftsinformatiker“ und arbeitet seit rund vierzehn Jahren im Java-Umfeld. In den letzten Jahren lag sein Hauptschwerpunkt im JEE-Umfeld. Zurzeit ist er als Freiberufler in ganz Deutschland unterwegs. In seiner Freizeit leitet er die Java User Group Augsburg.

http://ja.ijug.eu/15/1/21

So wird Testen groovy

Kai Spichale, adesso AG

Spock ist ein schlankes Test- und Spezifikations-Framework für Java- und Groovy-Applikationen. Seine Besonderheit ist eine Groovy-DSL, um Tests übersichtlich zu strukturieren. Dank seiner Mocking-Funktionen lassen sich neben zustandsbasierten auch verhaltensbasierte Tests entwickeln. Von diesen Möglichkeiten können auch Java-Entwickler leicht Gebrauch machen, weil die Einstiegshürden für Groovy vergleichsweise niedrig sind.

Java ist eine beliebte und weitverbreitete Sprache. Warum sollte man deshalb Groovy einsetzen? Die Groovy-Entwickler argumentieren, dass Java in die Jahre gekommen sei. Frühe Entwicklungsentscheidungen erscheinen aus heutiger Sicht suboptimal [1]. Groovy-Code ist fast immer übersichtlicher und einfacher als ein Pendant in Java. Beispielsweise ist die Behandlung von Strings, Collections sowie XML und JSON mit Groovy viel einfacher als mit Java. Meta-Programmierung, funktionale Programmierung, Operator-Überladung und wahlweise dynamische Typisierung machen Groovy sehr leistungsstark.

Groovy läuft auf der JVM und ist deshalb nahtlos in konventionelle Java-Projekte integrierbar. Aus technischer Sicht ist es einfach nur eine weitere Java-Bibliothek im Klassenpfad. Zudem sind aufgrund der Ähnlichkeit die Einstiegshürden für Java-Programmierer relativ gering.

Das heutige Einsatzgebiet von Groovy reicht von kleinen Skripten bis hin zu komplexen Web-Applikationen mit dem Web-Framework Grails. Beliebt ist Groovy auch zur Entwicklung domänenspezifischer Sprachen wie beim Build-Management-Werkzeug Gradle oder bei der Groovy-Bean-Konfiguration im Spring-Framework 4.

Ein ganz anderes Anwendungsgebiet, das zugleich Thema dieses Artikels ist, sind automatisierte Tests für Java- und Groovy-Applikationen. In diesem Zusammenhang kommt das Spock-Framework zum Einsatz. Das Testen ist zugleich eine gute Möglichkeit für ein Entwicklungsteam, Groovy auszuprobieren und Erfahrung zu sammeln.

Ein JUnit-Test wird mit Groovy genauso implementiert wie mit Java, indem man „public void"-Methoden als Testfälle mit der Annotation „@Test" markiert. Als Alternative bietet Groovy in seiner Standardbibliothek die Klasse „groovy.util.GroovyTestCase". Über diese werden durch Vererbung zusätzliche Assertions bereitgestellt, die JUnit standardmäßig nicht bietet. Groovy-TestCases lassen sich sehr einfach mit der Kommandokonsole ausführen; IDE-Unterstützung ist in der Regel kein Problem. Trotz dieser Möglichkeiten hat sich das Spock-Framework durchgesetzt.

Das Spock-Framework

Der Name des Frameworks stammt nicht aus der Fernsehserie „Star Trek", sondern ist ein Mix aus „specification" und „mock", beides bekannte Begriffe aus der testgetriebenen und verhaltensgetriebenen Entwicklung. Das Spock-API ist leicht verständlich und geprägt von einer ausdruckstarken Groovy-DSL. Anhand dieser werden die Testmethoden, hier „Feature-Methoden" genannt, in Blöcke eingeteilt, um die Tests übersichtlich zu strukturieren. Ein Block beginnt mit einem Label und erstreckt sich bis zum Beginn des folgenden Blocks beziehungsweise dem Ende der Feature-Methode.

Das Spock-Framework kennt die Blöcke „setup", „when", „then", „expect", „cleanup" und „where". Die Blöcke sind bestimmten Testphasen zugeordnet und werden in definierter Reihenfolge ausgeführt. In der ersten Phase wird der gleichnamige „setup"-Block ausgeführt, um die zu testenden Objekte zu erzeugen und für den Test vorzubereiten. In der folgenden Stimulus-Phase kommt der „when"-Block zur Ausführung, in der Response-Phase dann der „expect"-Block und in der abschließenden Aufräum-Phase der „cleanup"-Block.

Alternativ zum „when"- und „then"-Block kann ein „expect"-Block erfolgen, der die Stimulus- und Response-Phase zusammenfasst. Der „where"-Block definiert Testdaten, die iterativ durch Mehrfachausführung der Feature-Methode angewandt werden. Weitere Details zur Strukturierung und zur Begriffswelt der Spock-Tests stehen in der Spock-Dokumentation [2].

Alle Spock-Testklassen erweitern die Klasse „spock.lang.Specification" und können wie gewöhnliche JUnit-Tests ausgeführt werden. Dies hat den Vorteil, dass die JUnit-

```
import spock.lang.Specification
import spock.lang.Stepwise

@Stepwise
class MathUtilSpec extends Specification
{
  MathUtil math
  def setup() {
     math = new MathUtil()
  }
def "0 times 5 equals 0"() {
     setup: MathUtil math = new MathUtil()
     when: int result = math.multiply(0,5)
     then: result == 0
  }
  def "2 times 5 equals 10"() {
     expect: math.multiply(2,5) == 10
  }
}
```

Listing 1

Test-Infrastruktur wiederverwendet werden kann, ohne zusätzliche Werkzeuge oder spezielle Plug-ins installieren oder konfigurieren zu müssen.

Listing 1 zeigt einen einfachen Spock-Test zur Überprüfung der Klasse „MathUtil". Per Konvention enden alle Spock-Tests auf „Spec", um zwischen Spock- und anderen JUnit-Tests unterscheiden zu können. Die Feature-Methode „0 times 5 equals 0" enthält einen „setup"-Block, der das zu testende Objekt „math" erzeugt. Im „when"-Block wird die Methode „multiply()" ausgeführt und das Ergebnis im „then"-Block überprüft.

Die zweite Feature-Methode „2 times 5 equals 10" hat eine andere Struktur, denn das getestete Objekt wird in der Methode „setup()" erzeugt. Ein weiterer Unterschied besteht im Aufbau der Feature-Methode, denn diese besteht nur aus dem „expect"-Block. Die abweichende Struktur der Feature-Methode wurde gewählt, um zu zeigen, wie flexibel die DSL eingesetzt werden kann.

```
import spock.lang.Specification
import spock.lang.Unroll
class InterestCalculatorSpec extends
Specification {

InterestCalculator calc

  @Unroll
  def „future value of #value value and
  #rate rate after #years years“() {
      setup:
InterestCalculator calc = new Interest-
Calculator()
      expect:
      calc.computeCompoundedInterest(value,
      rate, years) == futureValue

      where:
      rate | years
      0.02 | 1
      0.02 | 2
      0.04 | 3

      value << [100, 100, 200]

      futureValue = value*(1+rate)**years
  }
}
```

Listing 2

```
def „send email after booking“() {
  EmailService emailService = Mock()
  BookingService bookingService = new BookingService(„emailService“: emailSer-
vice)

  when: bookingService.book(user, item)

  then:1*emailService.send(email)
}
```

Listing 3

Die Ausdrücke im „expect"- und „then"-Block müssen „true" sein, um den Test zu bestehen. Zahlen ungleich Null, nicht leere Strings und Nicht-Null-Referenzen werden von Groovy standardmäßig als „true" interpretiert.

Wie das Beispiel in *Listing 1* zeigt, sind die Test-Namen als Literale definiert. Schwer lesbare Methoden-Namen mit beispielsweise Camel-Case-Konvention entfallen. Mit den Annotationen „@Title" und „@Issue" können weitere Informationen angegeben werden, die insbesondere für den generierten Testreport relevant sind.

Die Annotation „@Stepwise" führt Testmethoden in einer bestimmten Reihenfolge aus. Obwohl Tests in der Regel unabhängig und in beliebiger Reihenfolge ausführbar sein sollten, gibt es Fälle, in denen dieses Feature nützlich ist – beispielsweise dann, wenn das zu testende Verhalten zustandsbasiert ist und das Zurücksetzen oder Initialisieren des Zustands zu aufwändig wäre.

Neben der Annotation „@Ignore", die aus JUnit übernommen wurde, bietet das Spock-Framework ebenfalls „@IgnoreIf", um Feature-Methoden nur unter bestimmten Bedingungen auszuführen, sowie „@IgnoreRest", um alle nicht mit dieser Annotation gekennzeichneten Feature-Methoden zu deaktivieren. Im „then"-Block kann mit den Methoden „thrown()" und „notThrown()" überprüft werden, ob Exceptions im „when-Block" wie erwartet geworfen wurden.

Datengetriebene Tests

Für Äquivalenzklassen-Tests und Grenzwert-Analysen muss der Testcode mehrfach mit unterschiedlichen Testdaten ausgeführt werden. Bei konventionellen JUnit-Tests sind Testcode und Testdaten nicht klar voneinander getrennt, sodass der Testcode üblicherweise dupliziert wird, um beispielsweise zusätzliche Äquivalenzklassen abzudecken.

Für diese Art von Aufgaben bietet das Spock-Framework verschiedene Techniken, die in *Listing 2* vorgestellt sind. Beispielhaft wird das Verhalten der Klasse „InterestCalculator" getestet. Das zu testende Objekt wird im „setup"-Block erzeugt und der berechnete Zinseszins im „expect"-Block mit den erwarteten Werten verglichen. Im „where"-Block werden die Testdaten definiert, sodass diese sauber vom Rest des Tests getrennt sind. Der Block enthält eine zweispaltige Datentabelle. Für jede Zeile dieser Tabelle wird die Feature-Methode einmal ausgeführt. Die Spaltennamen „rate" und „years" sind gleichzeitig die Variablen-Namen, mit denen die Testdaten referenziert sind.

Datentabellen sind nicht die einzige Möglichkeit, um Testdaten bereitzustellen, denn diese sind nur eine besondere Form der Data Pipes. Die Variable „value" wird mithilfe des Shift-Operators „<<" an einen Data Provider gebunden. Im Beispiel ist der Data Provider eine einfache Liste. Data Pipes mit mehreren Variablen – etwa zur Verarbeitung von Datenbankabfragen mit SQL – werden ebenfalls unterstützt. Am Ende der Iteration werden alle Data Provider automatisch durch Aufruf der Methode „close()" geschlossen, auch wenn das in diesem Fall nicht notwendig ist.

Auch die dritte Möglichkeit der Testdaten-Bereitstellung ist in *Listing 2* zu sehen. Hierbei handelt es sich um direkte Wertzuweisungen, wie dies mit der Variablen „futureValue" geschieht. In jeder Iteration wird die Zuweisung wiederholt.

Die Annotation „@Unroll" zählt jede Iteration einzeln im Testbericht auf. Ohne diese würde die Feature-Methode nur einmal aufgelistet werden. Der Name der Feature-Methode enthält Platzhalter, um die Testberichte verständlicher zu machen. Die Platzhalter „value", „rate" und „years" werden automatisch zur Laufzeit ersetzt. Laut *[3]* ist „@Unroll" nicht standardmäßig aktiviert, weil die Anzahl der Feature-Methoden sehr groß sein kann.

Objekte können gemeinsam von mehreren Iterationen genutzt werden, indem sie entweder in Klassenvariablen oder in mit „@Shard" annotierten Instanz-Variablen gehalten werden. In beiden Fällen kann auf die Variablen nur im „where"-Block zugegriffen werden.

Abschließend zu diesem Beispiel sei noch darauf hingewiesen, dass Groovy im Gegensatz zu Java für Zahlen mit Nachkommastellen standardmäßig „java.math.BigDecimal" verwendet *[4]*. Sehr praktisch ist ebenfalls die automatische Konvertierung der Integer in BigDecimal bei den Rechenoperationen des Beispiels.

Verhaltensbasiertes Testen mit Mock-Objekten

In den bisherigen Beispielen wurde stets die Korrektheit der getesteten Objekte auf Basis ihrer Zustände beziehungsweise Rückgabewerte sichergestellt. Ein anderer Ansatz ist das verhaltensbasierte Testen, in dessen Mittelpunkt nicht der Zustand der Objekte, sondern deren Interaktionen stehen. Im Beispiel von *Listing 3* wird eine Buchungskomponente getestet. Der Test soll überprüfen, ob eine E-Mail an den angemeldeten Benutzer zur Buchungsbestätigung gesendet wird; er braucht diese jedoch nicht zu verschicken. Stattdessen wird überprüft, ob der E-Mail-Service wie erwartet aufgerufen wird.

Als E-Mail-Service kommt aus diesem Grund ein Mock-Objekt zum Einsatz. Dieses überprüft, welche Methoden mit welchen Parametern aufgerufen werden und wie oft die Aufrufe stattfinden. Es implementiert den Typ, für den es eingesetzt werden soll, sodass es auch von statisch typisiertem Code verwendet werden kann, und hat initial kein Verhalten, auch wenn seine Methoden aufgerufen werden können. Die Standard-Rückgabewerte sind entweder „false", „0" oder „null". Die Methode „equals()" liefert nur dann „true", wenn das Mock-Objekt mit sich selbst verglichen wird. Die Methode „hashCode()" gibt einen eindeutigen HashCode zurück.

In *Listing 3* wird das Mock-Objekt durch die Methode „Mock()" erzeugt und anschließend dem „BookingService" übergeben. Dieser nutzt den „EmailService" zum Versenden der E-Mails. Die Zeile „1*emailService.send(email)" bedeutet, dass der Aufruf der Methode „send()" mit dem Parameter „email" auf dem Mock-Objekt „emailService" genau einmal stattfinden soll. Die Kardinalität der Interaktionen kann sehr flexibel definiert werden. Der Ausdruck „(_..3)" bedeutet beispielsweise, dass der Aufruf höchstens dreimal stattfinden darf. Auch für die Parameter können unterschiedliche Regeln definiert werden. Der Ausdruck „_ as String" heißt, dass der Parameter vom Typ „String" und nicht „Null" sein darf.

Hamcrest Matcher

Ein weiteres wichtiges Feature des Spock-Frameworks ist die Unterstützung für Hamcrest Matcher *[5]*, um Bedingungen an Objekten überprüfen zu können. Zu diesen gehören beispielsweise die Größe von Collections oder die ersten Buchstaben einer Zeichenkette. Die Kombination aus JUnit und Hamcrest ist üblich, weil die Hamcrest Matcher ausdrucksstärker sind als die einfachen JUnit-Assertions. Diese Vorteile gelten auch für Spock-Tests.

Listing 4 zeigt den Test der Methode „findPersonsOlderThan()", um Personen ab einem bestimmten Alter zu selektieren. Der „given"-Block ist lediglich ein Alias für „setup". Im „expect"-Block wird mit der Methode „that()" die gefundene Personenliste „list" mit den Hamcrest Matchern „hasSize()" und „olderThan()" überprüft. Der Matcher „hasSize()" gehört standardmäßig zur Hamcrest-Bibliothek und überprüft die Länge der Liste. Der benutzerdefinierte Matcher „olderThan()" erweitert „org.hamcrest.BaseMatcher" und testet das Alter der Personen. Neben der eigentlichen Bedingungsüberprüfung in der Methode „matches()" sind auch Beschreibungen definiert, die insbesondere bei Testfehlern helfen, die Ursache des Problems schnell zu erkennen.

```
def “find persons younger than 30”() {
  given:
  final list = service.findPersonsOlderThan(30)

  expect:
  that list, hasSize(3)
  that list, olderThan(30)
}

private olderThan(final int age) {
  [
     matches: { list -> list.every { age < it.age } },
     describeTo: { description ->
        description.appendText(„age greater than ${age}“)
     },
     describeMismatch: { list, description ->
        description.appendValue(list.toListString()).appendText(“ was not
         greater than ${age}”) }
  ] as BaseMatcher
}
```

Listing 4

Fazit

Groovy ist eine weitverbreitete und wichtige JVM-Sprache, die Java nicht ersetzen soll, sondern überall dort zum Einsatz kommt, wo Java Schwierigkeiten hat, beispielsweise bei der Verarbeitung von XML-Daten, zur Definition von DSLs oder Scripting. Die Einstiegshürden für Groovy sind relativ gering und das Aufgabengebiet „Testen" bietet einen idealen Einstiegspunkt für Groovy-Anfänger. In diesem Zusammenhang kann das Spock-Framework gegenüber konventionellen JUnit-Tests mit Java einen echten Mehrwert bieten, weil es durch seine DSL die Tests übersichtlich strukturiert. Die Mocking-Funktion erlaubt verhaltensbasierte Tests und entkoppelt die getesteten Objekte von anderen beteiligten Objekten. Die Unterstützung für Hamcrest Matcher rundet die Test-Möglichkeiten des Frameworks ab und macht das Spock-Framework zu einer interessanten Groovy-Alternative für herkömmliche JUnit-Tests mit Java.

Literatur und Links

[1] Kenneth A. Kousen, Making Java Groovy, 2013, Manning Pubn

[2] https://code.google.com/p/spock/wiki/Spock-Basics

[3] http://docs.spockframework.org/en/latest/data_driven_testing.html

[4] http://groovy.codehaus.org/Groovy+Math

[5] https://code.google.com/p/hamcrest

Kai Spichale
Kai.Spichale@adesso.de
http://spichale.blogspot.de

Kai Spichale ist Senior Software Engineer bei der adesso AG. Sein Tätigkeitsschwerpunkt liegt in der Konzeption und Implementierung von Java-basierten Software-Systemen. Er ist Autor verschiedener Fachartikel und hält regelmäßig Vorträge auf Konferenzen.

http://ja.ijug.eu/15/1/22

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:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Anzeigen:
Simone Fischer
anzeigen@doag.org

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

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

Titelfoto: © Elena Schweitzer / 123rf.com
Foto S. 09: © ruthblack / 123rf.com
Foto S. 10: © koya79 / 123rf.com
Foto S. 14: © alphaspirit / fotolia.com
Foto S. 24: © kamchatka / 123rf.com
Foto S. 33: © Dirk Reibetanz / fotolia.com
Foto S. 49: © ddp images / Sport Moments
Foto S. 53: © grufnar / fotolia.com
Foto S. 57: © hunthomas / 123rf.com
Foto S. 63: © blue67 / 123rf.com

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.

DevCamp

29. - 30. April 2015 | Frankfurt a. M.

Coming soon...

DOAG

www.doag.org