Neo4j-Tutorial: Graphdatenbanken zum Anfassen ▸87

S&S

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 12.2013

JAVA Mag

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Java 8
Lambda-Ausdrücke und Methodenreferenzen ▸21

OSGi at your Service
Generisches Ressourcenmodell ▸34

OSGi-Tests mit Groovy
Integrationstests implementieren ▸80

IT im Wandel

Warum wir jetzt umdenken müssen.

Sind Sie ein Gewinner des Wirtschafts-darwinismus? ▸40

Was der IT-Wandel für die Java-Community bedeutet ▸48

Crowd-governance für agile Teams ▸54

4 194586 709801 12

Anzeige

Next Generation IT

Die IT-Welt befindet sich in einem tiefgreifenden Wandel. Auch wenn an vielen Orten noch alles in Ordnung zu sein scheint, pfeift an anderen Stellen der scharfe Wind des globalisierten Wettbewerbs scharf und deutlich. Seine Botschaft lautet: Es wird in Zukunft schwieriger werden, sich auf ein funktionierendes Businessmodell zu verlassen; der Wandel bestimmt das Geschäft.

Es waren nicht die Telkos und Mobiltelefonanbieter, die dem Smartphone zum Durchbruch verhalfen, sondern der Quereinsteiger Apple mit seinem iPhone. Die mächtige und global vernetzte Autoindustrie hat nicht als erste ein überzeugendes Elektromobilitätskonzept auf die Straße gebracht, sondern ein Start-up aus Kalifornien (Tesla). Es sind in Deutschland nicht die Warenhäuser mit ihrer Versandhandelserfahrung aus Jahrzehnten, die das Internet als Vertriebskanal offensiv nutzen, sondern Amazon und eBay.

Aus dem Vorstand der BMW AG wird kolportiert, dass die Hauptmotivation, drei Milliarden Euro Entwicklungskosten in ein vollständig neues Elektroauto (den i3) zu investieren, allein von der Furcht vor Apple getrieben war (manager magazin, http://bit.ly/125s3Qt).

Gewiss kennen auch Sie eine Vielzahl solcher Geschichten, und die Liste ließe sich beliebig fortsetzen ... Aber haben Sie sich schon einmal gefragt, wie es mit Ihrer Branche, in der Sie arbeiten, weitergeht? Welcher Quereinsteiger wohl hinter der nächsten Ecke lauert, bloß um alles schneller, kundenorientierter, flexibler zu machen als Ihr bislang erfolgsverwöhntes Business?

All diese Newcomer, von denen hier die Rede ist, verbindet ein zentrales Merkmal: sie sind in gesteigertem Maße IT-getrieben. War bisher die Umsetzung komplexer Enterprise-Systeme die hohe Kunst der IT, ist es nun die Umsetzung webbasierter Anwendungen und Geschäftsmodelle. Und dies gekoppelt mit einem hohen Maß an Flexibilität sowie einem enormen Durchsatz an Features.

Der Wandel zielt also zuallererst auf – beständigen Wandel. Damit ist keine Trivialität gemeint, sondern ein Umdenken, das sich in der IT-Industrie schon längst vollzieht. Nehmen wir die Start-up-Kultur: da werden unfassliche Gelder in (mitunter windige) Geschäftsmodelle gesteckt, da hängen hochbegabte Leute ihren normalen Job an den Nagel, bloß um ein neues „Venture" (mit-)gestalten zu dürfen! Und warum? Weil der Wandel zur Norm wird, und vermutlich diejenigen, die bereit zum (zielgerichteten) Experiment sind, schon bald die Nase vorne haben werden.

Das ist aber noch nicht alles. Die Rede vom Wandel stellt alles, was wir seit Jahrzehnten, ja seit Jahrhunderten zum Thema „Industrie" verinnerlicht haben, auf den Kopf. Statt größtmöglicher Standardisierung, Prozesstreue und Serialität wird heute die Experimentierfreude zum Prinzip erhoben.

Deswegen lohnt es sich allemal, die Start-up-Kultur genauer zu beobachten. Hier lassen sich nicht nur interessante unternehmerische Prinzipien ablesen, sondern es wird auch ein alternativer Einsatz von Technologie sichtbar. Auffällig nämlich ist, dass viele Start-ups (auch die „gereiften" Start-ups à la Twitter oder SoundCloud) ein gewandelter Umgang mit technologischen Lösungen auszeichnet und dass sie nach anderen – moderneren – Methoden arbeiten.

Unser Schwerpunkt im Java Magazin beleuchtet verschiedene Aspekte dieses Wandels vor allem aus technologischer und methodischer Richtung. Uwe Friedrichsen beschreibt in seinem Beitrag, welchen gewandelten Anforderungen sich die IT als „Nervensystem" der Unternehmen stellen muss und Eberhard Wolff lenkt den Blick auf Technologien und Methoden, auf die man sich einstellen sollte.

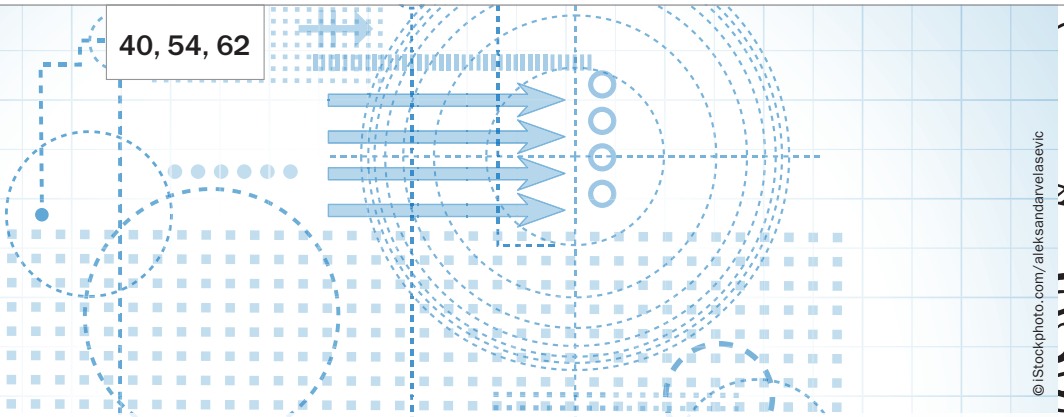
Weil wir diese Fragen so wichtig und vor allem so spannend finden, haben wir auch die Business Technology Days, die regelmäßig parallel zu unserer Konferenz JAX und W-JAX stattfinden, konsequent daran ausgerichtet (http://jax.de/wjax2013/special-days/business-technology-days). Auch das Magazin „Business Technology", das Sie als Abonnent des Java Magazins vierteljährlich kostenfrei erhalten, beschäftigt sich in vielen Beiträgen mit diesem Wandel.

Der Wandel ist in vollem Gange und es ist gut, dass es keine Analysten oder Verbände gibt, die glauben, uns die Richtung vorgeben zu können. Wohin die Reise geht, wissen wir alle nicht so genau. Notwendig ist ein großes Maß an Offenheit und Veränderungsbereitschaft, sowohl in technologischer als auch in geschäftlicher Hinsicht. Die Java-Plattform ist bereit, und ich bin sicher, Sie sind es auch.

In diesem Sinne wünsche ich Ihnen viel Spaß bei der Lektüre dieser Ausgabe

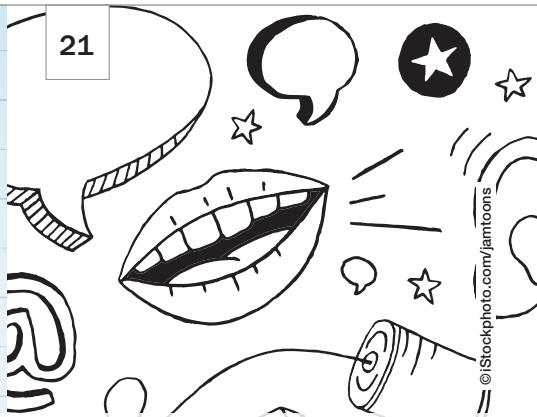
Sebastian Meyen, Chefredakteur

@smeyen



Post-Agile: Überleben im Wirtschaftsdarwinismus

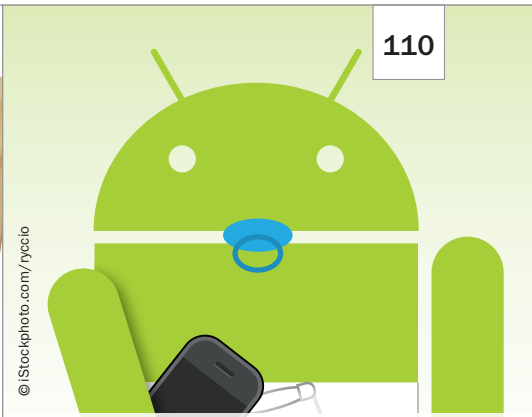
Die IT befindet sich im Wandel. Neue, hochinnovative Unternehmen mit einer extrem schnellen und flexiblen IT drängen in den Markt und sind den alteingesessenen Platzhirschen immer mehrere Schritte voraus. Agilität wird dann gerne als Wunderwaffe ins Feld geführt, um zumindest die Unternehmens-IT schneller und flexibler zu machen. Aber reicht das? Es wird Zeit, sich den Wandel und neue Trends – Continuous Delivery, DevOps, Crowd Governance, Liquid Democracy, Lean Innovation – genauer anzusehen. Unsere Autoren zeigen, wie man sich in der IT aufstellen muss, um zu den Gewinnern des Wirtschaftsdarwinismus zu gehören.



Effective Java: Let's Lambda!

In Java 8 gibt es neue Sprachmittel, die einen funktionalen Programmierstil unterstützen. Wir geben einen kompakten Einstieg in diese Erweiterungen. Beleuchtet werden dabei Syntaxvarianten, automatische Typdeduktion, SAM Types und der Zugriff auf Variablen des umgebenden Kontexts aus einem Lambda-Body heraus – alle Mittel, die man braucht, um Lambdas nutzen zu können.

Leserbriefe und Feedback zu den Artikeln des Java Magazins bitte an redaktion@javamagazin.de.



Der IT-Wandel in der Java-Welt
Das Java-Ökosystem ist vor allem im Enterprise-Bereich führend. Viele Technologien können auf eine ganze Dekade Geschichte zurückblicken – aber das Umfeld ist im Wandel begriffen. Durch den „Wirtschaftsdarwinismus" ändern sich die Herausforderungen, denen sich die IT stellen muss. Das wird an Java nicht spurlos vorbeigehen.

Neo4j 2.0
Neo4j ist eine in Java implementierte Graphdatenbank, die ursprünglich als hochperformante, in die JVM eingebettete Bibliothek genutzt wurde. Seit einigen Jahren steht sie als Serverdatenbank zur Verfügung. Nach dem positiven Feedback zum letzten Neo4j-Artikel soll nun das Handwerkzeug vorgestellt werden, mit dem jeder selbst Anwendungen mit Graphdatenbanken entwickeln kann.

Nexus 7
Mit dem Nexus 7 ist es 2012 gelungen, das Taschenbuchformat 7 Zoll für Tablets erfolgreich zu etablieren. Galt der 7-Zoll-Formfaktor zuvor als zu klein, zogen etliche Hersteller aufgrund des Erfolgs des Nexus 7 bald nach. Im Juli 2013 stellte der 7-Zoll-Pionier eine neue Version des Nexus 7 vor – bloße Modellpflege oder gibt es wirklich Neues zu vermelden? Ein Erfahrungsbericht.

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

Atmosphere 2.0 ist da: WebSocket für alle

Das asynchrone Cross-Browser-Framework Atmosphere ist in der Version 2.0 erschienen. Fast genau ein Jahr haben die Entwickler um den ehemaligen Sun-Veteranen Jeanfrançois Arcand (Verfasser beispielsweise des GlassFish Microkernels) gebraucht, um den Nachfolger des viel beachteten Atmosphere-1.0-Releases abzuschließen.

Mit dem „Real-Time-Client-Server-Framework für die JVM“ lassen sich portierbare Anwendungen in Groovy, Scala und Java schreiben. Neben einer JavaScript-Komponente enthält Atmosphere mehrere Serverkomponenten, die die wichtigsten Java-Webserver unterstützen. Ziel ist es, die Entwicklung von Anwendungen dadurch zu erleichtern, dass das Framework selbstständig und codeunabhängig den besten Kommunikationskanal zwischen Client und Server findet.

In der Version 2.0 wurde generell an der Performance und einem verbesserten Cache gearbeitet. Vor allem der WebSocket-Fallback soll zügiger vonstatten gehen. Neu unterstützt werden der Netty-I/O-Server, das Play!-Framework und Vert.x. Atmosphere wird auf GitHub entwickelt und vom Unternehmen Async-IO.org gesponsert.

▶ http://async-io.org/release.html

Stiller Java-Killer: Ceylon 1.0 beta

Ceylon ist eine JVM-Sprache, die laut ihrem Motto „mehr sagen“ möchte, aber relativ wenig von sich reden macht. Nun ist sie in der Version 1.0 beta erschienen und enthält damit die gesamte Sprachspezifikation, wie der Entwickler Gavin King auf der Projektseite mitteilt. Ceylon soll nicht nur auf der Java Virtual Machine laufen, sondern auch jede existierende JavaScript-Ausführungsumgebung unterstützen. Die Sprache sei mit nativem Code beider Plattformen interoperabel, so King.

Neben der kompletten Sprachspezifikation enthält diese Version ein Kommandozeilen-Toolset, das u. a. die Java- und JavaScript-Compiler enthält und Unterstützung für die Ausführung modularer Programme auf der JVM und Node.js bietet. Zur Ausstattung gehören außerdem ein typensicheres Metamodell, eine modulare Architektur und das Sprachmodul.

Parallel zur Sprache wurde die Eclipse-basierte Ceylon-IDE herausgegeben. Sie ermöglicht es nun, Ceylon-Programme in der Modullaufzeit zu starten. Außerdem wurden einige Eclipse-Funktionalitäten (Datei- und Paket-Refactorings, Merge Viewer) besser integriert.

▶ http://ceylon-lang.org/blog/2013/09/22/ceylon-1/

Projekt Avatar wird Open Source

Vor zwei Jahren angekündigt, auf der letzten JavaOne noch als Geheimtipp gehandelt, jetzt auf der Open-Source-Bühne angekommen: Oracles Projekt Avatar steht ab sofort unter https://avatar.java.net/ quelloffen zur Verfügung.

Avatar ist Teil von Oracles Masterplan, die JavaScript- und Java-Programmierung aneinander anzunähern. Neben der JavaScript-Engine Nashorn und dem NetBeans-HTML5-Plug-in „Easel“ ist Avatar das dritte JavaScript-Projekt, das in diese Kerbe schlägt. Es handelt sich dabei im Wesentlichen um ein Webframework, das HTML5-Frontends und Java-EE-Backends miteinander verbinden soll.

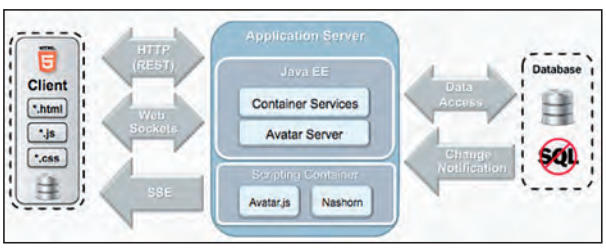
Drei Komponenten werden geboten. Zum einen Avatar.js, das es Entwicklern ermöglicht, auf Basis des Node-Programmiermodells JavaScript-Dienste auf dem Server zu schreiben. Veröffentlicht werden die Dienste über WebSocket, REST oder Server-Sent Events, sodass clientseitige Frameworks darauf zugreifen können.

Zweitens sollen JavaScript-Entwickler auf dem Server auch Java-EE-Services nutzen können. Avatar richtet sich also nicht nur an die Java-Community, auch JavaScript-Entwickler rücken ins Feld der potenziellen Konsumenten von Java-EE-Services.

Schließlich soll ein prinzipiell unabhängiges clientseitiges Framework bereitgestellt werden, mit dem HTML5-Komponenten an Services gebunden werden können. Dabei soll nur ein minimales JavaScript-Wissen notwendig sein; die Services werden wieder über WebSocket, REST und SSE verbreitet.

Derzeit läuft Avatar auf der Basis des GlassFish-Servers (Avatar-GF-Distribution), weitere Distributionen sollen folgen. Auf der Projektseite findet man bereits einige nützliche Tutorials für den Einstieg ins neue JavaScript-/Java-Zeitalter.

▶ https://avatar.java.net/



Avatar-Architektur. Bild: https://avatar.java.net/

Starthilfe für die Typesafe-Plattform: Activator-Projekt bietet Templates für reaktive Anwendungen

Das Scala-Unternehmen Typesafe hat das neue Projekt Activator ins Leben gerufen. Ziel von Activator ist es, Entwicklern den Einstieg in den eigenen Technologiestack zu erleichtern. Dessen Hauptkomponenten bestehen bekanntlich aus der Programmiersprache Scala, dem Middleware-Framework Akka und dem Webframework Play.

Activator kommt in Form einer lokalen Webapplikation, die beim Aufsetzen eines Projekts hilft, indem sie Templates für verschiedene Verwendungszwecke bzw. Technologiekombinationen bereithält. Jedes der derzeit 29 Templates enthält ein Tutorial, das die Funktionsweise des gewählten Technologiemixes erklärt. Der Projektcode lässt sich anzeigen und mit einem Klick in eine IDE übertragen – IntelliJ, Eclipse und NetBeans werden unterstützt. Zudem bietet Activator ein Compile-Tab, das das Compile-Ergebnis anzeigt, ein Testinterface, mit dem Tests vom Browser aus gestartet werden können, und ein Run-Tab, das Informationen über die laufende Anwendung übermittelt. Activator steht aktuell in Version 1.0 bereit.

▶ http://typesafe.com/activator

Venture-Kapital für Data Grid Hazelcast: Rod Johnson wird Vorstandsmitglied

Das In-Memory Data Grid Hazelcast bekommt prominente Unterstützung durch Spring-Erfinder Rod Johnson. Zusammen mit WebLogic-Gründer Ali Kutay und Venture-Kapitalgeber Salil Deshpande ist Johnson in den Vorstand des gleichnamigen Unternehmens Hazelcast aufgenommen worden. Einher geht die Vorstandsaufstockung mit einer neuen Kapitalrunde durch Bain Capital, die der Technologie neuen Auftrieb geben soll.

Hazelcast ist eine in Java geschriebene Open-Source-Plattform für die Datenverteilung auf Cluster-Systemen. Verteiltes Caching, hohe Skalierbarkeit, Memcache-Support und Integrationen mit Spring und Hibernate schreibt sich Hazelcast auf die Fahnen.

▶ http://www.hazelcast.com/

WebFX: JavaFX ist das neue HTML?

WebFX ist ein neues Open-Source-Projekt, das zeigen soll, wie man für das Erstellen von Webseiten JavaFX anstelle von HTML einsetzen kann. Das Ganze soll durch eine Kombination aus FXML, JavaScript und CSS passieren.

Warum aber sollte man freiwillig JavaFX anstelle von HTML für die Webentwicklung nutzen, mag man sich jetzt fragen. Projektersteller Bruno Borges sieht mit der neuen JavaScript-Engine Nashorn einen Performance-Boost auf JavaFX-Seiten zukommen: „Idea is to build an FX browser, a security layer, a navigation scheme where one FXML can tell the browser to go to another FXML and a protocol for server-side communication."

Das WebFX-Projekt befindet sich noch am Anfang, Samples und Browser können auf GitHub gefunden werden, Security-Layer und FX-Protokoll sind noch in der Planung. JavaFX-Interessierte sollten das Projekt im Auge behalten.

▶ https://github.com/brunoborges/webfx

Gradle 1.8 will dem Entwickler Zeit sparen

Das Build-System Gradle hat Version 1.8 erreicht und steht damit kurz vor dem zweiten Major-Release. Gradle versteht sich als Toolkit zum Bauen von Softwareprojekten und ist eine mögliche Alternative zu Tools wie Ant und Maven. In Gradle steht eine ausdrucksstarke Build-Sprache zur Verfügung, eine auf Groovy basierende erweiterbare DSL, mit der sich Build-Skripte schreiben lassen.

Gradle 1.8 unterstützt jetzt auch Builds für C/C++ und Assembler. Ab sofort werden Dependency-Informationen nur noch gespeichert, wenn sie im Heap benötigt werden. Zudem soll das Importieren eines Projekts deutlich schneller vonstattengehen.

Ab sofort konzentrieren sich die Arbeiten des Gradle-Teams auf das zweite Major-Release. Hier geht es laut Projektleiter Hans Dockter vor allem um eine Verschlankung des gesamten Tools.

▶ http://www.gradle.org/docs/current/release-notes

Autor des Monats

Uwe Friedrichsen ist ein langjähriger Reisender in der IT-Welt. Als Fellow der codecentric AG darf er seine Neugierde auf neue Ansätze und Konzepte sowie seine Lust am Andersdenken ausleben. Seine aktuellen Schwerpunktthemen sind verteilte, hochskalierbare Systeme und Architektur in post-agilen Umfeldern. Er teilt und diskutiert seine Ideen häufig auf Konferenzen, als Autor von Artikeln, Blog-Posts, Tweets und mehr.

Wie bist du zur Softwareentwicklung gekommen?

Anfang 1981 in der 9. Klasse kam unser Mathelehrer mit dem damals allerersten Schulcomputer auf einem Laborwagen (!) in den Unterricht und zeigte uns ein paar einfache Sachen. Ich fand das äußerst faszinierend, und als es dann Anfang der 10. Klasse eine Informatik-AG gab, war ich dabei. Der Rest ist Geschichte …

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Schwierig. Es gibt viele schöne Aspekte. Aber ich denke, für mich ist es immer noch am schönsten, wenn ich es schaffe, eine klare, gut verständliche Lösung für eine schwierige Aufgabe zu finden und umzusetzen.

Was ist für dich ein weniger schöner Aspekt?

Auch da gäbe es einiges zu nennen. Am ermüdendsten finde ich aber wohl ganz allgemein die in der IT weit verbreitete Unwissenheit und Inkompetenz auf allen Ebenen. Das beginnt mit schlampig hingeschludertem Code und endet bei lausigem IT-Management, basierend auf der stupiden Anwendung von verbreiteten, aber offensichtlich sinnlosen „Patentrezepten".

Wie und wann bist du auf Java gestoßen?

Das war Ende 1995. Java wurde als „das nächste große Ding" gehypt und wir haben es uns in meiner damaligen Firma genauer angesehen. Richtig intensiv habe ich 1999 begonnen, mit Java zu programmieren, als ich einen Webclient in Java entwickelt habe, um den existierenden Smalltalk-basierten Desktopclient abzulösen.

Wenn du für einen Tag König der Java-Welt wärst, was würdest du verändern?

Sehr schwierige Frage. Einiges nervt, aber Java ist wesentlich besser, als es bei dem aktuell äußerst hippen Java-Bashing den Anschein hat. Ich würde wohl den Umgang mit Null ändern. Die ewigen Null-Checks blähen den Code auf, und die dauernden NPEs machen die Anwendungen in Produktion instabiler. Sprachen wie z. B. Clojure haben das wesentlich eleganter gelöst. Ach ja, und natürlich mehr funktionale Aspekte …

Was ist zurzeit dein Lieblingsbuch?

Ein dediziertes Lieblingsbuch gibt es bei mir nicht. Sehr empfehlenswert finde ich aber nach wie vor „Release It!" von Michael Nygard, weil es sich mit produktionsfähiger Software befasst, ein bei Entwicklern leider häufig sträflich vernachlässigtes Thema – sollte Pflichtlektüre für jeden Softwareentwickler sein!

Was machst du in deinem anderen Leben?

Nun, ich bin verheiratet und habe zwei Kinder, was eigentlich schon ausfüllend genug ist. Da ich aber auch noch Musik- und Filmjunkie bin, gerne und viel lese und auch noch Sport treibe, müsste mein Tag eigentlich mindestens 36 Stunden haben, um das auch nur halbwegs unterzubringen.

JUG-Kalender* Neues aus den User Groups

WER?	WAS?	WO?
JUG Schweiz	06.11.2013 – Bereit für das Internet der Dinge	http://www.jug.ch
JUG Schweiz	07.11.2013 – Spring Framework 4.0	http://www.jug.ch
JUG Saxony	07.11.2013 – AngularJS: Logik am richtigen Platz	http://jugsaxony.org
JUG Frankfurt	13.11.2013 – Devops@Runtime mit MoSKito	https://sites.google.com/site/jugffm
JUG Düsseldorf	14.11.2013 – Eclipse-Abend	http://rheinjug.de
JUG Ostfalen	21.11.2013 – Copy & Paste & Bug	http://jug-ostfalen.de
JUG Augsburg	21.11.2013 – Einführung in Graphdatenbanken und Neo4j	http://jug-augsburg.de
JUG Stuttgart	25.11.2013 – Objektforum Stuttgart	http://www.jugs.de
JUG Frankfurt	27.11.2013 – Vert.x	https://sites.google.com/site/jugffm
JUG Ostfalen	28.11.2013 – Java on Tracks	http://jug-ostfalen.de
JUG Hessen	28.11.2013 – Neo4j	http://www.jugh.de
JUG Darmstadt	12.12.2013 – Apache Lucene & Solr: mal eben schnell was finden	http://jugda.wordpress.com
JUG Düsseldorf	12.12. 2013 – Kinder und Jugendliche mit Robotern fürs Programmieren begeistern	http://rheinjug.de

*Alle Angaben ohne Gewähr. Da Termine sich kurzfristig ändern können, überprüfen Sie diese bitte auf der jeweiligen JUG-Website.

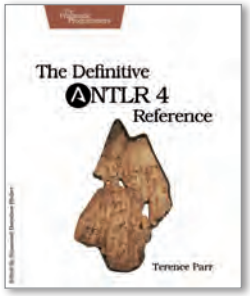
The Definitive ANTLR 4 Reference

von Terence Parr

In seinem ersten Buch erklärte Terence Parr noch den Namen ANTLR, **AN**other **T**ool for **L**anguage **R**ecognition. Im aktuellen Werk verzichtet er darauf. Schließlich ist ANTLR inzwischen bekannt genug. Bei dem Produkt handelt es sich um einen Lexer und Parser. Eingabedaten werden in Tokens umgewandelt und diese interpretiert. Der Autor nennt in der Einleitung ein paar bekannte Anwendungen: So parst die NetBeans-IDE C++-Dateien oder Twitter die Suchanfragen mit diesem Tool. Und bei Martin Fowler ist nachzulesen, wie er mittels ANTLR Domain-specific Languages erstellt. In der Version 4 ist dieses Tool nun einfacher und leistungsfähiger geworden. Musste der Entwickler in der Vorgängerversion für bestimmte Applikationen selbst noch einen abstrakten Syntaxbaum erzeugen, so geschieht dies in der neuen Version automatisch. Und Treewalker sowie Listener werden gleich mitgeliefert. Wie dies funktioniert, beschreibt der Autor im Detail. Aber nicht nur die neue Version ist besser, auch Terence Parr hat seinen Schreibstil verändert. Er setzt weniger voraus und holt so den Leser ganz am Anfang ab. Und er erklärt deutlich besser. Waren in seinem ersten Buch doch Kenntnisse über Lexing und Parsing fast schon zum Verständnis notwendig, so kommt nun auch ein Einsteiger in diesem Bereich gut zurecht. Lediglich Programmierkenntnisse werden vorausgesetzt. Kenntnisse der (E)BNF sind hilfreich, aber nicht notwendig.

Parr erläutert die Prinzipien von Lexern und Parsern und grenzt deren Funktionalität voneinander ab. ANTLR beherrscht beides – sogar in einer Grammatikdefinition. Das macht die Anwendung einfacher. Zum ersten Teil des Buchs gehört auch ein Überblick über die Möglichkeiten. Parr zeigt, wie beim Parsen ein Syntaxbaum aufgebaut und dieser durchwandert wird und welche Features ANTLR so alles bietet. In den Teilen II und III geht es dann in die Details. Parr zeigt anhand beispielhafter Grammatiken, wie CVS, JSON, R, XML und andere Sprachen zu erkennen sind – und zeigt auch, wie mittels ANTLR einfach Übersetzer, z. B. von JSON nach XML, erstellt werden können. Links-rekursive Grammatiken, Erkennung nicht eindeutiger Syntax, die erst im Kontext verständlich wird, Umschaltung zwischen Erkennungsmodi, Einbettung von Aktionen und vieles mehr erläutert der Autor ausführlich. Und wer sich bisher noch nicht mit dieser Thematik beschäftigt hat und daher mit den benannten Begriffen nichts anfangen kann, der versteht auch diese mit der Lektüre. Teil IV schließlich ist mit „ANTLR Reference“ betitelt. Aber auch dieser Teil ist nicht einfach nur eine Auflistung der ANTLR-API- oder Grammatik-Referenz. Auch hier findet der Leser zahlreiche Hinweise und Erklärungen. Insofern ist das Buch nicht nur Referenz, sondern auch User Guide. So macht es Spaß, es zu lesen und zu erkennen, was alles mit einem Lexer/Parser möglich ist.

Michael Müller



Terence Parr

The Definitive ANTLR 4 Reference

328 Seiten, 37,00 US-Dollar
The Pragmatic Programmers, 2013
ISBN 978-1-93435-699-9

NoSQL-Lösung mit großer Zukunft

HBase

Mit seinen Wurzeln in der Bigtable-Technologie hat sich HBase über die Jahre zur Grundlage vieler Google-Produkte und -Dienste entwickelt. Welche Ideen, Architektur und welches Datenmodell stecken hinter HBase? Wir nehmen die NoSQL-Lösung unter die Lupe.

von Lars George

Bevor wir uns HBase im Detail anschauen, ist es interessant, auch etwas über dessen Vergangenheit und damit Herkunft zu wissen. Anfang des 21. Jahrhunderts, also vor mehr als zehn Jahren, boomte das Internet und Information in Form von Dokumenten und Webseiten wuchsen unaufhörlich. Google, ein aufstrebendes Unternehmen mit Ziel, all diese Daten mithilfe einer Suchfunktion zur Verfügung zu stellen, stand vor dem Problem, diese Dokumente kosteneffektiv und zukunftssicher zu speichern und zu bearbeiten. Daraus entstand die erste Version des Google File Systems [1] (kurz GFS) und MapReduce [2]. Damit war es möglich, alle Webseiten im Internet zu archivieren und parallel im Batchbetrieb auszuwerten. Jeder neue Lauf über alle Daten erzeugte einen neuen Suchindex. Sehr schnell wurde aber klar, dass ein Dateisystem nicht ideal ist, um sich ständig ändernde Datensätze zu speichern. Es brauchte ein skalierbares, aber dennoch datenbankähnliches System, das eben diese Datensätze leicht zugänglich und aktualisierter macht. Daraus entstand dann Mitte der 2000er Jahre Bigtable [3], das als Grundlage für HBase diente. Google veröffentlichte nicht nur die Funktionsweise von GFS und MapReduce, sondern auch Bigtable. Dieses war 2006, als dessen technische Veröffentlichung herausgegeben wurde, schon mehrere Jahre erfolgreich im Einsatz. Über die weiteren Jahre wurde Bigtable Grundlage für viele Google-Produkte und -Dienste, und bestätigte damit dessen Idee und Implementierung.

Listing 1: Erster wichtiger Hadoop-Commit-Eintrag

```
r373007 | cutting | 2006-01-27 23:19:42 +0100 (Fri, 27 Jan 2006) | 1 line

Create hadoop sub-project.
...

r374733 | cutting | 2006-02-03 20:45:32 +0100 (Fri, 03 Feb 2006) | 1 line

Initial commit of code copied from Nutch.
...
```

Einige Jahre nach den technischen Veröffentlichungen von Google wurden im Rahmen des Apache-Nutch-Projekts diese Ideen aufgegriffen, denn zu dieser Zeit arbeitete Doug Cutting [4] – zusammen mit Mike Cafarella – an einer quelloffenen Implementierung eines Web-Crawlers und Indexers. Nutch brauchte dieselben Funktionen, wie GFS und MapReduce sie für Google boten, und deshalb wurden genau deren Funktionen in Nutch integriert. Doug ahnte aber vom allgemeinen Nutzen und startete ein neues Apache-Projekt namens Hadoop [5]. In Listing 1 sind die ersten Commit-Einträge zu sehen.

Kurz nach der Gründung von Hadoop begann Mike Cafarella auch Bigtable in dem quelloffenen Projekt zu implementieren. Daraus wurde dann die *Hadoop Database*, oder *HBase* in abgekürzter Form. Hier wiederum der erste Commit-Eintrag für HBase:

```
r525267 | cutting | 2007-04-03 22:34:28 +0200 (Tue, 03 Apr 2007) | 1 line

HADOOP-1045.  Add contrib/hbase, a Bigtable-like online database.
```

Über die Jahre wuchs HBase aus den Kinderschuhen heraus und wurde von einem *Contrib*-Modul in Hadoop zu einem Unterprojekt und dann endgültig zu einem Apache-Hauptprojekt. Dies spiegelt sich auch in den Versionsnummern wieder, die zuerst Hadoop folgen, aber dann von 0.20 auf 0.89 springen – nur aus dem Grund, um die Trennung des Projekts und die Änderung der Frequenz der Veröffentlichungen deutlich zu machen.

Datenmodell

Bigtable, und damit HBase, speichert seine Datensätze in einem logischen Modell, das man von relationalen Datenbanken her kennt. Es gibt also Tabellen, Zeilen und Spalten. Damit hören aber die Ähnlichkeiten auch auf: HBase hat keine Abfragesprache wie SQL, referenzielle Integrität, Transaktionen oder beliebige Indexe. Der Zugriff auf die Daten erfolgt mit einem relativ einfachen API, das nativ in Java zur Verfügung steht, aber

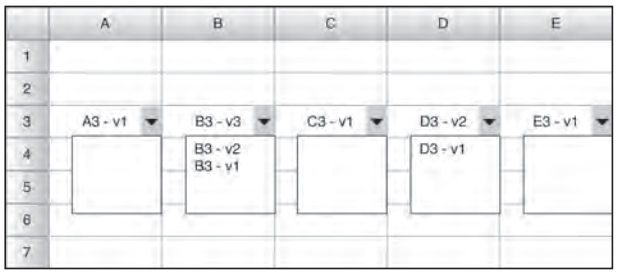


Abb. 1: Zeilen mit versionierten Werten

auch über andere Protokolle wie REST oder Thrift angesprochen werden kann. Dieses API kennt grundsätzlich nur vier Befehle: *Put, Get, Scan* und *Delete*. Dazu gesellen sich noch einige speziellere Befehle für atomare, serverseitige Funktionen, wie *Increment* und *checkAndPut* oder *checkAndDelete*.

Daten werden wie erwähnt in Tabellen und darin in Zeilen und Spalten abgelegt. Die Zeilen haben einen eindeutigen Schlüssel, der so genannte *Zeilenschlüssel (Row Key)*. Damit kann der Anwender auf die eigentlichen Spalten zugreifen, wiederum über einen Schlüssel, den *Spaltenschlüssel (Column Key* oder *Column Qualifier*). Ein weiteres Merkmal von HBase ist, dass es die eigentlichen Werte in den Spalten versioniert, und – so die systemweite Vorgabe – drei Versionen der Werte aufhebt, in den *Zellen (Cells)*. Hat man einen Zeilen- und Spaltenschlüssel, kann man sich den aktuellsten Wert – die zuletzt gespeicherte Zelle – geben lassen. Hat man aber auch noch einen *Zeitstempel (Timestamp)*, kann man auch auf einen historischen Wert zugreifen. Die Vorgabe ist immer der letzte gespeicherte Wert, und damit kann der Anwender eine ganze Zeile lesen, wie er es aus einer Datenbank gewöhnt ist, als wären nur die aktuellen Werte gegeben. **Abbildung 1** zeigt dies in Form einer Tabellenkalkulation.

Das API hat aber Funktionen, die dem Anwender erlauben, auch historische Werte auszulesen, zusammen mit den aktuellen oder für einen gegebenen *Versionsbereich (Time Range)* – ganz wie es die Anwendung benötigt. Im kanonischen Anwendungsfall von Bigtable, dem Web-Crawl, d. h. dem Speichern aller Internetseiten, werden zum Beispiel die drei letzten Versionen einer HTML-Seite aufbewahrt, denn damit kann Google feststellen, wie oft eine Seite aktualisiert, wie sie verändert wird (man denke an autogenerierte Inhalte) und damit den PageRank der Seite dementsprechend kalkulieren.

Schauen wir uns weiter die Architektur des Datenmodells von HBase an, dann finden wir einen Hinweis, wie dessen Skalierbarkeit erreicht wird. Anstatt dass der Anwender selbst Daten in Teilbereiche zerlegen muss, um diese von mehreren Servern bereitstellen zu lassen, ist bei HBase dies bereits eingebaut. Die Tabellen werden in *Regionen (Regions)* aufgeteilt, die von genau einem *Region Server* zur Verfügung gestellt werden. Ein solcher Server bietet seine Dienste im Auftrag von einer oder mehreren Regionen an. Damit kann eine Anwendung die Daten genau beim richtigen Server finden. Dazu gibt

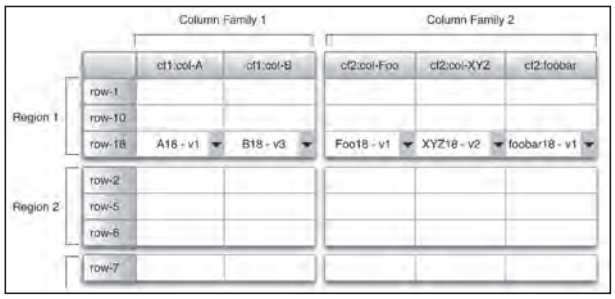


Abb. 2: Architektur der Datenspeicherung in HBase

es ein Nachschlageregelwerk, das auf internen Spezialtabellen beruht: *-ROOT-* und *.META.* (in 0.96 von HBase wurde *-ROOT-* entfernt, da nicht wirklich benötigt). Mithilfe dieser Tabellen kann der Server, der eine bestimmte Zeile (oder eine Anzahl von Zeilen innerhalb einer Region) bedient, gefunden werden. Die von HBase mitgelieferte, clientseitige Bibliothek in der Anwendung merkt sich diese Information für weitere Zugriffe. Damit wird recht schnell eine Karte aller Regionen und deren Server erstellt. Weitere Zugriffe sind dann direkt auf die Daten möglich, ohne weitere Nachschlageoperationen.

An dieser Stelle sollten zwei entscheidende Merkmale von HBase gegenüber anderen NoSQL-Datenbanken im Vergleich genannt werden: strikte Konsistenz und atomare Zeilenoperationen. Ersteres gründet sich auf der gerade genannten Zuordnung einer Region zu genau einem Server. Dieser kann dann alle Modifikationen (wie *Put* und *Delete*) so behandeln, dass immer eine strikte Konsistenz gewahrt bleibt. Dazu kommt, dass eine Zeile immer atomar verändert wird, also ganz oder gar nicht. Damit kann eine lesende Anwendung entweder die alten Werte oder aber die neuen Werte über alle Spalten und Familien lesen. Es kann nicht vorkommen, dass Modifikationen einer schreibenden Anwendung teilweise sichtbar sind (*Read Committed* nach ANSI/ISO).

Eine weitere Eigenheit des Datenmodells sind die *Spaltenfamilien (Column Famlies)*, welche die Spalten innerhalb einer Zeile gruppieren. Dies ist an die Funktion von spaltenorientierten Datenbanken angelehnt, die speziell für analytische Abfragen geeignet sind, denn dort werden bei einer sehr selektiven Abfrage nur die benötigten Daten der enthaltenen Spalten durchlesen. In HBase dienen die Spaltenfamilien der Gruppierung von zusammengehörigen Spalten, aber auf einer gröberen Ebene, d. h. viele Spalten in wenigen Gruppen. Zurück zum Web-Crawl-Beispiel von vorhin: Dort werden die echten Daten, also HTML-Seiten, in einer Familie abgelegt, und Metadaten in einer anderen. Auch die eingehenden und ausgehenden Links, die für die Berechnung des PageRanks nötig sind, werden in separaten Spaltenfamilien abgelegt, da diese auch separat durchlesen werden. **Abbildung 2** zeigt eine Zusammenfassung des Datenmodells.

Ein weiterer wichtiger Teil der Architektur in HBase ist die implizite Sortierung aller Schlüssel, d. h. die Zeilen und auch Spalten werden *lexikographisch* geordnet, im Falle der Spalten sogar innerhalb deren Familie getrennt. Diese Art der Sortierung beruht auf einer weiteren Eigenschaft des Datenmodells: Alle Schlüssel sind binär. In **Abbildung 2** kann man sehen, wie die Zeilen eher untypisch für Menschen sortiert werden, denn *row-18* kommt noch vor *row-2*. Lexikographisch sortiert bedeutet, dass ein Schlüssel Zeichen für Zeichen, genauer gesagt Byte für Byte, verglichen wird. Und da die *2* an Position *5* größer ist als *1* im vorherigen Schlüssel, werden alle Zeilen, die mit *row-1* anfangen vorneweg sortiert. Im Übrigen ist in diesem Beispiel wenigstens ein lesbarer Schlüsselwert gewählt worden, es hätte auch ein komplett binärer, nicht druckbarer Wert sein können – die Anwendung ist frei, irgendeinen Wert zu setzen.

Letztlich stellt die clientseitige Bibliothek von HBase auch noch zahlreiche Funktionen zur Verfügung, welche die Anwendung nach Bedarf nutzen kann. Dazu gehören Hilfsklassen für die Umwandlung von nativen Java-Typen in deren binärer Darstellung, wie *int, long* und *String*. Es gibt auch einen ganzen Satz an Filterkassen, die es erlauben, die Daten bereits auf der Serverseite zu begrenzen, damit diese nicht unnötigerweise über das Netzwerk an die Anwendung geschickt wird, nur um dort fallengelassen zu werden. Diese Filter erlauben nach beliebigen Schlüsseln oder Werten zu selektieren, und wenn nötig, kann der Anwender sogar seinen eigenen Filter schreiben und benutzen.

Implementierung

HBase verlässt sich auf das darunterliegende Dateisystem, um alle Daten sicher aufzubewahren. Dazu gehört auch die notwendige Replikation von Datenblöcken. Wie GFS dies für Bigtable macht, so ist das *Hadoop Distributed File System* (HDFS) dafür in Kombination mit HBase zuständig. Damit kann sich HBase selbst auf dessen eigene Dateien konzentrieren, denn HDFS übernimmt die Herstellung und Kontrolle der Prüfsummen, die Verteilung der Dateien in Blöcken über die verfügbaren Server, sowie die Sicherstellung, dass ein Ausfall eines Servers nicht bemerkbar ist.

Google selbst hat das Rad bei Bigtable nicht neu erfunden, sondern sich auf existierende Technologien gestützt, unter anderem die so genannten Log-Structured Merge Trees [6] (LSM-Trees), die Daten immer sequenziell schreiben und damit nicht dem unter Last merkbaren Problem der auf B-Tree basierenden relationalen Datenbanken ausgesetzt sind. Hier geht es um die IOPS [7] moderner Hardware: Festplatten können sequenziell schneller arbeiten als bei zufälligen Zugriffen. Grund hierfür ist, dass auch heutige Festplatten im Vergleich zu den Durchsatzraten immer noch eine um mehrere Faktoren größere Zeit für die Positionierung (Seek) der Schreib-/Leseköpfe benötigt. Ziel eines Speichersystems ist es deshalb, möglichst wenige Kopfpositionierungen zu verursachen und möglichst viel sequenziell zu lesen.

Dies ist genau das Ziel der LSM-Trees, die neue oder geänderte Daten zuerst in ein binäres Log schreiben, und dann sortiert im Speicher vorhalten. Sind genügend Daten vorhanden (also wird der Speicher voll, oder

ein vorbestimmter Schwellenwert wird erreicht), dann werden die Daten als neue Datei gespeichert. Über eine gewisse Zeit werden damit also immer wieder neue Dateien mit Daten im Dateisystem abgelegt. Diese können weiterhin von Anwendungsseite parametrisiert werden, indem dort im Datenschema festgelegt wird, ob eine Datei komprimiert werden soll, oder spezielle Zugriffshilfen (z. B. Bloom-Filter) mit erzeugt werden sollen. Die binären Logdateien werden solange vorgehalten, bis alle Veränderungen aus dem Speicher herausgeschrieben wurden. Damit kann HBase im Falle eines Serverfehlers den aktuellen Stand wiederherstellen.

Im Hintergrund laufen noch weitere Prozesse, die aus dem LSM-Trees-Modell stammen. Einer davon sind die *Compactions*, welche die obigen Datendateien in regelmäßigen Abständen prüfen und gegebenenfalls kombinieren. Dies ist vonnöten, um die Anzahl der Dateien gering zu halten (nicht jedes Dateisystem kann unendlich viele Dateien effizient verwalten), aber auch, um die Suche der Daten für eine Zeile möglichst schnell ablaufen zu lassen. Es ist einfacher, ein paar wenige Dateien zu prüfen, als viele hunderte.

Eine weitere asynchrone Operation ist die automatische *Aufteilung (Splits)* der Zeilen auf Regionen. Dies passiert, sobald die Größe aller Datendateien einer Region eine bestimmte voreingestellte Größe erreicht. Dann wird jener Hintergrundprozess angestoßen, der die Daten auf zwei neue Regionen aufteilt und die originale Region abschließend löscht. Davon bemerkt der Anwender im Normalfall nichts. Es kann aber zusätzliche I/O-Last auf einem HBase-Cluster erzeugen.

Mehr Details über HBase und dessen Architektur können in einem früheren Artikel nachgelesen werden [8]. Für die Betrachtung von HBase in diesem Artikel sind nun alle entscheidenden Merkmale genannt. Damit können wir uns nun anschauen, wo HBase gut oder weniger gut funktioniert – und warum.

Stärken und Schwächen

Wie eingangs erwähnt, fehlen HBase einige aus der relationalen Datenbankwelt bekannten Leistungen. Allem voran hat HBase keine Transaktionen, und das aus gutem Grund: Diese machen eine Skalierung einer Datenbank über Rechnergrenzen hinweg schwierig. Aber HBase bietet Ersatz in Form von serverseitigen, atomaren Operationen an, wie zum Beispiel *increment* oder *compare-and-set*-(CAS-)Befehle wie *checkAndDelete*. Mit deren Hilfe kann der Anwender sehr effizient Zähler verändern, oder vorher gelesene Daten nach einer Prüfung neu setzen. Dies machte sich Facebook in dessen Insights-Service [9] zunutze, in dem es einen (M)OLAP-Würfel für die spätere Reporterstellung immer aktuell hält. Der Reportgenerator braucht dann nur eine einzige Zeile aus HBase zu lesen, der Rest ist die Aufbereitung

mithilfe von Diagrammen und Tabellen. Tests innerhalb Facebook haben gezeigt, dass ein Cluster aus 100 Rechnern bis zu eine Million Zähleroperationen durchführen kann, also 10 K pro Server. Obwohl Transaktionen in HBase fehlen, kann eine Anwendung eine Zeile atomar aktualisieren, egal wie viele Spalten verändert werden und über wie viele Spaltenfamilien hinweg.

Neuere Versionen von HBase, also 0.94 und danach, unterstützen auch weitere interessante Erweiterungen, zum Beispiel die *regionslokalen Transaktionen*. Diese ermöglichen es, zusammengehörige Zeilen nicht über mehrere Regionen aufzuteilen, und damit eine atomare Veränderung über Zeilengrenzen hinweg durchzuführen. Im Prinzip entspricht das den *Entitätsgruppen (Entity Groups)* aus dem Google-Megastore-Projekt [10]. Dort werden nur kleine Teilbereiche atomar aktualisiert, so wie das auch in HBase der Fall ist. Kaum eine andere NoSQL-Lösung hat etwas Vergleichbares zu bieten.

HBase hat außerdem keine eigene deskriptive Abfragesprache, wie zum Beispiel SQL. Das kann zwar mithilfe von Tools wie Hive, Impala oder Drill umgangen werden, denn diese „übersetzen" SQL-Abfragen in solche, die HBase unterstützt, aber dennoch sollte sich der Anwender im Klaren sein, dass dies keine umgangssprachliche eierlegende Wollmilchsau ist. Der Grund hierfür ist das Datenmodell und das API, das HBase zur Verfügung stellt. Sollte eine SQL-Abfrage in eine direkte Leseoperation einer Zeile, oder wenige zusammenhängende Zeilen transformiert werden können, dann ist dies ideal. Ist aber die Abfrage so freizügig, dass das Tool keine Zeilen- oder Spaltenschlüssel benutzen kann, und damit die ganze Tabelle durchsuchen muss, dann kann

Schlüssel	Beschreibung
<userId>	Abfrage über alle Nachrichten für einen bestimmten Benutzer
<userId>-<date>	Abfrage über alle Nachrichten an einem bestimmten Datum für einen bestimmten Benutzer
<userId>-<date>-<messageId>	Abfrage über alle Teile einer Nachricht für einen bestimmten Benutzer
<userId>-<date>-<messageId>-<attachmentID>	Abfrage über alle Anhänge einer Nachricht eines bestimmten Benutzers

Tabelle 1: Beispiele für mögliche Werte des Startschlüssels

Technische Eigenschaften	
Datenmodell	Wide Column basierend auf Bigtable (modifizierter LSM-Tree)
Suchmöglichkeiten	Schlüssel direkt oder über Bereichsscan (Start-/Stoppschlüssel), MapReduce
Integration in BI-Tools	Fast alle kommerziellen und freien BI-Tools, entweder über Hive, Impala etc. mit JBDC/ODBC oder über die API mit Java oder REST bzw. Thrift
Typisches Einsatzszenario	Batch und interaktiv: Time-series Database, Message Store, CMS etc.
Horizontale Skalierbarkeit	Implizit über Verteilung von Regionen auf Server (Auto Sharding)
Hochverfügbarkeit	Eingebaut auf allen Ebenen (HBase, HDFS, ZooKeeper)
Implementiert in	Java
Unterstützte Betriebssysteme	Linux (Unix, Windows mit möglichen Einschränkungen und weniger getestet)
Monitoring-Werkzeuge	JMX oder Hadoop Metrics (Ganglia Integration mitgeliefert)
Backup-Lösung	Snapshots

Tabelle 2: Technische Eigenschaften von HBase

Lizenz und Support	
Aktuelles stabiles Release	0.96
Open Source?	Ja: Apache und kommerzielle Anbieter
Lizenz	Apache License v2.0
Kosten der kommerziellen Version	Abhängig vom Anbieter
Features der kommerziellen Version	Abhängig vom Anbieter
Zusätzlicher, professioneller Support	Cloudera, Hortonworks, MapR, IBM

Tabelle 3: Details zu Lizenzen und Support

Nutzung mit der JVM	
Java-APIs	Ja: nativ, binäre Daten
APIs für andere JVM-Sprachen	Ja: über Java-Bibliothek
APIs für Non-JVM-Sprachen	Ja: REST, Thrift
Object Mapper	3rd Party, z. B. Spring Data

Tabelle 4: Clients für Java und andere JVM-Sprachen

es vorkommen, dass eine Abfrage, die erwartungsgemäß sehr schnell sein sollte, auf einmal Minuten braucht (natürlich abhängig von der Tabellengröße).

Ein zusätzliches Problem ist, dass HBase eben nur zwei Indexe unterstützt, d. h. den globalen primären Index der Zeilenschlüssel, und dann innerhalb der Zeile den Index der Spaltenschlüssel – und beide nur lexikographisch sortiert in einer Richtung. Will die Anwendung aber weitere Indexe haben, um beispielsweise einen Anwender über ID, aber auch über Namen zu finden, oder einen Suchindex pro Zeile speichern, dann muss die Anwendung um diese Einschränkung herum arbeiten. Dies geht über Nachschlagetabellen (Lookup Tables), oder der Speicherung der Indexe in einer anderen Spaltenfamilie innerhalb der gleichen Zeile. Problematisch ist immer, diese Indexe mit dem Hauptdatensatz synchron zu halten. Verschiedene Ansätze wurden innerhalb der HBase-Gemeinde über die Jahre diskutiert. Facebook Messages [11] benutzt den letztgenannten Ansatz, wobei es den Suchindex des Benutzerkontos in einer eigenen Spaltenfamilie ablegt. Damit macht Facebook Gebrauch von den atomaren Modifikationen innerhalb einer Zeile: Der Hauptdatensatz und der Index sind damit immer im Einklang.

Ein interessanter Seiteneffekt der Speicherung der Daten in HBase liegt darin, dass es nur echte Daten speichert, also keine Platzhalter braucht wie das aus Datenbanken mit festen Schemata bekannte *NULL*. **Abbildung 3** veranschaulicht, was damit gemeint ist, denn jede Zeile ist wirklich in sich selbst geschlossen. Weitere Zeilen können die gleichen, aber auch beliebig andere Spalten enthalten.

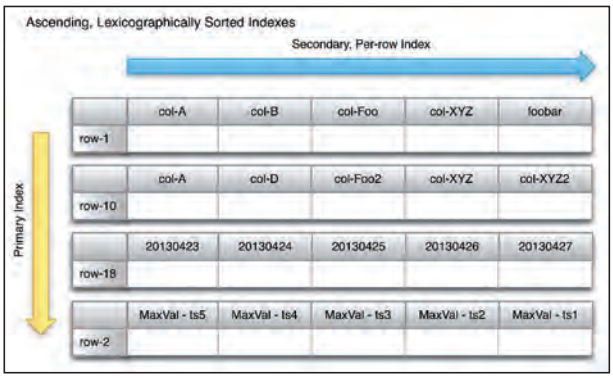


Abb. 3: Spalten pro Zeile können beliebig und völlig verschieden sein

Die Anwendung muss auch nichts weiter als die Spaltenfamilien definieren, danach kann sie während des Schreibvorgangs beliebige Spalten anlegen. Eine Tabelle hat initial keine einzige Spalte, aber wenn die Anwendung etwas in *col-A* schreibt, dann existiert die Spalte von nun an mit dem gegebenen Wert. Ein Rückschluss daraus ist, dass es keine Spalten *ohne* Wert geben kann. Dies kann sich zunutze gemacht werden, zum Beispiel

als eingebettete Datensätze oder Indexe. Generell ist also eine der Stärken von HBase die Behandlung von dünn besetzten Tabellen.

HBase hat neben dem *Put,* um Daten zu speichern, dem *Get,* um Daten einer Zeile zu lesen, und dem *Delete,* um Daten zu löschen, auch einen *Scan*-Befehl. Interessant ist, dass dieser sehr genau eingeschränkt werden kann, über Start- und Stoppschlüssel, sowie Zeitstempel. Wenn nun eine Anwendung nicht weiß, welche Schlüssel in einer Tabelle genau gespeichert sind, weil die Schlüssel beispielsweise aus Benutzerdaten generiert werden (MD5 Hash des Namens oder der Name im Klartext), dann können die Start- und Stoppschlüssel der Abfrage so gesetzt werden, dass sie automatisch einen bestimmten Bereich erfassen. Tabelle 1 listet einige Beispiele für den Startschlüssel auf und beschreibt das Ergebnis.

Die weitere Variante der Abfrage mit Zeitstempel funktioniert grundsätzlich, weil jede Zelle wie oben beschrieben einen Zeitstempel für die Versionierung der Werte mitspeichert. So kann die Anwendung zum Beispiel alle Werte, die nach einem bestimmten Datum verändert wurden, sehr effizient abfragen – was sehr nützlich ist für schrittweise, zeitgesteuerte Datenverarbeitung. Aber diese Variante hat noch einen weiteren Vorteil, denn sie kann ganze Datendateien innerhalb des Speichersystems in HBase ausschließen. Wie oben erwähnt, werden Dateien regelmäßig auf das Dateisystem geschrieben und diese Dateien haben unter anderem auch die Information, was der aktuellste und älteste Eintrag darin ist. Wenn nun für Daten aus dem letzten Tag gefragt werden, können Dateien, die älter sind, einfach übersprungen werden. Ein Datenschema, das sich dies zunutze machen kann, hat den enormen Vorteil, eine ganze Tabelle augenscheinlich in sehr schneller Zeit durchsuchen zu können – der Trick ist aber, dass nur wenige Daten wirklich gelesen werden müssen.

Zukunft

HBase hat eine große Zukunft vor sich. Die Konkurrenz aus dem NoSQL- und dem relationalen Datenbanklager ist Ansporn und Hinweis zugleich. In den letzten Jahren hat sich immer wieder gezeigt, wie Ansätze anderer Systeme sich nach kurzer Zeit auch in HBase wiederfanden. Apache und quelloffene Projekte generell erlauben es, solche Verbesserungen übernehmen zu können. Auch auf HDFS-Ebene findet dies statt: Kommerzielle Anbieter hatten vor einiger Zeit bestimmte Anforderungen erhoben, die innerhalb weniger Monate zu entsprechenden Änderungen in HDFS geführt und das Gleichgewicht wieder hergestellt haben.

Die letzten Versionen von HBase brachten interessante Neuerungen, zum Beispiel Coprocessors in 0.92 – diese fügen serverseitige Erweiterungen hinzu, die durch den Anwender selbst in Java geschrieben werden können, also ähnlich den *Stored Procedures* aus der RDBMS-Welt. Oder 0.94 [13], das einiges an Leistungssteigerungen bereitstellte und den Betrieb vereinfachte. Die aktuelle Version 0.96 (liegt zum Zeitpunkt des Schreibens in Form eines Release Candidates vor) stellt Snapshots zur Verfügung (diese wurden auch in 0.94.6.1 nachträglich eingefügt) und Google Protocol Buffer basierte RPC-Protokolle – damit ist ab dieser Version ein rollendes Aktualisieren der Software möglich, ohne den Cluster stoppen zu müssen. Snapshots wiederum sind essenziell, um Daten im laufenden Betrieb von HBase in einer Datensicherung ablegen zu können. Zuvor war dies nur im Offlinebetrieb möglich.

An allen Ecken und Enden wird verbessert und hinzugefügt, und die Entwicklergemeinschaft arbeitet gemeinsam auf eine Version 1.0 hin. Benutzer wie Facebook, eBay oder Apple zeigen, dass HBase im Mainstream nicht nur angekommen, sondern als produktives System akzeptiert worden ist. Alleine diese genannten Anwender besitzen riesige HBase-Cluster, z. B. Facebook mit mehr als zwei Petabyte komprimiert gespeicherten Daten, also mehr als 6 Petabyte roh, wenn man den Replikationsfaktor (dreifach) innerhalb von HDFS hinzurechnet, verteilt über mehrere separate Cluster. Kann man da noch zweifeln?

Lars George ist EMEA Chief Architect bei Cloudera und entwickelt datenorientierte Lösungen für Kunden und Partner. Er ist Autor des O'Reilly Buchs „HBase – The Definitive Guide" und hat auf vielen NoSQL- und Hadoop-bezogenen Konferenzen vorgetragen, darunter Hadoop Summit und Hadoop World, QCon, FOSDEM, JAX und ApacheCon.

Links & Literatur

[1] http://research.google.com/archive/gfs.html

[2] http://research.google.com/archive/mapreduce.html

[3] http://research.google.com/archive/bigtable.html

[4] http://en.wikipedia.org/wiki/Doug_Cutting

[5] http://hadoop.apache.org/

[6] http://nosqlsummer.org/paper/lsm-tree

[7] http://en.wikipedia.org/wiki/IOPS

[8] https://jaxenter.de/magazines/Java-Magazin-1111-165858

[9] http://www.cs.duke.edu/~kmoses/cps516/puma.html

[10] http://research.google.com/pubs/pub36971.html

[11] https://www.facebook.com/UsingHbase

[12] http://www.slideshare.net/cloudera/building-realtime-big-data-services-at-facebook-with-hadoop-and-hbase-jonathan-gray-facebook

[13] http://blog.cloudera.com/blog/2012/05/apache-hbase-0-94-is-now-released/

[14] http://www.hbasecon.com/2012/sessions/facebook-messages-application-server-using-hbase/

Anzeige

Anzeige

Anzeige

Anzeige

Lambda-Ausdrücke und Methodenreferenzen

Effective Java: Let's Lambda!

Wie bereits im letzten Beitrag unserer Reihe gesehen, gibt es in Java 8 neue Sprachmittel, die einen eher funktionalen Programmierstil in Java unterstützen werden. Diese neuen Sprachmittel sind die Lambda-Ausdrücke und die Methodenreferenzen. Damit wollen wir uns in diesem Beitrag näher befassen.

von Klaus Kreft und Angelika Langer

Diskussionen über Spracherweiterungen in Java für funktionale Programmierung hat es schon vor einigen Jahren gegeben. Seit der Freigabe von Java 5 wurde intensiv darüber nachgedacht, wie solche Erweiterungen aussehen könnten. Es gab drei konkrete Vorschläge in dieser als „Closure-Debatte" bekannt gewordenen Anstrengung [1]. Neil Gafter, vormals als Compilerexperte bei Sun tätig, hatte sogar einen Prototyp-Compiler für den Vorschlag gebaut, an dem er mitgewirkt hatte. Dennoch hat sich keine Konvergenz der drei Closure-Vorschläge ergeben. Es hatte auch keiner der drei Vorschläge die uneingeschränkte Unterstützung von Sun Microsystems. Als Sun Microsystems dann auch noch von Oracle übernommen wurde, verlief die Closure-Diskussion ergebnislos im Sande. Es sah zunächst so aus, als würde es in Java keine Erweiterungen für die funktionale Programmierung geben.

Im Jahr 2009 setzte sich dann die Erkenntnis durch, dass Java ohne Closures (oder Lambdas, wie sie fortan hießen) gegenüber anderen Programmiersprachen veraltet aussehen könnte. Erstens gibt es Closure- bzw. Lambda-artige Sprachmittel in einer ganzen Reihe von Sprachen, die auf der JVM laufen. Zweitens braucht man auf Multi-CPU- und Multi-Core-Hardware eine einfache Unterstützung für die Parallelisierung von Programmen. Denn was nützen die vielen Cores, wenn die Applikation sie nicht nutzt, weil sie in weiten Teilen sequenziell und nur in geringem Umfang parallel arbeitet?

Nun bietet das JDK mit seinen Concurrency Utilities im *java.util.concurrent*-Package umfangreiche Unterstützung für die Parallelisierung. Die Handhabung dieser Concurrency Utilities ist aber anspruchsvoll, erfordert Erfahrung und wird allgemein als schwierig und fehleranfällig angesehen. Eigentlich bräuchte man für die Parallelisierung bequemere, weniger fehleranfällige und einfach zu benutzende Mittel. Doug Lea, der sich

schon seit vielen Jahren um die Spezifikation und Implementierung der Concurrency Utilities in Java kümmert, hat dann prototypisch eine Abstraktion *ParallelArray* gebaut, um zu demonstrieren, wie eine Schnittstelle für die parallele Ausführung von Operationen auf Sequenzen von Elementen aussehen könnte [2]. Die Sequenz war einfach ein Array von Elementen mit Operationen, die paralleles Sortieren, paralleles Filtern sowie das parallele Anwenden von beliebiger Funktionalität auf alle Elemente der Sequenz zur Verfügung gestellt hat. Dabei hat sich herausgestellt, dass eine solche Abstraktion ohne Closures bzw. Lambdas nicht gut zu benutzen ist.

Deshalb gibt es seitdem bei Oracle unter der Leitung von Brian Goetz (der vielen Lesern vielleicht als Autor des Buchs „Java Concurrency in Practice" bekannt ist) ein „Project Lambda", d. h. eine Arbeitsgruppe, die die neuen Lambda-Sprachmittel definiert und gleichzeitig neue Abstraktionen für das JDK-Collection-Framework spezifiziert und implementiert hat [3]. Ein *ParallelArray* wird es in Java 8 zwar nicht geben; das war nur ein Prototyp, der Ideen lieferte. An seine Stelle treten so genannte *Streams*. Und aus dem anfänglich als Closure bezeichneten Sprachmittel sind im Laufe der Zeit Lambda-Ausdrücke sowie Methoden- und Konstruktorreferenzen entstanden. Diese Lambda-Ausdrücke bzw. Methoden-/Konstruktorreferenzen wollen wir uns im Folgenden genauer ansehen [4].

Wie sieht ein Lambda-Ausdruck aus?

Wir haben im letzten Beitrag bereits Lambda-Ausdrücke gezeigt, und zwar am Beispiel der Verwendung der *forEach*-Methode. In Java 8 haben alle Collections eine *forEach*-Methode, die sie von ihrem Superinterface *Iterable* erben. Das *Iterable*-Interface gibt es schon seit Java 5; es ist erweitert worden und sieht in Java 8 so aus:

```
public interface Iterable<T> {
  Iterator<T> iterator();

  default void forEach(Consumer<? super T> action) {
    for (T t : this) {
      action.accept(t);
    }
  }
}
```

Das *Iterable*-Interface hat zusätzlich zur *iterator*-Methode, die es schon immer hatte, eine *forEach*-Methode bekommen. Die *forEach*-Methode iteriert über alle Elemente in der Collection und wendet auf jedes Element eine Funktion an, die der Methode als Argument vom Typ *Consumer* übergeben wird. Die Benutzung der *forEach*-Methode sieht dann zum Beispiel so aus:

(int x) -> x+1	Parameterliste mit einem einzigen Parameter mit expliziter Typangabe
int x -> x+1	Falsch: Wenn man den Parametertyp angibt, muss man die runden Klammern verwenden
(x) -> x+1	Parameterliste mit einem einzigen Parameter ohne explizite Typangabe; der Compiler deduziert den fehlenden Parametertyp aus dem Kontext
x -> x+1	Parameterliste mit einem einzigen Parameter ohne explizite Typangabe, in diesem Fall darf man die runden Klammern weglassen
(int x,int y) -> x+y	Parameterliste mit zwei Parametern mit expliziter Typangabe
int x,int y -> x+y	Falsch: Wenn man den Parametertyp angibt, muss man die runden Klammern verwenden
(x,y) -> x+y	Parameterliste mit zwei Parametern ohne explizite Typangabe
x,y -> x+y	Falsch: Bei mehr als einem Parameter muss die runde Klammer verwendet werden
(x,int y) -> x+y	Falsch: Man darf Parameter mit und ohne Typangabe nicht mischen, entweder alle haben eine explizite Typangabe oder keiner
() -> 42	Die Parameterliste darf leer sein

Tabelle 1: Beispiele für Parameterlisten

() -> System.gc()	Body bestehend aus einem einzigen Ausdruck
(String[] args) -> (args != null) ? args.length : 0	Der ?:-Operator ergibt auch einen einzigen Ausdruck

(String[] args)
-> { if(args != null)
return args.length;
else
return 0;
}
	Body bestehend aus einer *if*-Anweisung, hier werden geschweifte Klammern gebraucht, weil es eine Anweisung und kein Ausdruck ist

(int x) -> x+1	Body bestehend aus einem einzigen Ausdruck
(int x) -> return x+1	Falsch: Mit *return* fängt eine Anweisung an und kein Ausdruck; die *return*-Anweisung muss mit einem Semikolon enden und gehört in geschweifte Klammen
(int x) -> { return x+1; }	So ist es richtig

Tabelle 2: Lambda-Body-Beispiele

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
numbers.forEach(i -> System.out.println(i));
```

Als *Consumer* haben wir einen Lambda-Ausdruck übergeben (*i -> System.out.println(i)*), der alle Integer-Werte aus der Collection nach *System.out* ausgibt.

Ein Lambda-Ausdruck besteht aus einer Parameterliste (das ist der Teil vor dem „->“-Symbol) und einem Rumpf (der Teil nach dem „->“-Symbol). Für Parameterliste und Rumpf gibt es mehrere syntaktische Möglichkeiten. Hier die vereinfachte Version der Syntax für Lambda-Ausdrücke:

```
LambdaExpression:
    LambdaParameters '->' LambdaBody
LambdaParameters:
    Identifier
    '(' ParameterList ')'
LambdaBody:
    Expression
    Block
```

Lambda-Parameterliste

Die Parameterliste ist entweder eine kommagetrennte Liste in runden Klammern oder ein einzelner Bezeichner ohne runde Klammern. Wenn man die Liste in Klammern verwendet, dann kann man sich entscheiden, ob man für alle Parameter den Parametertyp explizit hinschreiben will oder ob man den Typ weglässt und ihn vom Compiler automatisch bestimmen lässt. Tabelle 1 zeigt ein paar Beispiele.

Lambda-Body

Der Rumpf ist entweder ein einzelner Ausdruck oder eine Liste von Anweisungen in geschweiften Klammern. Tabelle 2 zeigt einige Beispiele.

Das Prinzip für die Syntax ist recht einfach. Wenn die Parameterliste oder der Rumpf ganz simpel sind, dann darf man sogar die Klammern weglassen; wenn sie ein bisschen komplexer sind, muss man die Klammern setzen.

Typdeduktion und SAM-Typen

Die Syntax für Lambda-Ausdrücke ist kurz und knapp. Es stellt sich die Frage: Wo nimmt der Compiler all die Information her, die wir weglassen dürfen? Wenn wir beispielsweise in der Parameterliste die Typen weglassen, dann muss der Compiler sich die Typen selbst überlegen. Wie macht er das? Man lässt bei den Lambda-Ausdrücken grundsätzlich den Returntyp und die Exception-Spezifikation weg. Woher nimmt der Compiler diese Information? Was ist eigentlich überhaupt der Typ eines Lambda-Ausdrucks? Dazu hatten wir im letzten Beitrag bereits erläutert, dass der Compiler den Typ eines Lambda-Ausdrucks aus dem umgebenden Kontext deduziert. Sehen wir uns das noch einmal genauer an.

Zunächst einmal hat man sich beim Design der Lambda-Ausdrücke überlegt, dass das Typsystem von Java nach Möglichkeit nicht gravierend geändert werden soll. Man hätte prinzipiell hingehen können und eine neue Kategorie von Typen für Lambda-Ausdrücke erfinden können. Dann hätte es neben primitiven Typen, Klassen, Interfaces, Enum-Typen, Array-Typen und Annotation-Typen auch noch Funktionstypen gegeben. Funktionstypen hätten Signaturen beschrieben, z. B. *void(String,String) IOException* für einen Lambda-Ausdruck, der zwei Strings als Parameter nimmt, nichts zurückgibt und *IOExceptions* wirft. Diesen heftigen Eingriff ins Typsystem wollte man aber vermeiden. Stattdessen hat man nach einer Möglichkeit gesucht, herkömmliche Typen für die Lambda-Ausdrücke zu verwenden.

Man hat sich also überlegt, welche schon existierenden Typen in Java einem Funktionstyp am ähnlichsten sind und dabei festgestellt, dass es eine ganze Menge

Interfaces gibt, die nur eine einzige Methode haben. Beispiele sind *Runnable*, *Callable*, *AutoCloseable*, *Comparable*, *Iterable* usw. Diese Interfaces beschreiben Funktionalität, und ihre einzige Methode hat eine Signatur mit Parametertypen, Returntyp und Exception-Spezifikation – also genau der Information, die auch ein Funktionstyp repräsentieren würde. Also hat man sich eine Strategie überlegt, wie man Lambda-Ausdrücke auf Interfaces mit einer einzigen Methode abbilden kann.

Solche Interfaces mit einer einzigen abstrakten Methode haben deshalb in Java 8 im Zusammenhang mit den Lambda-Ausdrücken eine besondere Bedeutung. Man bezeichnet sie als *Functional Interface Types* (bisweilen auch *SAM Types* genannt, wobei SAM für „Single Abstract Method" steht). Man kann sie mit einer speziellen Annotation, nämlich *@FunctionalInterface*, markieren. Sie sind die einzigen Typen, die der Compiler für Lambda-Ausdrücke verwenden kann.

Der SAM Type für einen Lambda-Ausdruck wird vom Java-Entwickler niemals explizit spezifiziert, sondern immer vom Compiler im Rahmen einer Typdeduktion aus dem Kontext bestimmt, in dem der Lambda-Ausdruck vorkommt. Sehen wir uns dazu Beispiele von Lambda-Ausdrücken in einem Zuweisungskontext an:

```
BiPredicate<String,String>           sp1 = (s,t) -> s.equalsIgnoreCase(t); //1
BiFunction<String,String,Boolean>  sp2 = (s,t) -> s.equalsIgnoreCase(t); //2
```

Auf der linken Seite der beiden Zuweisungen stehen Variablen vom Typ *BiPredicate<String,String>* bzw. *BiFunction<String,String,Boolean>*. *BiPredicate* und *BiFunction* sind Interfaces aus dem Package *java.util.function*, das es in Java 8 im JDK gibt. Die Interfaces sehen (vereinfacht) so aus:

```
public interface BiPredicate<T, U> {
  boolean test(T t, U u);
}
public interface BiFunction<T, U, R> {
  R apply(T t, U u);
}
```

Auf der rechten Seite der Zuweisungen steht in beiden Fällen der gleiche Lambda-Ausdruck. Wie passen linke und rechte Seite der Zuweisung zusammen?

Der Compiler schaut sich zunächst einmal an, ob die linke Seite der Zuweisung ein SAM Type ist. Das ist in beiden Zuweisungen der Fall. Dann ermittelt der Compiler die Signatur der Methode in dem SAM Type. Das *BiPredicate*-Interface in Zeile //1 hat eine *test*-Methode mit der Signatur *boolean(String,String)*. Das *BiFunction*-Interface in Zeile //2 hat eine *apply*-Methode mit der Signatur *Boolean(String,String)*.

Nun schaut der Compiler den Lambda-Ausdruck auf der rechten Seite an und prüft, ob der Lambda-Ausdruck eine dazu passende Signatur hat. Die Parametertypen fehlen im Lambda-Ausdruck. Da auf der linken Seite *Strings* als Parameter verlangt werden, nimmt der Compiler an, dass auf der rechten Seite *s* und *t* vom Typ *String* sein sollten. Dann wird geprüft, ob die *String*-Klasse eine Methode *equalsIgnoreCase* hat, die einen *String* als Argument akzeptiert. Diese Methode existiert in der *String*-Klasse; sie gibt einen *boolean*-Wert zurück und wirft keine checked Exceptions. Die Exception-Spezifikation passt also, der Returntyp passt im ersten Fall auch und im zweiten Fall mithilfe von Autoboxing.

Wie man sieht, hat der Compiler im Laufe dieses Deduktionsprozesses nicht nur die fehlenden Parametertypen des Lambda-Ausdrucks bestimmt, sondern auch den Returntyp und die Exception-Spezifikation. Außerdem hat er einen SAM Type für jeden der Lambda-Ausdrücke gefunden.

Deduktionskontext

Ein Lambda-Ausdruck kann im Sourcecode nur an Stellen stehen, wo es einen Deduktionskontext gibt, den der Compiler auflösen kann. Zulässig sind Lambda-Ausdrücke deshalb nur an folgenden Stellen:

- auf der rechten Seite von Zuweisungen (wie im obigen Beispiel)
- als Argumente in einem Methodenaufruf
- als Returnwert in einer *return*-Anweisung
- in einem Cast-Ausdruck

Der Deduktionsprozess ist in allen Fällen ähnlich. Den Zuweisungskontext haben wir uns im obigen Beispiel bereits angesehen: Bei der Zuweisung ist der Typ auf der linken Seite der Zuweisung der Zieltyp, zu dem der Lambda-Ausdruck auf der rechten Seite kompatibel sein muss. Beim Methodenaufruf ist der deklarierte Parametertyp der aufgerufenen Methode der Zieltyp, zu dem der Lambda-Ausdruck kompatibel sein muss. Bei der *return*-Anweisung ist der deklarierte Returntyp der Methode, in der die *return*-Anweisung steht, der Zieltyp. Beim Cast-Ausdruck ist der Zieltyp des Casts der Zieltyp für den Lambda-Ausdruck.

Es kann aber auch vorkommen, dass ein Lambda-Ausdruck in einem zulässigen Kontext vorkommt und die Typdeduktion dennoch scheitert. Hier ist ein Beispiel:

```
Object o = (s,t) -> s.equalsIgnoreCase(t);  // error: Object is not a functional
                                            // type
```

Das ist ein Zuweisungskontext und deshalb prinzipiell erlaubt, aber der Typ *Object* auf der linken Seite ist kein SAM-Typ. Also scheitert die Typdeduktion. Hier kann man sich behelfen, indem man einen Cast einfügt.

```
Object o = (BiPredicate<String,String>)(s,t) -> s.equalsIgnoreCase(t);
```

Jetzt steht der Lambda-Ausdruck in einem Cast-Kontext, und der Zieltyp des Casts ist ein SAM-Typ, mit dem der Compiler die erforderliche Typdeduktion durchführen kann.

Wir haben nun die Syntax für Lambda-Ausdrücke kennengelernt und gesehen, dass der Typ eines Lambda-Ausdrucks immer vom Compiler aus dem Kontext deduziert wird und immer ein SAM-Typ sein muss. Was darf nun im Rumpf eines Lambda-Ausdrucks stehen? Genauer gesagt, auf welche Variablen und Felder hat man im Lambda-Body Zugriff?

Variable Binding

Im Lambda-Body hat man natürlich Zugriff auf die Parameter und lokalen Variablen des Lambda-Ausdrucks. Manchmal möchte man aber auch auf Variablen des umgebenden Kontexts zugreifen. Hier ist ein einfaches Beispiel. Wir verwenden darin den SAM Type *IntUnaryOperator* aus dem *java.util.function*-Package. Dieser Typ sieht so aus:

```
@FunctionalInterface
public interface IntUnaryOperator {
  int applyAsInt(int operand);
}
```

Das Beispiel selbst verwendet diverse Abstraktionen aus dem Stream-Framework und sieht so aus:

```
private static void test() {
  int factor = 1000;                                          //1
  IntUnaryOperator times1000 = (int x) -> { return x * factor; };  //2
  Arrays.stream(new int[]{1, 2, 3, 4, 5}).map(times1000).forEach(
                                              System.out::println); //3
}
```

Nur kurz zur Erläuterung: In Zeile //3 machen wir aus einem *int*-Array einen *Stream*, dessen *map*-Methode wir benutzen, um alle Elemente in dem Array mithilfe der Funktion *times1000* auf einen neuen *int*-Wert abzubilden. Anschließend werden die neuen Werte nach *System.out* ausgegeben. Eigentlich geht es aber um den blau eingefärbten Lambda-Ausdruck.

Wir verwenden im Lambda-Body nicht nur den Parameter *x* des Lambda-Ausdrucks, sondern auch die Variable *factor* aus dem umgebenden Kontext. Das ist erlaubt. Alle Variablen, die im Lambda-Ausdruck verwendet werden, aber nicht im Lambda-Ausdruck selbst definiert wurden, haben dieselbe Bedeutung wie im umgebenden Kontext. Die einzige Voraussetzung ist, dass die betreffenden lokalen Variablen „effectively final" sind, d. h. sie dürfen nicht geändert werden – weder im Lambda-Ausdruck noch im umgebenden Kontext. Folgendes wäre also falsch:

```
private static void test() {
  int factor = 1000;
  IntUnaryOperator times1000 = (int x) -> { return x * factor; };
  ...
  factor = 1_000_000;  // error: local variable used in lambda must be final or
                       // effectively final
  ...
}
```

Dieses Binden von Namen in einem Lambda-Ausdruck an lokale Variablen, die außerhalb des Lambda-Ausdrucks definiert sind, ähnelt dem Binding, das auch in lokalen und anonymen Klassen erlaubt ist. Lokale und anonyme Klassen hatten schon immer Zugriff auf *final*-Variablen des umgebenden Kontexts. In Java 8 hat man übrigens die Regeln gelockert. Analog zu den Lambda-Ausdrücken haben in Java 8 auch die lokalen und anonymen Klassen Zugriff auf alle „effectively final"-Variablen des umgebenden Kontexts. Eigentlich ist alles so wie vorher, nur muss man das *final* nicht mehr explizit hinschreiben; der Compiler ergänzt es einfach, sobald eine Variable in einer lokalen oder anonymen Klasse (oder in einem Lambda-Ausdruck) verwendet wird.

Lambda-Ausdrücke haben außerdem Zugriff auf Felder der Klasse, in der sie definiert sind. Hier ist ein Beispiel:

```
class Test {
  private int factor = 1000;
  public void test() {
    IntUnaryOperator times1000 = x -> x * factor;
    Arrays.stream(new int[]{1, 2, 3, 4,
      5}).map(times1000).forEach(System.out::println);
    factor = 1_000_000;   // fine
  }
}
```

Dieses Mal ist *factor* keine lokale Variable in der Methode, in der der Lambda-Ausdruck vorkommt, sondern ein Feld der Klasse, in der der Ausdruck definiert ist. Bei Feldern wird nicht verlangt, dass sie *final* oder „effectively final" sein müssen. Der Lambda-Ausdruck hat ganz normalen, uneingeschränkten Zugriff darauf. Auch dies gilt für Lambda-Ausdrücke wie bisher für Inner Classes.

Die Ähnlichkeit der Regeln für Inner Classes und Lambda-Ausdrücke ist nicht verwunderlich. Denn Lambda-Ausdrücke ähneln anonymen Klassen, die Interfaces mit genau einer abstrakten Methode implementieren. Verglichen mit anonymen Klassen verzichten die Lambda-Ausdrücke dabei auf jeglichen Syntax-Overhead. Dafür muss der Compiler bei ihnen deutlich mehr Arbeit leisten und, wie weiter oben beschrieben, die fehlende Information aus dem Kontext deduzieren.

Methoden- und Konstruktorreferenzen

Neben den Lambda-Ausdrücken gibt es die Methoden- und Konstruktorreferenzen, die von der Syntax her noch kompakter als die Lambda-Ausdrücke sind. Wenn man in einem Lambda-Body ohnehin nichts weiter tut, als eine bestimmte Methode aufzurufen, dann kann man den Lambda-Ausdruck häufig durch eine Methodenreferenz ersetzen. Das lässt sich an unserem *forEach*-Beispiel von oben demonstrieren. Hier ist noch einmal das Originalbeispiel:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
numbers.forEach(i -> System.out.println(i));
```

Anstelle des Lambda-Ausdrucks kann man eine Methodenreferenz verwenden. Dann sieht es so aus:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
numbers.forEach(System.out::println);
```

Alles bisher über Lambda-Ausdrücke Gesagte gilt auch für Methodenreferenzen: Sie dürfen nur in einem Kontext vorkommen, in dem der Compiler eine Typdeduktion machen und einen SAM Type für die Methodenreferenz bestimmen kann. Der Deduktionsprozess ist ähnlich, lediglich mit dem Unterschied, dass der Compiler für eine Methodenreferenz noch mehr Informationen deduzieren muss. Beispielsweise fehlt bei einer Methodenreferenz nicht nur der Typ der Parameter, sondern auch jegliche Information über die Anzahl der Parameter.

Syntaktisch betrachtet besteht eine Methodenreferenz aus einem *Receiver* (das ist der Teil vor dem „::“-Symbol) und einem Methodennamen (das ist der Teil nach dem „::“-Symbol). Der Receiver kann – wie im obigen Beispiel – ein Objekt sein; es kann aber auch ein Typ sein. Der Methodenname ist entweder der Name einer existierenden Methode oder *new*; mit *new* werden Konstruktoren referenziert. Sehen wir uns einige Beispiele an.

StringBuilder::new ist eine Konstruktorreferenz. Der Receiver ist in diesem Fall kein Objekt, sondern ein Typ, nämlich die Klasse *StringBuilder*. Offensichtlich wird ein Konstruktor der *StringBuilder*-Klasse referenziert. Die *StringBuilder*-Klasse hat aber eine ganze Reihe von überladenen Konstruktoren. Welcher der Konstruktoren mit *StringBuilder::new* gemeint ist, hängt vom Kontext ab, in dem die Konstruktorreferenz auftaucht. Hier ist ein Beispiel für einen Kontext, in dem die Konstruktorreferenz *StringBuilder::new* vorkommt:

```
ThreadLocal<StringBuilder> localTextBuffer =
                                    ThreadLocal.withInitial(StringBuilder::new);
```

Die *withInital*-Methode der Klasse *ThreadLocal* sieht so aus:

```
public static <T> ThreadLocal<T> withInitial(Supplier<? extends T> supplier) {
  return new SuppliedThreadLocal<>(supplier);
}
```

Der verwendete SAM Type *Supplier* sieht so aus:

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

Der Compiler deduziert aus diesem Kontext, dass die Konstruktorreferenz *StringBuilder::new* vom Typ *Supplier<StringBuilder>* sein muss, d. h. eine Funktion, die keine Argumente nimmt und einen *StringBuilder* zurückgibt. Es ist also in diesem Kontext der No-Argument-Konstruktor der *StringBuilder*-Klasse gemeint. Hier ist ein anderer Kontext, in dem die Konstruktorreferenz *StringBuilder::new* vorkommt:

```
char[] suffix = new char[] {'.','t','x','t'};
Arrays.stream(new String[] {"readme", "releasenotes"})
    .map(StringBuilder::new)
    .map(s->s.append(suffix))
    .forEach(System.out::println);
```

Hier taucht die Konstruktorreferenz als Argument der *map*-Methode eines *Stream<String>* auf. Die betreffende *map*-Methode sieht so aus:

```
public interface Stream<T>
  <R> Stream<R> map(Function<? super T, ? extends R> mapper);
}
```

Der verwendete SAM Type *Function* sieht folgendermaßen aus:

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

In diesem Kontext deduziert der Compiler, dass die Konstruktorreferenz *StringBuilder::new* vom Typ *Function<String,StringBuilder>* sein muss, also eine Funktion, die einen *String* als Argument nimmt und einen *StringBuilder* zurückgibt. Es ist also in diesem Kontext der Konstruktor der *StringBuilder*-Klasse gemeint, der einen *String* als Argument akzeptiert.

Wie man sieht, sind Methoden- und Konstruktorreferenzen sehr flexibel, weil mit einem einzigen syntaktischen Gebilde wie *StringBuilder::new* eine ganze Reihe von Methoden bzw. Konstruktoren bezeichnet werden und der Compiler den richtigen von allein herausfindet.

In den obigen Beispielen haben wir Konstruktorreferenzen gesehen. Der Receiver ist dabei immer ein Typ. Bei Methodenreferenzen ist als Receiver neben einem Typ alternativ auch ein Objekt erlaubt. Das sieht man am Beispiel von *System.out::println*. Wir haben diese Methodenreferenzen mehrfach als Argument der *forEach*-Methode benutzt. Zum Beispiel hier:

```
char[] suffix = new char[] {'.','t','x','t'};
Arrays.stream(new String[] {"readme", "releasenotes"})
    .map(StringBuilder::new)
    .map(s->s.append(suffix))
    .forEach(System.out::println);
```

Die betreffende *forEach*-Methode sieht so aus:

```
public interface Stream<T>
  void forEach(Consumer<? super T> action);
}
```

Und der verwendete SAM Type *Function* so:

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
}
```

In diesem Kontext muss die Methodenreferenz *System.out::println* vom Typ *Consumer<? super StringBuilder>* sein, also eine Methode, die einen *StringBuilder* oder einen Supertyp von *StringBuilder* als Argument nimmt und nichts zurückgibt. Nun ist das Objekt *System.out* vom Typ *PrintStream* und die Klasse *PrintStream* hat eine passende nicht statische *println*-Methode, die ein *Object* (also einen Supertyp von *StringBuilder*) als Argument nimmt. Diese *println*-Methode ist aber nicht statisch und benötigt daher für den Aufruf ein Objekt vom Typ *PrintStream*, auf dem sie gerufen wird, und das *Object*, das als Argument übergeben wird. Eigentlich hat die *println*-Methode die Signatur *void(PrintStream,Object)*, d. h. sie braucht zwei Objekte für den Aufruf.

Wenn man nun als Receiver für die *println*-Methode nicht den Typ *PrintStream* angibt, sondern ein *PrintStream*-Object wie z. B. *System.out*, dann ist das erste Argument bereits versorgt, und die Methodenreferenz hat die Signatur *void(Object)*, d. h. sie braucht nur noch ein Objekt für den Aufruf.

Es macht also einen Unterschied, wie ich eine Methodenreferenz hinschreibe. Die Referenz *PrintStream::println* hat die Signatur *void(PrintStream,Object)* mit zwei Argumenten; die Referenz *System.out::println* hat die Signatur *void(Object)* mit nur einem Argument.

Die Verwendung von Objekten als Receiver in einer Methodenreferenz ist nur für nicht statische Methoden möglich, denn statische Methoden kann man über den Typ aufrufen; sie brauchen kein Objekt, auf dem sie aufgerufen werden.

Zusammenfassung und Ausblick

Wir haben uns in diesem Beitrag die Lambda-Ausdrücke und Methoden- bzw. Konstruktorreferenzen näher angesehen. Betrachtet haben wir die Syntaxvarianten, die automatische Typdeduktion, die besondere Bedeutung der Functional Interface Types (aka SAM Types) und den Zugriff auf Variablen des umgebenden Kontexts aus einem Lambda-Body heraus. Damit hat man alle Mittel in der Hand, um Lambda-Ausdrücke und Methoden- bzw. Konstruktorreferenzen nutzen zu können.

Im nächsten Beitrag sehen wir uns weitere Sprachneuerungen an, die mit Java 8 freigegeben werden: die Default-Methoden. Interfaces dürfen in Java 8 nicht nur abstrakte Methoden haben, sondern auch Methoden mit einer Implementierung. Damit sind Interfaces keine reinen Abstraktionen mehr. Wir sehen uns an, was das bedeutet.

Angelika Langer arbeitet selbstständig als Trainer mit einem eigenen Curriculum von Java- und C++-Kursen.

www.AngelikaLanger.com

Klaus Kreft arbeitet selbstständig als Consultant und Trainer.

www.AngelikaLanger.com

Links & Literatur

[1] Kreft, Klaus; Langer, Angelika: „Understanding the closures debate“: http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html

[2] Goetz, Brian: „Accelerate sorting and searching with the ParallelArray classes in Java 7“: http://www.ibm.com/developerworks/java/library/j-jtp03048/index.html

[3] Project Lambda: http://openjdk.java.net/projects/lambda/

[4] Langer, Angelika; Kreft, Klaus: „Lambda Tutorial“: http://www.AngelikaLanger.com/Lambdas/Lambdas.html

Design Patterns automatisieren mit Xtends Active Annotations

Schluss mit Copy and Paste!

Design Patterns werden oft als besonders fortschrittlich oder professionell dargestellt. Und Softwareentwickler, die nicht zumindest das Buch der Gang of Four [1] gelesen haben oder aus dem Stand erklären können, wie z. B. ein Visitor-Pattern zu implementieren ist, gelten als weniger erfahren. Aber sind Design Patterns wirklich das Nonplus-ultra? Oder vielmehr ein Ergebnis fehlender Sprachmittel, um diese Muster wiederverwendbar – also als Bibliothek – zu implementieren?

von Sven Efftinge

In der Java-Welt gibt es auffällig viele Quellcodemuster. Dazu zählen nicht nur die allgemein gängigen Entwurfsmuster der Gang of Four (GoF), sondern vor allem auch unzählige frameworkspezifische Idiome. Richtig interessant wird es, wenn wir die eigenen Projekte betrachten: Die konkrete Bedeutung und Implementierung eines „Service" oder einer „Entität" variiert von Projekt zu Projekt sehr stark. Hier wird dann häufig in Wikis oder anderer technischer Dokumentation erklärt, wie so ein Projektkonzept in Java-Konzepte übersetzt wird – ein Programmier- oder Entwurfsmuster wird beschrieben, damit die Entwickler es möglichst genau und fehlerfrei kopieren.

Natürlich wissen wir als Entwickler, dass Copy and Paste keine nachhaltige Programmiertechnik ist. Der dadurch entstehende redundante Quelltext verlangsamt zukünftige Veränderungen immens und macht unseren Code unleserlich. Die Wartbarkeit leidet.

Ein einfaches Beispiel

Stellen Sie sich vor, Sie implementieren ein typisches Datenmodell für eine Businessanwendung und wollen diesem Modell das Observer-Pattern beibringen. Das ist im Prinzip recht einfach: Sie müssen lediglich die Möglichkeit bieten, einen *Observer* an den Entitäten anzumelden und dann in allen Setter-Methoden bei Änderungen entsprechende *Events* zu erzeugen und zu werfen. Wie viele Setter-Methoden gibt es wohl in einem typischen Businessdomänenmodell? Das variiert natürlich stark, aber ich denke, mehrere Hundert bis Tausend sind keine Seltenheit. „Na gut", denkt Ihr Manager, „da setzen wir jemanden ein bis zwei Wochen dran und dann ist das erledigt." Klar hat der Kollege hier und da einen Fehler eingebaut (bei so stupider Arbeit ist man leicht mal unkonzentriert), aber die findet und behebt man im Laufe der Zeit. Den Beton an den Füßen spüren wir leider erst, wenn neue Anforderungen bzgl. dieses Musters entstehen. Stellen Sie sich vor, dass im Nachhinein auch Pre-Change-Events und die Möglichkeit, ein Veto durchzusetzen, unterstützt werden müssen. Oder wir wollen eine gänzlich andere Technologie benutzen, z. B. das Data Binding aus JavaFX. Wieder den armen Kollegen dran setzen? Dafür muss es doch auch eine bessere Lösung geben.

Makros

Paul Graham schrieb 2002 in einem sehr berühmten Essay [2] über Lisp Folgendes: „In the OO world you hear a good deal about ‚patterns'. When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough – often that I'm generating by hand the expansions of some macro that I need to write."

Aha, in Lisp benutzt man also so genannte *Makros*, um sich wiederholende Muster automatisch zu expandieren. Das heißt statt dieselben Muster immer und immer wieder von Hand aus Büchern oder anderen Dokumenten abzuschreiben, wird ein Makro implementiert und dort genau einmal erklärt, wie dieses Muster aussieht. Ein Makro in Lisp ist dabei nichts anderes als eine Funktion, die nicht wie üblich zur Laufzeit ausgeführt wird, sondern zur Übersetzungszeit. Als Ergebnis wird wieder Quellcode ausgespuckt. Es handelt sich also letztendlich um eine Codegenerierungstechnologie, aber eben um eine sehr mächtige und nützliche, die direkt in die Sprache integriert ist.

Xtend Active Annotations

Ich möchte Ihnen in diesem Artikel zeigen, wie Sie auch in Ihrem Java-Projekt beliebige Entwurfsmuster und Idiome in einer Bibliothek implementieren können und automatisch zu leserlichem Java-Quelltext expandieren

können. Das Wundermittel heißt *Active Annotations* und ist ein Feature von Eclipse Xtend (Kasten: „Eclipse Xtend erweitert Java“). Es ist ähnlich wie bei Makros in Lisp tief in die Sprache integriert, lebt aber im Gegensatz zu Lisp in der Java-Welt. Das bedeutet statische Typisierung ohne Überraschungen bzgl. der Java-Interoperabilität und nahtlose Integration in die Eclipse-IDE. Am besten lässt sich das an einem konkreten Beispiel erläutern. Ich habe mich hier für die Automatisierung eines Neo4j-Idioms entschieden, weil es zu diesem Thema einen weiteren Artikel in dieser Ausgabe gibt (s. Seite 87), und um zu zeigen, dass wir redundante Strukturmuster in beliebigen Java-Frameworks finden.

Problemstellung: Java und Neo4j

Neo4j ist eine in Java implementierte Graphdatenbank, die Daten in Graphen statt in Tabellen strukturiert speichert. Neo4j bietet ein sehr einfaches Programmiermodell. Im Grunde besteht es aus nur zwei Konzepten: *Nodes* und *Relationships*. Schön unkompliziert, aber leider ohne Schema oder Klassenmodell, das in statisch typisierter Form das eigene Datenmodell widerspiegelt. Deshalb müssen wir Java-Entwickler erst einmal ohne Autovervollständigung und statische Analyse zur Übersetzungszeit auskommen. Stattdessen arbeitet man mit *Nodes* in etwa wie mit einer *HashMap*. In Listing 1 sehen Sie an einem Beispiel, wie dies konkret aussieht.

Das ist natürlich ein sehr fehleranfälliges und überhaupt nicht idiomatisches Programmiermodell für Java. Deshalb beschreiben die Neo4j-Entwickler in [3], wie man eine *Node* in ein Domänenobjekt kapseln kann. Listing 2 zeigt den konkreten Vorschlag von der Neo4j-Webseite.

Mit solchen Klassen kann man schon eher Java-artig programmieren. Nur eben die Deklaration der Klassen und vor allem die Wartung scheint recht umfangreich. Bevor wir diese Aufgabe wieder unserem Lieblingskollegen aufdrücken, möchte ich Ihnen zeigen, wie solche Muster mit einer Active Annotation automatisch generiert werden können.

Listing 1

```
Node person = graphDb.createNode();
// schreiben
person.setProperty( "firstName", "Han" );
person.setProperty( "name", "Solo" );
person.setProperty( "age", 42 );
// und lesen
System.out.println(person.getProperty("firstName")
   +" "+person.getProperty("name")+" ist "
   +person.getProperty("age")+" Jahre alt.");
```

Listing 2

```
public class Person {
  private final Node underlyingNode;
  public Person(Node personNode) {
    this.underlyingNode = personNode;
  }
  protected Node getUnderlyingNode() {
    return underlyingNode;
  }
  public String getName() {
    return (String) underlyingNode.getProperty("name");
  }
  @Override public int hashCode() {
    return underlyingNode.hashCode();
  }
  @Override public boolean equals(Object o) {
    return o instanceof Person
        && underlyingNode.equals(((Person) o).getUnderlyingNode());
  }
  @Override public String toString() {
    return "Person[" + getName() + "]";
  }
}
```

Active Annotations

Das Ziel soll sein, Datenobjekte wie in Listing 2 in kompakterer Form zu deklarieren und an genau einer Stelle zu definieren, wie die Implementierung von Eigenschaften aussieht. Wir trennen dazu den Quellcode in zwei Aspekte: einen fachlichen und einen technischen Teil. Der fachliche Anteil wird in Xtend implementiert. Das ist nötig, weil wir die Klassen mit einer Active Annotation deklarieren wollen und der Java-Compiler diese nicht versteht bzw. als normale Annotation interpretieren würde. Unser Muster würde deshalb nicht expandieren. Die Deklaration der Person erfolgt folgendermaßen:

```
@NeoNode class Person {
  String firstName
  String name
  Integer age
}
```

Eclipse Xtend erweitert Java

Xtend ist eine statisch getypte Programmiersprache, die unter einer Open-Source-Lizenz bei eclipse.org entwickelt wird. Im Gegensatz zu anderen JVM-Sprachen wird Xtend-Code in verständlichen Java-Quelltext übersetzt. Dies stellt einerseits Transparenz sicher und sorgt andererseits dafür, dass Xtend überall dort verwendet werden kann, wo heute Java zum Einsatz kommt. Das schließt unter anderem auch Plattformen wie GWT, Android oder RoboVM ein. Auch alle Java-Werkzeuge wie JavaDoc, Findbugs oder Profiler können problemlos genutzt werden. Xtend selbst bringt keine eigene Standardbibliothek mit, sondern ist darauf ausgerichtet, mit dem JDK und gängigen Java-Bibliotheken, wie z. B. Googles Guava zu arbeiten. Die Sprache ist so entworfen, dass gängige Java-Idiome, wie z. B die JavaBeans-Konvention, direkt unterstützt werden. Die Konzepte von Xtend erlauben es uns Entwicklern, wirklich leserlichen Quelltext ohne jeglichen Boilerplate und andere Arten von Redundanzen zu schreiben. Das ist besonders an Stellen wichtig, die häufig gelesen und verändert werden müssen. Xtend ist also nicht unbedingt eine Alternative zu Java, sondern eine komplementäre Erweiterung.

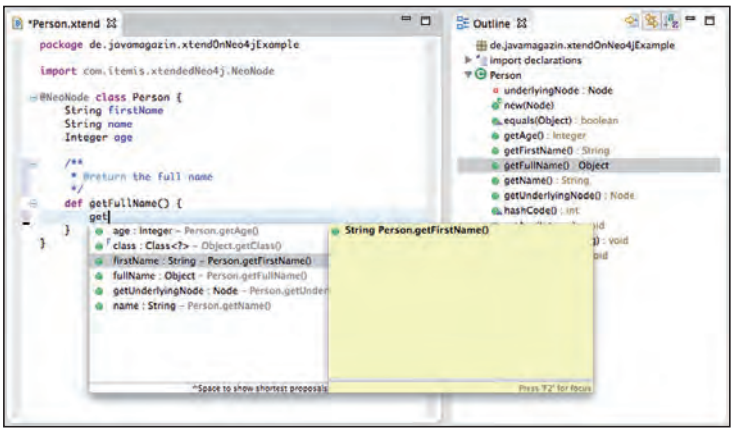


Abb. 1: Die Auswirkungen der Annotation sind sowohl in der Autovervollständigung als auch in der Outline-View sichtbar

Wie Sie sehen, enthält die mit *@NeoNode* annotierte Klasse wirklich nur noch Informationen über das Schema, aber keinerlei Implementierung, um diese auf eine Neo4j-Node abzubilden. Wir müssen nun zunächst die verwendete Annotation deklarieren. Dies können wir entweder in Java oder aber ebenfalls in Xtend tun. Ich verwende Xtend, weil es wesentlich einfacher ist, vor allem, wenn wir als Nächstes den Annotationsprozessor implementieren. Eine wichtige Einschränkung von Active Annotations ist, dass diese nicht im selben Projekt deklariert und verwendet werden dürfen. Das liegt daran, dass bei der Übersetzung eines Projekts die referenzierten Annotationsprozessoren ausgeführt werden und deshalb bereits übersetzt sein müssen. Wir deklarieren die Annotation *@NeoNode* also in einem separaten Projekt:

```
@Active(NeoNodeProcessor)
@Target(ElementType.TYPE)
annotation NeoNode { }
```

Eine Annotation wird aktiv, indem sie mit *@Active* annotiert wird. Die *@Active*-Annotation bekommt als Parameter den Klassennamen des Prozessors. Dieser Prozessor wird dann während der Übersetzung vom Compiler aufgerufen und bietet uns damit die Möglichkeit, am Übersetzungsvorgang teilzunehmen. Wir werden dies im Folgenden nutzen, um aus unserer einfachen *Person*-Klasse (Listing 2) mit drei Feldern eine Neo4j-kompatible Klasse wie in Listing 1 zu erzeugen.

Annnotationsprozessor

Wir beginnen mit der Deklaration. Da *@NeoNode* auf Klassen angewendet wird, erweitert unser Prozessor die Klasse *AbstractClassProcessor*. Ein Prozessor kann an drei unterschiedlichen Übersetzerphasen teilnehmen. Die erste Phase ist dazu da, neue Klassen, Interfaces oder auch Annotationen und Enums zu registrieren. Man könnte also aus einer Klasse beliebig viele neue Java-Typen erzeugen, was wir für dieses Beispiel aber nicht benötigen. In der zweiten Phase bekommen wir die Möglichkeit, das Java-Modell zu verändern. In der dritten Phase können wir beliebigen Text generieren oder aktualisieren. Das ist z. B. nützlich, um Konfigurationsdateien wie z. B. eine *web.xml* automatisch mit dem Quelltext zu synchronisieren. Für unser Beispiel benötigen wir lediglich die zweite Phase, weshalb wir die *doTransform*-Methode überschreiben. Die Details über die verschiedenen Phasen und der Zuständigkeiten sind im Java-Doc erläutert:

```
class NeoNodeProcessor extends AbstractClassProcessor {
  override doTransform(MutableClassDeclaration annotatedClass,
                       extension TransformationContext context) {
    // Transformationscode hier
  }
}
```

Beim Aufruf bekommen wir die Klassendeklaration vom Übersetzer eingereicht. Wie der Name *MutableClassDeclaration* schon suggeriert, ist dieses Objekt veränderbar. Der zweite Parameter (*TransformationContext*) wird als so genannte *extension* benutzt und bietet viele nützliche Extension-Funktionen [4], die in diesem Kontext wichtig sind. Wir werden z. B. als Erstes ein Feld vom Typ *Node* erzeugen, weshalb wir uns mittels der *extension*-Funktion *newTypeReference()* eine Typreferenz für *Node* erzeugen:

Listing 3

```
annotatedClass.addConstructor [
  addParameter("node", nodeType)
  body = ['''
    this.underlyingNode = node;
  ''']
]
annotatedClass.addMethod("getUnderlyingNode") [
  returnType = nodeType
  body = ['''
    return this.underlyingNode;
  ''']
]
```

Listing 4

```
for (field : declaredFields) {
  annotatedClass.addMethod("get"+field.simpleName.toFirstUpper) [
    returnType = field.type
    body = ['''
      return  («field.type.simpleName»)underlyingNode
                .getProperty("«field.simpleName»");
    ''']
  ]
  annotatedClass.addMethod("set"+field.simpleName.toFirstUpper) [
    addParameter(field.simpleName, field.type)
    body = ['''
      underlyingNode.setProperty(
            "«field.simpleName.toUpperCase»", «field.simpleName»);
    ''']
  ]
}
```

```
val nodeType = Node.newTypeReference()
val declaredFields = annotatedClass.declaredFields

annotatedClass.addField("underlyingNode") [
  type = nodeType
]
```

In der zweiten Zeile merken wir uns die deklarierten Felder in einer lokalen Variable. Jetzt können wir das Feld für die *Node* erzeugen. Wir rufen dazu *addField* an der *annotatedClass* auf und geben als ersten Parameter den Namen des Felds an. Im Block mit den eckigen Klammern wird das Feld initialisiert. Hier muss lediglich der Typ gesetzt werden.

Die Blöcke in eckigen Klammern sind übrigens Lambda-Ausdrücke und funktionieren wie in Java 8, mit dem Unterschied, dass sie syntaktisch ein wenig leichtgewichtiger sind. Xtend arbeitet also nicht nur mit allen Bibliotheken seit Java 5 zusammen, sondern ist auch für die Zukunft gerüstet. Einen Vergleich zwischen den Lambdas in Xtend und in Java 8 finden Sie unter [5].

Als Nächstes erzeugen wir einen Konstruktor, der das Feld initialisiert, sowie eine *Getter*-Methode für die *Node*. Dies geschieht analog zur Erzeugung eines Felds (Listing 3).

Der Rumpf (*body*) einer Methode oder eines Konstruktors wird in Form von Java-Code implementiert. Wir benutzen hierzu die Templateausdrücke aus Xtend, die es nicht nur erlauben, mehrzeilige Stringliterale zu schreiben, sondern auch ein sehr intelligentes Whitespace Handing besitzen. Dies sorgt dafür, dass sowohl die Templates als auch der generierte Code vernünftig formatiert sind. Mehr Informationen zum Templateausdruck finden sie in der Dokumentation [6].

Das Feld für die *Node* und die Zugriffmethode könnte man natürlich auch in eine Basisklasse auslagern. Dann müsste nur noch der Konstruktor von Hand geschrieben werden. Der wesentlich spannendere Teil ist die Erzeugung der Zugriffsmethoden für die einzelnen Eigenschaften. Dazu iterieren wir über die *declaredFields* und erzeugen für jedes Feld eine *Getter*- sowie eine *Setter*-Methode, die entsprechend auf die Neo4j-Node delegiert (Listing 4).

Nun müssen nur noch *equals* und *hashcode* überschrieben werden, die dann ebenfalls jeweils auf die *Node* delegieren. Und zu guter Letzt entfernen wir einfach die *declaredFields*, da wir diese nicht benötigen. Den vollständigen Prozessor finden Sie in Listing 5.

Der Übersetzungsvorgang

Das Annotation Processing geschieht während der Übersetzung, d.h. einerseits automatisch im Build-Prozess (z.B. im Maven-Compiler), andererseits auch in der IDE. Tatsächlich wird schon während des Tippens übersetzt, sodass Sie die Änderungen bereits in ungespeicherten Editoren sehen und benutzen können. Dabei wird die erzeugte Struktur ganz transparent gezeigt, weil sie ja auch das Programmiermodell darstellt. Alle Funktionen in Eclipse wissen deshalb, was eine mit *@NeoNode* annotierte Klasse ist und wie sie von außen aussieht. Der Screenshot in **Abbildung 1** zeigt es genauer: Hier reflektieren sowohl die Autovervollständigung als auch die Outline-View die Auswirkungen unserer Annotation. Sie können natürlich auch normale Funktionen in Ihrem Neo4j-Modell einfügen.

Weiterhin können Änderungen am Prozessor direkt im gleichen Workspace gemacht werden. Das resultiert in sehr schnellen und agilen Turn-Arounds. Für die Entwicklung einer Active Annotation empfiehlt sich ein Test-First-Ansatz, da Sie auf diese Weise den Prozessor auch debuggen können. Für diesen Zweck wird eine separate Bibliothek bereitgestellt, die das Testschreiben zum Kinderspiel macht. Wie das genau geht, wird in der Dokumentation erklärt. Weiterhin gibt es auch einen Example-Wizard in Eclipse, der Ihnen zwei Projekte mit jeder Menge Beispielen und dazugehörigen Tests erzeugt. Das in diesem Artikel beschriebene Beispiel finden Sie zusammen mit einem Unit Test auf GitHub [7].

Ausblick

Xtend selbst bietet noch viele andere spannende Möglichkeiten, um perfekte APIs und Abstraktionen zu definieren. Ich setze Xtend gerne in Java-Projekten genau an den Stellen ein, wo Java selbst einfach nicht mehr reicht und man ansonsten auf schlecht integrierte externe Lösungen wie XML und Templatesprachen oder auch solche Dinge wie Reflection zurückgreifen würde. Zum Einstieg empfiehlt sich die Implementierung der Unit Tests mit Xtend. Xtend selbst ist leider in diesem Artikel ein wenig kurz gekommen. Es ist aber für erfahrene Java-Entwickler sehr leicht zu erlernen, da es viele Dinge aus Java übernimmt.

Auch die Active Annotations konnten hier nur im Ansatz gezeigt werden. Das Potenzial als flexible und gleichzeitig mächtige Alternative zu klassischen Codegenerierungslösungen dürfte jedoch deutlich geworden sein. Für weitere Anregungen möchte ich auf den Blog eines Kollegen [8] und meinen eigenen Blog [9] verweisen. Über eine Suche auf GitHub lassen sich ebenfalls bereits viele Implementierungen finden. Und sollten Sie Fragen haben oder einmal nicht weiterkommen, wird Ihnen die freundliche Community [10] gerne weiterhelfen.

Sven Efftinge ist Projektleiter von Eclipse Xtend und Xtext bei eclipse.org. Er arbeitet als Development-Manager bei itemis und leitet den Standort in Kiel. In seiner Freizeit verbringt er gern viel Zeit mit seinen Kindern oder mit einem Kite auf der Ostsee.

Links & Literatur

[1] Design Patterns: Elements of Reusable Object-Oriented Software

[2] http://www.paulgraham.com/icad.html

[3] http://docs.neo4j.org/chunked/stable/tutorials-java-embedded-entities.html

[4] http://www.eclipse.org/xtend/documentation.html#extensionMethods

[5] http://blog.efftinge.de/2012/12/java-8-vs-xtend.html

[6] http://www.eclipse.org/xtend/documentation.html#templates

[7] https://github.com/svenefftinge/xtended_neo4j

[8] http://mnmlst-dvlpr.blogspot.de

[9] Bhttp://blog.efftinge.de

[10] https://groups.google.com/forum/#!forum/xtend-lang

Listing 5

```
class NeoNodeProcessor extends AbstractClassProcessor {

  override doTransform(MutableClassDeclaration annotatedClass,
                       extension TransformationContext context) {
    val nodeType = Node.newTypeReference
    val declaredFields = annotatedClass.declaredFields

    annotatedClass.addField("underlyingNode") [
      type = nodeType
    ]

    annotatedClass.addConstructor[
      addParameter("node", nodeType)
      body =  ['''
        this.underlyingNode = node;
      ''']
    ]

    annotatedClass.addMethod("getUnderlyingNode") [
      returnType = nodeType
      body = ['''
        return this.underlyingNode;
      ''']
    ]

    for (field : declaredFields) {
      annotatedClass.addMethod("get"+field.simpleName.toFirstUpper) [
        returnType = field.type
        body = ['''
          return («field.type.simpleName»)underlyingNode
                .getProperty("«field.simpleName»");
        ''']
      ]

      annotatedClass.addMethod("set"+field.simpleName.toFirstUpper) [
        addParameter(field.simpleName, field.type)
        body = ['''
          underlyingNode.setProperty(
              "«field.simpleName»", «field.simpleName»);
        ''']
      ]
    }

    annotatedClass.addMethod("hashCode") [
      returnType = primitiveInt
      body = ['''
        return this.underlyingNode.hashCode();
      ''']
    ]

    annotatedClass.addMethod("equals") [
      returnType = primitiveBoolean
      addParameter("o", object)
      body = ['''
        return o instanceof «annotatedClass.simpleName» &&
            underlyingNode.equals(
            ((«annotatedClass.simpleName»)o).getUnderlyingNode());
      ''']
    ]

    annotatedClass.addMethod("toString") [
      returnType = string
      body = ['''
        return "«annotatedClass.simpleName»[«
          declaredFields.map[
            simpleName+':"+get'+simpleName.toFirstUpper+"()+"
          ].join('','')»"]";
      ''']
    ]
    declaredFields.forEach[remove]
  }
}
```

Generisches Ressourcenmodell

NEUE SERIE

OSGi at your Service

Viele kennen OSGi als eine dünne Schicht über Java, um modulare Software zu entwickeln. OSGi übernimmt dabei die Prüfung, ob Abhängigkeiten einzelner Module in sich konsistent sind und das System somit wie gewünscht funktioniert. Ein Blick hinter die Kulissen, wie OSGi Abhängigkeiten beschreibt, lohnt sich allemal.

von Florian Pirchner

Dieser Artikel zeigt, wie OSGi auf Basis des generischen *Requirement-and-Capability*-Modells Abhängigkeiten zwischen Bundles beschreibt. Konzepte wie OBR (OSGi Bundle Repository), Resolving und Provisioning werden in der OSGi-Welt immer wichtiger. Finales Ziel ist, dass OSGi auf Basis dieses Modells alle Abhängigkeiten erkennt und mittels definierter Bundle Repositories automatisch installiert. Auf Resolving und Provisioning wird in diesem Artikel nicht eingegangen. Details hierzu folgen in den nächsten Artikeln dieser Serie.

OSGi in Kürze

OSGi basiert auf der Grundidee, Java-Applikationen in Module zu unterteilen. Jedes dieser Module (Bundle genannt) kapselt Funktionen. Ein Bundle wird durch seinen *Symbolic Name* und die Version eindeutig bestimmt. Im Gegensatz zu herkömmlichen JAR-Files definiert ein Bundle Schnittstellen.

Aufgrund von Einträgen im *MANIFEST.MF* definiert ein Bundle, welche Java-Packages nach außen sichtbar sein sollen. Ebenfalls muss ein Bundle definieren, welche sichtbaren Java-Packages es von anderen Bundles konsumieren möchte. Auf den ersten Blick scheint dies etwas umständlich. Es ergibt bei genauerer Betrachtung allerdings sehr viel Sinn. Durch diesen Mechanismus erkennt OSGi, welche Java-Packages zur einwandfreien Ausführung eines Bundles benötigt werden. Benötigt Bundle A bspw. das Package *org.foo*, das von OSGi jedoch nicht gefunden werden kann, wird Bundle A nicht korrekt ausgeführt, und OSGi reagiert mit einem Fehler darauf.

Die Definition von Abhängigkeiten erfüllt noch einen weiteren Zweck: Laut Java-Spezifikation darf eine Klasse nur einmal von einem Classloader geladen werden. Im klassischen SE-Ansatz, in dem die Context-Klassen immer vom gleichen Classloader geladen werden, bedeutet das, dass eine Klasse, die einmal geladen wurde, nicht wieder ausgetauscht werden kann. Die OSGi-Spezifikation umgeht dieses Problem, indem jedes Bundle seinen eigenen Classloader verwendet und Klassen eines Bundles nur mit dem jeweiligen Bundle Classloader geladen werden dürfen. Somit ist es möglich, Klassen im laufenden Betrieb auszutauschen. Ein Bundle besitzt einen Lifecycle und kann installiert, gestartet, gestoppt und deinstalliert werden. Beim Deinstallieren von Bundles wird auch der mit dem Bundle verbundene Classloader entfernt – was dazu führt, dass auch die mit dem Classloader geladenen Klassen aus der JVM entfernt werden.

Zusätzlich erlaubt diese Vorgehensweise, dass Klassen mit gleichem Namen, aber unterschiedlicher Version mehrfach im System geladen werden können. Benötigt Bundle A Klasse *Foo* in der Version 1 und Bundle B die gleiche Klasse in der Version 2, so ist es in OSGi möglich, dies umzusetzen. Im Gegensatz zu SE verwendet OSGi keinen globalen Klassenpfad, sondern delegiert das Laden von Klassen an Bundles. Stellt Bundle B bspw. die Klasse *Bar* zur Verfügung und exportiert das Java-Package, so importiert Bundle A dieses Java-Package und versucht die Klasse *Bar* zu laden. Der Classloader von Bundle A prüft, ob die Klasse im eigenen Bundle verfügbar ist. Falls nicht, delegiert der Classloader das Laden der Klasse an Bundle B. Der globale Klassenpfad mit all seinen Problemen gehört dank OSGi der Vergangenheit an.

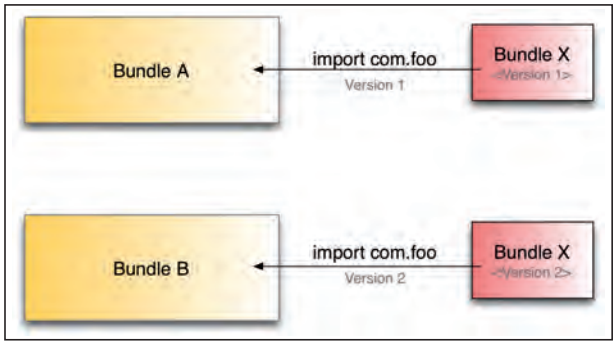


Abb. 1: Imports Packages

Artikelserie

Teil 1: Generisches Ressourcenmodell. Beschreibung von Abhängigkeiten auf Basis des Requirement-and-Capability-Modells

Teil 2: Requirement-Capability-Modell unter Verwendung der Resolver- und Provisioning-Spezifikation

Abbildung 1 veranschaulicht diesen Fall nochmals. Die Klasse *com.foo.Bar* ist in zwei Versionen geladen. Wobei Bundle A die Klasse in der Version 1 konsumiert und Bundle B die Klasse in der Version 2. Die Manifesteinträge für diesen Fall sehen folgendermaßen aus:

```
Bundle X_1 Version 1:
  Export-Package: com.foo;version="1.0.0"

Bundle X_2 Version 2:
  Export-Package: com.foo;version="2.0.0"

Bundle A:
  Import-Package: com.foo;version="[1.0.0,1.0.1)"

Bundle B:
  Import-Package: com.foo;version="[2.0.0,2.0.1)"
```

Sie können diesen Fall leicht selbst nachstellen, indem Sie in Eclipse 4 Bundles erstellen. Bitte nennen Sie die Provider-Bundles *X_1* und *X_2* und vergeben die notwendigen Bundle-Versionen.

Mithilfe der OSGi-Launchconfiguration können Sie dann unterschiedliche Bundle-Kombinationen auswählen, und mit Validate-Bundles zeigt Ihnen PDE die Probleme, die beim Auflösen aufgetreten sind (**Abb. 2**).

Internes Handling

Welche Modelle verwendet OSGi, um feststellen zu können, ob Bundles in sich konsistent sind und aufgelöst werden können? Die OSGi-Spezifikationen „Resource API Specification“ [1], „Bundle Wiring API Specification“ [2] und „Framework Namespaces Specification“ [3] beschreiben die damit verbundenen Konzepte. Ebenfalls bieten die Bücher „OSGi in Depth“ und „Enterprise OSGi in Action“ [4] von Manning eine gute Übersicht.

Generisches Requirement-Capability-Modell: Dieses Modell trennt sich von der Vorstellung, dass Manifestheader wie *Import-Package*, *Export-Package*, *Require-Bundle* usw. existieren. Stattdessen definiert es ein Modell bestehend aus Ressourcen, Requirements und Capabilities. Zur Definition von Requirements und Capabilities wurden neue Manifestheader definiert. Bestehende Manifestheader, wie die oben genannten, werden vom Framework automatisch in das Modell überführt.

Resource: Die Basis für das Auflösen von Abhängigkeiten bildet die „Resource API Specification“. Diese definiert ein generisches Requirement-Capability-Modell, um mithilfe dessen alle Abhängigkeiten eines Systems beschreiben zu können. Wie später angeführt, ist dieses Modell nicht auf Import- und Exportdeklarativen beschränkt, sondern bietet weitere Möglichkeiten, um Abhängigkeiten definieren zu können. Einträge in Manifestdateien werden vom Framework in das generische Modell überführt, was ein zentrales Handling ermöglicht.

Das Basiselement dieser Spezifikation ist die Ressource. Eine Ressource wird als ein Element definiert, das in die OSGi-Umgebung installiert werden kann und dort

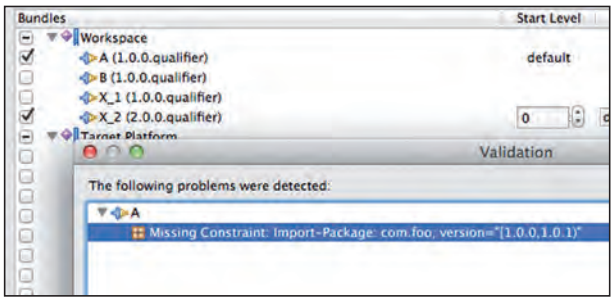


Abb. 2: Fehler beim Auflösen von Bundle A und X_2

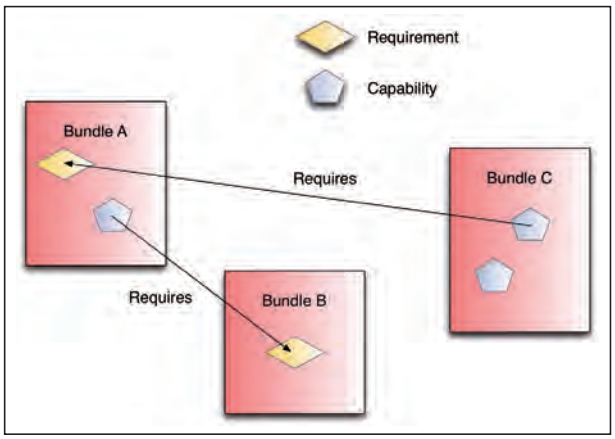


Abb. 3: Bundle-Abhängigkeiten dargestellt über generisches Modell

ihre Fähigkeiten (Capabilities) bereitstellt, sofern alle Requirements der Ressource bedient werden konnten.

Capability: beschreibt eine Fähigkeit einer Ressource, die dem System zur Verfügung gestellt wird. Ein Export-Package ist bspw. eine Capability eines Bundles.

Requirement: eine Notwendigkeit, die durch eine Ressource gefordert wird, z. B. ein Import-Package.

Namespace: definiert vorgegebene Typen von Capabilities und Requirements. Ein Requirement kann nur dann von einer Capability befriedigt werden, wenn beide im gleichen Namespace definiert sind. Mittels der „Framework Namespaces Specification" definiert OSGi Typen von Requirements und Capabilities, die von der OSGi-Core-Spezifikation verwendet werden. Ein Namespace definiert, welche Attribute und Direktiven sowohl für Requirements als auch für Capabilities erlaubt sind und welche Bedeutung Capabilities haben. Ein Requirement befriedigt nur dann eine Capability, wenn beide im gleichen Namespace liegen und eventuell ein Filter am Requirement zur Capability passt.

Abbildung 3 zeigt den Sachverhalt. Bundle A hat ein Requirement und befriedigt dieses durch die Capability von Bundle C. Bundle B befriedigt das Requirement wiederum mit der Capability von Bundle A.

Auf in die Praxis

Der oben beschriebene Ansatz bildet die Basis für OSGi, um Bedarfe von Bundles ermitteln und deren Integrität sicherstellen zu können. Nun werden die beschriebenen, abstrakten Konzepte auf das Beispiel (**Abb. 1**) angewandt.

Es werden unter anderem die klassischen Eigenschaften wie die Identität eines Bundles und die Abhängigkeit auf das generische Requirements-Capability-Modell umgelegt. Es ist darauf zu achten, dass einige der Namespaces nicht in der Manifestdatei angeführt werden dürfen, da diese automatisch auf Basis der klassischen Manifestheader in das generische Modell überführt werden. Ein Hinweis darauf ist in den einzelnen Fällen angegeben.

Des Weiteren wird auf Einfachheit gesetzt. Verkomplizierende Konzepte wie die Unterscheidung von Attributen und Direktiven werden nicht vorgenommen. Diese Themen werden im nächsten Artikel einbezogen.

Ressourcenidentität

Die Identität einer Ressource definiert sich auf Basis des Symbolic Name, der Version und des Typs einer Ressource. Die „Framework Namespaces Specification" definiert einen Namespace namens *osgi.identity*. Er definiert Attribute und Direktiven, um die Identität eines Bundles (bzw. einer Ressource) generisch definieren zu können. Unter [5] kann die Definition im Detail nachgeschlagen werden. Die Hauptelemente des *osgi.identity*-Namespace sind:

- *osgi.identity:* für Bundles entspricht dies dem Symbolic Name
- *type*: der Typ der Ressource; *osgi.bundle* oder *osgi.fragment*
- *version:* die Version der Ressource

All diese Attribute sind an einer Capability anzuwenden und definieren, was die Ressource zur Verfügung stellt. Es existieren noch weitere Attribute und Deklarativen, die jedoch nicht behandelt werden. Die Beschreibung der Identität von Bundle X_2 sieht auf Basis des generischen Modells folgendermaßen aus:

```
Provide-Capability:
  osgi.identity;
  osgi.identity:=X_2;
   version=2.0.0;
  type=osgi.bundle
```

Auf Basis dieser Beschreibung ist Bundle X_2 im generischen Modell eindeutig definiert, mit dem Namen, der Version und dem Typ. *osgi.identity* darf nicht in der Manifestdatei verwendet werden. Das Framework stellt sicher, dass die Manifestheader mit *bundle-symbolic-name* usw. automatisch in das generische Modell überführt werden.

Import- und Export-Package

Um Import- und Exportheader generisch beschreiben zu können, wurde der Namespace *osgi.wiring.package* definiert. Dieser setzt auf dem Konzept des Bundle Wiring auf, das in der „Bundle Wiring API Specification" beschrieben wird, wobei unter anderem jedes Requirement mit einer Capability mittels eines „Bundle Wire" verbunden wird.

Importheader stellen Requirements und Exportheader stellen Capabilities dar. Auch hier gilt, dass das Framework sicherstellt, dass die Manifestheader in das generische Modell überführt werden und der *osgi.wiring.package*-Namespace in der Manifestdatei für Imports

und Exports nicht verwendet werden darf. Die Definition wird folgendermaßen in das generische Modell überführt:

```
Provide-Capability:
  osgi.wiring.package;
  osgi.wiring.package=com.foo;
  version = 2.0.0;
  bundle-symbolic-name = X_2;
  bundle-version = 2.0.0
```

Etwas verwirrend ist der *bundle-symbolic-name*, der das Bundle definiert, welches das Package exportiert. Es scheint sich um eine redundante Information zu handeln. Da die Überführung des Export-Package-Headers in das generische Modell erst durch das Framework zur Laufzeit erfolgt, stellt dies kein Problem dar. Die Überführung der Definition des Import-Packages von Bundle B in das generische Modell erfolgt folgendermaßen:

```
Require-Capability:
  osgi.wiring.package;
    filter:="(&(osgi.wiring.package=com.foo)(version=2.0.0))"
```

Ein weiterer Namespace ist *osgi.wiring.bundle* und steht stellvertretend für den Require-Bundle-Header. Dieser ist dem *osgi.wiring.package* sehr ähnlich und wird nicht weiter ausgeführt. Zusätzlich ist auch noch der *osgi.wiring.host*-Namespace definiert, um Fragmente und dessen Host-Bundle beschreiben zu können.

Execution Environment

Bisher wurden Requirements und Capabilities beschrieben, die automatisch durch das Framework aus vordefinierten Manifestheadern in das generische Modell überführt werden. „Execution Environments“ ist der erste Namespace, der auch in der Manifestdatei angegeben werden darf. Ziel dieses Requirements ist die Beschreibung der notwendigen Ablaufumgebung, bspw. die Version der JVM oder die benötigte OSGi-Version. Ein Bundle könnte folgende Capability zur Verfügung stellen, um aufzuzeigen, welche Java-Versionen kompatibel sind:

```
Provide-Capability:
  osgi.ee;
    osgi.ee="JavaSE";
    version:List<Version>="1.6,1.7"
```

Ein anderes Bundle wiederum kann ein Requirement definieren, um sicherzustellen, dass nur Bundles dieses befriedigen können, die auch Java 1.5 unterstützen:

```
Require-Capability:
  osgi.ee;
    filter:="(&(osgi.ee=JavaSE)(version=1.5))"
```

Das Bundle, das nur Java 1.6 und 1.7 unterstützt und das mittels der Capability zum Ausdruck bringt, kann das Requirement von Java 1.5 nicht befriedigen.

OSGi Extender

Wer kennt das Problem nicht: Eine Applikation wird zwar einwandfrei von OSGi aufgelöst, aber dennoch nicht wie gewünscht gestartet. Die Erfahrung zeigt, dass es genau dann sinnvoll ist, OSGi-DS, OSGi-CM oder OSGi-Blueprint zu aktivieren. Oft fehlt eine OSGi-Erweiterung, die Services auflöst, Blueprint-Beans erzeugt oder Konfigurationen an Services sendet. Um diesem Problem entgegenzuwirken, wurde der *osgi.extender*-Namespace definiert. Dieser ist in der „Common Namespaces Specification“ im OSGi-Kompendium definiert. Ein Bundle, das Blueprint zur Verfügung stellt, kann folgende Capability definieren:

```
Provide-Capability:
  osgi.extender;
    osgi.extender="osgi.blueprint";
    version:Version="1.0"
```

Ein anderes Bundle, das eine Blueprint-Implementierung zum Ablauf benötigt, kann das dafür definierte Requirement definieren:

```
Require-Capability:
  osgi.extender;
    filter:="(&(osgi.extender=osgi.blueprint)(version>=1.0))"
```

Durch Verwendung dieses Namespace kann OSGi auf fehlende Abhängigkeiten hinweisen.

Fazit

Mithilfe des generischen Requirement-Capability-Modells ist es OSGi möglich, sämtliche Abhängigkeiten eines Systems zu erkennen und zu verifizieren. Neue Typen von Abhängigkeiten können einfach hinzugefügt werden und stehen dem Resolver anschließend zur Verfügung.

Besonders interessant wird dieses Modell unter Verwendung der Resolver- und Provisioning-Spezifikation, die Queries an Bundle Repositories richtet. Dies erlaubt eine neue Art und Weise des Provisionings. Dazu mehr in der nächsten Ausgabe des Java Magazins.

Florian Pirchner ist selbstständiger Softwarearchitekt, lebt und arbeitet in Wien. Aktuell beschäftigt er sich als Project Lead in einem internationalen Team mit dem Open-Source-Projekt „Iunifera.org – OSGi-services for business applications“. Auf Basis von Modellabstraktionen und der Verwendung von OSGi-Spezifikationen soll ein hoch erweiterbarer und einfach zu verwendender Kernel für Businessapplikationen geschaffen werden.

Links & Literatur

[1] http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf – chapter 6

[2] http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf – chapter 7

[3] http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf – chapter 8

[4] http://www.manning.com/alves/, http://www.manning.com/cummins/

[5] http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf – chapter 8

Anzeige

Agilität ist nur ein Anfang

Überleben im Wirtschaftsdarwinismus

Die IT befindet sich im Wandel. Manche spüren es schon, manche ignorieren es noch. Neue, hochinnovative Unternehmen mit einer extrem schnellen und flexiblen IT drängen in den Markt und sind den alteingesessenen Platzhirschen immer mehrere Schritte voraus. Agilität wird dann gerne als Wunderwaffe ins Feld geführt, um zumindest die Unternehmens-IT schneller und flexibler zu machen. Aber reicht das? Und warum produzieren viele Unternehmen mit Agilität nur noch mehr Mist in noch kürzerer Zeit? Es wird Zeit, sich den Wandel einmal genauer anzusehen und zu überlegen, wie wir uns in der IT aufstellen müssen, um zu den Gewinnern im Wirtschaftsdarwinismus zu gehören.

W-JAX Speaker

von Uwe Friedrichsen

Bevor wir aber richtig einsteigen, erst eine Frage: Was hat ein Beitrag mit den Begriffen *Wirtschaftsdarwinismus* und *Agilität* in der Überschrift im Java Magazin zu suchen? Hier erwarten wir doch eher Artikel von Architektur bis hart am Code im Java-Ökosystem und weniger „weiche" Themen wie z. B. Agilität. Darauf gibt es zwei Antworten:

- Dieser Wandel hat auch viel mit konkreter Technologie zu tun. Es gibt viele Auslöser und viele Auswirkungen – von eher „weichen" Faktoren bis hin zu ganz konkreter Technologie. So „weich" ist das Thema letztlich gar nicht, auch wenn ich in diesem Artikel wahrscheinlich auf einer etwas größeren Flughöhe bin, als man es von der Mehrzahl der Artikel dieses Magazins gewohnt ist. Eberhard Wolff wird im Anschluss ab Seite 48 den technischen Aspekt näher beleuchten.
- Indem man den hier beschriebenen Wandel versteht, lernt man auch besser einzuschätzen, welche konkreten Technologien und Techniken man auf seine To-do-Liste setzen sollte und welche vielleicht nicht ganz so wichtig sind.

Sollte Ihnen das für Ihre heutige Tagesform dennoch zu „weich" sein, dann ist das natürlich auch okay. In dem Fall wünsche ich Ihnen viel Spaß bei der Lektüre der anderen Artikel.

Der Wirtschaftsdarwinismus

Für die, die jetzt noch dabei sind: Es gibt viele Wege, sich dem Wandel zu nähern – ich versuche es über eine kurze Begriffsklärung:

Der *Darwinismus* ist eine Quelle vielfältiger Missverständnisse. Gerade das im Englischen lautende Selektionskriterium „*Survival of the Fittest*" wird im Deutschen häufig fälschlicherweise mit „Überleben der Stärksten" übersetzt. Doch „Fittest" hat in dem Zusammenhang nichts mit Stärke zu tun, sondern leitet sich von dem englischen Wort „to fit", zu Deutsch „passen" ab: Mit „Fittest" sind die Spezies gemeint, die sich am besten den äußeren Bedingungen anpassen können. Eine bessere Übersetzung wäre also „*Überleben derjenigen, die sich am besten auf Veränderungen einstellen können*".

Genau dies kann man heute mehr und mehr in der Wirtschaft beobachten: Wer sich nicht zügig auf veränderte Marktanforderungen einstellen kann, ist schnell weg vom Fenster. Manche von üppigen Margen verwöhnte Branchen schlafen noch ihren Dornröschenschlaf, aber alle anderen spüren den *Wirtschaftsdarwinismus* in vollem Umfang.

In der IT kommt dies in Form der Forderung an, in immer kürzerer Zeit immer mehr umsetzen und liefern zu können – natürlich ohne dass die Qualität dadurch schlechter würde. Anders formuliert zählt die Zeit von der ersten Formulierung einer Anforderung bis zur Unterstützung im Betrieb. Lange Analyse-, Konzeptions-, Test- oder Inbetriebnahmephasen sind keine Option mehr. Was tun?

Als Lösung wird immer wieder Agilität propagiert: Fix mal Scrum oder Kanban eingeführt, noch eine Prise XP dazu und alles wird gut – wird es aber häufig nicht. Nach der ersten Euphorie stellen viele fest, dass es überall hakt und klemmt. Anstatt besser zu werden, bauen sie nur schneller unwartbare Systeme und nach kurzer Zeit ist es schlimmer als je zuvor.

Agilität ist also nicht die Wunderwaffe zum Überleben im Wirtschaftsdarwinismus – zumindest nicht Agilität allein. Aber was benötigt man denn, um zu den Gewinnern zu gehören?

Beobachtungen und Folgerungen

Beginnen wir also noch einmal von vorne mit zwei Beobachtungen und der Konsequenz, die sich daraus ergibt:

1. Beobachtung: Der Wirtschaftsdarwinismus betrifft alle Branchen. Wir sind auf allen Ebenen einer wachsenden Globalisierung ausgesetzt: Der Bäcker beliefert nicht mehr nur sein Viertel, sondern häufig ganze Landkreise, den Joghurt aus Südbayern gibt es an der Nordseeküste zu kaufen und Autohersteller haben schon lange nicht mehr das eigene Land, sondern die ganze Welt als Zielmarkt. Verstärkt wird das Ganze noch durch die vielfältigen Einflüsse des Internets: Wenn ich einen Artikel mal nicht bei mir vor Ort bekomme, er mir hier zu teuer erscheint oder der Service nicht stimmt, dann schaue ich eben im Netz nach und werde fast immer fündig.

Entsprechend wächst der Konkurrenzdruck, weil immer mehr Wettbewerber pro Marktteilnehmer verfügbar sind. Der Käufer wird immer mehr umworben, seine Ansprüche steigen und Unternehmen müssen sich immer ein bisschen schneller als die Mitbewerber auf die Wünsche und Vorlieben ihrer Kunden einstellen, um sie nicht zu verlieren.

Und auch wenn man hin und wieder auf Unternehmen stößt, die von dieser Entwicklung noch unberührt erscheinen, sind die meisten Unternehmen bereits heute dem Wirtschaftsdarwinismus voll ausgesetzt – Tendenz steigend.

2. Beobachtung: IT ist heute für nahezu alle Unternehmen überlebensnotwendig.

Praktisch kein Unternehmen jenseits der Kleinstbetriebsgrenze kann heute noch ohne IT überleben. Fiele die IT komplett aus, stünde entweder die Produktion des Unternehmens still oder wäre zumindest so langsam und arbeitsintensiv, dass eine wettbewerbsfähige Produktion nicht möglich wäre. Und je größer ein Unternehmen ist, desto höher ist die Abhängigkeit von einer funktionierenden IT: Eine ganze Reihe an Unternehmen – vielfach auch in nicht IT-nahen Branchen – stufen einen IT-Ausfall von nur wenigen Stunden als existenzbedrohend ein.

Die IT ist heute in den meisten Unternehmen nicht mehr nur „Unterstützer", sondern das „zentrale Nervensystem" des Geschäfts. Ohne IT geht gar nichts mehr. Das gilt nicht nur für den laufenden Betrieb, sondern auch für jedwede Änderungen und Neuerungen auf

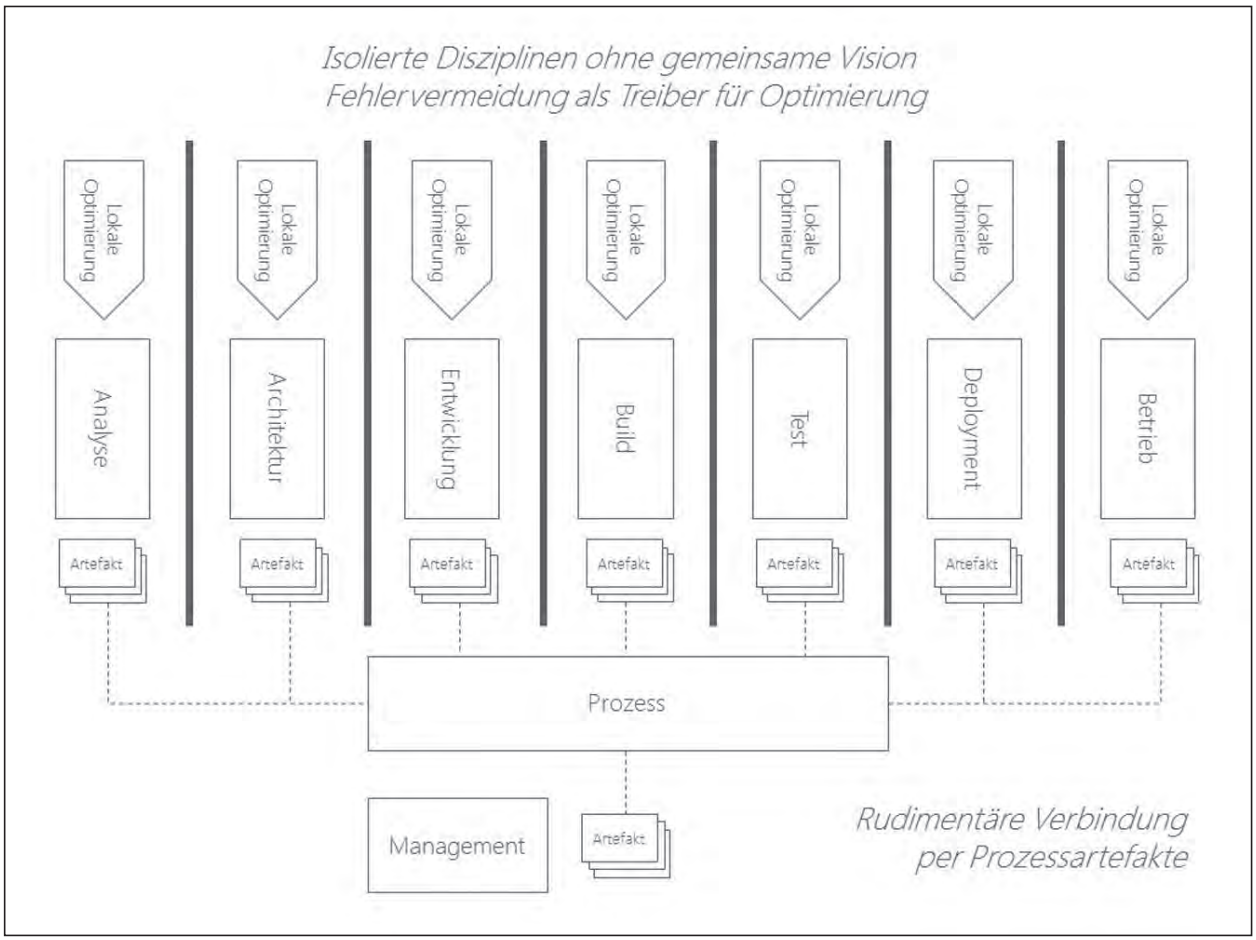


Abb. 1: Defizite im heutigen Software-Engineering

Geschäftsseite. Neue Produkte oder verbesserte Dienstleistungen sind ohne entsprechende IT-Unterstützung praktisch nicht mehr möglich.

Konsequenz: Die IT ist ein zentraler Schlüsselfaktor, um im Wirtschaftsdarwinismus zu den „Überlebenden" gehören zu können. Sie können als Unternehmen noch so innovativ sein, Sie können die Bedürfnisse Ihrer Kunden noch so gut verstehen oder gar antizipieren: wenn Ihre IT nicht mitspielt, hilft Ihnen das gar nichts.

Nur wenn Ihre IT in der Lage ist, Ihre Ideen zeitnah in Ihre Systeme zu integrieren, dann können Sie Ihre Ideen auch zeitnah an den Markt und Ihre Kunden bringen. Kann Ihre IT das nicht, werden Sie immer häufiger frustriert und hilflos beobachten müssen, wie andere Wettbewerber Ihre Idee früher an den Markt bringen und Ihnen Kunden abziehen. Um im Wirtschafsdarwinismus nicht das Nachsehen zu haben, ist es daher essenziell, eine IT zu haben, die neue Anforderungen mit minimaler Durchlaufzeit umsetzt und bereitstellt – natürlich ohne dabei die Zuverlässigkeit der Systeme zu kompromittieren.

Stand heute

Wir befassen uns seit vielen Jahren mit der Verbesserung und Optimierung der IT. Wir haben dabei sehr viel nützliches (und manchmal auch nutzloses) Wissen angesammelt. Es gibt jede Menge Prozessframeworks zur Softwareentwicklung, von V-Modell und RUP bis hin zu den vielen unternehmenseigenen Modellen, die sich fast jedes größere Unternehmen leistet. Und es gibt für jede der zentralen IT-Disziplinen – Analyse, Architektur, Entwicklung, Test, Build, Deployment, Betrieb, Management – jede Menge Best Practices und hilfreiches Wissen.

Eigentlich müsste es doch ein Leichtes sein, die im vorherigen Abschnitt formulierte Anforderung an die IT zu erfüllen. Trotzdem gibt es weiterhin viele Unternehmen, bei denen die durchschnittliche Durchlaufzeit von der ersten Formulierung eines neuen Features bis hin zum Betrieb des Features in den IT-Systemen sechs Monate oder länger dauert. Durchlaufzeiten von wenigen Tagen oder zumindest wenigen Wochen? Fast überall Fehlanzeige – zumindest wenn man sich an die vorgegebenen IT-Prozesse hält.

Immer wieder werden diese Prozesse dann umgangen, wenn die Zeit richtig drängt. Dann werden alle bestehenden Regeln per „Emergency Task Force", „Notfallfreigabe" oder anderer kreativer Begrifflichkeiten außer Kraft gesetzt und das Feature wird schnellstmöglich in Produktion gebracht – egal wie.

Aber das kann doch nicht die Lösung sein. Warum ist die IT so langsam? Es gibt sicherlich viele Defizite, die man betrachten könnte, aber ich möchte mich hier auf zwei essenzielle Defizite beschränken (**Abb. 1**).

Lokale Optimierung der Disziplinen: Wir optimieren die verschiedenen IT-Disziplinen nur lokal: Die Anforderungsexperten schauen, wie man die Anforderungserhebung noch besser machen kann, die Architekten, wie man die Architekturarbeit verbessern kann usw. Aber kaum jemand schaut darauf, wie man die IT-Produktionskette durch sinnvolles Verbinden und Vernetzen der Einzeldisziplinen mit Hinblick auf ein Gesamtziel optimiert.

Auch die bekannten Prozessframeworks wie RUP bringen da nichts: Die Prozesse verbinden die verschiedenen Disziplinen nur notdürftig miteinander. Die Disziplinen stehen weiterhin weitestgehend isoliert nebeneinander und als Schnittstellen werden „Artefakte", d. h. irgendwelche Dokumente über den Zaun geworfen.

Der ergänzende Blick, ob es neben den Einzeldisziplinen vielleicht noch andere, übergreifende Aspekte gibt, die für eine deutliche Verbesserung sorgen könnten, bleibt in der Regel aus. Ein Grund dafür könnte das zweite Defizit sein.

Fehlervermeidung als primäres Ziel: Sowohl die lokalen Optimierungen als auch die verbindenden Prozessframeworks haben Fehlervermeidung als primäres Ziel: Nicht das Produkt oder eine möglichst schnelle Bereitstellung stehen im Fokus, sondern die Vermeidung von Fehlern um jeden Preis.

Agile Werte und Prinzipien sind ohne einen soliden IT-Unterbau nur schöne Worte.

Exemplarisch kann man das z. B. bei der Anforderungserhebung sehen: Fast immer geht es bei Artikeln oder Vorträgen zu dem Thema darum, wie man Anforderungen noch vollständiger, noch redundanzfreier, noch unmissverständlicher erfassen kann. Die Verkürzung der Durchlaufzeiten hingegen ist (fast) nie ein Thema.

Fehlervermeidung ist sicherlich ein wichtiges Ziel und die IT hat an der Stelle definitiv immer noch einige Hausaufgaben zu machen, aber mit einem solchen Primärziel ist es nicht verwunderlich, dass man kaum ernstzunehmende Ansätze sieht, den Anforderungen aus dem Wirtschaftsdarwinismus effektiv zu begegnen.

Radikale Mehrwertorientierung

Wie aus dem zuvor Geschriebenen ersichtlich wird, muss eine neue Ausrichtung für eine Next Generation IT her, um den Anforderungen aus dem Wirtschaftsdarwinismus in der IT zu begegnen. Ich möchte die-

Abb. 2: Radikale Mehrwertorientierung als neues Zielbild einer Next Generation IT

ses neue Primärziel einmal marktschreierisch *radikale Mehrwertorientierung* nennen. Dieses Ziel setzt sich aus den folgenden gleichberechtigten Teilzielen zusammen, die ganzheitlich betrachtet werden müssen (**Abb. 2**):

- *Kurze Durchlaufzeiten*: Neue Features kommen nicht mehr in Monaten oder Jahren, sondern in Tagen oder Wochen. Oft muss ein neues Feature auch nicht sofort vollumfänglich ausgerollt werden. Häufig reichen auch schon Teilumsetzungen für ein erfolgreiches Ausrollen im Markt.
- *Kontinuierlicher Durchsatz:* Die IT setzt Anforderungen mit einer gleichbleibenden, hohen Produktivität um. Es reicht nicht, genau ein Feature schnell umsetzen zu können, während alles andere stillsteht. Es gibt immer eine Menge wichtiger Anforderungen verschiedenster Art, die seitens der IT umgesetzt werden müssen und es ist wichtig, dass die IT ein verlässlicher Partner der Geschäftsbereiche ist. Dazu gehört, eine hohe Planbarkeit bzgl. Durchlaufzeiten und Lieferterminen für neue Features anbieten zu können – und natürlich müssen sich die Termine auch mit den Erwartungshaltungen der vom Wirtschaftsdarwinismus getriebenen Geschäftsbereiche in Einklang bringen lassen.
- *Hohe Flexibilität:* Sich ändernde Anforderungen und Prioritäten auf Geschäftsseite sind der Normalfall, nicht die Ausnahme. Wir als IT müssen in der Lage sein, diese neuen Anforderungen und Umpriorisierungen schnell zu berücksichtigen und umzusetzen. Es kann nicht sein, dass wir die unter Druck stehenden Fachbereichsmitarbeiter auf irgendeinen Termin in ferner Zukunft vertrösten.
- *Hohe Zuverlässigkeit:* Eine schnelle Umsetzung allein genügt nicht. Es muss auch das Richtige umgesetzt werden, es muss richtig umgesetzt werden und stabil in Produktion laufen. Natürlich ist hohe Qualität weiterhin ein wichtiges Ziel und häufig ist die Messlatte höher als das, was in den traditionellen Vorgehensweisen bislang geliefert wird. Aber es geht nicht mehr um Fehlervermeidung als alleiniges höchstes Ziel, sondern es geht darum, eine hohe Zuverlässigkeit *zusammen* mit den anderen Teilzielen herzustellen.

Agilität als Lösung?

An dieser Stelle wird dann häufig auf Agilität verwiesen. Denn:

- Kurze Durchlaufzeiten? Klingt nach Kanban!
- Kontinuierlicher Durchsatz? Ganz klar Scrum!
- Hohe Zuverlässigkeit? Lean: „Build quality in"! Da haben wir es doch!
- Und hohe Flexibilität ist doch eh das Motto aller agilen Ansätze!

Die Lösung ist also schon da. Wozu dann dieser Artikel? Nun, ganz so einfach ist es leider nicht. Eine agile Vorgehensweise ist sicherlich ein wichtiger Baustein auf dem Weg zu einer Next Generation IT, die den Anforderungen des Wirtschaftsdarwinismus gerecht wird. Allein schon zur Beherrschung der Komplexität (im Sinne der Systemtheorie), die den Geschäftsdomänen innewohnt, ist ein agiles Vorgehen unerlässlich, da klassische, plangetriebene „Teile-und-Herrsche"-Strategien in solchen Umfeldern nicht funktionieren.

Aber ein agiles Vorgehen alleine reicht nicht aus und auch die agilen Werte und Prinzipien sind ohne einen soliden IT-technischen Unterbau und eine angemessene Qualifikation der beteiligten Individuen nur schöne Worte. Hart ausgedrückt: Im schlimmsten Fall hilft einem ein agiles Vorgehen nur, um noch mehr Mist in noch kürzerer Zeit zu produzieren.

Darwinistische Softwareentwicklung

Wenn Agilität alleine nicht ausreichend ist, was braucht man dann? Kurz gesagt muss man erst einmal die technischen, organisatorischen und personenbezogenen Voraussetzungen schaffen, damit ein agiles Vorgehen auf der methodischen Ebene seine Stärken ausspielen kann. Es müssen mehrere Konzepte miteinander verknüpft werden, um dem Ziel der radikalen Mehrwertorientierung gerecht werden zu können.

An dieser Stelle kommen eine Reihe aktueller Trendthemen ins Spiel. Letztlich sind viele aktuelle Trends und Konzepte nämlich nichts anderes als Teilantworten auf die Anforderungen des Wirtschaftsdarwinismus an die IT. In Summe möchte ich den hier vorgestellten Ansatz in Ermangelung einer besseren Bezeichnung *Darwinistische Softwareentwicklung* nennen. Diese lässt sich durch die folgenden Bausteine beschreiben (**Abb. 3**):

- *Agilität:* Das Ziel ist lauffähige Software in Verbindung mit sehr guten Reaktionsmöglichkeiten auf sich ändernde Anforderungen. Ergänzt wird dies durch Techniken zum Management von Komplexität auf Basis der Nutzung der Effekte von Kollaboration, Interaktion und indirekter Steuerung (vielen besser

bekannt unter dem häufig missverstandenen Begriff „Selbstorganisation“).

- *Lean:* Das Vermeiden von unnötigen Tätigkeiten und Qualität als Invariante, nicht als Teil eines Prozesses, dazu schnelle und regelmäßige Lieferungen sowie die ständige Suche nach Verbesserungen. Lean wird häufig mit Agilität gleichgesetzt, was aber nicht stimmt.
- *DevOps:* Das Einreißen der Mauern zwischen Entwicklung und Betrieb und den Parteien dazwischen mit dem gemeinsamen Ziel, neue Features möglichst zügig und qualitativ hochwertig in Produktion zu bringen.
- *Continuous Delivery:* Eine hochgradige Automatisierung der Build- und Lieferprozesse, um überhaupt in der Lage zu sein, schnell und häufig neue Features ausliefern zu können, ohne dass die Kosten explodieren und die Qualität aufgrund immer wiederkehrender (hochgradig langweiliger und dadurch fehlerträchtiger) Routinetätigkeiten leidet.
- *Feature Flow*: Der Continuous-Delivery-Fluss, der beim Bauen der Software einsetzt, muss durch einen

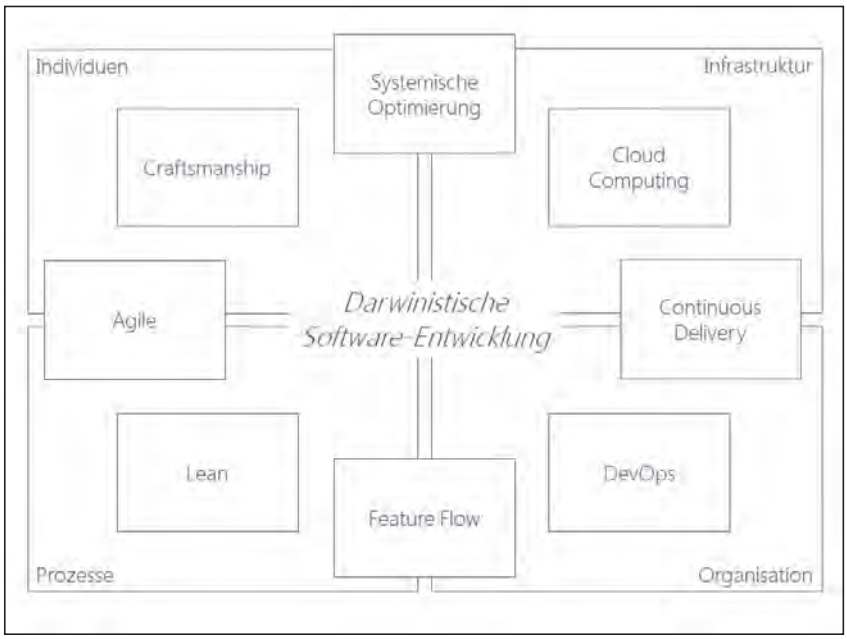


Abb. 3: Prinzipien der Darwinistischen Softwareentwicklung

Es warten viele Herausforderungen auf uns, aber nie war die IT-Welt spannender als heute.

zweiten Fluss am Anfang der Kette ergänzt werden: Es wird ein kontinuierlicher Fluss von Anforderungen, Konzeption und Umsetzung benötigt. Dabei werden Features vertikal in Teilfeatures zerlegt, individuell priorisiert, konzipiert und umgesetzt. Dies bedeutet trotz einiger vielversprechender, existierender Ideen noch eine Menge Konzeptarbeit, insbesondere für die Anforderungserhebung und Architektur, weil die Methoden beider Disziplinen nicht für einen sich kontinuierlich ändernden Zielzustand optimiert sind, sondern von einem zumindest für die Laufzeit eines Projekts eher stabilen Zielzustand ausgehen.

- *Cloud Computing:* Provisioning und De-Provisioning von Infrastruktur-Services on Demand als automatisierbarer Self-Service. Es geht nicht darum, Rechenkapazitäten in der Public Cloud einzukaufen, sondern um das hinter Cloud Computing stehende Provisioning-Modell: Ein zusätzlicher Rechner muss „auf Knopfdruck" da sein und genauso einfach wieder in den Pool zurückgegeben werden können. Vollständige Entwicklungsumgebungen für Features müssen in Minuten bereitgestellt werden können und nicht in Wochen oder Monaten. Ohne eine Infrastruktur, die ein solches Provisioning-Modell unterstützt, lassen sich die benötigte Durchlaufzeit und Flexibilität nicht realisieren.
- *Craftsmanship:* Der konsequente Mehrwertgedanke und die Professionalisierung des Individuums. Der beschriebene Ansatz stellt hohe Anforderungen an das Können der beteiligten Individuen. Ein „geht schon irgendwie" reicht nicht mehr – zumindest wenn man mehr will, als nur schnell Schrott liefern. Neben der Professionalisierung des Individuums geht es außerdem um die Professionalisierung der Zusammenarbeit. Das hat nichts mit trockener oder spaßfreier Kommunikation zu tun, sondern mit konsequentem Zusammenarbeiten an einem gemeinsamen Ziel und mit gegenseitigem Respekt und Wertschätzung der Tätigkeiten der anderen – auch außerhalb der eigenen Domäne.

Die ganzheitliche Verknüpfung und Umsetzung dieser Grundbausteine ergibt die Grundlage einer Next Generation IT:

- Features fließen kontinuierlich durch eine hochgradig optimierte und automatisierte Softwareentwicklungspipeline
- Projekte (und auch langwierige Planungs- und Budgetierungszyklen) verlieren ihre Bedeutung
- Die konsequente Automatisierung der Routinetätigkeiten schafft Freiraum für Exzellenz
- Durchlaufzeiten von wenigen Tagen bis herunter zu Stunden sind damit erzielbar – ohne das Kompromittieren der Qualitätsziele

Im Internetsektor gibt es heute schon Unternehmen, die mehrere Releases pro Tag freigeben – bei hochkomplexen Systemlandschaften und extremen Verfügbarkeitsanforderungen. Es ist also keine bloße Utopie, die hier beschrieben ist.

Zusammenfassung

Der Wirtschaftsdarwinismus betrifft uns alle – die meisten schon heute, ein paar wenige vielleicht erst morgen. Eine flexible, zuverlässige und schnelle IT ist ein zentraler Schlüsselfaktor, um sich in dem Umfeld als Gewinner positionieren zu können. Die heute üblichen Verfahren und Konzepte, die primär auf Fehlervermeidung fußen, sind dafür nicht geeignet.

Einfach nur ein agiles Vorgehen einzuführen ist auch keine Lösung. Stattdessen müssen verschiedene Konzepte wie Agilität, Lean, DevOps, Continuous Delivery, Cloud Computing und Craftsmanship ganzheitlich miteinander verknüpft werden, um eine Next Generation IT zu implementieren, die den Anforderungen des Wirtschaftsdarwinismus gerecht wird.

Natürlich ist es noch ein weiter Weg, die genannten Konzepte ganzheitlich in die Praxis umzusetzen, natürlich gibt es auch noch eine Reihe weiterer disruptiver Themen, die unsere Aufmerksamkeit erfordern wie z. B. Big Data, Ubiquitous Computing oder Social Media und natürlich bedarf es noch einer Menge Feinarbeit und Augenmaß bis zum Ziel (falls es so etwas überhaupt geben sollte), aber wir kennen jetzt die Richtung, in die wir uns bewegen müssen.

Um abschließend den Bogen zum Anfang des Artikels zu schlagen: Vielleicht hat Ihnen der Artikel ein wenig geholfen, Ihre Prioritäten in all den Trends und Hypethemen neu zu sortieren, die tagtäglich auf Sie einprasseln und Sie haben eine Idee, mit welchen Themen Sie sich als Nächstes konkret befassen wollen.

Und falls Sie ein Experte in einer der Software-Engineering-Disziplinen sein sollten: Vielleicht haben Sie ja schon Ideen, wie man Ihre Disziplin besser auf die neuen Anforderungen ausrichten kann und helfen mit, die IT von morgen mit Leben zu füllen.

Es warten jede Menge Fragen und Herausforderungen auf uns, aber nie war die IT-Welt spannender als heute. Ich persönlich freue mich auf die Next Generation IT. Sie auch?

Uwe Friedrichsen ist ein langjähriger Reisender in der IT-Welt. Als Fellow der codecentric AG darf er seine Neugierde auf neue Ansätze und Konzepte sowie seine Lust am Andersdenken ausleben. Seine aktuellen Schwerpunktthemen sind verteilte, hochskalierbare Systeme und Architektur in post-agilen Umfeldern. Er teilt und diskutiert seine Ideen häufig auf Konferenzen, als Autor von Artikeln, Blog Posts, Tweets und mehr.

Was der Wandel für die Java-Community bedeutet

Technologien für den Schritt nach Agilität

Durch den „Wirtschaftsdarwinismus“ ändern sich die Herausforderungen, denen sich die IT stellen muss. Aber welche konkreten Technologien sind relevant? Und wie beeinflussen die Herausforderungen den typischen Java-Entwickler?

von Eberhard Wolff

W-JAX Speaker

Das Java-Ökosystem ist vor allem im Enterprise-Bereich führend und das auch schon sehr lange. Viele Technologien können auf eine ganze Dekade Geschichte zurückblicken – aber das Umfeld ändert sich. Das wird sicher an Java nicht spurlos vorbei gehen. Oft ist aber kein radikaler Wandel, sondern eine Ergänzung des „üblichen“ Technologiestacks ausreichend. Dieser Artikel gibt einen Überblick über die Technologien und dient als Hinweis auf die verschiedenen Trends. Er geht also in die Breite – und nicht so sehr in die Tiefe bei den einzelnen Technologien.

Continuous Delivery

Agile Prozesse versprechen, dass Features in Software wesentlich schneller umgesetzt werden können. Allerdings ist das Ergebnis eines agilen Prozesses „nur“ auslieferbare Software – ob sie dann wirklich in Produktion laufen wird, steht auf einem anderen Blatt. Genau dort setzt Continuous Deployment [1] an: Das Ziel ist es, Software regelmäßig in Produktion zu bringen. Dazu dient eine Deployment-Pipeline, in der die Software verschiedene Stufen durchläuft, bis sie schließlich in Produktion landet (**Abb. 1**). Diese Pipeline soll möglichst schnell und zuverlässig durchlaufen werden – und darauf zielen die Technologien für die Pipeline auch ab.

Der erste Schritt in der Pipeline ist der „Commit Stage“. Er wird angestoßen, wenn ein Entwickler Code in die Versionskontrolle eincheckt. Konkret wird in diesem Schritt der Code kompiliert, Unit Tests durchgeführt und gegebenenfalls eine statische Codeanalyse durchgeführt. Werkzeuge für diesen Schritt sind: Continuous-Integration-Server wie Jenkins [2] oder Hudson [3] und Werkzeuge für statische Codeanalyse wie SonarQube [4]. Diese Technologien haben sich weitestgehend durchgesetzt. Also ändert sich in diesem Bereich für Java-Entwickler noch nicht so viel.

Der nächste Schritt sind Akzeptanztests. Im Gegensatz zu Unit Tests werden in diesem Schritt die fachli-

Abb. 1: Deployment-Pipeline

chen Anforderungen getestet. Klassisch werden dazu oft Testpläne aufgestellt und die einzelnen Anforderungen manuell abgetestet. Oft werden die Tests schon sehr detailliert beispielsweise in Excel-Checklisten beschrieben. Tests so genau zu beschreiben, ist aufwändig. Die Durchführung ist dann nur noch Routine. Aber solche Routineaufgaben sollten automatisiert werden. Dazu sind die Testpläne wegen der aufwändigen Erstellung oft schon detailliert genug – aber noch nicht so formal, dass sie automatisch ausgeführt werden können.

Grundsätzlich gibt es für diesen letzten Schritt unterschiedliche Werkzeuge: Die Tests können auf der Ebene der Oberfläche ansetzten. Dann kann das Team beispielsweise Selenium [5] einsetzen, um die Oberfläche automatisiert zu testen. Dann muss der Testplan nur einmal „aufgenommen“ werden, um dann automatisiert „abgespielt“ zu werden.

Alternativ können Werkzeuge für BDD (Behaviour-driven Design) genutzt werden. Dabei werden Anforderungen textuell beschrieben und automatisiert getestet. Sie entsprechen einem vorgegebenen Format, das durch natürliche Sprache auch für Endanwender verständlich ist und durch das Framework abgetestet werden kann (Listing 1). BDD-Tools mit Java-Unterstützung sind beispielsweise JBehave [6] oder Thucydides [7]. Sie interpretieren die textuelle Beschreibung des Tests und rufen dann die Logik des Systems auf, um zu testen, ob es sich wie erwartet verhält. Gerade im Bereich BDD gibt es aber auch zahlreiche alternative Werkzeuge.

Es schließen sich dann automatisierte Kapazitättests an: Sie messen die Performance der Anwendung und können über das GUI erfolgen. Eine Alternative ist es, ein eigenes API zu implementieren, mit dem Szenarien direkt ausgeführt werden. Für beide Fälle können Werkzeuge wie Grinder [8] oder JMeter [9] genutzt werden.

Schließlich wird die Anwendung manuell getestet. In diesem Schritt ist es aber nur noch notwendig, neue Features zu testen, für die keine automatisierten Tests vorliegen. So können sich die Tester darauf fokussieren, explorativ nach Fehlern zu suchen, weil die Routinetätigkeiten durch Automatismen erledigt sind. Das bedeutet, dass nicht alle Tests sofort automatisiert werden. Wenn Tester immer wieder dieselben Tests manuell ausführen, ist eine Automatisierung natürlich sehr sinnvoll.

Am Ende steht das Release. Das ist aber nicht das erste Mal, dass die Software installiert wird: Schon für die Akzeptanztests und die automatisierten Kapazitätstests sind produktionsnahe Umgebungen notwendig – sodass auch Deployment-Prozesse automatisiert und optimiert werden müssen. Dazu gibt es unterschiedliche Möglichkeiten: DSLs wie Chef [10] oder Puppet [11] erlauben es, den gewünschten Zustand eines Servers zu definieren. Beispielsweise kann definiert werden, welche Linux-Packages installiert sein müssen oder wie bestimmte Konfigurationsdateien aussehen sollen. Ergeben sich Unterschiede zwischen dem aktuellen Zustand der Server und der definierten Konfiguration, so können die Werkzeuge diese Unterschiede beheben. Auf diese Weise können Server nicht nur einmal installiert, sondern auch bei Konfigurationsänderungen aktualisiert werden. Die Konfiguration kann als DSL-Skript auch zusammen mit der eigentlichen Software versioniert werden. So kann sichergestellt werden, dass die Konfiguration auch immer zur Software passt.

Da verschiedene Releases parallel in verschiedenen Testphasen sein können, ist es nicht ausreichend, einfach nur einen Server auf das jeweils aktuelle Release zu bringen – es müssen ausreichend Server mit den entsprechenden Releases zur Verfügung stehen. Ideal wird das durch eine virtualisierte Umgebung unterstützt, in der neue virtuelle Server ohne größeren Aufwand erstellt werden. In diesem Bereich hat VMware mit seinen Produkten eine starke Marktdurchdringung gerade im Enterprise-Sektor. Diese Produkte bieten auch schon länger die Möglichkeit, neue virtuelle Rechner durch Aufrufe an entsprechende APIs zu starten. Dadurch ist es recht einfach möglich, in einer Continuous-Delivery-Pipeline automatisch neue Maschinen zu starten und die entsprechende Software auf ihnen zu installieren. In letzter Zeit werden aber auch in diesem Bereich Open-Source-Lösungen immer populärer wie beispielsweise OpenStack [12]. Hinter diesem Projekt haben sich mittlerweile zahlreiche Firmen versammelt, die die Technologie unterstützen.

Eine andere mögliche Basis für Continuous Delivery sind PaaS-Cloud-Dienste: Sie bieten eine Ablaufumge-

Listing 1

```
Gegeben ist, dass ein Kunde einen Sparplan abgeschlossen hat
und er hat 3 Monate 100 € eingezahlt
wenn er den Sparplan auflöst
sollte er 306 € bekommen
```

Mehr zum Thema

Wer sich weiter mit dem Thema Continuous Delivery beschäftigen möchte, vor allem mit den für den Einsatz notwendigen Architekturen, sollte einen Blick auf die letzte Ausgabe des Java Magazins (11.2013) werfen, die Continuous Delivery in einem großen Heftschwerpunkt beschreibt.
www.javamagazin.de

bung an, in der nur noch die Anwendungen installiert werden müssen und es können typischerweise auch beliebig viele Umgebungen mit unterschiedlichen Versionen der Anwendung installiert werden. Technologien im Bereich PaaS für Java sind bereits in einer Artikelserie im Java Magazin ausführlich dargestellt worden [16]. Ein Tool, das im Rahmen der Serie noch nicht vorgestellt wurde, ist Docker – ein sehr einfaches PaaS, das ohne Weiteres auch auf eigenen Rechnern installiert werden kann [13]. Während also bei Chef oder Puppet immer noch die Server komplett ab Betriebssystem aufgebaut werden, stellt ein PaaS die Ablaufumgebung bereit, die sonst manuell installiert werden müsste. Die Ablaufumgebungen sind also standardisiert und werden nicht individuell erzeugt, wie dies mit Chef oder Puppet der Fall wäre.

Oft wird die Automatisierung des Deployments als zentraler Bestandteil von Continuous Delivery wahrgenommen – aber in Wirklichkeit ist sie nur eine Voraussetzung für andere Techniken im Bereich des Testings. Erst durch das Zusammenspiel der verschiedenen Praktiken können Anwendungen wesentlich schneller und zuverlässiger in Produktion gebracht werden. Die Automatisierung allein ist aber auch schon wichtig: So können Fehler beim Aufsetzen der Umgebungen vermieden und die Umgebungen exakt passend zum jeweiligen Softwarestand aufgebaut werden. Wenn also nur die Automatisierung allein umgesetzt wird, kann das schon erhebliche Vorteile für die Produktivität der Teams haben.

DevOps = Entwicklung + Betrieb

Klassisch sind IT-Organisationen oft in den Betrieb und die Entwicklung aufgeteilt. Diese Abteilungen haben auch unterschiedliche Ziele: Der Betrieb ist um Stabilität bemüht, während die Entwicklung neue Features in Produktion bringen will – aber am Ende müssen beide Abteilungen gemeinsam für die Nutzer und die Organisation möglichst optimale IT-Dienste anbieten.

Mindestens bei der Installation neuer Releases müssen die Abteilungen eng zusammenarbeiten. Continous Delivery ändert gerade in diesem Bereich den technologischen Ansatz. DevOps ist eine organisatorische Reaktion. Der Name ist schon Programm: Die Bereiche Entwicklung (Development) und Betrieb (Operations) wachsen zusammen. Dafür gibt es unterschiedliche Gründe: Als Konsequenz aus Continuous Delivery werden viele klassische Betriebsprozesse anders. Es geht nicht mehr darum, einen Server zu installieren, sondern die Installation der Anwendungen und Infrastrukturen muss automatisiert werden. Dadurch werden klassische Betriebsaufgaben eher zu Softwareentwicklung – eben der Entwicklung von Software für die automatisierte Installation von Anwendungen. Betriebler wissen am besten, welche Anforderungen solche Systeme erfüllen müssen. Letztendlich entstehen so gemischte Teams – wie sie bei agilen Teams auch schon Gang und Gäbe sind. Nur bleiben agile Teams bei den Skills auf Entwicklung und Anforderungsanalyse beschränkt, während DevOps auch den Betrieb in Betracht zieht.

Durch DevOps ergeben sich auf technischer Ebene weitere Potenziale. So kann die Entwicklung beispielsweise beim Monitoring zusätzliche Werte liefern, um so mehr über den Zustand der Anwendung zu erfahren. Oder die Entwicklung kann in die Software Schalter einbauen, um bestimmte Features zu deaktivieren. Dadurch kann bei Problemen in Produktion ein Teil der Software deaktiviert werden, statt das gesamte System ausfallen zu lassen. Solche Ansätze wären zwar auch ohne gemischte Teams möglich, aber oft entstehen erst so die notwendigen Diskussionen und auch die Tools können dann auf dem „kurzen Dienstweg“ entstehen. Technisch können beispielsweise Klassiker wie JMX für die Integration der Software aus dem Betrieb dienen.

DevOps ist also keine Technologie – sondern ein Vorgehen. Letztendlich wird die agile Vision erweitert, möglichst schnell möglichst viel Wert zu schaffen – und zwar dadurch, dass auch der Betrieb sich an den in der Entwicklung schon dominierenden agilen Werten orientiert.

Sizing

Continuous Delivery und DevOps helfen dabei, neue Features in Produktion zu bringen. Aber auch nicht funktionale Anforderungen können mit neuen Technologien ganz anders gelöst werden. Ein Beispiel ist Sizing und Performance: Es wäre optimal, wenn die Anwendungen einfach immer gerade so viele Ressourcen bekommen, dass sie für die Anwender ausreichend performant sind. Meistens wird aber die gewünschte Performance festgelegt und ein Mengengerüst für die Anzahl der Nutzer und Daten erstellt. Anhand dessen wird dann die notwendige Hardware beschafft. Dieses Konzept hat einige Nachteile:

- Die Kosteneffizenz ist gering. Hardwareressourcen für Lastspitzen werden beschafft und vorgehalten – selbst wenn diese nur sehr selten auftauchen.
- Es kann nicht flexibel auf ungeplante Spitzen reagiert werden. Wenn die Hardware eben doch nicht ausreicht, muss ein neuer, aufwändiger Beschaffungsprozess angestoßen werden.
- Der Ausbau ist oft nur grobgranular möglich. Wenn die Anwendung auf einem Cluster von zwei Servern läuft, kann er nur sinnvoll um mindestens einen Server erweitert werden – was die Kapazität gleich um 50 Prozent erhöht.

Schon wegen Continuous Deployment kann dieser Ansatz nicht durchgehalten werden. Wenn mehrere Umgebungen zur Verfügung stehen sollen, auf denen die Software installiert und getestet werden kann, erzwingt dieses unflexible Modell die Beschaffung von vielen Hardwareressourcen, die aber nur während der Entwicklung genutzt werden. Daher ist Continuous Delivery nur auf virtualisierten Umgebungen oder in einer Cloud wirklich sinnvoll realisierbar. So können die Ressourcen flexibel nur dann zur Verfügung gestellt werden, wenn sie wirklich benötigt werden.

Wenn diese Möglichkeiten einmal zur Verfügung stehen, kann auch mit dem Sizing der Produktionsumgebungen anders umgegangen werden: Statt statisch die notwendigen Ressourcen zur Verfügung zu stellen, können je nach Last neue Server in das System integriert werden. So werden nur jeweils die Ressourcen genutzt, die bei der aktuellen Last tatsächlich notwendig sind. Und bei einer ungeplanten Spitze werden entsprechend mehr Server hinzugefügt. Technisch ist es für diese Ansätze notwendig, dass neue Server zur Laufzeit gestartet werden. Dazu sind Clouds oder entsprechende Virtualisierungsmechanismen notwendig.

Damit geht ein radikal anderes Verständnis von Systemen einher: Während früher Systeme statisch Ressourcen belegt haben, werden ihnen mit diesem Paradigma dynamisch Ressourcen zugewiesen. Das ist kosteneffizienter und flexibler.

Die Softwarearchitektur muss natürlich damit umgehen können, dass die Ressourcen nicht mehr „für immer“ zur Verfügung stehen. Beim Hochskalieren muss die Last also möglichst schnell auf mehr Server verteilt werden. Und wenn die Last geringer wird, muss es möglich sein, Server zu deaktivieren. Damit dies möglich ist, darf auf den Servern kein Zustand gehalten werden wie beispielsweise die Session von Benutzern. Beim Hochskalieren ist auf den neuen Servern noch kein Zustand vorhanden, sodass sie nur für neue Nutzer zur Verfügung stehen. Und die Server können auch nicht mehr ohne Weiteres heruntergefahren werden, weil dann die Daten einiger Nutzer nicht mehr zur Verfügung stehen. So groß ist die Änderung aber nicht: Die Server möglichst zustandsfrei zu halten, ist schon lange als eine Best Practice etabliert.

Wesentlich für dieses Vorgehen ist also lediglich, dass die Zahl der genutzten Server je nach Last reguliert wird. PaaS-Clouds bieten dazu von Haus aus Möglichkeiten. Aber auch andere Clouds, die nur Infrastruktur anbieten, haben in diesem Bereich Möglichkeiten. So kann die Amazon-Cloud durch den Elastic Load Balancer aufgrund bestimmter Regeln neue Server starten – beispielsweise wenn die Bearbeitung eines Requests zu lange dauert [15].

Solche fortgeschrittenen Technologien sind aber nicht immer notwendig: Wenn die Infrastruktur es möglich macht, neue Server zu starten, kann ein einfaches Skript, das die aktuelle Performance misst und dann gegebenenfalls neue Server startet, bereits ausreichend sein.

Ausfallsicherheit

Auch bei der Ausfallsicherheit ergeben sich neue Ansätze: Klassisch wird vor allem auf eine hohe Verfügbarkeit der Hardware gesetzt. Aber dieser Ansatz hat seine Grenzen: Es hat sich wohl fast schon jeder in einer Situation wiedergefunden, in der die angeblich so ausfallsicheren Systeme dann doch ausfallen. Üblicherweise sind die Auswirkungen dann erheblich: Die Anwendung stellt ihren Dienst ein und gegebenenfalls gehen auch Daten verloren. Daher streben IT-Organisationen an, die Verfügbarkeit der Hardware immer weiter zu optimieren.

Aber ab einer bestimmten Anzahl Server werden Ausfälle nicht mehr zu einem Sonderfall, sondern kommen ständig vor. Firmen wie Google haben tausende von Servern ständig im Einsatz – spätestens dann ist es komplett unrealistisch, sich darauf zu verlassen, dass die Hardware schon nicht ausfallen wird. Dann muss die Software damit umgehen können, dass einzelne Rechner nicht zur Verfügung stehen. Oft werden daher Server redundant vorgehalten – in einigen Fällen sogar in unterschiedlichen Rechenzentren. Das führt dann zu erheblich höheren Kosten.

Wenn allerdings schon die beschriebenen Voraussetzungen für Flexibilität gegeben sind, kann auf den Ausfall eines Servers reagiert werden: Es können neue Server gestartet werden – gegebenenfalls auch in anderen Rechenzentren oder in einer Cloud. Die Software muss mit dem Ausfall eines Servers umgehen können, denn wie bereits erläutert, werden Server abgeschaltet, wenn die Last zurückgeht – und das ist einem Ausfall nicht unähnlich.

Auch für die Ausfallsicherheit gilt also: Wenn die Anwendungen keinen Zustand auf den Servern halten, ist der Ausfall eines einzelnen Servers durchaus verschmerzbar. Irgendwo müssen Daten aber aufbewahrt werden – dazu bieten sich NoSQL-Datenbanken an, wie der Kasten: „Ein Beispiel: NoSQL“ im Detail diskutiert.

Auch für die Softwarearchitektur hat diese Idee Auswirkungen: Systeme müssen mit Ausfällen rechnen und sinnvoll mit ihnen umgehen. Beispielsweise gibt [14] Hinweise, wie mit solchen Situationen sinnvoll umgegangen werden kann. Oft kann bei dem Ausfall eines Diensts statt den gelesenen Werten einfach ein Default-Wert genutzt oder bei schreibenden Zugriffen die Werte zunächst gepuffert werden. Dieser triviale Schritt sorgt dann dafür, dass der Ausfall des Diensts zwar schlechtere Ergebnisse liefert, aber nicht zu einem vollständigen Ausfall führt. Während also eine klassische Architektur den Ausfall eines Servers direkt an den Nutzer weitergibt, können diese Architekturen mit dem Ausfall einiger Server umgehen. So entstehen Systeme, die gegenüber der Infrastruktur widerstandsfähig sind.

Was bedeutet das für Java? Im Wesentlichen ergeben sich Konsequenzen weniger für die Technologien sondern für die Architektur. Beispielsweise sollte die Anwendung zustandslos sein, sodass der Ausfall eines Servers nicht zu einem Datenverlust führt. Letztendlich werden die Anwendungen so ausfallsicherer: Sie sind nicht mehr an die Verfügbarkeit einzelner Server gebunden, sondern gehen in der Software mit dem Ausfall um.

Die wesentliche Änderung ist die Ablaufumgebung: Statt einige hoch performante und ausfallsichere Server zu nutzen, laufen die Anwendungen in virtualisierten Umgebungen. Je nachdem, wie viele Kapazitäten gerade benötigt werden, können mehr oder weniger Ressourcen hinzugefügt werden. Dazu kann eine virtualisierte Infrastruktur, eine Cloud oder auch eine PaaS-Lösung genutzt werden.

Fazit

Mit Continuous Delivery und DevOps gibt es zwei Konzepte, mit denen agile Ansätze auch auf den Betrieb und die Produktivstellung von Anwendungen adaptiert werden. Java-Entwickler sollten sich daher mit Technologien für Continuous-Delivery-Pipelines vertraut machen. Durch DevOps werden Entwickler und Architekten in Zukunft viel dichter mit dem Betrieb zusammenarbeiten und auch Skills im Bereich von Betrieb aufbauen. Gerade mit Continuous Delivery geht eine deutlich flexiblere Infrastruktur einher, die dann beim Sizing und bei der Flexibilität auch in der Produktion weitere Vorteile ermöglichen – wenn die Softwarearchitektur entsprechend angepasst wird. Neue Technologien wie NoSQL setzen diese Ansätze schon geschickt um und können daher als Vorbild dienen.

Eberhard Wolff arbeitet als freiberuflicher Architekt und Berater. Außerdem ist er Java-Champion und Leiter des Technologiebeirat der adesso AG. Sein technologischer Schwerpunkt liegt auf Spring, NoSQL und „Cloud".

http://ewolff.com

Links & Literatur

[1] Humble, Jez; Farley, David: „Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", Addison-Wesley, 2010

[2] http://jenkins-ci.org/

[3] http://hudson-ci.org/

[4] http://www.sonarqube.org/

[5] http://docs.seleniumhq.org/

[6] http://jbehave.org/

[7] http://thucydides.info/

[8] http://grinder.sourceforge.net/

[9] http://jmeter.apache.org/

[10] http://www.opscode.com/chef/

[11] http://puppetlabs.com/puppet/

[12] http://www.openstack.org/

[13] http://www.docker.io/

[14] Nygard, Michael T.: „Release It!: Design and Deploy Production-Ready Software", Pragmatic Programmers, 2007

[15] http://aws.amazon.com/elasticloadbalancing/

[16] Wolff, Eberhard; et al: „PaaS – Die wichtigsten Java-Clouds auf einen Blick", entwickler.press, 2013 (nur als E-Book)

Ein Beispiel: NoSQL

Gerade für das Sizing und die Flexibilität sind zustandslose Anwendungen wünschenswert. Dabei gibt es allerdings ein Problem: Eine Anwendung muss einen Zustand verwalten können. Es gibt also eigentlich nur die Möglichkeit, den Zustand nicht in der Anwendung sondern in der Datenbank zu halten. Dennoch sollte die Datenbank den Kriterien aus dem Artikel bezüglich Ausfallsicherheit und Sizing gerecht werden. Das ist bei relationalen Datenbanken oft nicht der Fall – daher ist es in diesem Zusammenhang sinnvoll, einen Blick auf NoSQL-Datenbanken zu werfen, die ganz andere Lösungen parat haben:

- Für das Sizing bieten NoSQL-Datenbanken meistens horizontale Skalierbarkeit an. Dabei werden die Daten und Anfragen auf mehrere Server verteilt. Abhängig von der Datenmenge können so also mehr oder weniger Server genutzt werden. Dadurch können Kapazitäten dynamisch erweitert werden – gegebenenfalls müssen dazu natürlich Daten auf neue Server verteilt werden, was die Dynamik etwas einschränkt.
- Für die Ausfallsicherheit werden die Daten auf mehrere Server repliziert. Wenn ein Server ausfällt, sind die Daten immer noch auf mehreren anderen Servern vorhanden, sodass der Ausfall toleriert werden kann. Dafür muss es allerdings Kompromisse geben: Die Replikation führt dazu, dass die Daten auf Server kopiert werden müssen. Das bedeutet, dass die Daten nicht sofort auf allen Servern zur Verfügung stehen. Sie sind also nicht immer konsistent: Eine Anfrage an einen Server kann ein anderes Ergebnis bringen, als die Anfrage bei einem anderen Server, wenn die Daten noch nicht vollständig kopiert worden sind.

Deutlich wird hier das Konzept hinter der Architektur von NoSQL-Datenbanken: Durch Redundanz kann Ausfallsicherheit erzeugt werden – und zwar in diesem Fall mithilfe der Software. Und durch die Aufteilung der Last auf viele Maschinen kann ein flexibles Sizing erreicht werden. Diese Ansätze können auch in eigenen Architekturen genutzt werden. Außerdem können NoSQL-Datenbanken also Daten auch entsprechend den Anforderungen aus der neuen Welt gespeichert werden – sodass die Anwendungen trotzdem einen Zustand halten können.

Interessanterweise haben NoSQL-Datenbanken auch Auswirkungen auf Continuous Delivery und DevOps: Die Datenbanken sind auch bezüglich der Schemas viel flexibler, sodass bei Continuous Delivery neue Releases der Software einfacher installiert werden können, da aufwändige Schemamigrationen entfallen. Bezüglich DevOps ändern NoSQL-Datenbanken das Zusammenspiel zwischen Entwicklung und Betrieb: Bei relationalen Datenbanken ist der Betrieb im Cluster für die Software transparent. Bei NoSQL haben Entwickler hingegen die Wahl: Lieber Daten schneller lesen – aber potenziell Inkonsistenzen in Kauf nehmen – oder ist die Konsistenz doch wichtiger? Die Daten werden repliziert, sodass Änderungen bei Repliken erst mit Verzögerung eintreffen. Bei Leseoperationen kann also von den Replikaten gelesen werden – die Daten sind dann möglicherweise veraltet – oder die Konsistenz ist wichtiger. Gerade beim Betrieb von Clustern verlagern sich also Herausforderungen vom Betrieb zur Entwicklung.

Anzeige

Agile Teams an die Macht!

Crowdgovernance

IT-Governance und agiles Arbeiten in der Softwareentwicklung passen nicht zusammen – da sind sich zumindest die Vertreter der agilen Welt recht einig. Crowdgovernance vereint praktische Ideen von Scrum, Liquid Democracy und Communityprozessen miteinander. Damit unterstützt es Teams, den roten Faden zu finden und ihr Handeln an einer gemeinsamen Strategie auszurichten.

von Sebastian Mancke

Die Spatzen pfeifen es bereits seit Langem von den Dächern: Softwareentwicklung muss agil durchgeführt werden. Alles andere bringt nichts – und kaum jemand traut sich heute noch öffentlich etwas anderes zu behaupten. Dennoch hapert es in großen Projekten und großen Organisationen oft noch sehr. Meist wird an etablierten Strukturen festgehalten und die Umstellung auf agiles Arbeiten erfolgt nur zögerlich. Der Grund hierfür ist nicht selten die Angst davor, die Kontrolle zu verlieren und die Geschehnisse nicht mehr lenken zu können. Aus dieser Angst heraus wird oft an einer klassischen und zentralistischen IT-Governance festgehalten.

Klassische IT-Governance

IT-Governance an sich ist nichts Schlechtes und verfolgt plausible Ziele:

- Konsistenz der IT-Systeme
- Vermeidung von Fehlentscheidungen
- Unterstützung strategischer Ziele
- Minimierung von Risiken
- Effizienz
- Homogene IT-Landschaften
- Zukunftsfähigkeit des Unternehmens

In der Praxis führt dies jedoch oft zu einem starren Konstrukt von Regeln und dem Versuch, alles aus zentraler Hand zu steuern. Folgende „Bad Smells“ sind häufig anzutreffen:

- Zentrale Technologie-Whitelists, die projekt- oder sogar konzernweit vorschreiben, welche Technologie eingesetzt werden darf.
- Es fehlt die Möglichkeit für jedermann, einfach Einfluss auf die geltenden Vorgaben zu nehmen.
- Zentrales Design aller Schnittstellen.
- Ein verbindlicher „Standardstack“ oder ein Framework, das für jedes Projekt verwendet werden muss (häufig irgendeine veraltete Eigenentwicklung).
- Trennung zwischen Architekt und Entwickler.
- Eine zentrale Architekturabteilung, die alles absegnen oder sogar selbst entwerfen möchte.

Natürlich hängt der Nutzen oder auch Schaden immer von der konkreten Ausführung und dem Umgang damit ab. Eine gut gemachte Technologie-Whitelist kann durchaus helfen, wenn sie die Entscheidungen motiviert und den Trade-off zwischen Regelung und Freiraum richtig vermittelt. Auch kann eine zentrale Architekturabteilung eine agile Arbeit der Teams unterstützen, wenn sie mit den richtigen Mitarbeitern besetzt ist. Doch viel zu häufig führen die zentralistischen Governance-Ansätze lediglich zu einem großen Katalog von IT-Vorgaben und Richtlinien. Als Ergebnis unpassender IT-Vorgaben gibt es zwei typische Reaktionen:

- Ein Teil der Mitarbeiter findet sich mit den unpassenden Vorgaben ab und akzeptiert, dass es in dem Unternehmen nicht möglich ist, die richtigen Entscheidungen zu treffen. Da man nur das verantworten kann, worauf man auch Einfluss hat, sinkt in der Folge meist die Identifikation und Übernahme der Verantwortung für die eigenen Ergebnisse.
- Der andere Teil der Mitarbeiter akzeptiert die Vorgaben nicht, sondern trifft einfach die richtigen Entscheidungen. Da es nicht opportun ist, Regeln zu missachten, werden widerstrebende IT-Entscheidungen aber meist nicht offen kommuniziert, sondern als U-Boot betrieben. Im Ergebnis führt dies zu besseren Ergebnissen als die erste Variante. Es birgt aber ein ganz neues Risiko: Durch die fehlende Transparenz wird die Chance genommen, die IT-Governance an die neuen Ereignisse anzupassen und strategisch zu steuern. In der Folge wird der Graben zwischen Governance und gelebter IT immer breiter.

IT-Governance und Innovation

Das höchste Gut eines Unternehmens ist die Innovation. Innovation kann aber nur dann geschehen, wenn es ausreichenden Raum und die Motivation für freie Weiterentwicklung gibt. Jede Regel macht diesen Innovationsraum kleiner und jede Verlagerung der Verant-

wortung, weg von den Entwicklern, verringert deren Motivation zu gestalten. Eine zentralistisch ausgeübte IT-Governance führt damit mutmaßlich zu einer Innovationsbremse.

Steuerung und trotzdem agil?

Es gibt die Theorie, dass sich gute Ideen von alleine durchsetzen und agile Teams von selbst das Richtige entscheiden. Soll die Antwort also sein, den Anspruch an Steuerung und Governance komplett abzulegen? Nein!

Es ist nicht planbar, wie viel Zeit vergeht, bis Teams ohne Unterstützung eine klare Richtung finden. Außerdem führt der Weg dorthin oft über viele Umwege, die Zeit, Scope und Budget gefährden. Damit birgt eine ungesteuerte Entwicklungsmannschaft das Risiko unnötiger Kosten und führt oft zum Scheitern des Projekts. Governance ist also erstrebenswert. Aber wie kann sie gestaltet werden, um der Agilität und Innovation förderlich zu sein? Folgende Regeln sollten als Eckpfeiler einer agilen Governance dienen.

Grundsatz 1: Die Entscheidungshoheit liegt in den Teams!

Der oberste Grundsatz einer agilen Governance ist es, die Gestaltungsverantwortung bei den Mitarbeitern anzusiedeln, bei denen auch die Umsetzung liegt: Wer etwas tut, **darf** entscheiden, wie es getan wird. Entscheidungsverantwortung ist aber kein Selbstbedienungsladen: Wer etwas tut, **muss** auch selbst über sein Schicksal entscheiden.

Im Gegenzug muss das Team die Konsequenzen der getroffenen Entscheidungen selbst tragen. Wenn etwas nicht funktioniert, kann also nicht mit dem Zeigefinger auf falsche Vorgaben verwiesen werden.

Grundsatz 2: Befähige deine Mitarbeiter!

Die Anforderungen an Teams sind sehr hoch, wenn diese ihr Schicksal selbst in die Hand nehmen sollen. Es ist wichtig, die Teams nicht im Regen stehen zu lassen. Sie müssen dahin entwickelt werden, ihren Anforderungen auch angemessen gegenüber zu stehen. Dazu ist eine ausreichende und passende Skill-Verteilung nötig. Dies kann durch Teaming- und Schulungsmaßnahmen geschehen.

Grundsatz 3: Das Gesamtbild muss klar sein!

Damit die Teams selbst steuern können, müssen sie die Richtung kennen. Es muss klar sein, welche Ziele die Organisation oder das Projekt verfolgt, wie der grobe Plan dahin aussieht und was die eigene Rolle ist. Nur auf Basis einer ausreichenden Wissensgrundlage können die richtigen Entscheidungen abgeleitet werden.

Grundsatz 4: It's all about Communication!

Ein guter Überblick stellt sich nicht von alleine ein. Das Wissen über die Organisation, die Ziele und die Arbeit von anderen Teams muss aktiv ausgetauscht und vermittelt werden. IT-Organisationen sind schnelllebig. Es reicht nicht, alle Informationen irgendwo zu dokumentieren und zum Lesen zur Verfügung zu stellen. Die einzige Lösung ist der Aufbau einer guten Informations- und Kommunikationskultur. Das gegenseitige Austauschen von Informationen muss Kern der täglichen Arbeit sein und von allen gelebt werden. Die Kunst hierbei ist es, den Spaß an Kommunikation zu wecken, ohne eine Informationsflut zu schaffen. Informationsmanagement hat noch einen Vorteil: Es schafft Identifikation.

Grundsatz 5: Ermögliche Heterogenität

Im Ziel einer Governance steht oft, die gesamte IT völlig einheitlich zu gestalten, oder die Heterogenität zumindest möglichst gering zu halten. Das ist falsch. Natürlich ist es praktisch, wenn alle Systeme gleich sind. Dies darf aber keine Forderung auf oberster Ebene für Technologieentscheidungen sein. Technologie entwickelt sich schnell weiter und Weiterentwicklung braucht Vielfalt. Es darf kein unkontrollierbarer Technologiezoo entstehen, vor allem aber darf nicht versucht werden, die Vielfalt zu unterbinden. Das Wichtigste ist, zu lernen, wie Systeme effizient gemanagt werden können.

Grundsatz 6: Förderung von Zielen, nicht von Verboten!

Schon bei der Kindererziehung merken wir: Verbote sind zwar manchmal unumgänglich, helfen den Kindern aber meist nicht dabei, das Richtige zu tun. Erwachsene Menschen sind genauso. Das Aussprechen von Verboten lähmt meist nur. Es führt dazu, sich mit dem zu beschäftigen, was nicht getan werden soll. Viel wichtiger ist es jedoch, die Mitarbeiter auf Alternativen zu dem Verbot zu fokussieren und deren Handlungen damit nicht zu stoppen, sondern in eine positive Richtung umzulenken.

Viel wirksamer als ein Verbot ist es häufig auch, die Konsequenzen einer Entscheidung deutlich zu machen. Wo dies möglich ist, sollte man dafür sorgen, dass die Personen, die diese Entscheidung treffen, auch selbst mit den Konsequenzen ihrer Entscheidungen konfrontiert sind. Ein gutes Beispiel hierfür ist der „you build it, you run it"-Grundsatz [1].

Von der Theorie zur Praxis

Es ist meist leicht, ein paar agile Grundsätze zu formulieren. Dies hat aber meist noch keine Wirkung auf eine Organisation. Im Folgenden sind eine Reihe ganz konkreter Methoden definiert, die dabei helfen sollen, die obigen Grundsätze zu leben und zu etablieren. Die Liste ist weder vollständig, noch ist es wichtig, alles darin einzuführen.

Liquid Democracy und Votings

Die Piratenpartei hat neue Maßstäbe in der toolgestützten Meinungs- und Entscheidungsbildung gesetzt. Auch wenn Projekte und Unternehmen im Kern meist nicht basisdemokratisch organisiert sind, lassen sich diese Elemente sehr effektiv nutzen. Die Mitbestimmung und kollektive Meinungsbildung bietet zwei große Vorteile: Entscheidungen, die von vielen Köpfen getroffen

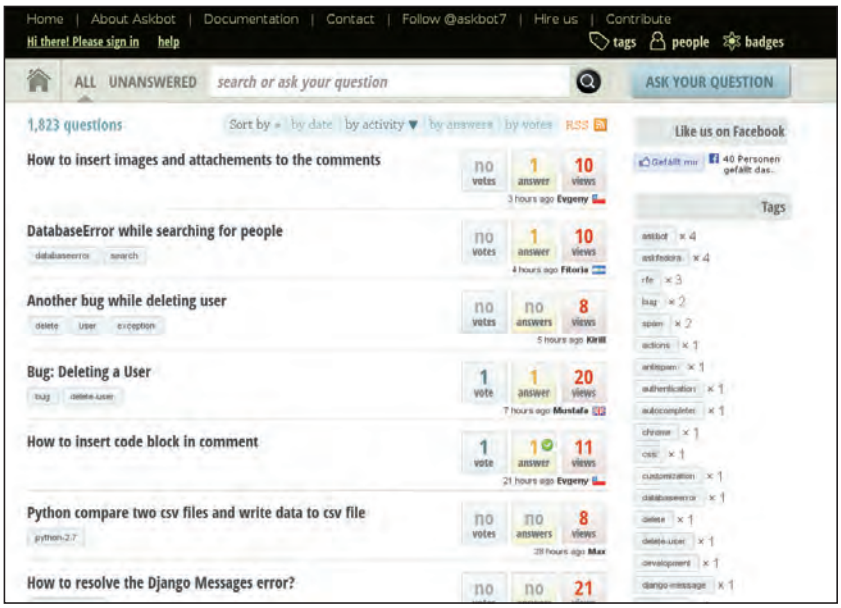


Abb. 1: Votingtool-Beispiel

wurden, sind robust und geraten bei Führungswechseln nicht so schnell ins Wanken. Außerdem liefert die Mitbestimmung eine sehr hohe Identifikation mit dem Unternehmen sowie eine hohe Akzeptanz für die so getroffenen Entscheidungen. Um Mitbestimmung zu organisieren, bieten sich Votingfunktionen bestehender Tools an. Es können aber auch Softwaretools aus dem Liquid-Democracy-Umfeld verwendet werden (**Abb. 1**):

- Askbot [2]
- Liquid Feedback [3]
- Idea Torrent [4]

Die Kunst beim Aufbau einer internen Votingplattform liegt darin, diese mit passenden Inhalten zu beleben. Dabei muss einerseits dafür gesorgt werden, dass relevante Entscheidungen auch wirklich den Weg auf die Plattform finden und andererseits sollte vermieden werden, ständig alle Kleinigkeiten kollektiv zu diskutieren. In der Praxis hat sich gezeigt, dass es gut ist, eine wechselnde Gruppe von Moderatoren zu etablieren, die eine leichte Steuerung auf die Inhalte ausüben. Des Weiteren ist es sehr wichtig, dass das Management die getroffenen Entscheidungen anerkennt. Natürlich hat es grundsätzlich immer ein Vetorecht (Kasten: „Kontrollverlust des Managements?“). Wird jedoch oft von diesem Vetorecht Gebrauch gemacht, führt dies zu einer starken Demotivation der Mitarbeiter.

Kontrollverlust des Managements?

Den Kern der Crowdgovernance bildet die Verantwortungsübernahme durch die Mitarbeiter. Dies hat die Annahme zur Grundlage, dass nur die wirklich handelnden Personen in der Lage sind, schnell die richtigen Detailentscheidungen zu treffen. Das kann nur dann funktionieren, wenn das Management sich auch darauf einlässt, nicht mehr alle Entscheidungen selbst zu treffen oder zu kontrollieren.

Oft führt dies zu starken Ängsten bei Team-Leads und Abteilungsleitern. Da sie ihre Rolle und ihre Einflussmöglichkeiten schwinden sehen. Das neue Rollenverständnis des Managements muss im Kern haben, den Raum für eine innovative Arbeit zu schaffen und gemeinsam mit den Mitarbeitern die Vision und die Strategie zu formen.

Mitarbeiter in die Verantwortung zu bringen, heißt aber nicht, die Führungsverantwortung loszulassen. Die Verantwortung für das Handeln der Organisation bleibt beim Management. Damit haben auch alle demokratisch hergeleiteten Entscheidungen eine Grenze. Wenn die Dinge nicht in die richtige Richtung laufen und klare Fehlentscheidungen entstehen, ist das Management weiterhin in der Verantwortung. Es kann dezent einlenken oder im Notfall auch abrupt eingreifen.

Technologieratings

Die Idee der Technologieratings geht auf das ThoughtWorks Technology Radar [5] zurück. Die darin enthaltenen Empfehlungen betrachten die gesamte IT-Welt. Dieselben Ratings können auch intern durchgeführt werden. Dabei werden Technologien in unterschiedlichen Kategorien gesammelt und anschließend gemeinsam bewertet. Abhängig vom Organisationskontext sollten diese Kategorien individuell gewählt werden. Der Autor hat z. B. mit folgender Einteilung gute Erfahrungen gemacht:

- Programmiersprachen und Frameworks
- Tooling
- Testing
- Architektur, Paradigmen, Vorgehen

In der anschließenden Bewertung sollte es jedem Mitglied der Organisation möglich sein, die Vorschläge in folgenden Kategorien zu bewerten:

- **Einführen:** Die Technologie ist in der Organisation noch nicht weit verbreitet, ihre breite Verwendung wird aber empfohlen.
- **Testweise einsetzen:** Die Technologie ist sehr interessant und birgt kein großes Risiko, sodass sie bereits testweise eingesetzt werden soll (insbesondere in nicht kritischen Komponenten mit kurzer Lebensdauer).
- **Bewerten/Evaluieren:** Die Technologie ist interessant. Sie sollte angeschaut, bewertet und weiter beobachtet werden. Ein Einsatz in Projekten sollte jedoch noch nicht erfolgen.
- **Beibehalten:** Die Technologie wird schon in der Breite verwendet und sollte auch weiterhin beibehalten werden.
- **Vermeiden/Abschaffen:** Die Technologie sollte, wenn möglich, nicht gewählt werden. Wo sie bereits im Einsatz ist, sollte erwogen werden, davon weg zu migrieren.

In einem einzelnen Team empfiehlt es sich, ein solches Rating gemeinsam mit Post-its oder an einem White-

Anzeige

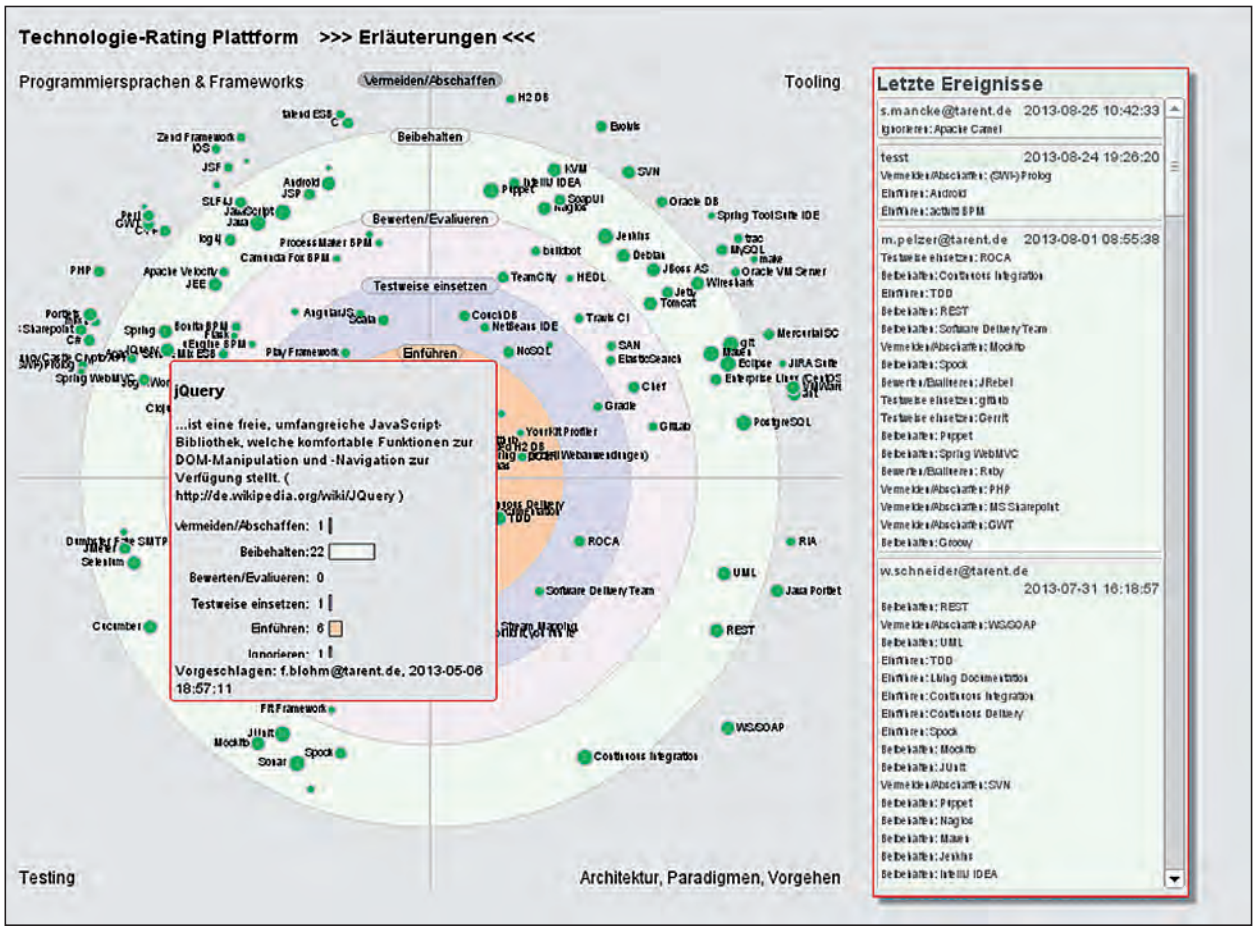


Abb. 2: Grafische Darstellung der Techrating-Ergebnisse

board durchzuführen und die dabei entstehende Diskussion zu nutzen. Wenn jedoch ein Technologiebild für eine größere Organisation erstellt werden soll, wird ein Tooling benötigt. Hierzu hat der Autor eine einfache Webanwendung entwickelt. Sie steht unter der MPL-Lizenz und kann unter [6] bezogen oder auch unter [7] direkt verwendet werden. Die Anwendung unterstützt die Sammlung und Bewertung der Ideen und bietet eine grafische Darstellung der Ergebnisse (**Abb. 2**).

Es empfiehlt sich, ein bis zweimal pro Jahr eine Aktualisierung des Techratings vorzunehmen. Das Ergebnis kann als Orientierung und Richtlinie in der Organisation verwendet werden. Die Teams, die vor Architekturentscheidungen stehen, haben jetzt die Möglichkeit, zu erkennen, welche Entscheidungen konsensfähig sind und welche andere Teams eher nicht treffen würden. Da das Techrating-Ergebnis einer kollektiven Meinungsbildung ist, besitzen die darin getroffenen Aussagen meist eine sehr hohe Akzeptanz.

Aber auch die Organisationsführung kann die Ergebnisse nutzen, um zu sehen, ob neue Strategien angenommen werden oder auch, um neue Strategien aus dem Techrating abzuleiten.

Beispiel: Vision Statement der Deutschen Post DHL

- Provider of Choice
- Investment of Choice
- Employer of Choice

Beispiel eines Leitsatzes

Wir richten die Entwicklung unserer Software am Nutzen unserer Kunden und Anwender aus! Hierzu entwickeln wir in kurzen Zyklen und legen besonderen Wert auf automatisches Deployment und Testautomatisierung sowie auf die Integration mit Open-Source-Software und der Community.
SWE-Leitsatz, tarent AG

Vision und Leitsätze formulieren

Softwareentwickler, die in einem Team eingebunden sind, fokussieren meist sehr stark auf die kurzfristigen Teamziele. Das ist wichtig, aber manchmal geht dabei der Blick auf das Große und Ganze verloren. Daher ist es von großer Bedeutung, das Gesamtbild der Unternehmung klar und deutlich zu machen. Hierzu können Visionen und Leitsätze formuliert werden. Eine Vision gibt Ziel und Richtung und ist dabei die abstrakteste Ebene einer Zielbeschreibung. Leitsätze hingegen geben eine Orientierung, wie ein Ziel erreicht werden kann. Beides kann auf verschiedenen Ebenen formuliert werden.

Die Kästen zeigen jeweils ein Beispiel einer Vision (Kasten: „Beispiel: Vision Statement der Deutschen Post DHL“) und eines Leitsatzes (Kasten: „Beispiel eines Leit-

satzes“). Weitere sind unter [8] zu finden. Zur weiteren Beschäftigung mit der Produktvision ist weiterhin der Artikel von Roman Pichler zu empfehlen [9], in dem er die Produktvision als „the projects true north“ beschreibt.

Eine Vision kommt meist vom Management oder sogar dem Kunden. Leitsätze hingegen sollten aus der Mitte der handelnden Personen kommen. Eine verbreitete Form von Leitsätzen sind z. B. auch die Definions of Done von Scrum-Teams. Um ein hohes Commitment zu unternehmensweiten Leitsätzen zu bekommen, sollten diese allgemein verabschiedet oder breit abgestimmt werden. Hierbei können die oben beschriebenen Votingmechanismen zum Einsatz kommen.

Etablierung einer Unternehmenskommunikation

Eine Vision ist recht statisch, da sie das Ziel für einen langen Weg beschreibt. Um ständig zu wissen, wo lang der Weg gerade verläuft, ist eine regelmäßige Unternehmenskommunikation wichtig. Dies kann z. B. in Form von wöchentlichen Inforunden, Townhall-Meetings oder auch Unternehmensblogs mit allgemeinen Informationen erfolgen. Da diese Informationen nicht in direkter Verbindung mit der täglichen Arbeit stehen, werden sie oft als überflüssig empfunden. Sie sind jedoch elementare Voraussetzung dafür, dass Mitarbeiter in der Lage sind, selbst sinnvolle Entscheidungen zu treffen.

Teamreviews

Teamreviews sind ein klassisches Scrum-Element – evt. sogar das wichtigste. Im Review schließt das Team die Arbeit des Sprints ab und präsentiert dem Product Owner die Ergebnisse zur Abnahme. Die gesamte Organisation ist eingeladen, dem Review beizuwohnen und Feedback zu geben. Damit bietet das Review Raum für zwei wichtige Bestandteile einer lateralen IT-Governance: Informationsaustausch und gegenseitige Kritik. Doch vor allem Letzteres muss geübt werden.

Der Austausch von Kritik wird oft als sehr unangenehm empfunden. Daher sind die meisten Scrum-Meetings sehr „kuschelige“ Veranstaltungen, die nur an der Oberfläche kratzen. Wenn die Teams jedoch gelernt haben, ohne Scham über Erfolge und Defizite zu sprechen, ist es möglich, mit den Reviews eine sehr gute Kultur der gegenseitigen Kontrolle zu etablieren.

Um die Reviews zu beleben, sollten die Teams sich rege gegenseitig besuchen. Am besten ist bei jedem Review mindestens ein Vertreter eines jeden Teams dabei. Dies kann gerne rollieren und sollte auch dann beibehalten werden, wenn die Teams keinen direkten fachlichen Bezug haben. Sollte sich zeigen, dass es zu viele Teams sind, und das Ganze zu viel Zeit in Anspruch nimmt, ist es manchmal besser, die Reviews zu teilen und die Präsentation an die Organisation auf eine Re-

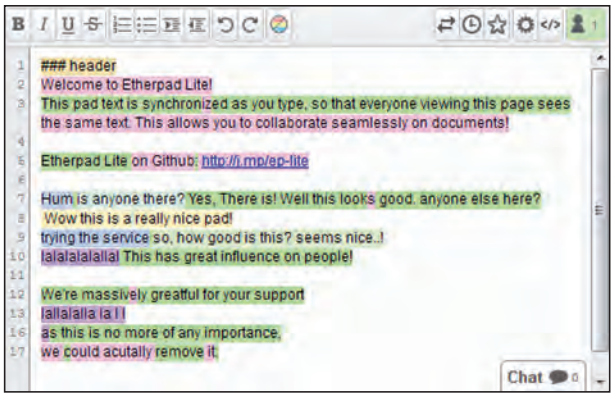


Abb. 3: Etherpad: Realtime Collaborative Editing

viewmesse zu konsolidieren, bei der alle Teams zusammen präsentieren.

Warum sind Reviews so wichtig? Durch sie ist es sehr effizient möglich, auf dem Laufenden zu bleiben, und zu wissen, was in anderen Teams gerade so passiert. Dies ist die Grundlage dafür, die Entscheidungen im eigenen Team sinnvoll an der Allgemeinheit auszurichten und zu sehen, ob gerade in dieselbe Richtung gearbeitet wird. Des Weiteren bieten die Reviews eine Plattform zur gegenseitigen Erinnerung und Ermahnung an gemeinsam getroffene Verabredungen – sie schaffen völlige Transparenz. In einer gut gelebten Reviewkultur ist es einem Team nicht möglich, die Company-DoD zu unterschreiten oder von den Empfehlungen gemeinsamer Technologieratings abzuweichen, ohne dies zu diskutieren und zu reflektieren.

Communities of Practice

Scrum setzt eine primäre Verortung der Mitarbeiter in Teams voraus. Die Zugehörigkeit zu Abteilungen der Organisation wird dadurch meist abgebaut oder verblasst. Es gibt also nicht mehr die Designer, die Architekten oder die Tester. Damit diese Gruppen eine gemeinsame Linie und Ausrichtung entwickeln können, brauchen sie jedoch einen regen Austausch. Hierzu gibt es die Communities of Practice (CoP). Das sind offene Gruppen, die sich regelmäßig zu einem definierten Themenkomplex treffen, um Erfahrungen auszutauschen und ein gemeinsames Vorgehen abzustimmen. Eine CoP ist kein beschlussfähiges Gremium, das für alle verbindliche Entscheidungen trifft. Die Entscheidungshoheit bleibt weiterhin im Team – die CoP erarbeitet jedoch den organisationsweiten Konsens und legt damit die Strategie fest, an der sich die Teams ausrichten sollen. Es ist nicht leicht, eine CoP gut zu gestalten. Folgende Aspekte helfen hier, den Drive zu behalten:

- Ein guter Moderator, der selbst für das Thema brennt und die Gruppe antreibt.
- Ein dynamisches, aber striktes Agendamanagement. Bewährt hat sich die Vorabsammlung von Agendapunkten. Diese werden dann zu Beginn des Meetings kurz bewertet und entsprechend ihrer Priorität in Time Boxes besprochen. Die gewonnenen Erkenntnisse müssen dabei natürlich dokumentiert und allen zugänglich gemacht werden.
- Aufhören, wenn es nicht gut ist. CoPs können sehr dynamisch gestartet und wieder gestoppt werden. Wenn es gerade keine wichtigen Themen gibt, sollte es auch keine CoP geben. Es sollte aber jemand beobachten, wann der Bedarf wieder entsteht.

Für die Durchführung der CoPs eignet sich die Unterstützung durch ein Realtime Collaboration Tool. Hierbei hat sich das Etherpad [10] bewährt (**Abb. 3**). Es ermöglicht das gleichzeitige Editieren von Text mit mehreren Personen und ist somit perfekt, um ad hoc eine Agenda abzustimmen und die Ergebnisse eines Meetings zu protokollieren.

Talks

Für einen regen Austausch über neue Trends und Best Practices sollten regelmäßig Talks gehalten werden. Folgende Formate haben sich in der Praxis bewährt:

- **Der Lightning Talk** ist ein Kurzvortrag (z. B. 20 oder 30 min), in dem ein Thema kurz angerissen wird. Sowohl zuhören als auch Vorbereitung sind leichtgewichtig.
- **Der Tech Talk** ist ein längerer Technologievortrag (1–2 Stunden), der Interessierten bereits einen guten Einblick in eine Materie gibt.
- **Die Knowledge Session** ist ähnlich wie der Tech Talk, fokussiert aber auf fachliche und technische Themen aus der eigenen Organisation, z. B. wie funktioniert eigentlich diese oder jene Komponente in unserem Produkt?

Die Rolle des Architekten

Der Autor vertritt die Meinung, dass es die etablierte Rolle des Architekten in einer agilen Organisation nicht geben sollte. Die Trennung zwischen Entwickler und Architekt beißt sich direkt mit dem Anspruch einer ganzheitlichen Verantwortungsübernahme des gesamten Teams für das komplette System. Dies umfasst Architektur, Code, Qualität, Systemumgebung, Sicherheit und Benutzbarkeit.

Sofern die Entwickler eines Teams nicht bereits zu stark mit dem klassischen Bild des Architekten vorbelastet sind, schadet es aber nichts, den Begriff Architekt analog zu dem Begriff eines Seniorentwicklers zu verwenden. Dies kann im Rahmen der Personalentwicklung hilfreich sein, um Mitarbeitern eine klare Selbst- und Fremdeinschätzung zu ermöglichen.

Auch wenn die Rolle des Architekten in agilen Teams fehl am Platz ist, sind dessen Fähigkeiten natürlich weiterhin von großer Bedeutung. Sie sollten aber von jedem Teammitglied nach seinen Möglichkeiten erbracht werden. Neben der technischen Lösungskompetenz steht hier insbesondere die Kommunikationsfähigkeit im Vordergrund.

Labs

Bevor eine neue Technologie eingesetzt wird, sollte sie erprobt werden. Hierzu gibt es das Format der Labs. Ein Lab ist ein Experiment mit einem klaren fachlichen und zeitlichen Scope und einem standardisierten Vorgehen:

- Öffentliche Formulierung der Problemstellung und Zielerwartung
- Definition von zeitlichem und fachlichem Scope
- Bewerbung auf die Teilnahme an dem Lab
- Konzentrierte gemeinsame Durchführung, z. B. in einem extra Projektraum
- Öffentliche Dokumentation des Ergebnisses
- Vorstellung der Erkenntnisse in einem Lightning Talk

Es sollte sich bemüht werden, die Labs teamübergreifend durchzuführen. Damit wird das Inseldenken aufgebrochen und die gewonnenen Erkenntnisse bilden direkt einen breiteren Konsens in der Organisation.

Fortbildungs- und Schulungskonzepte

Im IT-Bereich gehört die Weiterbildung eigentlich zur täglichen Arbeit. Dennoch tritt in vielen Organisationen oder lange laufenden Projekten oft eine gewisse Trägheit ein, die mit Fortbildungsmaßnahmen belebt werden sollte. Schulungsmaßnahmen bieten eine wunderbare Möglichkeit einer dezenten Steuerung. Die meisten Mitarbeiter setzen bevorzugt die Sachen ein, die sie gut beherrschen. Aus einer gemeinsamen Wissensgrundlage erfolgt damit häufig schon ohne weitere Steuerung eine recht homogene IT-Landschaft.

Um Fortbildung zu organisieren, sollte ein Fortbildungskatalog aufgebaut werden. Dieser hinterlegt für jedes Thema die geeigneten Maßnahmen. Die Themen selbst sollten sich aus unterschiedlichen Quellen speisen, wie Ergebnisse des Technologieradars, Vertriebs- oder Produktanforderungen, Fortbildungswünsche in Mitarbeitergesprächen und offensichtliche Skill-Bedarfe, die durch Team-Leads oder Scrum Master identifiziert werden.

Für unterschiedliche Themen eignen sich auch ganz unterschiedliche Fortbildungsmaßnahmen. Bei Fortbildung denken die meisten zunächst an die klassische Schulung durch einen externen Coach, die entweder in einem Schulungsunternehmen oder inhouse durchgeführt wird. Dies ist jedoch nur eine Möglichkeit, meist sind jedoch die Alternativen effektiver:

- Etablierung von **Subject Matter Experts (SME):** Das sind ausgewiesene Experten für ein Thema. Sie bekommen das Ziel, sich in ein Thema vertieft einzuarbeiten und dieses in der Organisation nach innen und außen zu vertreten sowie an Kollegen weiterzugeben [11].
- **Interne Workshops und Schulungen:** In größeren Organisationen hat man für viele Themen bereits fähige Personen, die das Thema auf ausreichendem Level an andere vermitteln können. Neben dem Kostenaspekt bringt die interne Schulung noch die Vorteile, dass die Kommunikation miteinander angeregt und dabei die gemeinsame Diskussion und Meinungsbildung gefördert wird.
- **Konferenzbesuche** sind ein etabliertes Mittel, um den Trends und aktuellen Entwicklungen zu folgen. Die Reichweite davon erhöht sich, wenn die Konferenzbesucher angehalten sind, im Nachgang kurz über die interessantesten Themen zu berichten.
- Für viele Mitarbeiter ist das **Selbststudium** die effektivste Möglichkeit, sich in neue Themen einzuarbeiten. Dies kann durch explizite Freiräume und ggf. auch durch kontrollierte Lernziele unterstützt werden.

Bei allen Fortbildungsmaßnahmen ist darauf zu achten, dass das Erlernte auch zeitnah angewendet werden kann. Wenn nicht, dann festigt es sich nicht, sondern verblasst schnell wieder. Der praktische Einsatz „Training on the Job“ hingegen ist für die meisten Personen das effektivste Mittel.

Fazit

Ich hoffe, dass Ihnen die konkreten Ideen ein paar Anregungen liefern konnten. Das Ziel sollte es nicht sein, möglichst viele Sachen parallel anzustoßen. Vielmehr ist es als Werkzeugkasten zu verstehen, aus dem gut überlegt die entsprechenden Tools genommen werden können. Wichtiger als die Anzahl der Werkzeuge ist die Etablierung und damit Schlagkraft des einzelnen Tools. Der Autor freut sich natürlich über weitere Ideen und Anregungen sowie über Erfahrungen bei der Einführung der Maßnahmen.

Sebastian Mancke ist Head of Technology bei der tarent AG. Er hat viele Jahre Erfahrung in der Softwareentwicklung und Projektleitung innerhalb der tarent AG sowie in Großprojekten von Kunden.

s.mancke@tarent.de

Links & Literatur

[1] http://tnw.co/qKMVxm

[2] https://askbot.com/

[3] http://liquidfeedback.org/open-source/projekt/

[4] http://www.ideatorrent.org

[5] http://www.thoughtworks.com/radar

[6] http://www.tarent.org/techrating

[7] http://www.techrating.org

[8] http://bit.ly/14Mg9uj

[9] http://bit.ly/15y45fi

[10] http://etherpad.org

[11] http://en.wikipedia.org/wiki/Subject-matter_expertv

Lean Modeling: mit natürlicher Sprache zum Modell

Abnehmen, ohne zu hungern

Modellierungswerkzeuge und domänenspezifische Sprachen sind in den letzten Jahren in der Softwareentwicklung zunehmend populär geworden. Für die meisten Java-Entwicklungsumgebungen existiert heute eine Vielzahl von ausgereiften Werkzeugen, um Modelle oder Modellierungssprachen zu nutzen oder selbst zu erstellen. Leider werden diese mächtigen Technologien noch viel zu selten genutzt. Eine leichtgewichtige Alternative zu klassischen Modellierungswerkzeugen und DSL-Tools ist Lean Modeling. Dieser Ansatz verwendet die Ideen von Acceptance Test-driven Development, um natürlichsprachliche Spezifikationen in der Anwendungsentwicklung zu nutzen.

von Florian Heidenreich, Dr. Mirko Seifert, Christian Wende und Tobias Nestler

Ausgangspunkt ist die Grundidee, natürlichsprachliche Texte als Spezifikation zu nutzen. Diese entstand ursprünglich im Kontext von Acceptance Test-driven Development, mit dem Ziel, lesbare, wartbare und ausführbare Akzeptanztests mit natürlicher Sprache zu beschreiben. Anhand von verschiedenen Beispielen wird gezeigt, wie einfach Texte als Modelle interpretiert und verarbeitet werden können. Wir bezeichnen dieses Vorgehen als „Lean Modeling".

Die Hauptaufgabe jedes Softwareentwicklers besteht darin, die Wünsche bzw. Anforderungen eines Kunden in eine lauffähige Software zu verwandeln. Dazu müssen diese Anforderungen zunächst möglichst genau erfasst werden, um sicherzustellen, dass man ein einheitliches Verständnis davon hat, was am Ende des Entwicklungsprozesses entstehen soll. Die Mittel, um ein solches gemeinsames Verständnis zu erreichen, sind vielfältig. Beispielsweise können Workshops genutzt werden, um Anforderungen zu erfassen oder das Verhalten der Anwendung kann in Fachspezifikationen beschrieben werden. Genauso gut kann man in regelmäßigen Abständen dem Kunden die Software vorlegen, um die gemeinsamen Vorstellungen abzugleichen.

Unabhängig davon, welchen Weg man bevorzugt: Die Anforderungen an die Anwendung werden fast immer aus einer fachlichen Perspektive und meist in natürlicher Sprache erfasst. Dadurch ergeben sich bereits die ersten beiden zentralen Herausforderungen. Zum einen muss die fachliche Perspektive mit einer technischen in Einklang gebracht werden, zum anderen müssen die natürlichsprachlichen Beschreibungen in formale Spezifikationen (z. B. Programmcode) übersetzt werden. Diesen beiden Aufgaben widmet sich jeder Softwareentwickler täglich.

Um die Kommunikation zwischen Kunden, Fachexperten und Entwicklern zu vereinfachen, wurden in der Vergangenheit verschiedenste Ansätze ausprobiert und mit unterschiedlichem Erfolg angewendet. Beispielsweise wurden Entity-Relationship-Diagramme genutzt, um gemeinsam über das Datenmodell einer Anwendung zu beraten. UML-Diagramme wurden eingesetzt, um neben dem Datenmodell auch andere Aspekte (z. B. Use Cases, Verhaltensprotokolle, Zustandsmodelle) zu erfassen. Besondere Popularität haben in den letzten Jahren Ansätze mit domänenspezifischen Sprachen (DSLs) erfahren, die es erlauben, fachliche Anforderungen besonders domänennah zu erfassen. Das gemeinsame Ziel all dieser Modellierungstechniken war und ist es, spezielle Abstraktionen einzuführen, mit denen Anwendungsteile einfacher und genauer spezifiziert werden können. Dieses Ziel teilt der Ansatz, der in diesem Artikel vorgestellt wird. Allerdings setzt Lean Modeling die Schwerpunkte anders als bisherige Modellierungsmethoden (Kasten: „Die Ziele von Lean Modeling").

Abstraktion, Modellierung – Muss das sein?

Warum sind zusätzliche Abstraktionsebenen bei der Softwareentwicklung so erstrebenswert? Nun ja, man kann sicher jede Anwendung auch ausschließlich mithilfe von Programmcode beschreiben. Dem Verständnis der Anwendung ist das aber sicher nicht zuträglich. So ist beispielsweise das Datenmodell einer Applikation zwar vollständig im Programmcode enthalten, aber nicht direkt ersichtlich. Um den Überblick zu behalten, braucht selbst ein erfahrener Entwickler eine Möglichkeit, die

für das Datenmodell irrelevanten Teile des Codes auszublenden. Für Kunden und Fachexperten, die normalerweise nicht programmieren können und wollen, gilt dies noch viel mehr. Deshalb werden oft Fließtexte, Diagramme und Tabellen genutzt, um Informationen für alle an der Softwareentwicklung Beteiligten auf einem adäquaten Abstraktionsniveau darzustellen. Wir brauchen also ein gewisses Maß an Abstraktion, um die Komplexität zu beherrschen und um große Anwendungen zu verstehen. Modellierung bzw. Modellierungswerkzeuge stellen damit ein essenzielles Instrument für die Softwareentwicklung dar.

Werkzeuge – Der Stand der Technik

Je einfacher wir Modelle erstellen und verarbeiten können, desto effektiver können wir Abstraktionsebenen einführen und wechseln. Nun stellt sich die Frage, wie gut uns existierende Modellierungsansätze und -werkzeuge dabei unterstützen. Betrachtet man hier auf der einen Seite Werkzeuge mit fest definierten Modellierungssprachen (z. B. UML-Werkzeuge), so stellt man fest, dass man die vorgegebenen Sprachen zwar sofort nutzen kann, man bzgl. der Wahl der Abstraktionen aber stark eingeschränkt ist. Will man beispielsweise mit der UML Eingabeformulare modellieren, so muss man sich schon etwas verbiegen (z. B. Stereotypen benutzen).

Auf der anderen Seite gibt es Modellierungswerkzeuge, mit denen man eigene Sprachen entwickeln kann, d. h. die Abstraktion kann hier frei gewählt werden. Im Eclipse-Umfeld sind hier in den letzten Jahren viele Werkzeuge zur Entwicklung von textuellen und grafischen DSLs entstanden [1], [2], [3], [4]. Mit MPS steht zudem ein kommerzielles DSL-Werkzeug von JetBrains zur Verfügung [5]. Man kann mit all diesen Werkzeugen gezielt neue Abstraktionen einführen und hat dazu noch die Wahl, die neuen Konzepte (z. B. Formularfelder) grafisch oder in Form von Text auszudrücken.

Das klingt sehr komfortabel und einfach. Es stellt sich also die Frage, warum noch immer viele Softwareprojekte auf eigene DSLs und die schönen, komfortablen Editoren verzichten. Eine Ursache dafür ist unserer Erfahrung nach, dass die o. g. Werkzeuge nicht zugänglich genug oder zu schwerfällig sind.

Was muss man mit modernen DSL-Werkzeugen tun, um sich eine neue Sprache, d. h. eine neue Abstraktionsebene, zu schaffen? Zuallererst muss man sich natürlich mit den entsprechenden Werkzeugen und Konzepten vertraut machen. Hier betritt man meist unbekanntes Terrain. Bei DSL-Werkzeugen im Eclipse-Umfeld muss man sich mit der Entwicklung von Eclipse-Plug-ins befassen, was sicherlich kein triviales Thema ist. Auch bei der Nutzung von MPS muss man die gewohnte Entwicklungsumgebung erst einmal verlassen und sich in ein neues (Meta-)Werkzeug einarbeiten. Dieser Lernprozess betrifft natürlich nicht nur einen Entwickler im Team, sondern alle. Damit jeder von den neuen Abstraktionen profitieren kann, muss der Umgang mit DSLs leicht von der Hand gehen.

Kurzum, man steht vor einer substanziellen und teilweise sehr steilen Lernkurve. Hat man diese überwunden, was sicherlich erstrebenswert ist, ist man in der Lage, innerhalb relativ kurzer Zeit (z. B. weniger Tage) neue Sprachen zu erstellen. Wäre es aber nicht wunderbar, wenn die Einführung einer neuen Abstraktionsebene so schnell und so einfach wäre wie das Anlegen einer neuen Klasse?

Hoppla, die Lösung liegt vor unseren Füßen

Hier kommt das Stichwort „Acceptance Test-driven Development (ATDD)“ ins Spiel. Auf den ersten Blick scheint dieses Thema keinen direkten Bezug zur Modellierung zu haben. Schaut man aber genauer hin, so ergeben sich doch wichtige Parallelen.

Das zentrale Ziel von ATDD ist es, die Diskrepanz zwischen fachlichen Anforderungen (wie sie ein Kunde formulieren würde) auf der einen Seite und der technischen Überprüfung der Anforderungen (wie sie ein Entwickler in einem Test manifestiert) auf der anderen Seite zu überwinden. Damit widmet sich ATDD also dem oben angesprochenen Abstraktionsproblem.

Die Kernidee von ATDD ist es dabei, natürlichsprachliche Szenariobeschreibungen direkt zum Test von Anwendungen zu nutzen. ATDD nutzt demnach natürliche Sprache als sofort verständliches und natürliches Instrument, um fachspezifische Anforderungen darzustellen und gleichzeitig automatisiert auszuwerten.

Wenn man mit ATDD-Werkzeugen Dokumente in natürlicher Sprache einsetzen kann, um ausführbare Tests zu beschreiben, warum sollte man nicht das gleiche Prinzip bei der Implementierung von Software nutzen?

Probieren wir es doch einfach mal aus!

Ein zentraler Bestandteil jeder Anwendung ist das Datenmodell. Es besteht aus den Entitäten, Eigenschaften und Beziehungen der Anwendungsdomäne und wird klassischerweise mithilfe von ER-Diagrammen, SQL-Schemata oder UML-Klassendiagrammen modelliert. Würden wir

Die Ziele von Lean Modeling

- Geringere Einstiegshürde im Vergleich zu klassischen Modellierungsansätzen (keine Vorkenntnisse nötig)
- Einfache Zusammenarbeit mit Fachexperten durch Verwendung von natürlicher Sprache (keine technischen DSLs)
- Direkte Verwendung der Modelle für die Implementierung steigert die Entwicklungseffizienz (Codegenerierung oder Interpretation der Texte)
- Leicht zugängliche Methode zur Softwarespezifikation (eine Methode pro Satz, Matching über einfache Muster mit Platzhaltern)
- Spezifikation von Tests, Datenmodellen, Geschäftsregeln u.v.m. mit einem einheitlichen Mechanismus
- Extrem kurze Feedbackschleife bei der Entwicklung von Modellierungssprachen (keine Codegenerierung für die DSL selbst)

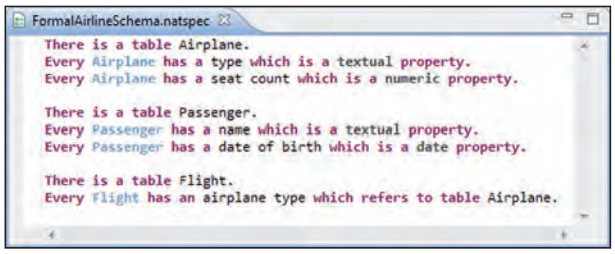


Abb. 1: Datenschema im NatSpec-Editor

natürliche Sprache (z. B. Englisch) nutzen wollen, um ein Datenschema zu beschreiben, so könnte das für eine Anwendung im Kontext einer Fluggesellschaft beispielsweise aussehen wie in **Abbildung 1** dargestellt.

Eine solche Beschreibung des Datenschemas klingt zwar noch etwas mechanisch, ist aber sowohl für Menschen als auch für Maschinen verständlich.

Wenn wir die Beschreibung als (textuelles) Modell betrachten, möchten wir daraus natürlich automatisch die SQL-Befehle zur Erzeugung des Schemas ableiten, genau wie es uns DSL-Werkzeuge erlauben. Dazu bedienen wir uns der prinzipiellen Funktionalität von ATDD-Werkzeugen: Wir bilden die einzelnen Sätze des Dokuments auf Code, d. h. auf einzelne Methoden ab. Beispielsweise ordnen wir dem Satz „There is a table Airplane." eine Methode zu, die eine Tabelle erzeugt. Analog dazu wird der nächsten Zeile eine Methode zugeordnet, die eine Spalte vom Typ *VARCHAR* anlegt.

Listing 1

```
@TextSyntax("There is a table #1.")
public Table createTable(String name) {
  Table newTable = new Table(name);
  allTables.add(newTable);
  return newTable;
}
```

Listing 2

```
@TextSyntax("Every #1 has a #2 which is a #3 property.")
public void addColumn(Table table, List<String> nameParts, String
typeName) {
  String columnName = getColumnName(nameParts);
  String type = getType(typeName);
  Column newField = new Column(columnName, type);
  table.addColumn(newField);
}

@TextSyntax("Every #1 has an #2 which refers to table #3.")
public void addReference(Table table, List<String> nameParts,
Table otherTable) {

  String columnName = getColumnName(nameParts);
  Column newColumn = new Column(columnName, "INT");
  table.addColumn(newColumn);
}
```

Praktisch umgesetzt wird diese Zuordnung zwischen Sätzen und Methoden bei den meisten ATDD-Werkzeugen (z. B. bei Cucumber [6]) über spezielle Annotationen an den Methoden, die angeben, welchem Satz bzw. welchen Satzmustern die Methode entspricht. Für die Beispiele in diesem Artikel wird exemplarisch das Werkzeug NatSpec [7] verwendet, da es im Gegensatz zu herkömmlichen ATDD-Werkzeugen keine Einschränkungen bzgl. der möglichen Sätze aufweist und damit auch für den Einsatz außerhalb des Testens bestens geeignet ist.

NatSpec verwendet die *@TextSyntax*-Annotation zur Verknüpfung von Methoden und Sätzen. Um beispielsweise den ersten Satz aus **Abbildung 1** mit der Methode *createTable()* zu verknüpfen, genügt der Code aus Listing 1.

Die Zeichenkette innerhalb der *@TextSyntax*-Annotation gibt an, welchen Sätzen die Methode zugeordnet wird. Platzhalter (z. B. *#1*) können verwendet werden, um die Methode mit Wörtern aus Sätzen zu parametrisieren. So kann *createTable()* zum Erzeugen verschiedener Tabellen verwendet werden. NatSpec verwendet hier einfache Platzhalter, die lediglich den Index des Parameters angeben. Andere ATDD-Werkzeuge setzen hier auf reguläre Ausdrücke, um die Menge der erlaubten Wörter noch stärker einzuschränken. Diese Satzmuster stellen das Gegenstück zu Grammatiken dar, die bei Parser-basierten DSL-Werkzeugen zum Einsatz kommen.

Innerhalb der *createTable()*-Methode wird ein neues Objekt vom Typ *Table* angelegt. Dieses Objekt repräsentiert eine Tabelle in dem Datenschema, das wir spezifizieren möchten. Für die anderen Sätze aus **Abbildung 1** lassen sich ebenfalls Methoden mit entsprechenden Annotationen definieren. Diese sind in Listing 2 zu sehen.

Auch hier werden Platzhalter verwendet, um mit zwei Methoden insgesamt vier Sätze aus **Abbildung 1** abzudecken. Eine Besonderheit stellen hier die Parameter vom Typ *List<String>* dar, da diese sich mehreren Wörtern im Satz zuordnen lassen. So können beispielsweise die Namen der Spalten im Datenmodell (z. B. „seat count") im Text normal beschrieben werden und dann später, d. h. im Code, in die SQL-Schreibweise (*seat_count*) umgewandelt werden.

Das war ja einfach, aber reicht das aus?

Dieses erste Beispiel zeigt die zentrale Idee und das prinzipielle Vorgehen bei Lean Modeling. Man kann leicht sehen, dass Standard-Java-Wissen ausreicht, um zu verstehen, wie das Ganze funktioniert. Jeder Entwickler, der schon einmal Annotationen benutzt hat, kann entsprechende Methoden schreiben. Nicht einmal mit der Syntax für reguläre Ausdrücke muss man vertraut sein, um die Satzmuster zu definieren. Hier kommen wir dem Ziel von Lean Modeling, die Einstiegshürde so gering wie möglich zu halten, schon sehr nahe (Kasten: „Die Ziele von Lean Modeling").

Gleichzeitig wirft das Beispiel aber auch eine Menge Fragen auf: Was haben wir hier eigentlich getan? In erster Linie haben wir eine Menge relativ eintöniger Satz-

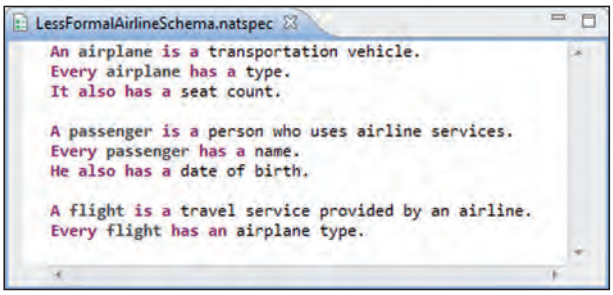


Abb. 2: Alternatives Datenschema im NatSpec-Editor

muster auf Java-Methoden abgebildet, die wiederum Objekte erzeugen. Welchen Vorteil bringt es, Texte zu schreiben, die so ähnlich wie Programmcode klingen? Welche Texte können überhaupt auf Methoden abgebildet werden? In welchem Verhältnis steht dieses Vorgehen zu anderen DSL-Werkzeugen?

Obwohl das Datenschema aus **Abbildung 1** sicherlich für Menschen lesbar ist, die mit den Konzepten relationaler Datenbanken (Tabelle, Spalte, Typ, Fremdschlüssel) vertraut sind, so mutet der Text doch noch sehr starr an. Alternativ könnte man das Schema auch wie in **Abbildung 2** dargestellt beschreiben.

Dieser Text klingt schon wesentlich weniger formal. Der Text enthält keine Typinformationen für die Eigenschaften (z. B. *seat count* oder *name*), dafür aber eine Beschreibung der einzelnen Tabellen. Auch dieser Text lässt sich mithilfe der o. g. Annotationen auf drei Methoden abbilden. Diese sind in verkürzter Form in Listing 3 zu sehen, die vollständige Implementierung kann von GitHub [8] bezogen werden.

Diese drei Methoden funktionieren ähnlich wie die aus Listing 1 und 2 und definieren Satzmuster, die alle sieben Sätze aus **Abbildung 2** abdecken. Allerdings bedient sich die zweite *createColumn()*-Methode einer besonderen Funktionalität von NatSpec. In der *@TextSyntax*-Annotation ist nur der erste Parameter (*nameParts*) mit einem Platzhalter versehen. Woher kommt also der Wert für den zweiten Parameter (*table*), wenn nicht aus dem Satz?

Hier wird der Kontext des Satzes genutzt, um einen Wert für den Parameter zu finden. Dieser Kontext besteht aus allen Sätzen vor dem aktuellen Satz, d. h. allen Aussagen, die zuvor gemacht worden. Wird durch einen vorhergehenden Satz ein Objekt vom Typ *Table* erzeugt, so kann dieses Objekt als Argument für die Methode *createColumn()* benutzt werden. Das erscheint auf den ersten Blick etwas ungewöhnlich, ist aber tatsächlich ein elementares Konzept von natürlichen (und auch von formalen) Sprachen. Wir beziehen uns bei unseren Aussagen (z. B. „Es war einfach grandios“) sehr oft auf zuvor Gesagtes (z. B. „Gestern war ich beim Konzert von Spinal Tap“). Bei Programmiersprachen entspricht das in etwa dem Scoping bei der Auflösung von Namen. Durch den Kontextbezug erspart man sich unnötige Referenzen und kann so Sachverhalte kompakter und natürlicher ausdrücken.

Listing 3

```
@TextSyntax("A #1 is a #2")
public Table createTable(String name, List<String> description) {...}

@TextSyntax("Every #1 has a #2")
public void createColumn(String tableName, List<String> nameParts) {...}

@TextSyntax("It also has a #1")
public void createColumn(List<String> nameParts, Table table) {...}
```

Neben der Nutzung des Kontexts für die Bestimmung zusätzlicher Parameter nutzt das zweite Beispiel auch Synonyme, um mit der Methode *createTable()* sowohl Sätze, die mit „A“ als auch Sätze, die mit „An“ beginnen, zu verarbeiten. Synonyme können bei NatSpec frei definiert werden und sind insbesondere nützlich, um Wörter, die sowohl in der Einzahl als auch in der Mehrzahl vorkommen, abzudecken.

Insgesamt zeigt dieses zweite Beispiel, dass man bzgl. der Wahl der Sätze sehr frei ist. Das gleiche Datenschema lässt sich auf verschiedene Weise ausdrücken. Man kann zum einen sehr generische Satzmuster definieren, die auf viele Sätze passen. Dadurch ergibt sich dann eine gleichförmigere, aber auch monotonere Beschreibung des Schemas. Man kann aber auch sehr spezielle Satzmuster nutzen, die im Extremfall sogar nur auf einen Satz passen. Auch hat man die Wahl, ob man Informationen über das Datenschema im Text oder in den dazugehörigen Methoden ablegt. So wurden die Typen der einzelnen Felder in der ersten Variante im Text spezifiziert, während die zweite Variante diese Information im Code hinterlegt.

Hier wird schnell deutlich, dass die Gestaltung der textuellen Spezifikationen viel Spielraum lässt. Dies ist ähnlich wie bei „klassischen“ DSL-Werkzeugen, bei denen man ja ebenfalls in der Gestaltung der Grammatik frei ist und letztendlich selbst entscheiden muss, in welcher Art die Inhalte der jeweiligen Sprache repräsentiert werden sollen. Hier gilt es, ein gutes Gespür für die richtigen Satzmuster zu entwickeln.

Jetzt schreiben wir doch wieder eine Grammatik?

Offensichtlich gibt es demnach eine Beziehung zwischen den Grammatiken der textuellen DSL-Tools und den Satzmustern, die wir in den o. g. Beispielen definiert haben. Beide Mechanismen dienen dazu, die gültigen Sätze der Sprache zu definieren. Allerdings gibt es hier auch entscheidende Unterschiede: Zum einen generieren die meisten DSL-Tools aus den Grammatiken Parser und Editoren. Die Annotationen werden dagegen interpretiert, und es wird ein und derselbe Editor für alle Dokumente verwendet. Der Hauptvorteil der Interpretation besteht darin, dass man die Sprache ändern kann (z. B. durch Änderung einer Annotation) und sofort sieht, ob die existierenden Texte der neuen Sprache genügen. Im Fall von NatSpec kann man dies leicht am Syntax-Highlighting erkennen. Nur Sätze, die einer Methode

zugeordnet werden können, erscheinen im Editor hervorgehoben. So verkürzt sich die Feedback-Schleife bei der Entwicklung einer Sprache drastisch. Beispielsweise muss man bei der Nutzung von Eclipse keine zweite Eclipse-Instanz starten; die Sprache wird im selben Eclipse-Workspace definiert und auch genutzt.

Weiterhin muss man Sprachen nicht ausgehend von einer Grammatik entwickeln, sondern kann auch von Beispielsätzen ausgehen. So kann man einen neuen Satz in ein Dokument einfügen und sich dafür eine neue, passende Methode mit der entsprechenden *@TextSyntax*-Annotation automatisch per Quick Fix erzeugen lassen. Dieses beispielgetriebene Vorgehen ist sehr intuitiv und pragmatisch, wird von DSL-Werkzeugen, die einen Parser generieren, aber nicht unterstützt.

Ein weiterer Unterschied, der insbesondere den Freunden des Compilerbaus auffallen sollte, ist das Fehlen von rekursiven Aufrufen in den Satzmustern. Will man verschachtelte Strukturen aufbauen, so ist dies jedoch über den Kontext der Sätze möglich.

Wo bleibt der Codegenerator?

Um aus den textuellen Modellen Java-Code oder auch andere Artefakte zu erzeugen, wird natürlich ein Codegenerator benötigt. Bei normalen DSL-Werkzeugen verwendet man üblicherweise eine Templatesprache, um über das Modell zu traversieren und entsprechenden Text zu erzeugen. Das funktioniert natürlich auch mit NatSpec. Die Methoden aus Listing 1, 2 und 3 erzeugen ja bereits ein Modell. Dieses kann mit einer beliebigen Templatetechnologie in Code verwandelt und im Workspace abgelegt werden. Für unser Beispiel verwenden wir der Einfachheit halber einen normalen *StringBuilder*, der ausgehend von einer *main()*-Methode den Code für unser Datenmodell erzeugt und im Workspace abspeichert. Für die Codegenerierung unterscheidet sich das Vorgehen also nicht wesentlich von bekannten DSL-Tools.

Ein Datenmodell für SQL-Tabellen – Ist das alles?

Die bisherigen Beispiele waren sehr einfach gewählt, um die prinzipiellen Ideen hinter Lean Modeling darzustellen. Natürlich lassen sich auch andere Anwendungsfälle mit dem gleichen Vorgehen adressieren. So ist beispielsweise die Definition von Formularen im User Interface neben dem Datenschema eine wichtige Aufgabe während der Entwicklung. Als Beispielanwendung möchten wir eine Bestellapplikation für Android Devices betrachten. Sie ist in **Abbildung 3** zu sehen.

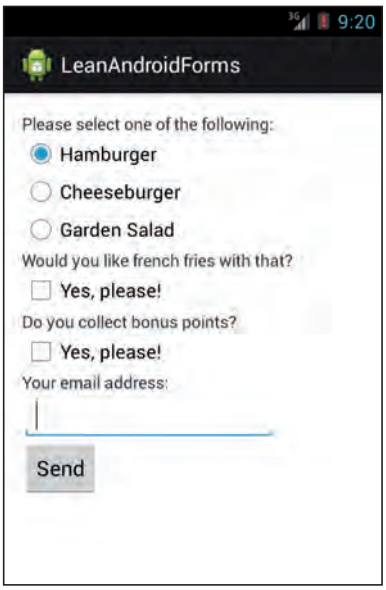


Abb. 3: Beispielanwendung im Android-Simulator

Für die Entwicklung einer solchen Oberfläche können wir ebenfalls textuelle Spezifikationen einsetzen. Ein Beispiel ist in **Abbildung 4** zu sehen.

Der letzte Satz stellt die primitivste Art, ein Formularelement zu beschreiben, dar. Er enthält keinerlei Parameter und wird auf ein einfaches Eingabefeld für E-Mail-Adressen abgebildet. Trotzdem ist es sinnvoll, Formularelemente auf diese Weise zu spezifizieren, da der Satz für einen Fachexperten natürlich zugänglicher und verständlicher ist als der entsprechende Android-XML-Code.

Der zweite und dritte Satz sind parametrisierte Formularelemente, die wir jeweils auf ein Textfeld und eine Checkbox abbilden. Der Text für das Textfeld ist dabei variabel. Das User Interface kann also parametrisiert werden, und der zugehörige Implementierungscode wird automatisch durch die Codegenerierung erzeugt. Darüber hinaus sind die beiden Sätze wesentlich einfacher zu verstehen als die dazugehörigen XML-Fragmente.

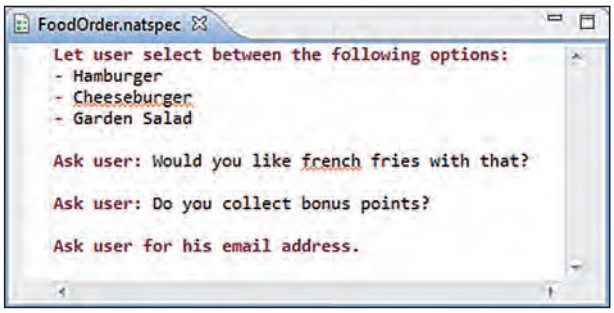


Abb. 4: Formularbeschreibung im NatSpec-Editor

Der erste Satz und die Anstriche danach stellen das komplexeste Formularelement in unserem Beispiel dar. Hier wird eine Auswahl von verschiedenen Optionen definiert, die später auf eine *RadioBoxGroup* abgebildet wird. Es wird hier sehr deutlich, dass man mit derartigen textuellen Regeln auf einem viel höheren Abstraktionsniveau als mit Sourcecode arbeiten kann.

Die Methoden zum ersten Satz sind in Listing 4 zu sehen. Es werden zwei Methoden benötigt. Die erste (*addOptionSet()*) legt ein neues Set von Optionen an. Danach können mithilfe von *addOption()* weitere Optionen in das *OptionSet* aufgenommen werden. Auch diese Methode bedient sich des Kontexts, um das *OptionSet* zu bestimmen, zu dem die neue Option hinzugefügt werden soll.

Was geht noch so?

Mit textuellen Beschreibungen lassen sich natürlich auch andere Bestandteile von Applikationen spezifizieren. Zustandsmodelle, Protokollbeschreibungen, Prozessabläufe u.v.m. sind denkbar. Auch Varianten der Applikation, wie sie bei einer Produktlinie existieren, können beschrieben werden. Für die technische Dokumentation von Anwendungen, insbesondere ihrer Komponenten und Schnittstellen lassen sich textuelle, natürlichsprachliche Modelle ebenfalls einsetzen. Ob und wann der Einsatz einer textuellen Modellierung wirklich sinnvoll ist, hängt natürlich inhärent mit der Frage nach dem Wert einer abstrakteren Darstellung zusammen. Lässt sich ein Teil einer Anwendung mit einem textuellen Modell wesentlich kürzer und kompakter spezifizieren, so ist das ein erster Indikator für einen gewinnbringenden Einsatzbereich.

Listing 4

```
@TextSyntax("Let user select between the following options:")
public OptionSet addOptionSet() {
  String text = "Please select one of the following:";
  OptionSet optionSet = new OptionSet(nextID++, text);
  elements.add(optionSet);
  return optionSet;
  }

@TextSyntax("- #1")
public void addOption(List<String> parts, OptionSet optionSet) {
  String text = new StringUtils().explode(parts, " ");
  Option option = new Option(nextID++, text);
  optionSet.addOption(option);
}
```

Fazit

Die zentrale Idee des Lean Modeling besteht darin, einfache Texte bzw. Sätze auf Java-Methoden abzubilden. Dieser Ansatz, der aus der ATDD-Community stammt, erlaubt es, ohne zusätzliche Frameworks und in der gewohnten Entwicklungsumgebung Texte zum Modellieren und Spezifizieren von Anwendungen zu nutzen. Setzt man geeignete Werkzeuge ein, muss man zudem nicht auf den Komfort klassischer DSL-Workbenches, z. B. Syntax-Highlighting, Code-Completion oder Hyperlinks, verzichten.

Das im Artikel vorgestellte Werkzeug NatSpec wird bei DevBoost intensiv in Kundenprojekten eingesetzt, um mehr Erfahrungen mit diesem Modellierungsansatz zu sammeln. Für den Einsatz zum Testen von Anwendungen hat sich die Vorgehensweise bereits etabliert und wird auch schon seit Längerem in anderen Unternehmen erfolgreich eingesetzt. Momentan wird auf verschiedenen Gebieten damit experimentiert, neue Einsatzbereiche außerhalb des Testens zu erschließen. Unter anderem wird NatSpec genutzt, um Datenbankschemata und Schnittstellen zu beschreiben oder auch, um Anforderungen strukturiert zu erfassen. Später wird es auch möglich sein, Word-Dokumente ähnlich wie im NatSpec-Editor zu bearbeiten. So könnten Fachexperten mit einem vertrauten Editor arbeiten, aber dennoch Dokumente erstellen, die direkt für die Entwicklung weiterverwendet werden können.



Florian Heidenreich, Dr. Mirko Seifert und **Christian Wende** entwickeln neben NatSpec die Open-Source-Tools EMFText, JaMoPP und JUnitLoop. Ihr Unternehmen, die DevBoost GmbH, bietet Produkte und Dienstleistungen zur Effizienz- und Qualitätssteigerung in der Softwareentwicklung an.

www.devboost.de

Tobias Nestler arbeitet für die SAP AG und interessiert sich insbesondere für Werkzeuge zur Erfassung von Anforderungen und Testautomatisierung.

Links & Literatur

[1] http://www.eclipse.org/Xtext/

[2] http://www.emftext.org

[3] http://www.eclipse.org/modeling/gmp/

[4] http://www.eclipse.org/graphiti/

[5] http://www.jetbrains.com/mps/

[6] http://cukes.info

[7] http://www.nat-spec.com

[8] https://github.com/DevBoost/JavaMagazin_Lean_Modeling_Example

JSR 352 – Batch Applications for the Java Platform

Ein Standard für die Batchentwicklung

Die Batchverarbeitung gehört zu den ältesten Verarbeitungsformen in der IT überhaupt, und doch gab es im Java-Bereich bisher keinen Standard dafür. Das hat sich nun mit dem finalen Release des Java Specification Requests 352 (Batch Applications for the Java Platform, JSR 352) geändert. Dieser zielt nicht nur auf die Enterprise Edition und ist dort Teil von Java EE 7, sondern explizit auch auf die Standard Edition. Eine erste Implementierung gibt es bereits im GlassFish 4, weitere werden bald folgen. Höchste Zeit, sich die Spezifikation genauer anzusehen.

von Tobias Flohre

Der Java Community Process (JCP) dient dazu, Standards im Java-Bereich zu etablieren. Jeder kann daran teilnehmen [1]. Wenn jemand einen neuen Standard etablieren möchte, reicht er einen JSR ein. Das Executive Committee, bestehend aus Firmen und Einzelpersonen [2], entscheidet dann, ob der JSR generell zugelassen wird. Wenn das der Fall ist, wird eine Expertengruppe gegründet, die den Standard ausarbeitet. Am Ende entscheidet das Executive Committee, ob der JSR in der ausgearbeiteten Fassung akzeptiert wird.

Der JSR 352 enthält die Standardisierung der Batchverarbeitung im Java-Bereich [3]. Eingereicht wurde er im Oktober 2011 von IBM und dann auch angenommen. Chris Vignola wurde Specification Lead. In der Expertengruppe sind IBM, Credit Suisse, Oracle, VMware und Red Hat sowie Simon Martinelli und Michael Minella als Einzelpersonen vertreten, wobei Michael Minella seit etwa einem Jahr Project Lead von Spring Batch ist. Seit Mai dieses Jahres ist die Spezifikation nun final und Teil von Java EE 7. Obwohl IBM einen starken Enterprise-Hintergrund hat, ist das Ziel des JSRs allerdings nicht nur die Java Enterprise Edition, sondern explizit auch die Standard Edition.

Das Ergebnis orientiert sich stark an Spring Batch. So wurden die Begrifflichkeiten der Domäne und das Programmiermodell fast eins zu eins übernommen, und auch die Konfiguration eines Jobs in XML ähnelt einer Spring-Batch-Konfiguration stark. Der Ablauf eines Jobs ist ebenfalls identisch.

Spring Batch wird die Spezifikation in der Version 3.0 (angekündigt für Herbst/Winter 2013) implementieren. Bisher gibt es bereits eine Implementierung im GlassFish 4 Application Server. Die Spezifikation schreibt übrigens keine bestimmte Art der Dependency Injection vor, sodass eine Implementierung einerseits CDI, aber andererseits auch Spring DI verwenden kann.

Ganz pragmatisch werde ich zunächst einen einfachen Job vorstellen, der nichts anderes macht, als Daten auf die Konsole zu loggen, und die entsprechenden JSR-352-Artefakte bei ihrem Vorkommen erläutert.

Die Job-XML

Die Job-XML beschreibt den Ablauf unseres Batchjobs (Listing 1). Dabei kann ein Job mehrere Steps enthalten. Über das Attribut *next* kann dabei die Reihenfolge der Steps bestimmt werden. Es gibt *Batchlet*- und *Chunk*-basierte Steps, wobei die *Batchlet*-basierte Verarbeitung alle Freiheiten lässt. Das referenzierte Batchartefakt (*myBatchlet* in Listing 1) muss nur das Interface *javax.batch.api.Batchlet* implementieren, und bei Start des Steps wird einmal die dort vorhandene Methode *process* aufgerufen. Diese Art der Verarbeitung eignet sich für vorbereitende oder abschließende Tätigkeiten wie das Kopieren und Löschen von Dateien oder für die Anbindung von Legacy-Code, der nicht *Chunk*-basiert ausgeführt werden kann.

Sobald es um die Verarbeitung einer großen Menge von Datensätzen geht, ist die *Chunk*-basierte Verarbeitung die richtige Wahl. Ein Datensatz wird dabei als *Item* bezeichnet. Das Attribut *item-count* bestimmt die Anzahl der Datensätze, die innerhalb einer Transaktion

Artikelserie

Teil 1: Java Config, Spring-Data-Support und Co.
Teil 2: JSR 352 – Batch Applications for the Java Platform
Teil 3: Spring XD – Ordnung für Big-Data-Datenströme

verarbeitet werden sollen, und die Menge dieser Datensätze wird als *Chunk* bezeichnet. Drei Komponenten betreiben die Verarbeitung der Datensätze: *Reader*, *Processor* und *Writer*.

Dependency Injection und Scoping für Batchartefakte

Wie schon erwähnt, schreibt die Spezifikation nicht vor, auf welche Art und Weise die referenzierten Komponenten erzeugt werden. Während es im Java-EE-Umfeld vermutlich meistens per CDI geregelt werden wird, wird Spring Batch sicherlich weiterhin Spring DI verwenden. Die in diesem Artikel gezeigten Beispiele zeigen Komponenten für einen CDI-Container.

Im Gegensatz zur DI schreibt die Spezifikation allerdings Scopes für Batchartefakte vor. Alle innerhalb eines *<job />*-Tags referenzierten Artefakte haben Job-Scope, werden also für jeden Joblauf neu erzeugt. Alle innerhalb eines *<step />*-Tags referenzierten Artefakte haben Step Scope, werden also für den Step erzeugt und danach wieder entfernt. Das sind insbesondere Reader, Processor und Writer, und im Beispiel in Listing 1 sind das auch alle referenzierten Artefakte. Zusätzlich gibt es noch den Scope Partition Step für die parallelisierte Verarbeitung.

Reader, Processor und Writer

Ein Reader ist dafür verantwortlich, Datensätze einzulesen. Dafür implementiert er das Interface *javax.batch.api.chunk.ItemReader*, dessen wichtigste Methode die *readItem*-Methode ist. Diese wird vom Framework aufgerufen, um jeweils ein Item zu lesen. Der Reader signalisiert mit der Rückgabe von *null*, dass alle Items gelesen wurden und der Step abgeschlossen werden kann. Listing 2 zeigt eine Implementierung, die eine feste Anzahl von Items zurückgibt.

Im Processor werden die eingelesenen Items nun beliebig fachlich verarbeitet. Der Processor muss dafür das Interface *javax.batch.api.chunk.ItemProcessor* mit der Methode *processItem* implementieren (Listing 3). Das Framework übergibt das eingelesene Item an diese Methode, die ein beliebiges Objekt als Ergebnis der Verarbeitung zurückgeben kann. Gibt sie *null* zurück, so wird das Item herausgefiltert und nicht an den Writer gegeben.

Der Writer schreibt die verarbeiteten Daten in eine beliebige Ressource. Er implementiert das Interface *javax.batch.api.chunk.ItemWriter*, dessen wichtigste Methode *writeItems* ist. Listing 4 zeigt ein Beispiel, das alle zu schreibenden Items loggt.

Ablauf einer Chunk-basierten Verarbeitung

Der Writer erhält im Gegensatz zum Reader und Processor eine Liste von Items. Wie kommt es dazu? **Abbildung 1** zeigt die Verarbeitung eines Chunk-orientierten Steps. Sie stammt direkt aus der Spezifikation. Es werden immer abwechselnd *ItemReader* und *ItemProcessor* aufgerufen, bis *item-count*-Datensätze gelesen und verarbeitet wurden. Die verarbeiteten Datensätze werden

Listing 1

```
<job id="simpleJob">
  <step id="batchletStep" next="chunkStep">
    <batchlet ref="myBatchlet"/>
  </step>
  <step id="chunkStep">
    <chunk item-count="2">
      <reader ref="dummyItemReader"/>
      <processor ref="logItemProcessor"/>
      <writer ref="logItemWriter"/>
    </chunk>
  </step>
</job>
```

Listing 2

```
@Named
public class DummyItemReader extends AbstractItemReader {

  private String[] input = {"1","2","3","4","5",null};
  private int index = 0;

  @Override
  public Object readItem() throws Exception {
    return input[index++];
  }

}
```

Listing 3

```
@Named
public class LogItemProcessor implements ItemProcessor {

  private static final Logger log = Logger.getLogger(LogItemProcessor.class);

  @Override
  public Object processItem(Object item) throws Exception {
    log.info("ItemProcessor: "+item);
    return item;
  }

}
```

Listing 4

```
@Named
public class LogItemWriter extends AbstractItemWriter {

  private static final Logger log = Logger.getLogger(LogItemWriter.class);

  @Override
  public void writeItems(List<Object> items) throws Exception {
    log.info("ItemWriter: "+items.toString());
  }

}
```

nun gesammelt an den *ItemWriter* übergeben, damit dieser sie batchoptimiert schreiben kann. Danach wird ein Commit für die Transaktion ausgelöst, eine neue Transaktion eröffnet und erneut gelesen und verarbeitet.

Start eines Jobs

Zur Interaktion mit einer Job-Runtime definiert die Spezifikation das Interface *javax.batch.operations.JobOperator*. Mithilfe von Implementierungen dieses Interface können Jobs gestartet und gestoppt werden. Zusätzlich definiert die Spezifikation die Laufzeitobjekte *javax.batch.runtime.JobInstance, javax.batch.runtime.JobExecution* und *javax.batch.runtime.StepExecution*, die Daten zu bestimmten Läufen halten. Das Verhältnis von Job, JobInstance und JobExecution ist in **Abbildung 2** dargestellt: Während ein Job den fachlichen Typ einer Verarbeitung darstellt, wird eine JobInstance für jeden geplanten Lauf erzeugt. Eine JobExecution stellt dann die tatsächliche Ausführung dar, von der es in der Regel genau eine pro JobInstance gibt. Falls aber eine JobExecution fehlschlägt und der geplante Lauf erneut gestartet wird, wird eine weitere JobExecution zur JobInstance erzeugt.

Der *JobOperator* kann über die Factory-Klasse *javax.batch.runtime.BatchRuntime* ermittelt werden. Der folgende Codeausschnitt zeigt den Start unseres Beispieljobs. Dabei ist *simpleJob* der Name der Job-XML-Datei ohne Dateityperweiterung:

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
Properties props = new Properties();
jobOperator.start("simpleJob", props);
```

Jobparameter und Properties

Der JSR 352 führt ein interessantes Property-Konzept ein. Praktisch auf jeder Ebene der Job-XML können Properties definiert werden, die dann im Folgenden zur Ersetzung in weiteren Property-Definitionen verwendet werden können, wenn sich diese in der gleichen Hierarchie befinden. Listing 5, direkt aus der Spezifikation, zeigt, wie das geht. Die Property *infile.name* wurde hier zu *postings.txt* aufgelöst.

Der Zugriff auf die Property in Batchartefakten (in diesem Fall sinnvollerweise ein *ItemReader*) erfolgt über Injizieren per Annotation:

```
@Inject @BatchProperty(name="infile.name")
String fileName;
```

Insgesamt definiert die Spezifikation noch weitere Quellen für Properties:

- *#{jobParameters['xxx']}*: Zugriff auf Jobparameter, die beim Start des Jobs mitgegeben wurden.
- *#{systemProperties['xxx']}*: Zugriff auf System-Properties.
- *#{partitionPlan['xxx']}*: Bei paralleler Verarbeitung Zugriff auf Daten, die speziell für diesen Thread erstellt wurden.

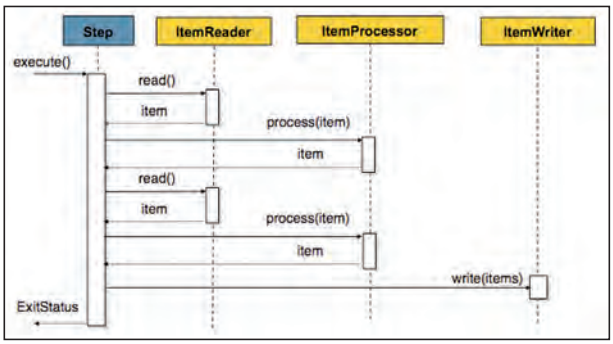


Abb. 1: Chunk-orientierte Verarbeitung

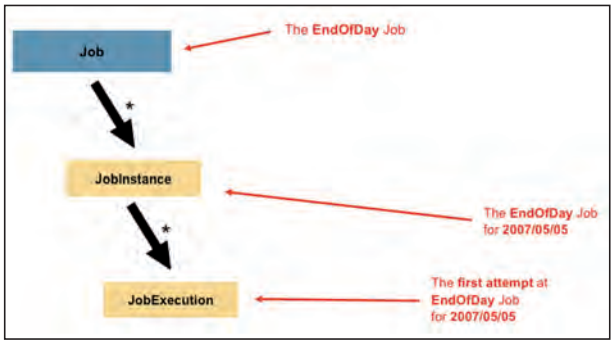


Abb. 2: Job, JobInstance und JobExecution

Restartfähigkeit

Ein wichtiges Feature bei der Batchverarbeitung ist die Wiederaufsetzbarkeit von fehlgeschlagenen Jobs. In der Regel sind die Daten dann zum Teil verarbeitet, und ein kompletter Neustart kommt nicht in Frage, da dann Daten doppelt verarbeitet werden. Die Komponenten, die beim Restart besonders aktiv werden müssen, sind *ItemReader* und *ItemWriter*. Dafür besitzen sie noch weitere Methoden (Listing 6).

Klar wird das am besten an einem konkreten Beispiel: Wir haben einen *ItemReader*, der Datensätze anhand eines SQL-Statements über einen *CURSOR* aus der Datenbank liest. Das SQL-Statement definiert eine feste Reihenfolge (*ORDER BY*). Die Methode *checkpointInfo* wird immer kurz vor dem Commit einer Chunk-Transaktion aufgerufen. Wir geben in diesem Fall die Anzahl der schon gelesenen *Items* als *Checkpoint* zurück. Die Methode *open* wird beim Start des Steps einmal aufgerufen. Im Restartfall wird ihr der in der letzten erfolgreichen Transaktion ermittelte *Checkpoint* über-

Listing 5

```
<job id="job1">
  <properties>
    <property name="filestem" value="postings"/>
  </properties>
  <step id="step1">
    <chunk>
      <properties>
        <property name="infile.name" value="#{jobProperties['filestem']}.txt"/>
      </properties>
    </chunk>
  </step>
</job>
```

geben. In unserem Beispiel wäre das also die Zahl der schon gelesenen *Items*. In der *open*-Methode wird nun das Statement ausgeführt, der *CURSOR* geöffnet, und falls es sich um einen Restart handelt und ein *Checkpoint* übergeben wurde, werden entsprechend viele *Items* übersprungen, damit die Verarbeitung mit noch nicht verarbeiteten *Items* beginnen kann.

Restartfähigkeit hängt also immer an der Implementierung des *ItemReaders* bzw. *ItemWriters*, es kann sie nicht generisch Out of the Box geben.

Skip und Retry

Im Normalfall schlägt ein Batchlauf fehl, sobald eine *Exception* durch die Batchartefakte geworfen wird. Mit der Skip-Funktionalität können bestimmte *Exceptions* als *skippable* markiert werden, sodass der Batchlauf nicht abbricht, sondern das *Item* überspringt. Die Maximalanzahl der zu überspringenden *Items* wird im Attribut *skip-limit* auf dem Tag *chunk* definiert:

```
<chunk item-count="2" skip-limit="25">
  <skippable-exception-classes>
    <include class="java.lang.Exception" />
    <exclude class="java.io.IOException" />
  </skippable-exception-classes>
</chunk>
```

Mit der Retry-Funktionalität können *Exceptions* als *retryable* markiert werden, sodass der Batchlauf nicht direkt abbricht, sondern zunächst versucht wird, die Verarbeitung erneut durchzuführen. Die Maximalanzahl der erneuten Versuche wird im Attribut *retry-limit* auf dem Tag *chunk* definiert:

```
<chunk item-count="2" retry-limit="25">
  <retryable-exception-classes>
    <include class="java.lang.Exception" />
    <exclude class="java.io.IOException" />
  </retryable-exception-classes>
</chunk>
```

Listener

Der JSR 352 definiert eine Reihe von Listenern, die auf bestimmte Ereignisse im Lebenszyklus eines Jobs reagieren. Diese können auf Job- und auf Step-Ebene registriert werden (Listing 7).

Auf Jobebene kann der *JobListener* registriert werden, der mit den beiden Methoden *beforeJob* und *afterJob* entsprechend bei Start bzw. Ende des Joblaufs aufgerufen wird. Auf Step-Ebene gibt es folgende Listener:

- Der *StepListener* hat die beiden Methoden *beforeStep* und *afterStep*.
- Der *ChunkListener* hat die Methoden *beforeChunk*, *onError* und *afterChunk*. Dieser Listener bezieht sich auf den Chunk, also die Menge der in einer Transaktion verarbeiteten Datensätze. Alle Methoden dieses Listeners werden innerhalb der Transaktion durchgeführt.
- *ItemReadListener*, *ItemProcessListener* und *ItemWriteListener* beziehen sich jeweils auf den Aufruf von Reader, Processor und Writer und besitzen jeweils eine *before**-, eine *after**- und eine *on*Error*-Methode.
- *RetryReadListener*, *RetryProcessListener* und *RetryWriteListener* werden aktiv, wenn eine retryable *Exception* geworfen wurde und haben demnach jeweils eine *onRetry*Exception*-Methode.
- *SkipReadListener*, *SkipProcessListener* und *SkipWriteListener* werden aktiv, wenn eine skippable *Exception* geworfen wurde, und haben demnach jeweils eine *onSkip*Item*-Methode.

Parallelisierung

Die Spezifikation sieht eine Art der lokalen parallelen Verarbeitung vor, bei der die Verarbeitung auf mehrere Threads aufgeteilt werden kann. Dabei werden die Daten partitioniert, und jeder Thread arbeitet nur auf der ihm zugewiesenen Partition. Ein Beispiel: Wir gehen wieder von einem Job aus, der über einen Cursor Daten aus einer Datenbank liest und dann weiterverarbeitet. Die Daten sind anhand eines Spartenkürzels geclustert. Statt

Listing 6

```
public interface ItemReader {
  public void open(Serializable checkpoint) throws Exception;
  public void close() throws Exception;
  public Object readItem() throws Exception;
  public Serializable checkpointInfo() throws Exception;
}

public interface ItemWriter {
  public void open(Serializable checkpoint) throws Exception;
  public void close() throws Exception;
  public void writeItems(List<Object> items) throws Exception;
  public Serializable checkpointInfo() throws Exception;
}
```

Listing 7

```
<job id="simpleJob">
  <listeners>
    <listener ref="myJobListener"/>
  </listeners>
  <step id="chunkStep">
    <listeners>
      <listener ref="myStepListener"/>
    </listeners>
    <chunk item-count="2">
      <reader ref="dummyItemReader"/>
      <processor ref="logItemProcessor"/>
      <writer ref="logItemWriter"/>
    </chunk>
  </step>
</job>
```

alle Daten sequenziell zu verarbeiten, bilden wir eine Partition pro Spartenkürzel. Jede Partition nutzt ein eigenes JDBC-Statement, das nach dem Spartenkürzel filtert, sodass sich die verschiedenen Threads nicht behindern.

Listing 8 zeigt einen statisch partitionierten Job mit zwei Partitionen, eine für die Sparte mit dem Spartenkürzel „L“ und eine für die Sparte mit dem Spartenkürzel „K“. Über die Attribute *partitions* und *threads* kann die Anzahl der Partitionen sowie die Anzahl der Threads, die diese Partitionen verarbeiten sollen, bestimmt werden. Die definierten Properties können in anderen Komponenten verwendet werden, indem über das Schlüsselwort *partitionPlan* auf sie zugegriffen wird. Listing 9 zeigt einen Teil des *partitionedJdbcItemReader*. Je nach Partition wird in das Feld *sparte* „K“ oder „L“ injiziert.

Anstatt eines statischen Partition-Plans kann auch eine Mapper-Komponente eingebunden werden, die die Partitionen erzeugt. Ein Mapper, der im obigen Beispiel funktioniert, wird in Listing 10 gezeigt. Der folgende Codeausschnitt zeigt die Job-XML:

```
<partition>
  <mapper ref="spartePartitionMapper" />
</partition>
```

Listing 8

```
<job id="partitionedJdbcJob">
 <step id="chunkStep">
  <chunk item-count="2">
   <reader ref="partitionedJdbcItemReader">
    <properties>
     <property name="spartenKuerzel" value="#{partitionPlan['sparte']}"/>
    </properties>
   </reader>
   <processor ref="logItemProcessor"/>
   <writer ref="logItemWriter"/>
  </chunk>
  <partition>
   <plan partitions="2" threads="2">
    <properties partition="0">
     <property name="sparte" value="L"/>
    </properties>
    <properties partition="1">
     <property name="sparte" value="K"/>
    </properties>
   </plan>
  </partition>
 </step>
</job>
```

Neben dem *PartitionMapper* sieht die Spezifikation noch drei weitere Komponententypen vor, mit denen ein partitionierter Step kontrolliert werden kann. Der *PartitionCollector* ist je Thread aktiv und bietet die

Möglichkeit, Daten über den Verlauf der Partition an eine zentrale Instanz zu schicken. Der *PartitionAnalyzer* ist im Eltern-Thread einmal aktiv und stellt die zentrale Instanz dar, die die Daten der verschiedenen *PartitionCollectors* erhält und verarbeiten kann. Der *PartitionReducer* wird aktiv, wenn Partitionen fehlschlagen bzw. erfolgreich beendet werden, und kann dann Kompensations- bzw. Aufräumarbeiten durchführen.

Spring Batch oder JSR 352?

Diese Frage ist so nicht korrekt, denn der JSR 352 ist eine Spezifikation, Spring Batch eine (zukünftige) Implementierung dieser Spezifikation. Korrekterweise muss man Implementierungen vergleichen, und da gibt es nun einmal noch nicht so viele. Der GlassFish zumindest implementiert tatsächlich nur die Spezifikation und kann so nicht mit Spring Batch mithalten, da die Komponenten, die vielen *ItemReader* und *ItemWriter*, weitergehende Managementmöglichkeiten und vieles mehr fehlen. Deswegen bin ich auch sehr gespannt auf IBMs Implementierung, da IBM ja treibende Kraft war.

Dass sich mit IBM und den Machern von Spring Batch die beiden wichtigsten Player im Batchmarkt auf einen Standard geeinigt haben, der Spring Batch sehr nahe kommt, ist eine gute Sache – aus beiden Blickwinkeln. Diejenigen, die bisher auf Spring Batch setzen, haben die Sicherheit, dass es in Zukunft andere Implementierungen mit dem gleichen Programmiermodell gibt, und dass man im Notfall wechseln kann. Diejenigen, die auf jetzt entstehende Implementierungen in Java-EE-Servern setzen, haben die Sicherheit, dass das Programmiermodell schon jahrelang erfolgreich ist.

Listing 9

```
@Named
public class PartitionedJdbcItemReader extends AbstractItemReader {

  @Inject @BatchProperty(name="spartenKuerzel")
  private String sparte;

  @Override
  public void open(Serializable checkpoint) throws Exception {
    // Cursor mit Jdbc-Statement öffnen, dabei sparte verwenden.
  }

  @Override
  public void close() throws Exception {
    // Cursor schließen.
  }

  @Override
  public Object readItem() throws Exception {
    // Item über Cursor lesen und zurückgeben.
  }

}
```

Listing 10

```
@Named
public class SpartePartitionMapper implements PartitionMapper {
  @Override
  public PartitionPlan mapPartitions() throws Exception {
    PartitionPlan partitionPlan = new PartitionPlanImpl();
    partitionPlan.setPartitions(2);
    partitionPlan.setThreads(2);
    Properties[] propertiesArray = new Properties[2];
    Properties properties = new Properties();
    properties.put("sparte","L");
    propertiesArray[0] = properties;
    properties = new Properties();
    properties.put("sparte","K");
    propertiesArray[1] = properties;
    partitionPlan.setPartitionProperties(propertiesArray);
    return partitionPlan;
  }
}
```

Fazit

Die Spezifikation JSR 352 wirkt rund mit kleinen Abstrichen. (Ich verstehe beispielsweise nicht, warum *ItemReader* und *ItemWriter* nicht über Generics parametrisierbar sind. Dass ein Item vom Typ *Object* ist, wirkt wie aus einer fernen, alten Zeit). Ab jetzt ist es möglich, anbieterneutrale Batchkomponenten zu entwickeln, die sowohl in Spring Batch als auch direkt in jedem Application Server verwendet werden können. Auch das Vermischen von Implementierungen geht nun, man kann also die Laufzeitumgebung des WebSpheres und Komponenten von Spring Batch verwenden, natürlich erst, sobald beide den JSR implementieren.

Würde ich in Zukunft Jobs streng nach dem JSR 352 entwickeln? Vermutlich nicht, da ich Konfigurationen in Java denen in XML vorziehe. Aber es ist gut zu wissen, dass ich im Notfall nicht viel tun muss, um meine Jobs darauf umzustellen.

Tobias Flohre arbeitet als Senior Softwareentwickler bei der codecentric AG. Seine Schwerpunkte sind Java-Enterprise-Anwendungen und Architekturen mit JEE/Spring, häufig mit Fokus auf Spring Batch. Er spricht regelmäßig auf Konferenzen und bloggt auf blog.codecentric.de.

tobias.flohre@codecentric.de.

Links & Literatur

[1] http://jcp.org/en/participation/membership

[2] http://jcp.org/en/participation/committee#SEEE

[3] http://jcp.org/en/jsr/detail?id=352

Kolumne: EnterpriseTales

von Lars Röwekamp

Project Avatar

Im Rahmen der diesjährigen JavaOne kündigte Oracle das Projekt Avatar an – mal wieder. Gleiches geschah nämlich bereits 2011 und 2012. Doch dieses Mal scheint deutlich mehr dahinter zu stecken, als nur ein paar Slides und Demos. Avatar ist ein Web Application Framework, das die Java-basierte Implementierung von „modernen HTML5-Anwendungen" deutlich vereinfachen soll. Grund genug für E.T., einen kleinen Blick hinter die Kulissen zu werfen.

Vorab ein kleiner historischer Abriss: Bereits 2011 priesen Cameron Purdy (heute Vice President of Development, Oracle) und Adam Messinger (heute CTO, Twitter) unter dem Motto „One to rule them All" ein neues Projekt namens Avatar an, das die Entwicklung moderner Webanwendungen im Java-Umfeld deutlich vereinfachen sollte. Neben ein paar Slides, aus denen noch nicht einmal hervorging, ob es sich um ein Framework, Infrastruktur oder eher eine Philosophie handeln sollte, gab es damals allerdings nicht viel zu sehen. Die wenigen, greifbaren Informationen ließen vermuten, dass ein „Etwas" geschaffen werden sollte, das vom HTML5- und JavaScript-Client bis hin zum Java-Backend ein einheitliches Programmiermodell bietet und dabei auch moderne Aspekte wie WebSocket und JSON berücksichtigt. Als Ziel wurde auch eine stärkere Vereinheitlichung der Zielplattformen genannt, da die, auf diesem Modell basierenden Anwendungen auch auf Smartphonebrowsern lauffähig sein sollten.

Etwas konkreter wurde es dann etwa zwölf Monate später, natürlich wieder pünktlich zur JavaOne. Im Rahmen der Keynote wurden erste Demos gezeigt und auf dem Java-Playground – eine Art Ausstellung der neuesten Technologien – konnten Interessierte einen Blick auf den Quellcode werfen. Das Ganze war aber nach wie vor eher eine Blackbox als ein offenes Projekt. Selbst ausprobieren oder gar herunterladen konnte man nach wie vor nichts. Bis jetzt ...

Avatar (2013 Remix)

Getreu dem Motto „und ewig grüßt das Murmeltier" ließ es sich Oracle auch in diesem Jahr nicht nehmen, das Projekt Avatar noch einmal vorzustellen. Anders als in den vergangenen Jahren steckte diese Mal aber tatsächlich auch eine gewisse Substanz dahinter – Open-Source-Lösung zum Herunterladen inklusive [1].

Das Projekt Avatar trägt der Tatsache Rechnung, dass moderne Webanwendungen mehr und mehr die gesamte UI-Logik – Modell, View und Controller – auf den Client verlagern und somit einen neuen Architekturansatz (*TSA*, Thin Server Architecture) verfolgen. Das Resultat sind Single Page Applications, die einmalig die gesamte Anwendung als „HTML5 & Friends"-Seite herunterladen und im Anschluss nur noch Daten mit dem Server austauschen. Bei Bedarf werden via JavaScript-basierter DOM-Manipulation Teile der Seite ein- und ausgeblendet.

Avatar bietet zur Realisierung dieses Modells zwei unterschiedliche Komponenten – JavaScript Server Side Services und Client Side Views. Während die Server Side Services (JSON-)Daten via REST, WebSocket

Porträt

Lars Röwekamp ist Geschäftsführer der open knowledge GmbH und berät seit mehr als zehn Jahren Kunden in internationalen Projekten rund um das Thema Enterprise Computing.

@mobileLarson

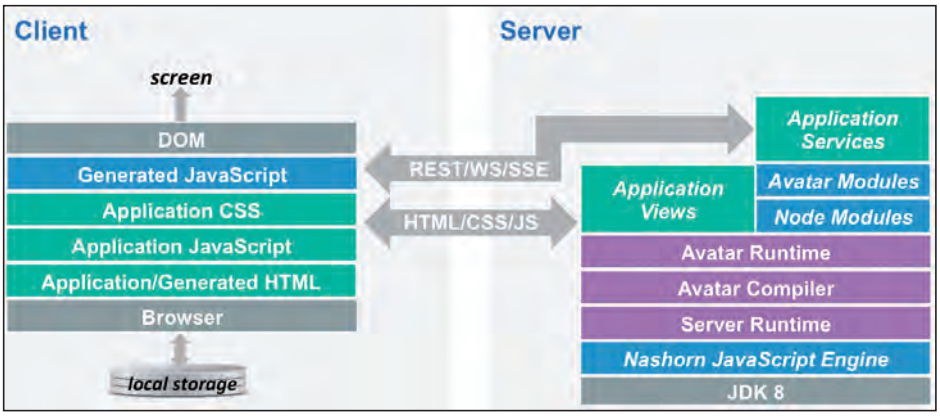


Abb. 1: Avatar-Architektur (Quelle: [2])

Listing 1: Serverservice „aktuelle Uhrzeit“

```
var avatar = require("org/glassfish/avatar");

var delta = 0;

avatar.registerRestService({
  url: "data/time",
  methods: ["GET", "PUT"]},
  function() {

    this.$onGet = function(request, response) {
      var current = new Date().getTime();
      return response.$send({time: (new Date(current-delta)).toISOString()});
    };

    this.$onPut = function(request, response) {
      var newTime = Date.parse(request.data.time);
      var current = (new Date()).getTime();
      delta = current - newTime;
      return response.$send(null);
    };
  }
);
```

Listing 2: Client-View-Module „aktuelle Uhrzeit“

```
<html>
  <head>...</head>
  <body>
    <script data-model="rest">
      var Server = function() { this.time = ''; };
    </script>
    <script data-type="Server"
      data-instance="server"
      data-url="data/time"></script>
    <span id="output">Time is #{server.time}</span>
    <button onclick="#{server.$get()}" id="refresh">Refresh</button><br>
    <input id="input"
      data-value="#{server.time}"/>
    <button onclick="#{server.$put()}" id="set">Set</button>
  </body>
</html>
```

und Server Sent Events an den Client ausliefern, erlauben die Client Side Views die Erstellung eines modernen HTML5-Frontends auf Basis einer Thin-Server-Architektur mit kaum bzw. rudimentären JavaScript-Kenntnissen (**Abb. 1**).

Anders als bei vielen anderen, ähnlichen Frameworks, in denen meist die Client- und Serverkomponenten eng miteinander verwoben sind, können bei Avatar beide Komponenten sowohl im Zusammenspiel als auch völlig losgelöst voneinander verwendet werden. So lässt sich zum Beispiel Avatar nur für die Visualisierung nutzen, während der Server die darzustellenden Daten mithilfe klassischer Java-EE-Services zur Verfügung stellt. Umgekehrt ist das Zusammenspiel ebenso möglich.

Aller Anfang ist ... leicht

Damit der Einstieg in das neue Framework möglichst reibungslos und schnell gelingt, hat Oracle eine Reihe von Tutorials und Beispielen zur Verfügung gestellt. Einmal durchgespielt, werden die in der Theorie eher abstrakten Konstrukte schnell greifbar. Listing 1 zeigt einen Avatar-Service, der auf dem Server deployt wird und von dort via REST-Get-Call die aktuelle Uhrzeit liefert. Gleichzeitig erlaubt der Service das Setzen der Uhrzeit via REST-Put-Call. Für die Auswertung des Service zur Laufzeit auf dem Server ist Avatar.js zuständig.

Zunächst erfolgt ein Zugriff auf das Avatar-Modul (*"org/glassfish/avatar"*) und somit auf **das** zentrale Objekt zur serverseitigen Verwaltung von Services. Im Anschluss wird mit dessen Hilfe der Service als REST-Service unter der relativen URL *date/time* registriert und eine Constructor-Funktion angegeben, die beim Aufruf der URL abgearbeitet werden soll. Mit *$onGet* wird die Funktion definiert, die bei einem GET aufgerufen werden soll, also die Funktion, die das aktuelle Datum inkl. Uhrzeit zurückliefert. Die *$onSend*-Funktion des Response-Objekts dient zur Rückgabe der angefragten Ressource an den Client – in diesem Fall ein einfacher String, der das aktuelle Datum inkl. Uhrzeit enthält. Mit *$onPut* wird das entsprechende Pendant zu *$onGet* zum Setzen der Uhrzeit deklariert. Möchte man das Ganze nun visualisieren, kann dazu ein Client-View-Module verwendet werden (Listing 2).

Bei der Betrachtung von Listing 2 wird schnell deutlich, dass man kein JavaScript-Experte sein muss, um mittels Avatar Client View einen REST-Call abzusetzen und das Ergebnis im Anschluss anzuzeigen. Mithilfe von Script-Tags wird das innerhalb der View zu verwendende Data-Modell deklariert und instanziiert. Der Zugriff auf die Instanz sowie das Absetzen der REST-Calls erfolgt via Expression Language (*EL*). Dank Two Way Data Binding wirken sich Änderungen am Modell direkt auf die UI aus – vice versa.

Push Services via SSE

Im obigen Beispiel muss die Uhrzeit durch ein regelmäßiges „Refresh“ des Clients aktualisiert werden, was nicht gerade als ergonomisch bezeichnet werden kann. Dank *Server Sent Event* (SSE) lässt sich diese Aufgabe leicht automatisieren und auf den Server verlagern. Das Beispiel muss dazu lediglich um einen Push Service auf Seiten des Servers (Listing 3) und ein Push Model auf Seiten des Clients (Listing 4) erweitert werden.

Die Registrierung des Push Service erfolgt ähnlich zur bereits gezeigten Registrierung eines REST-Service. Mit *$onOpen* kann eine Callback-Methode angegeben werden, die bei der Herstellung der Verbindung automatisch aufgerufen werden soll und dabei den aktuellen Kontext übergeben bekommt. Der Kontext kann genutzt werden, um zum Beispiel Daten an den Client zu senden oder einen Timer zu setzen. Läuft der Timer aus, ruft dieser die Callback-Methode *$onTimeout* auf und übergibt ebenfalls den bestehenden Kontext. In unserem Beispiel nutzen wir dieselbe Funktionalität – Timer neu setzen und aktuelle Uhrzeit an den Client senden – sowohl für *$onOpen* als auch für *$onTimeout*.

Damit der Client bzw. die Clientview mit dem Push Service kommunizieren kann, wird dort die Deklaration und Initialisierung eines entsprechenden Push-Modells benötigt. Dank der eben eingeführten Erweiterung wird die manuelle Abfrage der aktuellen Uhrzeit und somit auch der REST-Get-Call obsolet. Die Uhrzeit wird jede Sekunde einmal vom Server in das Modell „gepusht“, die View entsprechend automatisch aktualisiert.

Fazit

Neben den bereits gezeigten Aspekten hat Avatar noch eine ganze Reihe weiterer Features zu bieten: Ein eigenes Hash-Schema zur Abbildung von Navigation, ohne dabei jedes Mal einen Server-Request auslösen zu müssen, bidirektionaler WebSocket-Support, Item- und DataProvider zum Austausch von großen Datenmengen und eine Reihe vorgefertigter, auf *jQuery* basierender UI-Widgets inkl. Themes (bei Bedarf austauschbar mit *dijit*), sind nur einige davon.

Unabhängig von dem derzeitigen Funktionsumfang (*Early Access*!) und der eigentlichen Umsetzung ist auf jeden Fall interessant, dass Oracle mit einem HTML5- und JavaScript-basierten Framework den Puls der Zeit aufgreift und versucht, ihn in die Enterprise-Java-Welt zu portieren.

Ein einheitliches Programmiermodell auf Client und Server, das REST, Server Sent Events und WebSocket unterstützt und somit auch eine Thin Server Architecture, ist für den einen oder anderen Java-EE-Entwickler sicherlich erst einmal gewöhnungsbedürftig, scheint aber – für diesen speziellen Anwendungsfall – deutlich einfacher und komfortabler zu sein als JSF und Co.

Avatar soll übrigens nicht nur auf Java-EE-Servern – derzeit nur GlassFish – eingesetzt werden können, sondern auch in einer reinen Java-SE-Umgebung. Dann allerdings nur in Form eines rudimentären Stacks und somit zum Beispiel ohne JMS- und JPA-Support. Voraussetzung ist das JDK 8.

Insgesamt darf man gespannt darauf sein, wie sich Avatar bis zum Release 1.0 weiterentwickelt. Und natürlich darauf, ob und wie es in Zukunft von der Community angenommen wird. In diesem Sinne: stay tuned ...

Links & Literatur

[1] https://avatar.java.net

[2] https://avatar.java.net/docs/avatar-arch.png

Listing 3: Server Sent Event – Push Service

```
...
var getTime = function() {
  var current = new Date().getTime();
  return {time: (new Date(current - delta)).toISOString()};
};

...

avatar.registerPushService({
    url: "push/time"},
  function() {
    this.$onOpen = this.$onTimeout = function(context) {
      context.$setTimeout(1000);
      return context.$sendMessage(getTime());
    };
  }
);
```

Listing 4: Server Sent Event – Push Model

```
...
  <body>
    <script data-model="rest">
      var Server = function() { this.time = ''; };
    </script>
    <script data-model="push">
      var Updates = function() { this.time = ''; };
    </script>
    <script data-type="Server"
      data-instance="server"
      data-url="data/time">
    </script>
    <script data-type="Updates"
      data-instance="updates"
      data-url="push/time">
    </script>
    <span id="output">Time is #{updates.time}</span>
    <input id="input" data-value="#{server.time}"/>
    <button onclick="#{server.$put()}" id="set">Set</button>
  </body>
...
```

Anzeige

Anzeige

Effiziente Implementierung von Integrationstests

OSGi-Tests mal Groovy

Um bei Softwareprojekten den Anforderungen stets gerecht werden zu können, ist es wichtig, eine permanente Qualitätssicherung mithilfe automatisierter Tests durchzuführen. Die Open Services Gateway Initiative (OSGi) ist eine Softwareplattform, die ein dynamisches Komponentenmodell spezifiziert. In klassischen OSGi-Projekten werden häufig Unit Tests in Java implementiert. Das Schreiben dieser Tests kann sich jedoch als zeitaufwändig und umständlich herausstellen. Die Syntax von Java ist mittlerweile leicht angestaubt und wichtige Sprachfeatures wie Lambda-Ausdrücke kommen erst mit der nächsten Java-Version [1].

von Lars Pfannenschmidt und Dennis Nobel

Glücklicherweise ist die JVM in den vergangenen Jahren um weitere Sprachen erweitert worden. Dies bietet dem Entwickler die Möglichkeit, für verschiedene Aspekte eines Projekts unterschiedliche Programmiersprachen zu verwenden. Insbesondere die Skriptsprache Groovy eignet sich gut für das Implementieren von OSGi-Tests. Mit Groovy lassen sich vor allem Integrationstests, die in der OSGi-Umgebung ausgeführt werden können, effizient und in einer modernen Syntax kompakt schreiben.

Dieser Artikel demonstriert, wie sich für ein beispielhaftes OSGi-Projekt Groovy-Tests implementieren und in einer Equinox-OSGi-Umgebung ausführen lassen. Die Tests lassen sich zudem mithilfe des Build-Werkzeugs Maven vollständig automatisiert ausführen.

OSGi

OSGi enthält eine Reihe von Spezifikationen, die ein dynamisches Komponenten- und Servicemodell für Java beschreiben [2]. Der zentrale Bestandteil des OSGi-Komponentenmodells sind die Bundles. Sie besitzen einen Lebenszyklus, der sich beispielsweise durch Starten, Stoppen, Installieren und Deinstallieren steuern lässt. Jedes einzelne Bundle gibt seine zu exportierenden Java-Pakete sowie die benötigten Abhängigkeiten in Form von Paketimporten an. Dies ist ein wesentlicher Vorteil, der ein einfaches Verbergen und Entkoppeln von API und Implementierung erlaubt und zudem klare Abhängigkeiten zwischen Bundles definiert. Innerhalb von Bundles können OSGi Services registriert werden, die von anderen Komponenten referenziert werden können. Je nach Version der Spezifikation werden bereits eine Reihe von Basisservices, wie der Event-Admin-Service, HTTP-Service, Declarative-Services, Conditional Permission Admin etc., bereitgestellt. Da OSGi häufig im Embedded-Umfeld zum Einsatz kommt, können Applikationen dort in Abhängigkeit der Hardware nur in älteren Java-Versionen implementiert werden.

Integrative OSGi-Tests

Bei Softwaretests unterscheidet man unter anderem zwischen Unit und Integrationstests. Während bei Unit Tests nur eine einzelne Komponente getestet wird, lässt sich bei integrativen Tests das Zusammenspiel mehrerer Komponenten untersuchen. Einfache Java-Klassen, die keine OSGi-spezifischen Abhängigkeiten besitzen, lassen sich gut mit Unit Tests testen. Das Implementieren von Unit Tests für OSGi-Komponenten gestaltet sich mit steigender Komplexität der Komponente jedoch relativ aufwändig. Je nachdem, wie viele Abhängigkeiten eine Komponente besitzt, müssen entsprechend viele Mocks implementiert werden. Darüber hinaus ergeben sich durch das Komponentenmodell in OSGi deklarative Bestandteile, wie die Declarative-Services-(DS-) XML- und Manifest-Dateien, die man durch Unit Tests nur schwer abdecken kann. In DS-XML-Dateien können beispielsweise die OSGi Services einer Anwendung definiert werden [3]. Schleicht sich hier bei der Definition des Java-Interfacenamens ein Tippfehler ein, kann ein Unit Test, der ausschließlich die Java-Klasse fokussiert, diesen nicht aufdecken.

Um die genannten Probleme zu umgehen, bietet sich daher eine integrative Ausführung der Tests an. Damit lässt sich das Verhalten von OSGi-Komponenten innerhalb des Frameworks testen. In diesem Fall müssen darüber hinaus keine Mocks für Standard-OSGi-Services, wie dem Event-Admin, implementiert werden. Zudem lässt sich auch der Lebenszyklus von OSGi-Komponenten testen, beispielsweise wie diese auf Veränderung der Umgebung durch das Deinstallieren anderer Bundles reagieren.

Groovy für Tests

Java hat sich schon lange einen festen Platz unter den typisierten Programmiersprachen für Enterprise- und

Embedded-Entwicklung erobert und wird für seine vielen Frameworks, eine aktive Community sowie für eine stabile Entwicklung geschätzt. All das kann aber nicht darüber hinwegtäuschen, dass Java bei der Syntax und fehlenden Sprachfeatures wie Lambda-Ausdrücke nicht mehr mit modernen Programmiersprachen konkurrieren kann. Das ist für die Entwicklung von Tests besonders nachteilig. Häufig haben diese eher einen Skriptcharakter. Tests sollten daher so effizient und ausdrucksstark wie möglich implementiert werden können. Trotzdem möchte man typsicher auf die zu testenden Java-Klassen zugreifen und von der ausgeprägten Java-Frameworklandschaft profitieren können. Gerade weil das Schreiben von Tests bei einigen Entwicklern zudem nicht sonderlich beliebt ist, sollte man sich um eine Sprache bemühen, in der Tests schnell und einfach geschrieben werden können.

Groovy ist eine moderne dynamische Programmiersprache, die ebenfalls auf der JVM ausgeführt wird. Im direkten Vergleich zu Java besitzt Groovy eine sehr expressive Syntax mit zusätzlichen Konzepten und Funktionen aus Python, Ruby, Perl und Smalltalk, wie beispielsweise Closures, reguläre Ausdrücke, Metaprogrammierung, sowie eine native Syntax für Maps und Listen. Trotzdem lässt sich von Groovy problemlos auf Java-Klassen zugreifen. Daher eignet sich Groovy ideal für das Implementieren von Tests. Aufgrund der expressiven Syntax können die Tests wesentlich schneller und verständlicher implementiert werden. Dies macht sich vor allem bei der Anzahl der Quellcodezeilen bemerkbar. Auch wenn in OSGi-Projekten für die Entwicklung zum Teil nur ältere Java-Versionen verwendet werden können, spricht nichts dagegen, OSGi-Unit- und Integrationstests in Groovy zu implementieren. Ganz im Gegenteil – gerade in Projekten, in denen die Technologien sonst beschränkt sind, kann eine moderne Sprache wie Groovy eine neue Dynamik und mehr Performance ins Team bringen.

Die Pizzademo

Zur Demonstration der Tests wird ein Beispiel verwendet, das einen Pizzaservice in OSGi implementiert. Zweck des Beispiels ist es, typische Services und Techniken von OSGi zu verwenden, die dabei möglichst verständlich sein sollten. Die fachliche Bedeutung des Beispiels ist daher nebensächlich. Das komplette Beispiel inklusive der Tests ist auch auf GitHub (https://github.com/groovyosgi/testing) zu finden.

Das Beispiel stellt einen Pizzabestellvorgang nach. **Abbildung 1** zeigt die verschiedenen OSGi-Komponenten in Form eines Komponentendiagramms. Mithilfe des Pizza-OSGi-Service (*PizzaService*) lassen sich verschiedene Pizzen bestellen. Die Zutaten können mit einer Builder-Klasse konfiguriert werden. Die Bestellung wird innerhalb des Pizzaservice verarbeitet. Hierzu wird zunächst beim Payment-Service (*CreditcardPaymentService*) die Abrechnung der Pizza mittels einer vom Kunden übermittelten Kreditkartennummer veranlasst.

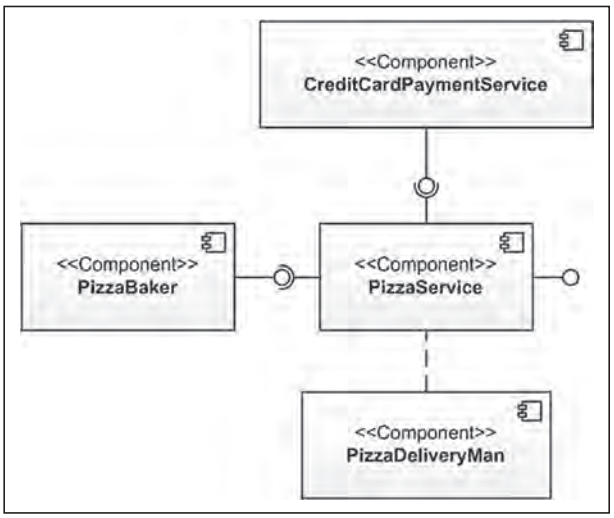


Abb. 1: Pizzademo-Komponentendiagramm

Der Payment-Service stellt den OSGi Service eines Drittanbieters dar, der für Tests zwingend gemockt werden muss. Desweiteren delegiert der Pizzaservice das Backen der Pizza an den Pizzabäcker-OSGi-Service (*PizzaBaker*). Sobald die Pizza fertig ist, wird der Pizzabote (*PizzaDeliveryMan*) über ein Event benachrichtigt: Die Pizza ist bereit zur Auslieferung. Die Kommunikation findet lose gekoppelt über den OSGi-Eventbus statt. Nachdem der Pizzabote die Pizza ausgeliefert hat, benachrichtigt dieser wiederum den Pizzaservice mithilfe eines weiteren Events über die erfolgreiche Auslieferung.

Die Beispielanwendung ist auf drei Bundles verteilt. Die Implementierung liegt getrennt von den Java-Interfaces in einem eigenen Bundle. Darüber hinaus ist der Payment-Service in einem separaten Bundle definiert, wie es auch bei einem echten Drittanbieterservice der Fall wäre. Alle Bundles werden mit Apache Maven und dem Tycho-Plug-in [4] gebaut. In Eclipse lassen sich die Projekte ohne Einschränkungen importieren und nutzen. Die Tests lassen sich sowohl mit Eclipse über vordefinierte Launcher-Dateien, als auch über Maven ausführen.

Ein erster Groovy-Test

Die Klasse *PizzaBuilder* bietet mittels Fluent-Interface-Design-Patterns [5] eine elegante Möglichkeit, Pizzaobjekte mit verschiedenen Zutaten zu erzeugen. Da es sich bei der Klasse um ein POJO ohne Abhängigkeiten zu OSGi handelt, genügt hierfür ein einfacher Unit Test, der sich aber ebenso mit Groovy schreiben lässt. Listing 1 zeigt einen JUnit-Test, der mit dem *PizzaBuilder* eine Pizza generiert und anschließend die Zutaten mit Assertions überprüft. Aufgrund der in Groovy enthaltenen Typinferenz, muss der Typ der Variable *Pizza* nicht explizit angegeben werden. Nur mithilfe des Rückgabetyps der *build*-Methode kann der Groovy-Compiler den Typ selbst herleiten. Für Assertions wird die Hamcrest-Matcher-Bibliothek [6] verwendet, mit der sich besonders ausdruckstarke Assertions formulieren lassen. Die *Assert*- und *Matcher*-Methoden sind im Test statisch

importiert. Da in Groovy zudem Funktionsklammern optional sind und sich auf *getter*-Methoden in einer verkürzten Notation zugreifen lässt, liest sich die Zusicherung *assertThat pizza.sauce, is(TOMATO)* fast wie natürliche Sprache. Dieser einfache Test ist gegenüber einer Java-Implementierung zwar nicht deutlicher kürzer, aber auf Grund der Kombination der genannten Groovy-Sprachfeatures und der Einbindung von etablierten Java-Frameworks deutlich besser lesbar.

Die Basistestklasse „OSGiTest.groovy“

Die abstrakte Klasse *OSGiTest* dient als Basisklasse für OSGi-Integrationstests und stellt Methoden für den Zugriff auf, sowie das Registrieren von gemockten Services bereit. Wie aus Listing 2 hervorgeht, muss von dem erbenden Test die abstrakte Methode *getBundleContext* implementiert werden. Hier muss der *BundleContext* bspw. über einen Activator zurückgegeben werden. Die Methode *bindBundleContext()* wird durch die JUnit-Annotation *@Before* vor der eigentlichen Testausführung initialisiert. Somit hat das Feld *bundleContext* eine Referenz auf den *BundleContext* des Host Bundles.

Durch die Referenz auf den *BundleContext* kann man nun auf bereits registrierte Services zugreifen oder, was für integrative OSGi-Tests häufig notwendig ist, eigene Services registrieren. Mittels der Methode *getService(Class<T>)* lässt sich nun unter Angabe des Serviceinterface via Delegation an *BundleContext.getServiceReference(String)* dessen Implementierung holen.

Das Registrieren einer eigenen Serviceimplementierung bzw. von Service-Mocks ist durch *registerMock(Object)* mit einem optionalen Parameter für Service-Properties möglich. Beim Registrieren des Service erhält man unter Verwendung des Serviceinterfacenamens eine Referenz auf ein *ServiceRegistration*-Objekt, um es später wieder deregistrieren zu können.

Zum Deregistrieren eines bereits selbst registrierten Mocks kann man die Methode *unregisterMock(Object)* nutzen. Die mit *@After* annotierte Methode *unregisterMocks()* deregistriert alle vom Test registrierten Mocks, um die Plattform nach Ausführung der Tests wieder so zu hinterlassen, wie sie vorgefunden wurde.

Listing 1

```
class PizzaBuilderTest {

  @Test
  public void buildPizzaWithBaconCheeseAndTomateSauce() {
    def pizza = PizzaBuilder.newPizza()
            .withBacon()
            .withCheese()
            .withSauce(Sauce.TOMATO)
            .build();

    assertThat pizza.cheese, is(true)
    assertThat pizza.bacon, is(true)
    assertThat pizza.sauce, is(TOMATO)
  }
}
```

Testen von OSGi-Anwendungen

In OSGi besitzt jedes Bundle einen eigenen Classloader. Für OSGi-Tests ist es vorteilhaft, wenn diese im Classloader des zu testenden Bundles laufen und zudem den gleichen *BundleContext* nutzen. Fragment-Bundles stellen ihren Inhalt dem Host Bundle zur Verfügung und laufen damit automatisch in dem gleichen Classloader. OSGi-Tests sollten daher in einem Fragment-Bundle des zu testenden Bundles implementiert werden. In den Tests lässt sich somit auf alle Klassen des zu testenden Bundles zugreifen, ohne dass deren Packages importiert werden müssen. Alle Imports des Host Bundles gelten automatisch für das Fragment-Bundle. Lediglich die Testabhängigkeiten wie JUnit, Hamcrest und Groovy müssen im Fragment-Bundle ergänzt werden.
In Bundles werden zudem OSGi-Service-Interfaces häufig in anderen Packages als deren Implementierung definiert. Während die Packages mit den Interfaces exportiert und anderen Bundles zugänglich gemacht werden, bleiben die Implementierungsklassen in der Regel versteckt. Nur mithilfe des Fragment-Bundles lässt sich auf diese internen Bundle-Klassen zugreifen.

Groovy-Integrationstests

PizzaService verwendet andere OSGi Services, um die Abwicklung der Bestellung zu realisieren. An dieser Stelle ist es sinnvoll, den Pizzaservice innerhalb des OSGi-Lebenszyklus zu testen. So kann beispielsweise überprüft werden, ob die Services korrekt referenziert werden und ob das Zusammenspiel der Komponenten wie gewünscht funktioniert.

Listing 3 zeigt den *PizzaServiceTest*, der von der zuvor erläuterten *OSGiTest*-Klasse erbt. In der *getBundleContext*-Methode wird der *BundleContext* des Activator aus dem Implementierungs-Bundle zurückgegeben. Obwohl die Sichtbarkeit der *getBundleContext*-Methode Package-privat ist, kann Groovy problemlos darauf zugreifen.

In dem Test soll geprüft werden, ob der *PizzaService* bei einer eingehenden Bestellung die *handleTransaction*-Methode des *CreditCardPaymentService* mit den korrekten Parametern aufruft. Da keine Implementierung für die Drittanbieterkomponente *CreditCardPaymentService* existiert, muss zunächst ein Mock definiert werden. Die Erstellung eines Mocks für ein Interface ist in Groovy sehr einfach. Es genügt, eine Map mit dem Ausdruck *as* zu einem Interface zu casten. Innerhalb der Map können Methoden als Schlüssel-Werte-Paar definiert werden, wobei der Schlüssel der Methodenname und der Wert eine Closure sein muss. Für den *CreditCardPaymentService* wird die Methode *handleTransaction* definiert. Innerhalb der Methode werden die

Parameter mit Assertions überprüft, so zum Beispiel ob Kundendaten richtig übermittelt wurden und ob der Preis der Pizza korrekt berechnet wurde. Am Ende der Methode wird die Variable *transactionCalled* auf den Wert *true* gesetzt. So kann später geprüft werden, ob die Methode überhaupt aufgerufen wurde. Der Mock kann nun mithilfe der *registerMock*-Methode aus der *OSGiTest*-Klasse in der OSGi-Laufzeitumgebung registriert werden.

Da nun alle benötigten Services vorhanden sind, wird der *PizzaService* automatisch gestartet und kann über die Methode *getService* referenziert werden. An dem *PizzaService* wird die Methode *placeOrder* mit einem vorher definierten *Order*-Objekt, bestehend aus einer Pizza und den Kundendaten, aufgerufen. So wird die interne Verarbeitung der Bestellung im *PizzaService* angestoßen, der wiederum die *handleTransaction*-Methode des Mocks aufrufen soll. Dies wird ganz am Ende des Tests durch die finale Assertion *assertThat transactionCalled, is(true)* sichergestellt.

In diesem Test wird nun deutlich, wie einfach sich mit Groovy Mocks erzeugen lassen, dessen Verhalten sich in Closures direkt definieren lässt. Prinzipiell lassen sich auch bekannte Java-Mocking-Frameworks wie Mockito [7] oder EasyMock [8] in Groovy verwenden. Darüber hinaus bietet auch Groovy selbst mit *MockFor* und *StubFor* Klassen an, mit denen sich spezifische Mocks erstellen lassen [9]. Es hängt vom konkreten Fall ab, an welcher Stelle welches Framework mehr Sinn macht.

Eventgetrieben

In vielen OSGi-Anwendungen spielt die eventbasierte Kommunikation eine wichtige Rolle. Hierfür existiert der OSGi-*EventAdmin*-Service, der als zentraler Kommunikationsbus fungiert. Komponenten können *EventHandler* am *EventAdmin*-Service registrieren, die auf

Listing 2

```
abstract class OSGiTest {

  static BundleContext bundleContext
  def registeredServices = [:]

  protected abstract BundleContext getBundleContext()

  @Before
  void bindBundleContext() {
    bundleContext = getBundleContext()
    assertThat bundleContext, is(notNullValue())
  }

  def <T> T getService(Class<T> clazz){
    def serviceReference = bundleContext.getServiceReference(clazz.name)
    assertThat serviceReference, is(notNullValue())
    return bundleContext.getService(serviceReference)
  }

  def registerMock(def mock, Hashtable properties = [:]) {
    def interfaceName = mock.class.interfaces?.find({it})?.name
    assertThat interfaceName, is(notNullValue())
    registeredServices.put(interfaceName, bundleContext.
                           registerService(interfaceName, mock, properties))
  }

  def unregisterMock(def mock) {
    registeredServices.get(mock.interfaceName).unregister()
    registeredServices.remove(mock.interfaceName)
  }

  @After
  void unregisterMocks(){
    registeredServices.each() { interfaceName, service ->
      service.unregister()
    }
    registeredServices.clear()
  }
}
```

Listing 3

```
class PizzaServiceTest extends OSGiTest{

  @Test
  public void assertHandleTransactionIsCalled() {

    def transactionCalled = false

    def paymentService = [
      handleTransaction: { String companyId, long creditCardNumber, String
                                                          cardHolderName, float price ->
        assertThat companyId, is(equalTo('LUIGIS_PIZZA_SERVICE'))
        assertThat creditCardNumber, is(524030324560)
        assertThat cardHolderName, is(equalTo("Max Mustermann"))
        assertThat price, is(5f)
        transactionCalled = true
      }
    ] as CreditCardPaymentService

    registerMock(paymentService)

    PizzaService pizzaService = getService(PizzaService)
    def pizza = PizzaBuilder.newPizza().withSauce(Sauce.BBQ).build()
    def customerInfo = new CustomerInfo("Max Mustermann", new Address(), 524030324560)
    pizzaService.placeOrder(new Order(pizza, customerInfo))

    assertThat transactionCalled, is(true)
  }

  @Override
  protected BundleContext getBundleContext() {
    Activator.context
  }
}
```

bestimmte Events reagieren und diese verarbeiten. Das Testen dieser Komponenten ist vor allem aufgrund der Asynchronität tückisch.

In dem Pizzaservice-Beispiel nimmt die Klasse *PizzaDeliveryMan* als OSGi-Komponente Pizzabestellungen über Events entgegen und verschickt wiederum ein bestimmtes Event, wenn die Auslieferung erfolgreich war. Um diese Komponente fokussiert aber integrativ testen zu können, ist es notwendig, im Test Events zu verschicken und zu überprüfen, ob das Event nach der erfolgten Auslieferung verschickt wurde.

Listing 4 zeigt Auszüge aus dem *PizzaDeliveryMan-Test*. Im Test wird zunächst ein *EventListener-Mock* auf das Eventtopic *PIZZA_DELIVERED* registriert. In der *handleEvent*-Methode, die aufgerufen wird, sobald ein Event mit dem entsprechenden Topic verschickt wird, wird die Boolean-Variable *pizzaHasBeenDelivered* auf den Wert *true* gesetzt. Im Folgenden wird am *EventAd-*

Listing 4

```
@Before
void setUp() {
  eventAdmin = getService(EventAdmin)
}

@Test
public void deliverPizzaOnEvent() {

  def eventHandlerProperties = ['event.topics':
                          PizzaDeliveryMan.EVENT_TOPIC_PIZZA_DELIVERED] as Hashtable

  def pizzaHasBeenDelivered = false

  registerMock([
    handleEvent: { Event event ->
      assertThat event.topic, is(equalTo(PizzaDeliveryMan.EVENT_TOPIC_PIZZA_DELIVERED))
    assertThat event.getProperty(PizzaDeliveryMan.EVENT_PROPERTY_ID), is(1)
    pizzaHasBeenDelivered = true
  }
  ] as EventHandler, eventHandlerProperties)

  def customer = new CustomerInfo("Max Mustermann", new Address(), 54325548936 as
                                                                          short)
  def pizza = PizzaBuilder.newPizza().withSauce(Sauce.BBQ).build()

  def properties = [
    (PizzaService.EVENT_PROPERTY_ID): 1,
    (PizzaService.EVENT_PROPERTY_CUSTOMER_INFO): customer,
    (PizzaService.EVENT_PROPERTY_PIZZA): pizza
  ] as Hashtable

  eventAdmin.postEvent(new Event(PizzaService.EVENT_TOPIC_PIZZA_DELIVERY,
                                                                  properties))

  waitFor({pizzaHasBeenDelivered})

  assertThat pizzaHasBeenDelivered, is(true)
}

private waitFor(Closure<?> condition, int timeout = 1000, int sleepTime = 50) {
  def waitingTime = 0
  while(!condition() && waitingTime < timeout) {
    waitingTime += sleepTime
    sleep sleepTime
  }
}
```

Listing 5

```
<repositories>
  <repository>
    <id>eclipse-kepler</id>
    <layout>p2</layout>
    <url>http://download.eclipse.org/releases/kepler</url>
  </repository>
  <repository>
    <id>eclipse-orbit</id>
    <layout>p2</layout>
    <url>http://download.eclipse.org/tools/orbit/downloads/drops/
                                        R20130517111416/repository/</url>
  </repository>
  [...]
</repositories>
```

Listing 6

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>${maven.compiler.version}</version>
  <configuration>
    <compilerId>groovy-eclipse-compiler</compilerId>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-eclipse-compiler</artifactId>
      <version>${groovy.eclipse.compiler.version}</version>
    </dependency>
    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-eclipse-batch</artifactId>
      <version>${groovy.eclipse.batch.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

min-Service ein Event gesendet, das eine Pizzabestellung in Form von verschiedenen Event-Properties beinhaltet. Innerhalb der Methode *postEvent* wird das Event asynchron verschickt. Prüft man in der folgenden Zeile die Variable *pizzaHasBeenDelivered* mit einer Assertion auf den Wert *true*, könnte die Verarbeitung des Events in der *PizzaDeliveryMan*-Komponente noch nicht abgeschlossen sein. Die Assertion kann fehlschlagen, obwohl die Komponente lediglich noch ein wenig Zeit zur Vollendung der Aufgabe benötigt. Daher ist es notwendig, im Thread des Tests eine Weile zu warten.

Eine naive Lösung wäre ein *Thread.sleep*-Aufruf vor der Assertion mit einer fest definierten Wartezeit. Diese sollte hoch genug gewählt sein, dass der Test auch bei einem stark ausgelasteten Rechner nicht fehlschlägt. Problematisch wird es nur, wenn sehr viele Tests eine entsprechende Wartezeit aufweisen. Daher wird im Test die Methode *waitFor* aufgerufen, die als Parameter eine Closure entgegennimmt. Diese kann eine beliebige Bedingung prüfen und muss lediglich einen Boolean-Wert zurückgeben. Die *waitFor*-Methode prüft iterativ die Bedingung und lässt den Thread für eine kurze Zeit schlafen, wenn die Bedingung noch nicht erfüllt ist. Die Schleife läuft solange, bis die Bedingung erfüllt ist oder eine maximale Wartezeit überschritten wird. Somit ist sichergestellt, dass der Test im Erfolgsfall nur so lange läuft, wie es zur Tätigung einer entsprechenden Assertion nach einer asynchronen Verarbeitung notwendig ist. Im Test ist die Bedingung, dass die Variable *pizzaHasBeenDelivered* den Wert *true* annimmt, nachdem das Event nach erfolgter Pizzaauslieferung empfangen wurde.

Und jetzt alles automatisiert

Da es für Softwareprojekte wichtig ist, möglichst schnell und genau den aktuellen Status identifizieren zu können, müssen Tests auch außerhalb der Entwicklungsumgebung automatisiert ausführbar sein. Hierfür wird Tycho verwendet, ein Maven-Plug-in für Eclipse-Artefakte und OSGi Bundles. Im Gegensatz zu einem reinen Maven-Projekt werden im so genannten *Manifest-First*-Ansatz die Abhängigkeiten aus der *MANIFEST.MF* herangezogen. Um diese in den Abhängigkeiten der einzelnen Projekte auflösen zu können, werden in die Parent Pom mehrere *Repositories* hinzugefügt (Listing 5). Bei den OSGi Bundles sind die *pom.xml*-Dateien sehr kompakt. Wichtig ist, dass *eclipse-plugin* als Packaging-Typ angegeben wird.

Die Pom-Definition der Tests hingegen ist aufwändiger. Damit das Fragment-Bundle *com.github.groovyosgi.testing.pizzaservice.test* als Test in Equinox ausgeführt wird, muss der *packaging*-Typ in diesem Fall *eclipse-test-*

plugin sein. Damit nun auch die in Groovy geschriebenen Tests von Tycho gefunden und ausgeführt werden können, muss der Groovy-Sourcecode via Maven kompiliert werden. Dies wird mithilfe des *groovy-eclipse-compiler*-Plug-ins realisiert (Listing 6). Da nun die *.class*-Dateien der JUnit-Tests im *target*-Ordner vorhanden sind, können diese durch Hinzufügen eines "**/*" *includes* in der *tycho-surefire-plugin*-Konfiguration ausgeführt werden. In der *tycho-surefire*-Konfiguration werden ebenfalls die für die Demo benötigten *Declarative Servcies* und der *EventAdmin* gestartet (Listing 7).

Mit dieser Konfiguration können durch den Befehl "*mvn clean package*" alle Projekte gebaut und die Tests ausgeführt werden. Dies kann ohne weitere Konfiguration von einem Build-Server mit vorhandener Maven-Installation automatisiert werden.

Fazit und Ausblick

Dieser Artikel präsentiert ein ausführliches Beispiel, das zeigt, wie sich Groovy für das Implementieren und automatisierte Ausführen von integrativen OSGi-Tests verwenden lässt. Mittlerweile sind die IDEs mit leistungsstarken Groovy-Editoren ausgestattet, die eine komfortable Groovy-Entwicklung ermöglichen. Das initiale Aufsetzen eines Maven-Builds, der zunächst den Groovy-Code kompiliert und die Tests anschließend in einer Equinox OSGi Runtime ausführt, ist zwar etwas aufwändiger, muss am Anfang des Projekts aber nur einmalig konfiguriert werden. Dafür erhält man mit Groovy eine moderne JVM-Sprache, mit der das Entwickeln von Tests spürbar schneller von der Hand geht und die Tests in der Regel auch ausdrucksstärker und einfacher zu überblicken sind. Die automatisierte Ausführung der Tests durch Maven und Tycho in einem CI-Prozess sorgt zudem für eine permanente Qualitätssicherung. Für Java-Entwickler ist der Umstieg auf Groovy auch sehr einfach, da Groovy der Java-Syntax sehr nah ist. Grundsätzlich lässt sich mit Groovy nichts machen, das nicht auch mit Java ginge – aber in Groovy ist es deutlich einfacher. Und gerade Tests sollten so einfach und schnell wie möglich implementiert werden können.

Listing 7

```
<plugin>
  <groupId>org.eclipse.tycho</groupId>
  <artifactId>tycho-surefire-plugin</artifactId>
  <version>${tycho.version}</version>
  <configuration>
    <includes>
      <include>**/*</include>
    </includes>
    <dependencies>
      <dependency>
        <type>eclipse-plugin</type>
        <artifactId>org.eclipse.equinox.ds</artifactId>
        <version>0.0.0</version>
      </dependency>
      <dependency>
        <type>eclipse-plugin</type>
        <artifactId>org.eclipse.equinox.event</artifactId>
        <version>0.0.0</version>
      </dependency>
    </dependencies>
    <bundleStartLevel>
      <bundle>
        <id>org.eclipse.equinox.ds</id>
        <level>1</level>
        <autoStart>true</autoStart>
      </bundle>
      <bundle>
        <id>org.eclipse.equinox.event</id>
        <level>2</level>
        <autoStart>true</autoStart>
      </bundle>
    </bundleStartLevel>
  </configuration>
</plugin>
```

Dennis Nobel ist IT-Berater bei der itemis AG und in verschiedenen Projekten mit den Schwerpunkten Java-, OSGi- und Webentwicklung sowie Continuous Integration tätig. Darüber hinaus interessiert er sich für Qualitätssicherung und verschiedene Programmiersprachen.

Lars Pfannenschmidt ist Senior Software Engineer. Er beschäftigt sich vor allem mit Themen rund um Internet of Things, mobilen Applikationen, agiler Softwareentwicklung und Machine Learning. Er ist Mitgründer der mobile.cologne User Group in Köln.

Links & Literatur

[1] Siehe Artikel von Angelika Langer und Klaus Kreft in dieser Ausgabe auf Seite 21

[2] OSGi Alliance | Technology | The OSGi Architecture. 2012. 31 Jul. 2013: http://www.osgi.org/Technology/WhatIsOSGi

[3] OSGi Declarative Services | OSGi-Wiki: http://wiki.osgi.org/wiki/Declarative_Services

[4] Tycho-Maven-Plug-in: http://eclipse.org/tycho/

[5] FluentInterface | Martin Fowler. 2005. 20. Dez 2005: http://martinfowler.com/bliki/FluentInterface.html

[6] Hamcrest: https://code.google.com/p/hamcrest/

[7] Mockito: https://code.google.com/p/mockito/

[8] Easymock: http://easymock.org/

[9] Groovy – Using MockFor and StubFor: http://groovy.codehaus.org/Using+MockFor+and+StubFor

Neo4j 2.0 hands-on

Graphdatenbank zum Anfassen

Nach dem positiven Feedback zum Neo4j-Artikel in der Ausgabe 10.2013 soll nun das Handwerkzeug vorgestellt werden, mit dem jeder selbst Anwendungen mit Graphdatenbanken entwickeln kann.

von Michael Hunger

Das Interesse an nicht relationalen Datenbanken hat sich im letzten Jahrzehnt deutlich verstärkt. Ein Grund dafür ist neben dem massiv angestiegenen Datenvolumen auch die wachsende Heterogenität der Daten und die zunehmende Komplexität der Beziehungen zwischen den verschiedenen Aspekten der Informationen, die verarbeitet werden müssen.

Da relationale Datenbanken mit ihrem „One size fits all"-Anspruch den konkreten Anforderungen oft nicht gewachsen waren, hat sich eine neue Generation von Datenbanken stark gemacht, solche Anwendungsfälle besser zu unterstützen.

Ausgehend von den großen Internetkonzernen wie Google, Amazon, Facebook wurden für ganz bestimmte Nutzungsszenarien und Datenmengen interne Datenbanken entwickelt (Bigtable, Dynamo, Cassandra), die deutliche Performanceverbesserungen zur Folge hatten. Diese Datenbanken wurden dann entweder Open Source zugänglich gemacht oder wenigstens ihre Konzepte in technischen Artikeln detailliert erläutert. Das bereitete die Grundlage für einen massiven Anstieg der Anzahl von Datenbanklösungen, die sich auf einige wenige Anwendungsfälle spezialisieren und diese dafür optimal abbilden.

Im Allgemeinen werden diese neuen Persistenzlösungen als „NoSQL" zusammengefasst – eine nicht sehr glücklich gewählte Bezeichnung. Unter dem Gesichtspunkt der polyglotten Persistenz steht dieses Kürzel aber eher für „Nicht *nur* (Not only) SQL". Relationale Datenbanken haben weiterhin ebenso ihre Daseinsberechtigung und Einsatzzwecke wie alle anderen Datenbanken auch.

Mit der großen Auswahl ist auch eine neue Verantwortlichkeit auf die Schultern der Entwickler gelegt worden. Sie müssen sich mit den vorhandenen Technologien kritisch auseinandersetzen und mit diesem Hintergrundwissen fundierte Entscheidungen für den Einsatz bestimmter Datenbanktechnologien für konkrete Use Cases und Datenstrukturen treffen.

Den meisten Datenbankanbietern ist das klar. Daher sind Informationsveranstaltungen, Trainings und Bücher über diese Technologien zurzeit hoch im Kurs. Ein wichtiger Aspekt in den Herausforderungen modernen Datenmanagements ist die Verarbeitung heterogener, aber stark vernetzter Daten, die im relationalen Umfeld spärlich besetzte Tabellen und den Verzicht auf Fremdschlüsselbeziehungen (da optional) nach sich ziehen würden.

Informationen über Entitäten aus unserem realen Umfeld sind ohne die sie verbindenden Beziehungen weniger als die Hälfte wert. Heutzutage werden aus den expliziten und impliziten Verknüpfungen entscheidungskritische Analysen erstellt, die für das schnelle Agieren am Markt unabdingbar sind.

Und so hat sich neben den in die Breite skalierbaren, aggregatorientierten NoSQL-Datenbanken auch die Kategorie der Graphdatenbanken etabliert, die sich auf die Verarbeitung stark vernetzter Informationen spezialisiert hat.

Neo4j als einer der ausgereiftesten Vertreter der Kategorie der Graphdatenbanken (RDF und Triple Stores bleiben bei diesen Betrachtungen außen vor) ist schon seit zehn Jahren in der Entwicklung und am Markt. Ursprünglich für die Echtzeitsuche von verschlagworteten Dokumenten über Sprachgrenzen (27 Sprachen) und Bedeutungshierarchien hinweg als Teil eines Onlinedokumentenmanagementsystems entwickelt, wird seine Entwicklung seit 2007 von Neo Technology offiziell gesponsert.

Neo4j ist eine in Java implementierte Graphdatenbank, die ursprünglich als hochperformante, in die JVM eingebettete Bibliothek genutzt wurde, aber seit einigen Jahren als Serverdatenbank zur Verfügung steht. Anders als andere Graphdatenbanken nutzt es einen eigenen, optimierten Persistenzmechanismus für die Speicherung und Verwaltung der Graphdaten. Mit einer mittels Java

NIO (Datei-Memory-Mapping usw.) implementierten Persistenzschicht, die Blöcke fester Größe zum Abspeichern von Knoten und Verbindungsinformationen nutzt, kann Neo4j die unteren Schichten optimal auf seine Bedürfnisse optimieren.

Da Graphdatenbanken, anders als aggregatorienterte Ansätze, auf feingranulare Elemente setzen, die miteinander verknüpft werden, ist es notwendig, für Änderungsoperationen einen Kontext bereitzustellen, in dem Änderungen entweder ganz oder gar nicht erfolgen. Bekanntlich sind dafür Transaktionen ziemlich gut geeignet. Neo4j selbst stellt eine komplette JTA-(und auch XA-2PC-)Transaktionsinfrastruktur zur Verfügung, die die gewohnten ACID-Garantien mitbringt und auch mit anderen transaktionalen Datenquellen integriert werden kann.

Vorhandene APIs

Neo4j stellt eine Reihe von Zugriffsmöglichkeiten bereit, die im Folgenden kurz erläutert werden sollen. Im Tutorial werden wir uns auf Cypher und das Java-API konzentrieren, aber auch einige andere Ansätze (z. B. Spring Data Neo4j oder die Integration in Ruby, JavaScript und Scala) anschauen.

Initial stand nur das auf *Node-* (Knoten) und *Relationship-* (Beziehungen) Klassen basierende, hochperformante Java-API zur Verfügung, das die Graphkonzepte in einer objektorientierten Art und Weise abbildet. Dieses kann auch heutzutage noch für Servererweiterungen genutzt werden (Listing 1).

Seit etwas mehr als zwei Jahren bietet Neo4j mit der Abfragesprache Cypher ein mächtiges Werkzeug zur Abfrage und Verarbeitung von Graphinformationen. Cypher ist wie SQL eine deklarative Abfragesprache, aber deutlich mächtiger in Bezug auf die Lesbarkeit, Repräsentation von Graphkonzepten wie Pfade und Graphmuster und die Verarbeitung von Listen von Werten. Cypher selbst ist in Scala implementiert und nutzt die funktionalen Eigenschaften der Sprache und die vorhandenen und einfach anwendbaren Parser (Parser Combinator und jetzt Parboild/PEG). In Neo4j 2.0 hat Cypher deutliche Erweiterungen erfahren. Der erste Teil des Tutorials wird sich vor allem auf diese Abfragesprache konzentrieren. Cypher kann sowohl mit dem Neo4j-Server als auch über das Java-API benutzt werden und wird so zum universellen API für Neo4j (Listing 2).

Der Neo4j-Server kann durch ein exploratives REST-API angesprochen werden, das die Graphkonzepte auf URIs abbildet und somit zwar einfach zu benutzen, aber nicht besonders performant ist. Daher wird in Neo4j 2.0 das REST-API nur noch für Managementaufgaben eingesetzt. Für alle zukünftigen Interaktionen mit dem Server setzen wir auf einen dedizierten HTTP-Endpunkt, der Abfragen in der Abfragesprache Cypher entgegennimmt und die Ergebnisse zurückstreamt. Dieser Endpunkt unterstützt auch Transaktionen, die mehrere HTTP-Requests überspannen können (Listing 3).

Für den hochperformanten initialen Import von Daten in Neo4j gibt es noch den *BatchInserter*. Dieses Low-Level-Java-API, das nur knapp über dem Persistenzlevel angesiedelt ist, kann genutzt werden, um große Datenmengen schnell aus einer vorhandenen Datenquelle (relationale DB, CSV-Dateien, Datengenerator) in einen Neo4j Datastore zu importieren. Hier gibt es keine Transaktionen, und der Zugriff erfolgt nur von einem Thread. Darauf kommen wir später noch einmal zurück.

Beispieldomäne und Modellierung

Mit Graphdatenbanken hat man bei Beispieldomänenmodellen die Qual der Wahl. Die meisten Domänen, die sich auf ein objektorientiertes oder relationales Modell zurückführen lassen, sind auch für Graphdatenbanken bestens geeignet. Meist kann man weitere interessante Details und Strukturen hinzufügen und das Datenmodell noch viel stärker normalisieren.

Da die Kosten für Beziehungen zwischen Entitäten gering sind, aber ihr Wert enorm ist, ist man gut beraten, das Modell so zu gestalten, dass die reichhaltigen Beziehungen einen deutlichen Mehrwert für die

Listing 1

```
// zentral wegspeichern, ist threadsafe
GraphDatabaseService graphDB = new GraphDatabaseFactory().
                                newEmbeddedGraphDatabase("/path/to/db");

try (Transaction tx = graphDB.beginTx()) {
  Node node1 = graphDB.createNode(Labels.Person);
  node2.setProperty("name","Peter");
  Node node2 = graphDB.createNode(Labels.Person);
  node2.setProperty("name","Andreas");
  node1.createRelationshipTo(node2, Types.KNOWS);

  for (Relationship r : node1.getRelationships(Direction.OUTGOING)) {
    processFriend(r.getEndNode().getProperty("name"));
  }
  tx.success();
}

// am Ende der Anwendung
gdb.shutdown();
```

Zugang zu Neo4j

Neo4j als Server ist ein einfacher Download von [1]. Einfach auspacken, starten und schon sollte alles funktionieren. Alle Treiber gegen das HTTP-API und auch die Neo4j-Shell funktionieren damit. Für die Nutzung von Java-/JVM-Sprachen ist die Maven-Dependency-Konfiguration sinnvoll (bitte die Version auf das aktuelle Release anpassen):

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
  <version>2.0.0-M05</version>
</dependency>
```

Siehe auch [2].

Beantwortung interessanter Fragen darstellen. Auch die Anwendung von verschiedenen Beziehungstypen als semantische Bereicherung ist sehr hilfreich (z. B. als Typen von sozialen Beziehungen: *KNOWS, LOVES, MARRIED_TO, WORKS_WITH, ...*).

Für dieses Tutorial bietet sich das Konzept der Publikation, genauer: Artikel in einer Zeitschrift, an. Dieses Modell ist einfach genug, um von jedem sofort verstanden zu werden, aber ausreichend komplex, um ein paar interessante Fragestellungen zu beleuchten.

Die Entitäten des Modells sind in den folgenden Graphen abstrakt und konkret dargestellt. Praktischerweise geht man von konkreten Daten und Anwendungsfällen aus, wenn man ein Graphmodell zusammen mit einem Domänenexperten skizziert.

Das hat den Vorteil, dass man anhand der konkreten Fälle testen kann, ob das Modell aussagekräftig genug ist und die Beziehungen in einer Art und Weise modelliert sind, die diese Szenarien unterstützt. Des Weiteren ist es ziemlich beeindruckend, wenn das Modell, das soeben noch auf dem Whiteboard zu sehen war, mit einigen Testdaten genauso als Inhalt der Datenbank visualisiert werden kann. Vor allem für nicht technische Beteiligte ist dieser Aha-Effekt oft hilfreich für die Akzeptanz einer neuen Technologie.

Graphdatenmodell

Neo4js Datenmodell (labeled Property-Graph) besteht aus vier grundlegenden Elementen.

Knoten bilden die Entitäten der realen Welt ab. Sie haben oft eine Identität und können anhand von relevanten Schlüsseln gefunden werden. Dabei ist es nicht notwendig, dass alle Knoten im Graphen äquivalent sind. Wie Elemente der Domäne können Knoten in mehreren Rollen und Kontexten genutzt werden. Diese Rollen oder Tags können durch verschiedene **Labels** (keine oder beliebig viele) an den Knoten repräsentiert werden. Mithilfe der Labels werden zusätzliche, strukturelle Metainformationen hinterlegbar. Die Labels sind auch ein geeignetes Mittel, um die Knoten des Graphen in verschiedene (auch überlappende) Sets zu gruppieren.

Beziehungen verbinden Knoten, um das semantische Netz zu formen, das das Graphmodell ausmacht. Beziehungen haben einen Typ und eine Richtung. Dabei kann aber jede Beziehung ohne Performanceeinbußen in beide Richtungen navigiert (traversiert) werden, anders als in einem Objektmodell, in dem Beziehungen nur einseitig sind. Daher ist es normalerweise nur dann sinnvoll, Beziehungen in beiden Richtungen anzulegen, wenn es semantische Bedeutung hat (z. B. *FOLLOWS* im Twitter-Graph).

Beziehungen erzwingen auch das einzig notwendige Integritäts-Constraint in Neo4j. Es gibt keine „Broken Links“. Alle Beziehungen haben einen gültigen Start- und Endknoten. Das bedingt, dass Knoten nur gelöscht werden können, wenn sie keine Beziehungen mehr haben.

Knoten und Beziehungen können beliebige **Attribute** (Schlüssel-Wert-Paare) enthalten; die Werte können alle primitiven Java-Datentypen (String, boolean, numerisch) und Felder davon sein.

Ein Beispiel für ein Datenmodell sehen Sie in **Abbildung 1** [3]. Dieses Datenmodell enthält folgende Entitäten mit folgenden (ausgehenden) Beziehungstypen:

- Publisher (Verlag) *[PUBLISHES]*
- Publication
- Issue (Ausgabe) *[ISSUE_OF, IN_YEAR, CONTAINS]*
- Tag (Schlüsselwort)
- Article *[TAGGED, RELATED_TO]*
- Author *[AUTHORED]*
- Reader *[RATED]*
- Year

Neben den einfachen CRUD-Operationen sind vor allem komplexere Abfragen interessant, zum Beispiel:

- In welchen Themen gibt es die besten Ratings?
- Welcher Autor ist am fleißigsten (pro Verlag)?

Listing 2

```
MATCH (u:Person)-[:KNOWS]->(friend)-[:KNOWS]->(fof)
  (fof)-[:LIVES_IN]->(city)-[:IN_COUNTRY]->(c:Country)
WHERE u.name = "Peter" AND fof.age > 30 and c.name ="Sweden"
RETURN fof.name, count(*) as connections
ORDER BY connections DESC
LIMIT 10
```

Listing 3

```
http> POST /db/data/transaction {"statements":[
{"statement":"CREATE (u:User {login:{name}}) RETURN u","parameters":{"name":"Peter"}}]}
==> 201 Created
{"commit":"http://localhost:7474/db/data/transaction/2/commit",
"results":[{"columns":["u"],"data":[{"row":[{"login":"Peter"}]}]}],
"transaction":{"expires":"Wed, 18 Sep 2013 14:36:26 +0000"},"errors":[]}

POST /db/data/transaction/4 {"statements": [{"statement":"MATCH (u:User) RETURN u"}]}
==> 200 OK
{"commit":"http://localhost:7474/db/data/transaction/4/commit",
"results":[{"columns":["u"],"data":[{"row":[{"login":"Peter"}]}]}],
"transaction":{"expires":"Wed, 18 Sep 2013 14:39:05 +0000"},"errors":[]}

http> POST /db/data/transaction/4/commit
==> 200 OK
==> {"results":[],"errors":[]}
```

Listing 4

```
BatchInserter inserter = BatchInserters.inserter(DIRECTORY.getAbsolutePath());
long node1 = inserter.createNode(map("name", "Peter"), Types.Person);
long node2 = inserter.createNode(map("name", "Michael"), Types.Person);
long relId = inserter.createRelationship(node1, node2, Types.KNOWS, map("since", 2009));
inserter.shutdown();
```

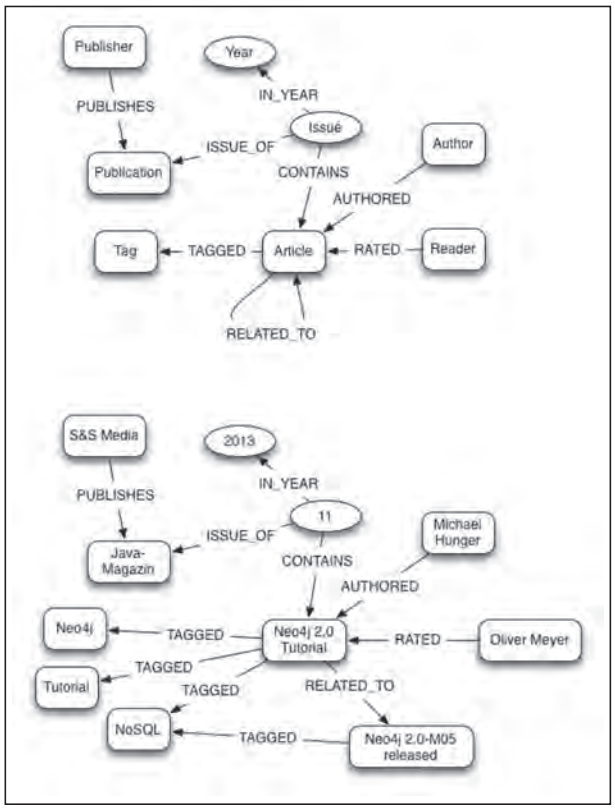


Abb. 1: Beispiel Datenmodell

- Welche Artikel sind zum Tag „NoSQL“ erschienen und in welchen Ausgaben?
- Wenn ich folgenden Artikel gut fand, welche anderen Artikel sind noch zu empfehlen?
  - Empfehlungsberechnung über Ratings, Autoren und Tags
  - Empfehlungsberechnung über mir ähnliche andere Leser und deren Präferenzen

Einführung in Cypher

Wie schon erwähnt, ist Cypher wie SQL eine deklarative Abfragesprache. Man teilt der Datenbank mit, an welchen Informationen man interessiert ist, nicht, wie sie konkret ermittelt werden sollen. Im Allgemeinen sind Graphdatenbanken auf lokale Abfragen ausgerichtet, d. h. man kann von einem Set von Startpunkten aus bestimmten Beziehungen folgen und währenddessen relevante Informationen aggregieren und filtern. Dabei wird aber meist nur ein kleiner Teil (Subgraph) der Daten betrachtet. Interessanterweise ist damit die Abfragegeschwindigkeit nur von der Anzahl der traversierten Beziehungen und nicht von der Gesamtgröße des Graphen abhängig. Dieser Ansatz ist relativ imperativ. Daher kommt in Cypher eine etwas andere Herangehensweise zum Einsatz.

Mustersuche im Graphen

Menschen sind ziemlich gut darin, in übersichtlichen Visualisierungen Muster zu erkennen. Wenn diese Muster formal dargestellt werden können, kann ein Algorithmus natürlich viel schneller und in vergleichsweise riesigen Datenbeständen nach diesen Mustern suchen, Informationen, die damit verknüpft sind, aggregieren, filtern und als Ergebnisse projizieren. Genau das erledigt Cypher für uns. Die Sprache erlaubt die „formale“ Deklaration von Mustern im Modell, und die Query-Engine sucht effizient nach diesen Mustern im Graphen und stellt die gefundenen Informationen inkrementell (lazy) bereit.

Normalerweise würde man diese Muster als Kreise und Pfeile in einem Diagramm aufzeichnen, in einer textuellen Abfragesprache hat man diese Möglichkeit aber nicht. Als alten Mailbox-, MUD- und UseNet-Nutzern war uns die Ausdruckskraft von ASCII-Art geläufig. Daher trafen wir die Entscheidung, Graphmuster in Cypher als ASCII-Art darzustellen.

Knoten werden mit runden Klammern eingeschlossen: *(u:User)*. Beziehungen als Pfeile werden als „-->“ dargestellt, wobei Zusatzinformationen in eckigen Klammern (*-[r:KNOWS*1..3]->*) stehen können. Diese Muster werden in Cypher in der *MATCH*-Klausel angegeben. Beispiel: Finde alle Artikel eines Autors und ihre Ausgabe:

```
MATCH (author:Author)-[:AUTHORED]->(article)<-[:CONTAINS]-(issue:Issue)
WHERE author.name = "Eberhard Wolff"
RETURN article.title, issue.number;
```

Wie man hier sehen kann, ist es ziemlich offensichtlich, was diese Abfrage darstellt – sogar Nichtentwickler können sie leicht verstehen, kommentieren und verändern. Die Hauptklauseln von Cypher sind:

- *MATCH:* Angabe von Mustern und Deklaration von Identifikatoren für Knoten und Beziehungen
- *WHERE:* Filterung der Ergebnisse
- *RETURN:* Projektion der Rückgabewerte (ähnlich zu SELECT in SQL), auch integrierte Aggregation
- *ORDER BY*, *SKIP*, *LIMIT:* Sortierung und Paginierung
- *WITH:* Verkettung von Abfragen mit Weitergaben von Teilergebnissen (kann auch Sortierung, Paginierung enthalten), ggf. Änderung der Kardinalität
- *CREATE*, *MERGE:* Erzeugen von Knoten und Beziehungsstrukturen
- *FOREACH:* Iteration über Liste von Werten
- *SET*, *REMOVE:* Aktualisieren von Informationen/Attributen auf Knoten und Beziehungen
- *DELETE:* Löschen von Elementen
- *CREATE INDEX*, *CREATE CONSTRAINT:* Verwaltung von Indizes und Constraints

Für einen schnellen Überblick über die möglichen Ausdrücke ist die Cypher-Reference-Card empfohlen [4]. Als eine interaktive Sandbox hilft die Neo4j-Onlinekonsole [5] zum Lernen und Ausprobieren, und für die Dokumentation von Graphmodellen die Neo4j GraphGists [6].

Einfache Anwendungsfälle: CRUD-Operationen

Zuerst einmal muss man Informationen in die Datenbank bekommen. Dazu bieten sich die erwähnten *CREATE*- und *MERGE*-Operationen an. *CREATE* er-

zeugt Strukturen ohne Überprüfung. *MERGE* versucht, existierende Muster anhand der Labels, Beziehungen und eindeutiger Attributschlüsselwerte zu finden und erzeugt sie ggf. neu, falls sie noch nicht im Graph vorhanden sind. So erfolgt das Hinzufügen eines Autors mit dem Namen als eindeutiger Schlüssel:

```
MERGE (a:Author {name:"Oliver Gierke"})
ON CREATE a SET a.company = "Pivotal" , a.created = timestamp()
RETURN a;
```

Abhängig von den Anwendungsfällen im Graphen kann die Firma als Attribut oder als Beziehung zu einem Firmenknoten abgelegt werden. Falls die Firma für andere Aspekte (z. B. Empfehlungen, Abrechnung, Sponsoring) relevant sind, würde sie als referenzierbarer Knoten modelliert werden und so als Datenpunkt wiederverwertbar sein.

Ein wichtiger Aspekt bei der Anwendung von Cypher ist die Benutzung von Parametern. Ähnlich wie in Prepared Statements in SQL werden diese eingesetzt, um das Cachen von Abfragen zu erlauben. Denn Statements, in denen literale Werte durch Parameter ersetzt wurden, sehen strukturell gleich aus und müssen nicht neu geparst werden. Außerdem verhindern die Parameter die Injektion von Abfragecode durch Nutzereingaben (siehe SQL-Injektion). Parameter sind benannt und werden in geschweifte Klammern eingeschlossen. Bezeichner (Parameter und andere), die Sonderzeichen enthalten, müssen übrigens mit Backticks „escaped“ werden (z. B. *'süße Träume'*). Die Abfrage würde dann so aussehen:

```
MERGE (a:Author {name:{name}})
ON CREATE a SET a.company = {company} , a.created = timestamp()
RETURN a;
```

Dabei würde der Ausführung der Abfrage eine Map mit Parametern übergeben.

Wie kann diese Abfrage nun gegen die Datenbank ausgeführt werden? Die Listings 5–7 zeigen einige Beispiele.

Das Erzeugen von Beziehungen und ganzen Pfaden erfolgt ähnlich:

```
MERGE (a:Author {name:{name}}),(i:Issue {number:{issue}})
CREATE (a)-[:AUTHORED]->(article:Article {props})<-[:CONTAINS]-(i)
RETURN article;
```

Um ein Element zu löschen, muss es zunächst mittels *MATCH* gefunden werden. Um einen Knoten mit seinen Beziehungen zu löschen, würde man folgendes Statement nutzen:

```
MATCH (a:Author)-[r]-()
WHERE a.name = {name}
DELETE a,r;
```

Ähnlich sieht das mit Aktualisierungen aus. Labels für Knoten können ebenso wie Attribute jederzeit hinzugefügt und entfernt werden:

```
MATCH (issue:Issue)
WHERE issue.number = {issue}
SET issue.date = {date}
SET issue:Special
RETURN issue;
```

Soweit zu den CRUD-Operationen.

Automatische Indizes

Die Mustersuche ist am effektivsten, wenn Knoten im Graphen fixiert werden können, z. B. durch die Beschränkung/Filterung eines Attributs auf einen Wert. Dann kann die Query-Engine die gewünschten Muster an diesen Knoten „verankern" und die Ergebnisse viel schneller in Graph-lokalen Traversals ermitteln.

Falls Indizes für dieses Label-Attribut-Paar vorhanden sind, werden diese für das Laden des initialen Sets von Knoten automatisch benutzt. Seit Neo4j 2.0 kann Cypher aber auch selbst Indizes verwalten:

```
CREATE INDEX ON :Author(name);
DROP INDEX ON :Author(name);

// automatische Nutzung des Indizes zum Auffinden der Knoten
MATCH (a:Author)
WHERE a.name = {name}
RETURN a;
```

Dies kann man auch sichtbar machen, indem man den Query-Plan für eine Abfrage anzeigen lässt. In der Neo4j-Shell würde das mittels des *PROFILE*-Präfixes erfolgen.

Wenn man die Nutzung eines Indexes erzwingen will, sollte das durch einen Hinweis (Hint) mit *USING* erfolgen. Das ist zurzeit auch noch notwendig, wenn mehrere Indizes genutzt werden sollen:

```
MATCH (a:Author), (i:Issue)
USING INDEX a:Author(name)
USING INDEX i:Issue(number)
WHERE a.name = {name}
RETURN a;
```

Listing 5: Java-API

```
// beide zentral speichern, sind threadsafe
GraphDatabaseService gdb = new GraphDatabaseFactory().newEmbeddedDatabase("path/
                                                                        to/db");
ExecutionEngine cypher = new ExecutionEngine(gdb);

String query=
    "MERGE (a:Author {name:{name}}) \n" +
    "ON CREATE a SET a.company = {company} , a.created = timestamp()\n" +
    "RETURN a";
// Transaktionsklammer bei einzelnem Statement hier nicht unbedingt notwendig,
// Cypher startet selbst eine Lesetransaktion
try (Transaction txn = gdb.beginTx()) {
  Map<String,Object> params = map("name", "Oliver Gierke", "company", "Pivotal");
  ExecutionResult result = cypher.execute(query, params);
  assertThat(result.getQueryStatistics().getNodesCreated(),is(1));
  for (Map<String,Object> row : result) {
    assertThat(row.get("a").get("name"),is(params.get("name")));
  }
  txn.success();
}

// am Ende der Anwendung
gdb.shutdown();
```

Listing 6: Neo4j-Shell

```
bin/neo4j-shell [-path data/graph.db]
neo4j-sh (0)$ export name="Oliver Gierke"
neo4j-sh (0)$ export company="Pivotal"
neo4j-sh (0)$ MERGE (a:Author {name:{name}}) ON CREATE a SET a.company =
                                    {company} , a.created = timestamp() RETURN a;
==> +-------------------------------------------------+
==> | a                                               |
==> +-------------------------------------------------+
==> | Node[13784]{created:1379555014972,              |
    | company:"Pivotal",name:"Oliver Gierke"}         |
    +-------------------------------------------------+
==> 1 row
==> Nodes created: 1
==> Properties set: 3
==> Labels added: 1
==> 2 ms
```

Listing 7: HTTP-Endpunkt

```
POST /db/data/transaction/commit {"statements":[
{"statement":"MERGE (a:Author {name:{name}})
ON CREATE a SET a.company = {company} , a.created = timestamp()
RETURN a","parameters":{"name":"Oliver Gierke","company":"Pivotal"}}]}
```

Listing 8

```
CREATE
  (JM_DE:Publication {name:'Java Magazin', language:'DE'}),
  (JM_DE)<-[:ISSUES_OF]-(JMNov2013 {month:11, title:'Java Magazin
                                                          11/2013'})
    -[:IN_YEAR]->(_2013 {year:2013}),
  Neo4j20Tutorial:Content {title:'Neo4j 2.0 Tutorial'}),
  (SnS:Publisher {name:'S&S Media'})-[:PUBLISHES]->(JM_DE),
  (JMNov2013)-[:CONTAINS]->(Neo4j20Tutorial),
  (Olli:Reader{name:'Oliver Meyer',handle:'@olli'})
    -[:RATED{rating:4}]->(Neo4j20Tutorial),
  (MH:Author:Reader{name:'Michael Hunger',handle:'@mesirii'})
    -[:AUTHORED]->(Neo4j20Tutorial),
  (Neo4j20Tutorial)-[:RELATED_TO]->
  (Neo4j20Rel:Content {title:'Neo4j 2.0-M05 released'})
    -[:TAGGED]->(NoSQL:Tag {name:'NoSQL'}),
  (Neo4j20Tutorial)-[:TAGGED]->(NoSQL),
  (Neo4j20Tutorial)-[:TAGGED]->(:Tag {name:'tutorial'}),
  (Neo4j20Tutorial)-[:TAGGED]->(:Tag {name:'Neo4j'});
```

Import

Mit diesem Handwerkszeug kann man nun relativ einfach Daten in Neo4j importieren. Dazu gibt es mehrere Möglichkeiten:

- Programmatischer Aufruf der genannten Cypher-APIs mittels eines Programms (in Java, Scala, Ruby, C# ...) und Übergabe der notwendigen Anfragen und Bereitstellung der Parameter aus einer Datenquelle (z. B. relationale Datenbank, CSV-Dateien oder Datengenerator).
- Generierung von Cypher-Statements in Textdateien (ähnlich SQL-Importskripten) und Import über die Neo4j-Shell. Dabei können große Blöcke von Statements (30 k–50 k), die atomar eingefügt werden sollen, von *BEGIN ... COMMIT*-Kommandos umgeben sein, um einen transaktionalen Rahmen zu schaffen.

Listing 9

```
BEGIN
// Knoten und Indizes
CREATE INDEX ON :Author(name);
CREATE (:Author {name:"Michael Hunger"});
CREATE (:Author {name:"Peter Neubauer"});
CREATE (:Author {name:"Eberhard Wolff"});
CREATE (:Author {name:"Oliver Gierke"});

CREATE INDEX ON :Publisher(name);
CREATE INDEX ON :Publication(name);

MERGE (pr:Publisher {name:"S&S Media"}),
    (pn:Publication {name:"Java Magazin"})
CREATE (pr)-[:PUBLISHES]->(pn);

CREATE INDEX ON :Tag(name);
CREATE (:Tag {name:"NoSQL"});
CREATE (:Tag {name:"Neo4j"});
CREATE (:Tag {name:"Spring Data"});
CREATE (:Tag {name:"Tutorial"});

CREATE INDEX ON :Issue(number);

CREATE INDEX ON :Article(title);

// Beziehungen
MERGE (pn:Publication {name:"Java Magazin"}),
   (i:Issue {number:201311}),
   (art:Article {title:"Neo4j 2.0 Tutorial"}),
   (au:Author {title:"Michael Hunger"})
CREATE (i)-[:CONTAINS]->(art),
   (i)-[:ISSUE_OF]->(pn),
   (au)-[:AUTHORED]->(art);
....

MATCH (t:Tag),(art:Article)
WHERE t.name IN ["NoSQL","Neo4j","Tutorial"]
AND art.title="Neo4j 2.0 Tutorial"
CREATE (art)-[:TAGGED]->tag;
....
COMMIT
```

- Import aus CSV-Dateien mittels eines existierenden Tools (z. B. der CSV-Batch-Inserter für den Import großer Datenmengen [7]).
- Import über einen der Neo4j-Treiber für die meisten Programmiersprachen [7] oder ein Mapping-Framework wie Spring Data Neo4j (dazu später mehr).

Beispiele

Ein einziges *CREATE*-Statement aus dem genannten *GraphGist* zeigt Listing 8, eine Reihe von *CREATE/MERGE*-Statements Listing 9.

Diese Datei aus Listing 9 kann nun mittels der Neo4j-Shell geladen werden:

```
./bin/neo4j-shell -path data/graph.db -file articles.cql
```

Beim Einfügen neuer Daten in eine bestehende Datenbank muss man davon ausgehen, dass die Knoten aus einem vorhergehenden Importschritt teilweise schon vorhanden sind. Dann bietet sich an, *MERGE* für Knoten zu benutzen, das einem „erzeuge, wenn nicht vorhanden“ entspricht:

```
MERGE (:Publication {name:"Java Magazin"});
```

Fazit und Ausblick

Das war ein erster Blick hinter die Kulissen von Graphdatenbanken und Neo4j mit seinen verschiedenen APIs. Im nächsten Teil wird es um komplexe Abfragen mittels Cypher gehen. Außerdem werden wir einen Blick auf das Web-UI für den Neo4j-Server werfen.

Software zu entwickeln, gehört zu **Michael Hungers** großen Leidenschaften. Der Umgang mit den beteiligten Menschen ist ein besonders wichtiger Aspekt. Zu seinen Interessen gehören außerdem Software Craftsmanship, Programmiersprachen, Domain Specific Languages und Clean Code. Seit Mitte 2010 arbeitet er eng mit Neo Technology zusammen, um deren Graphdatenbank Neo4j noch leichter für Entwickler zugänglich zu machen. Hauptfokus sind dort Integration in Spring (Spring Data Graph Project) und Hosting-Lösungen. Zurzeit hilft er der Neo4j-Community dabei, mit der Graphdatenbank ihre Wünsche wahr werden zu lassen. Michael arbeitet(e) an mehreren Open-Source-Projekten mit, ist Autor, Editor, Buchreviewer und Sprecher bei Konferenzen. Neben seiner Familie betreibt er noch ein Buch- und Kulturcafé (die-buchbar.de) in Dresden, ist Vereinsvorstand des letzten großen deutschen MUDs (mg.mud.de) und hat viel Freude an kreativen Projekten aller Art.

Links & Literatur

[1] http://neo4j.org/download

[2] http://www.neo4j.org/download/maven

[3] http://gist.neo4j.org/?github-neo4j-contrib%2Fgists%2F%2Fother%2FThePublicationGraph.adoc

[4] http://docs.neo4j.org/refcard/2.0

[5] http://console.neo4j.org

[6] http://gist.neo4j.org

[7] http://neo4j.org/develop/import

Anzeige

REST ohne Stress

Google Cloud Endpoints

Ideen für mobile Apps gibt es wie Sand am Meer. Daher wundert es nicht, dass man mit seiner Idee nicht alleine startet, sondern im Wettlauf mit anderen Teams. Oder der Kunde hätte die erste Version der App einfach nur gerne am liebsten schon vorgestern. Wie es möglich ist, dass zumindest das Thema Backend-Kommunikation kein Hindernis mehr darstellt, erfahren Sie hier.

von Andreas Feldschmid

Fast jede mobile App ist in der einen oder anderen Weise auf ein Backend angewiesen. Das Mittel der Wahl ist hier klassischerweise eine REST-Schnittstelle, über die JSON oder XML ausgetauscht wird. Obwohl es hier schon viele Frameworks (Jersey oder Restlet, Gson oder Jackson, SimpleXML) gibt, bleibt doch viel Boilerplate-Code zu schreiben. POJOs wollen annotiert, die Verbindung aufgebaut, Retry-Mechanismen eingebaut werden.

Und dann bleibt noch das Problem, wie man REST-Schnittstellen überhaupt beschreibt. Weder WSDL noch WADL haben sich hier bislang richtig durchgesetzt. Sowohl auf dem Server als auch auf dem Client muss die Schnittstelle jedenfalls erstellt und gepflegt werden. Kommt dann noch ein iOS-Client dazu, hat man diese Arbeit in Objective-C nochmals.

Google stellt Entwicklern mit den Cloud Endpoints ein mächtiges Werkzeug zur Verfügung, mit dem die REST-Schnittstelle im Backend und der nötige Clientcode auf geradezu magische Weise generiert werden können. In den nächsten Abschnitten beschreibe ich Schritt für Schritt anhand einer Beispielapplikation, wie man ein Cloud-Endpoints-Backend erstellt und dies aus einer Android-Applikation heraus aufruft. Auf iOS gehe ich nicht detailliert ein, kann aber aus eigener Projekterfahrung versichern, dass auch hier die generierten Client-Libraries einen guten Dienst tun.

Set-up

Als Entwicklungsumgebung wird Eclipse in der aktuellen Version (4.3, „Kepler“) verwendet. Mit dem Google-Plug-in für Eclipse [1] gelingen die Projektkonfiguration, das Deployment in die App-Engine sowie die Generierung der Client-Libraries noch komfortabler als per Kommandozeile. Falls noch nicht geschehen, kann unter dem gleichen Update-URL auch das Plug-in für Android-Entwicklung bezogen werden.

Für die Benutzung der App-Engine ist ein kostenloser Google-Account notwendig, mit dem unter appengine.google.com eine neue Applikation mit eindeutigem Namen angelegt wird. Für mein Beispiel verwende ich *iteratec-bib* als Application Identifier. Weiterhin wird das App-Engine-SDK für Java [2] benötigt. Die weiteren Schritte sind nun:

1. Ein Backend-Projekt mit den fachlichen Entitäten erstellen
2. Aus den Entitäten die REST-Schnittstelle generieren lassen
3. Client-Libraries für Android, iOS und JavaScript generieren lassen
4. Client-Library in einer App verwenden

REST-Backend in der App-Engine

In Eclipse wählt man im *New Project*-Wizard (STRG + N) Google | Web Application Project aus. Spätestens hier muss nun auch angegeben werden, wo das App-Engine-SDK entpackt wurde. Google Web Toolkit wird nicht verwendet, „Generate project sample code“ kann angehakt bleiben. Dies erstellt eine statische HTML-Seite und ein einfaches Servlet, mit dem wir verifizieren können, dass das Deployment erfolgreich klappt. Nach der Vergabe von Namen und Package landet man im erzeugten Projekt. Dieses kann ab sofort über einen Rechtsklick auf das Projekt und Auswahl von Google | Deploy to Appengine ausgeliefert werden. Beim ersten Mal muss man sich dazu im Plug-in einlog-

gen und die zuvor angelegte Application-ID (in unserem Fall *iteratec-bib*) in den Projekteinstellungen eintragen. Die Applikation ist ab sofort unter iteratec-bib.appspot.com erreichbar. Erstellen wir nun eine Klasse, die eine Entität darstellt und mit JPA-Annotationen versehen ist, wie etwa das Beispiel für Bücher in Listing 1.

Um nun einen REST-Endpoint für unsere Bücher zu generieren, reicht ein Rechtsklick auf die Klasse und Auswählen von Google | Generate Cloud Endpoint Class. Im generierten Book Endpoint wurden Methoden zum Anlegen, Lesen, Editieren, Löschen und Auflisten (CRUDL) von Büchern erzeugt. Ein Ausschnitt davon ist in Listing 2 zu sehen. Im Wesentlichen ist dies der nötige JPA-Code, um auf den Data Store zuzugreifen. Die nötige *EntityManagerFactory*-Implementierung, die auf die bereits vorhandene *persistence.xml* verweist, in der der Persistence-Provider für den Zugriff auf den App-Engine-Data-Store konfiguriert ist, wurde ebenfalls automatisch erzeugt. Die Methoden sind so annotiert, dass sie als REST-Endpoint bereitgestellt werden. Lobenswert: Anhand des Methodennamens wird entschieden, welche HTTP-Methode verwendet wird. Dies kann – wie vieles andere auch – durch Parameter der *@ApiMethod*-Annotation explizit konfiguriert werden.

Manuelles Testen und der Google APIs Explorer

Testen wir unseren Code zunächst lokal. Dazu reicht ein Rechtsklick auf das Projekt und Auswahl von Run As | Web Application. Daraufhin startet ein Jetty mit lokaler App-Engine-Laufzeitumgebung inklusive simuliertem High Replication Data Store (HRD). Um unseren REST-Endpoint aufzurufen, kann nun der Google APIs Explorer verwendet werden. Dazu wird der URL http://localhost:8888/_ah/api/explorer aufgerufen. Der APIs Explorer ist in JavaScript implementiert und kann sowohl für lokale als auch für auf appspot.com deployte Apps verwendet werden. Ein Wort der Warnung: Caching wird hier exzessiv verwendet, regelmäßiges Löschen der Browserdaten bzw. Browserneustarts sind anzuraten, falls Backend-Änderungen sich nicht auf den Explorer durchschlagen.

Der API-Explorer ruft das Discovery-Dokument auf, das sich unter localhost:8888/_ah/api/discovery/v1/apis/bookendpoint/v1/rest befindet und bietet basierend darauf die möglichen REST-Aufrufe an. Dieses Dokument findet man auch lokal im */war/WEB-INF*-Verzeichnis. Es enthält eine Beschreibung sämtlicher angebotenen Methoden und deren Parameter sowie komplexen Datenstrukturen und stellt somit eine Dokumentation der REST-Schnittstelle dar.

Um ein neues Buch anzulegen, wählen wir *bookendpoint.insertBook* aus und befüllen die nötigen Felder im *Request-Body*. Das Ergebnis der Ausführung dieses Requests ist in **Abbildung 1** zu sehen. Eine weitere Weboberfläche verbirgt sich unter http://localhost:8888/_ah/admin/ – hier lassen sich die Data-Store-Einträge betrachten und löschen. Das Anlegen und Editieren ist wohl noch auf der To-do-Liste.

Ebenso einfach lässt sich der APIs Explorer verwenden, um die deployte Applikation auf der App-Engine anzusprechen. Dazu wird einfach https://iteratec-bib.appspot.com/_ah/api/explorer aufgerufen.

Client-Libraries

Widmen wir uns nun dem Erzeugen der Client-Libraries. Für Android reicht ein Rechtsklick auf das Projekt und Auswählen von Google | Generate Cloud Endpoint Client Library. Der erzeugte Code sowie benö-

Listing 1

```
package de.iteratec.bib.model;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Book {
  @Id
  private String isbn;
  private String title;
  private String author;

  [getter/setter]
}
```

Listing 2

```
package de.iteratec.bib.model;
@Api(name = "bookendpoint", namespace = @ApiNamespace(ownerDomain =
                    "iteratec.de", ownerName = "iteratec.de", packagePath = "bib.model"))
public class BookEndpoint {

  /**
   * This inserts a new entity into App Engine Data Store. If the entity already
   * exists in the Data Store, an exception is thrown.
   * It uses HTTP POST method.
   *
   * @param book the entity to be inserted.
   * @return The inserted entity.
   */
  @ApiMethod(name = "insertBook")
  public Book insertBook(Book book) {
    EntityManager mgr = getEntityManager();
    try {
      if (containsBook(book)) {
        throw new EntityExistsException("Object already exists");
      }
      mgr.persist(book);
    } finally {
      mgr.close();
    }
    return book;
  }
}
```

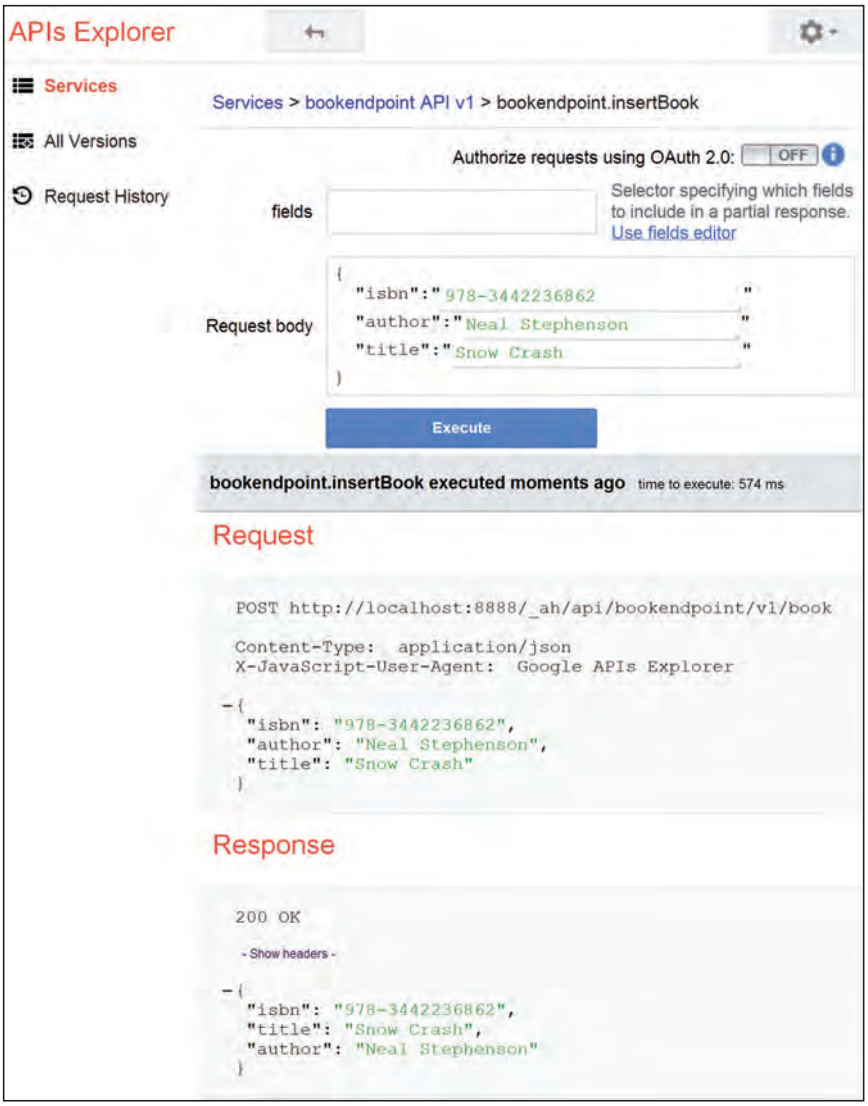


Abb. 1: Ausgabe des Google APIs Explorer

Android/Java-Client-Libraries

Die erstellten Client-Libraries für Java sind eigentlich nicht Android-spezifisch. Im Ordner */libs* wird eine Vielzahl von Dependencies abgelegt – je nachdem, von welcher Art Java-Client die Endpoints aufgerufen werden, sind andere Google API Extensions nützlich. Folgende sind für Verwendung unter Android notwendig und müssen in das Projekt unter */libs* kopiert werden:

- *google-api-client-*.jar* und *google-http-client-*.jar* – Zugriff auf das REST-Backend
- *google-http-client-android-*.jar* – Hilfsklassen, um je nach Android-Version zur Laufzeit passende HTTP-Implementierung zu wählen
- *google-http-client-gson-*.jar* – Extension für den *api-client,* um GSON für JSON Marshalling zu verwenden
- *gson-2.1.jar* – GSON-Bibliothek für JSON Marshalling

Diese unterstützen die Verwendung von OAuth unter Android. Beispiele dazu finden sich in der offiziellen Dokumentation:

- *google-api-client-android-*.jar*
- *google-oauth-client-*.jar*

tigte Dependencies landen im Unterordner *endpoint-libs*. Um Clientcode für iOS zu erzeugen, ist etwas mehr Arbeit nötig, siehe [3]. Für JavaScript reicht es, die bestehende Google-API-JavaScript-Bibliothek entsprechend zu konfigurieren [4]. Zurück zu Android: Die generierten *.java*-Dateien (oder wahlweise das *.jar*-Archiv) können in ein bestehendes Android-Projekt integriert werden. Zusätzlich müssen Dependencies in das */libs*-Verzeichnis des Android-Projekts übernommen werden, siehe Kasten „Android/Java-Client-Libraries".

Falls die Android-App bisher noch keinen Internetzugriff ausgeführt hat, ist es nun notwendig, die erforderliche Berechtigung im *AndroidManifest.xml* anzufordern:

```
<uses-permission android:name="android.permission.
                                    INTERNET" />
```

Der Code in Listing 3 konfiguriert zunächst einen Builder mit der nötigen HTTP- und JSON-Implementierung und ruft dann eine unserer REST-Methoden auf. Ergebnis ist eine Java-Liste von Buchobjekten, die geloggt wird. Diese ließen sich nun auch leicht in einer List-View anzeigen. Natürlich sollte der Aufruf asynchron in einem *AsyncTask* ausgeführt werden. Damit ist der erste Durchstich erfolgt, und wir haben erfolgreich unseren REST-Endpoint aus Android heraus konsumiert – war doch gar nicht so schwer.

Das Anlegen von neuen Objekten erfolgt ebenso einfach. Clientseitig werden die gleichen Java-Objekte befüllt, die im Backend erstellt wurden. Auch mit komplexen Objekten klappt die Übertragung ohne Probleme. Sollen mehrere Objekte (verschiedenartige oder beispielsweise eine Liste von Büchern) übertragen werden, muss darauf geachtet werden, dass nur ein „Top-Level"-Objekt unterstützt wird. In diesem Fall muss ein Wrapper-Objekt verwendet werden.

Customizing

Apps werden mobil genutzt und unterliegen somit völlig anderen Rahmenbedingungen als ein gut ans Netz angebundener Entwicklungsrechner – was Verbindungsqualität im Mobilfunknetz angeht, sollte man sich diesbezüglich durchaus Gedanken machen. Auch hier unterstützt Google mit einem konfigurierbaren *HttpBackOffIOExceptionHandler*. Der in Listing 4 eingesetzte enthält eine Strategie, die nach *IOExceptions* initial 500 ms wartet, diesen Wert mit jeder Exception um 50 Prozent erhöht, maximal zwei Sekunden zwischen Versuchen verstreichen lässt und nach maximal acht Sekunden komplett abbricht. Der *CustomHttpReq-*

Initializer, der jeden Request mit dem *ExceptionHandler* konfiguriert, kann beim Erstellen des Builders als dritter Parameter übergeben werden.

Ebenfalls in Listing 3 sieht man, wie Request-Parameter überschrieben werden können oder die *Basic Authentication* mit Username und Passwort hinzugefügt wird. Möchte man den Nutzer dagegen über einen bestehenden Google-Account des Android-Smartphones einloggen, verwendet man als *RequestInitializer* eine Instanz von *GoogleAccountCredential*, die sich mit dem Android-*AccountPicker* integrieren lässt. Über die im Hintergrund ausgetauschten OAuth-2.0-Tokens muss man sich dann zum Glück nicht mehr selbst kümmern.

Im Backend haben wir Anfangs JPA-Annotationen benutzt. Dies hat den Vorteil, dass das Eclipse-Plug-in hieraus direkt Endpoint-Methoden generieren konnte. Kann man auf die automatische Generierung verzichten, lassen sich aber auch andere Frameworks zum Zugriff auf den App-Engine-Data-Store problemlos einsetzen, wie etwa das beliebte Objectify [5]. Möchte man nicht alle Attribute der Entität über die REST-Schnittstelle ausliefern, werden diese mit *@ApiSerializationProperty (ignored = AnnotationBoolean.TRUE)* annotiert.

Werden die Anforderungen ans Backend komplexer, lohnt es sich, Dependency Injection einzuführen. Wie Google Guice konfiguriert werden muss, damit es mit den Cloud-Endpoints zusammenspielt, habe ich unter [6] beschrieben.

Fazit

Derzeit sind die Google-Cloud-Endpoints noch als „Previewrelease“ veröffentlicht, was bedeutet, dass das API sich innerhalb kurzer Zeit ändern kann und keine SLAs zugesichert werden. Die Dokumentation ist auf dem aktuellen Stand und mit Codebeispielen leicht verständlich. Zudem findet man auf den einschlägigen Seiten wie Stackoverflow Google-Mitarbeiter, die auch knifflige Fragen beantworten. Was man bei komplexeren Datenstrukturen nicht unterschätzen sollte, ist der Unterschied zwischen klassischem SQL und dem Google-App-Engine-Data-Store. Für kleinere Testprojekte, Konzeptstudien oder Hobbyprojekte eignet sich diese Art, ein Backend für eine mobile App zu betreiben, hervorragend. Bei iteratec haben wir die Cloud-Endpoints erfolgreich für die Umsetzung eines Prototypen für einen Kunden eingesetzt. Die gesparte Zeit konnte in weitere Features und UI-Feinschliff investiert werden.

Andreas Feldschmid ist Softwarearchitekt bei der iteratec GmbH, einem Softwareentwicklungs- und Beratungshaus in München. Er beschäftigt sich seit 2009 mit Android-Entwicklung und ist seit drei Jahren unter anderem für die fahrzeugseitige Entwicklung der BMW-Carsharing-Lösung für DriveNow und Alphacity zuständig.

Andreas.Feldschmid@iteratec.de

Links & Literatur

[1] https://developers.google.com/eclipse/docs/download

[2] https://developers.google.com/appengine/downloads#Google_App_Engine_SDK_for_Java

[3] https://developers.google.com/appengine/docs/java/endpoints/consume_ios

[4] https://developers.google.com/appengine/docs/java/endpoints/consume_js

[5] http://code.google.com/p/objectify-appengine/

[6] http://stackoverflow.com/questions/17128714/appengine-with-google-cloud-endpoints-and-guice

Listing 3

```
Builder builder = new Bookendpoint.Builder(
  AndroidHttp.newCompatibleTransport(),
  new GsonFactory(), null);
Bookendpoint bookingEndpoint = builder.build();

try {
  List<Book> books = bookingEndpoint.listBook().execute().getItems();
  if (books == null) {
    Log.i(TAG, "No Books stored in Backend");
    return;
  }
  for (Book book : books) {
    Log.i(TAG, String.format("ISBN: %s, Title: %s, Author: %s",
    book.getIsbn(), book.getTitle(), book.getAuthor()));
  }
} catch (IOException e) {
  Log.e(TAG, "Error while executing request", e);
}
```

Listing 4

```
public class CustomHttpReqInitializer implements HttpRequestInitializer {

  public void initialize(HttpRequest request) {
    request.getHeaders().setBasicAuthentication("user", "pass");
    request.setConnectTimeout(2 * 1000);
    request.setReadTimeout(2 * 1000);

    ExponentialBackOff backOffStrategy = new ExponentialBackOff.Builder()
    .setInitialIntervalMillis(500)
    .setMultiplier(1.5)
    .setMaxIntervalMillis(2 * 1000)
    .setMaxElapsedTimeMillis(8 * 1000)
    .build();
    request.setIOExceptionHandler(
    new HttpBackOffIOExceptionHandler(backOffStrategy));
  }

}
```

TFS SDK für Java zur Anbindung eines Taskmonitors

Krieg der Welten?

Mit dem TFS SDK für Java hat Microsoft eine Möglichkeit geschaffen, Java-Anwendungen mit dem Microsoft TFS arbeiten zu lassen. Mithilfe des SDKs wurde ein Taskmonitor für TFS gebaut, mit dem sich die Sprints, Stories und Tasks eines TFS-basierten Scrum-Projekts betrachten, verfolgen und beeinflussen lassen. Wir berichten von Erfahrungen, Problemen und Lösungen bei der Implementierung.

von Thomas Wilk und Denny Israel

Es war schon eine ungewöhnliche Projektsituation, der wir gegenüberstanden, als wir hörten, dass es sowohl ein .NET- als auch ein Java-Projekt werden sollte – ein Java-Taskmonitor mit dem Project-Management-System des Team Foundation Servers. Diese ungewöhnliche Zusammenstellung der Technologien überraschte uns und wurde zu unserer Aufgabe innerhalb des letzten halben Jahres. Dabei haben wir Türen geöffnet, Grenzen überquert und Dämme gebaut. Somit war schon früh klar, dass wir einen Artikel über unsere Erfahrungen verfassen würden. Dieser Artikel verfolgt die Absicht, Stolperfallen zu offenbaren, Kniffe im Umgang mit dem SDK zu zeigen und Eigenheiten des Java-SDKs gegenüber seinem .NET-Pendant darzustellen.

Architektur

Unser Taskmonitor selbst ist eine Java-Anwendung, die über das TFS SDK für Java Kontakt mit einem TFS aufnimmt und die Daten eines bestimmten Projekts ausliest. Hierzu zählen sowohl Projektmetadaten wie Objekttypen (Stories, Bugs, Tasks) und deren Workflows (*ToDo*, *In Progress*, *Done*) als auch der Inhalt des Projekts selbst (Sprints, Stories, Tasks). Diese Informationen werden ausgelesen und dargestellt sowie während der Laufzeit des Taskmonitors ständig aktuell gehalten. Die Aktualisierung wird dabei vom TFS ausgelöst, wie im Folgenden beschrieben. Der Taskmonitor bietet außerdem die Möglichkeit, in begrenztem Maße Änderungen an den Objekten vorzunehmen. Der Status von Tasks, der Taskbearbeiter sowie die aufgewendete Arbeitszeit können angepasst werden.

Auf TFS-Seite kommen zwei Komponenten hinzu. Zum einen ein Plug-in im TFS selbst, das für die Überwachung des TFS nach Änderungen zuständig ist. Hier werden Events abgefangen, die Datenänderungen melden, welche an den Taskmonitor weitergegeben werden. Zum anderen liegt neben dem TFS aber noch ein eigener Service, welcher Zusatzaufgaben übernimmt. **Abbildung 1** zeigt den Aufbau des Gesamtsystems inklusive TFS.

TFS SDK for Java

Der TFS bietet eine Web-Service-Schnittstelle an, mit deren Hilfe die meisten TFS-Funktionen genutzt werden können. Microsoft empfiehlt jedoch ausdrücklich, diese Web Services nicht direkt zu benutzen, sondern über entsprechende SDKs. Neben dem SDK für .NET bietet Microsoft auch ein SDK für Java an. Auch wenn dies im Prinzip ein Nachbau des .NET-SDKs ist, gibt es dennoch einige Unterschiede, die bei der Anbindung einer Java-Anwendung an den TFS ins Gewicht fallen. In unserem Beispiel basiert der Taskmonitor auf dem TFS SDK für Java und bezieht die meisten Informationen über dessen API.

Das SDK kapselt die gesamte Verbindung mit dem TFS und bietet den Zugriff auf die TFS-Daten und -Ein-

Listing 1

```
public class TfsConnection {
  private TFSTeamProjectCollection tfs;
  public void initialize() {
    final UsernamePasswordCredentials user =
      new UsernamePasswordCredentials("tfsuser", "tfspasswd");
    tfs = new TFSTeamProjectCollection(new URI("https://tfsurl/tfs/Collection"), user);
    ...
  }
}
```

Listing 2

```
private TFSTeamProjectCollection tfs; // siehe Listing 1
public List<String> getTaskTypes() {
  List<String> types = new LinkedList<>();
  WorkItemClient wic = tfs.getWorkItemClient();
  Project project = wic.getProjects().get("MeinScrumProjekt");
  for (WorkItemType wit : project.getWorkItemTypes()) {
    types.add(wit.getName());
  }
  return types;
}
```

stellungen über das API an. Microsoft bietet das SDK zum Download unter [1] an.

Nachdem man seine JVM mit einer entsprechenden System-Property (*com.microsoft.tfs.jni.native.base-directory*) versorgt hat, die den Pfad zur benötigten nativen Bibliothek enthält, gestaltet sich der Verbindungsaufbau recht einfach, wie Listing 1 zeigt. *TFSTeamProjectCollection* ist dabei der zentrale Anlaufpunkt für den gesamten Zugriff.

Wie sieht mein Projekt aus?

Die erste Aufgabe für die Anbindung des TFS ist die Ermittlung der Projektmetadaten. Hierzu zählen die dem Projekt zugeordneten Nutzer, die möglichen Status eines Tasks inklusive der möglichen Transitionen, die Tasktypen sowie die vorhandenen Sprints.

Die ersten beiden Punkte sind mit dem Java-SDK nicht so einfach möglich, da es keinen direkten Zugriff auf diese Informationen gibt. Eine Lösung dafür wird in den Abschnitten „Wo sind meine User?“ und „Wie sieht eigentlich mein Workflow aus?“ beschrieben.

Die Typen von Tasks hingegen können ausgelesen werden. Hierzu benötigt man zuerst das Projekt und kann sich von diesem die *WorkItemTypen* geben lassen. Die *WorkItems* in der TFS-Welt entsprechen zum Beispiel Issues in Jira. Diese können verschiedenen Typs sein, haben einen Status und diverse Informationen. Listing 2 zeigt die Ermittlung der Tasktypen über das Projekt, das über den *WorkItemClient* bezogen werden kann. Der *WorkItemClient* ist von zentraler Bedeutung, wenn es um die Arbeit mit *WorkItems* geht.

Die nächste Aufgabe ist die Ermittlung der vorhandenen Sprints. Da diese keine *WorkItems* sind, sondern den Ablauf des Projekts beschreiben, werden sie über eine separate Collection ermittelt. Im TFS werden sie als *Iterations* bezeichnet und können entsprechend ausgelesen werden. Listing 3 zeigt die Ermittlung der Sprints. Die konkreten Informationen aus einem Sprint werden aus dem so genannten *CommonStructureClient* bezogen. **Abbildung 2** zeigt dabei den Aufbau der Iterationen im TFS, die beliebig verschachtelt werden können.

Listing 3

```
private Project project; // siehe Listing 2
private CommonStructureClient commonStructureClient = project
.getWorkItemClient().getConnection().getCommonStructureClient();
public List<String> getSprints() {
  List<String> sprints = new LinkedList<>();
  for (Node irn : project.getIterationRootNodes().getNodes()) {
    for (Node spr : irn.getChildNodes().getNodes()) {
      NodeInfo nodeInfo = commonStructureClient.getNode(spr.getURI());
      String sprintName = irn.getName() + "\\" + spr.getName();
      sprints.add(sprintName);
    }
  }
  return sprints;
}
```

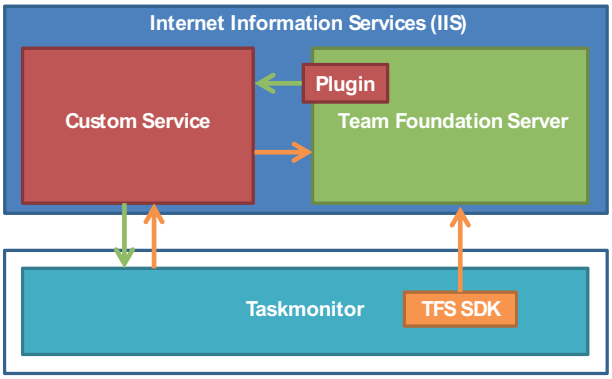


Abb. 1: Architektur des Taskmonitors mit TFS

Iterationen	Startdatum	Enddatum
Scrumboard	15.05.2013	28.06.2013
Release 1	15.05.2013	01.06.2013
Sprint 1	12.05.2013	13.05.2013
Sprint 2	27.05.2013	12.06.2013
Sprint 3	14.06.2013	14.07.2013
Sprint 4	30.06.2013	02.07.2013

Abb. 2: Iterationseinstellungen im TFS mit Projekt, Releases und Sprints

Für das Taskmonitorbeispiel haben wir die Ermittlung der Sprints auf die ersten beiden Ebenen unterhalb des Projekts beschränkt (Release und Sprint).

Wo sind meine Attachments?

Zu den Informationen, die man über ein *WorkItem* erhalten möchte, sei es nun eine Story, ein Bug oder ein Task, gehören die Anhänge. An ein solches Element können beliebige Dateien angehängt werden. Im Taskmonitor soll es möglich sein, sich diese Dateien zu öffnen. Hierzu müssen die Anhänge jedoch zum einen bekannt sein und zum anderen auch vom TFS bezogen werden können.

Ersteres lässt sich laut API einfach durch einen Aufruf von *getAttachments()* am *WorkItem* erreichen. Leider mussten wir feststellen, dass die erhaltene Collection trotz vorhandener Anhänge immer leer ist. Gleiches gilt für die Links (*getLinks()*), die an einem *WorkItem* hängen. Links können dabei normale Hyperlinks sein; es werden jedoch auch Beziehungen zwischen den *WorkItems* damit abgebildet. Hängt ein Task an einer Story, so besitzt der Task einen Link vom Typ *Parent* und die Story einen Link vom Typ *Child*.

Eher durch Zufall fanden wir heraus, dass ein Aufruf der Methode *open()* am *WorkItem* das Problem löst (Obwohl diese im Javadoc mit „Opens this work item for modification“ beschrieben wird). Sowohl Anhänge als auch Links können danach abgerufen werden. Weitere Nachforschungen ergaben, dass dies durchaus kein Fehler oder Bug im SDK, sondern eine Vereinfachung des Verhaltens seitens des SDKs ist. Das .NET-Pendant kann an dieser Stelle mit den abgestuften Öffnungsstatus *none*, *partial*, *readonly* und *full* aufwarten, wodurch eine granularere Steuerung möglich wird [2].

Das zweite Problem bezieht sich auf das Laden der Anhänge zum Anzeigen. Hierfür gibt es im Treiber die Möglichkeit, mit der Methode *downloadToFile()* den Anhang lokal zu speichern oder mit *getURL()* direkt vom Server zu laden. Letzteres ist in einer Serverumgebung reizvoller, da man die Dateien nicht zwischenspeichern und sich auch nicht um das Aufräumen kümmern muss. Der Zugriff auf den URL erfordert jedoch eine Authentifizierung gegenüber dem TFS. An dieser Stelle unterstützt das TFS SDK hilfreich mit einer Funktion für die Ausführung von HTTP-Requests durch den Treiber hindurch. Somit kann der Anhang über den URL unter Ausnutzung der Authentifizierung und Verschlüsselung des SDK bezogen werden. Listing 4 zeigt das Vorgehen. Dabei sind sowohl *GetMethod* als auch der bezogene *HttpClient* Klassen des TFS SDK. Der Client wird wie auch alle anderen bereits beschriebenen Funktionen von *TFSTeamProjectCollection* bezogen. Die Authentifizierung, die beim Erstellen der *TFSTeamProjectCollection*-Instanz ausgeführt wurde, gilt auch für die Ausführung der HTTP-Requests.

Wo sind meine User?

Ein wichtiges Element des Taskmonitors ist das Anzeigen und Zuweisen von Benutzern. Hierfür wird eine Liste der projektbezogenen Nutzer des Team Foundation Servers benötigt. In einer frühen Phase der Entwicklung wurde hierbei bereits festgestellt, dass dies allein mit dem TFS SDK für Java problematisch werden kann. Ein für genau diesen Zweck wichtiger Service steht im TFS SDK für Java nicht zur Verfügung, der *IGroupSecurityService*. Im .NET-Pendant steht dieser Service zur Verfügung. Jedoch war die Suche nach einem funktional ähnlichen Service im Java-SDK erfolglos.

Aufgrund dieses Umstands wurde die Zusatzkomponente Custom Service (**Abb. 1**) erstmalig in Erwägung gezogen und umgesetzt. Der Custom Service stellt hierbei einen REST-basierten Service dar, der uns im Java SDK fehlende Funktionalität für den Taskmonitor verfügbar macht. Daraus ergibt sich, dass diese Komponente rein funktional nur in C# umgesetzt werden konnte. Der Vollständigkeit halber ist deshalb der C#-Code hier mit angeführt (Listing 5).

Innerhalb des Codes wird zu Beginn das gewünschte Projekt mittels des Projektnamens extrahiert. Dies ist notwendig, um später die Menge aller Nutzer des TFS-Servers auf die Menge der Nutzer des jeweiligen Projekts einschränken zu können. In der folgenden Zeile wird nun der bereits erwähnte *IGroupSecurityService* aus der TFS Project Collection initialisiert. Weiterhin wird mithilfe dieses *IGroupSecurityService* und des Project-URI ein Array von Gruppenidentitäten generiert, über die im Folgenden iteriert wird. Zusätzlich wird zuvor noch eine leere Liste vom Typ *Identity* erzeugt, um die gewünschten Nutzer zu speichern.

Da Projekte innerhalb des TFS auch als Identitäten gehalten werden, wird innerhalb jeder Iteration der *foreach*-Schleife wiederum der *IGroupSecurityService*

Listing 4

```
private TFSTeamProjectCollection tfs;
public InputStream getAttachment(String url) throws IOException {
  final HttpMethod attGet = new GetMethod(url);
  tfs.getHTTPClient().executeMethod(attGet);
  return attGet.getResponseBodyAsStream();
}
```

Listing 5

```
Project _project = GetWorkItemStore(projectName).Projects[projectName];
IGroupSecurityService _gss = (IGroupSecurityService)GetTfsTPC().
                                          GetService(typeof(IGroupSecurityService));
Identity[] _projectGroups = _gss.ListApplicationGroups(_project.Uri.ToString());
List<Identity> _projectMembers = new List<Identity>();
foreach (Identity _projectGroup in _projectGroups){
  Identity[] _groupMembers = _gss.ReadIdentities(SearchFactor.Sid, new String[] {
                                      _projectGroup.Sid }, QueryMembership.Expanded);
  foreach (Identity _member in _groupMembers){
    if (_member.Members != null){
      foreach (string _memberSid in _member.Members){
        Identity _memberInfo = gss.ReadIdentity(SearchFactor.Sid,_memberSid,
                                                    QueryMembership.None);
        if (_projectMembers.Contains(_memberInfo) == false){
          if (!_memberInfo.SecurityGroup){
            _projectMembers.Add(_memberInfo);
}}}}}}
```

Listing 6

```
<WORKFLOW>
  <STATES>
    <STATE value="To Do">
      <FIELDS>
        <FIELD refname="Microsoft.VSTS.Common.ClosedDate">
          <EMPTY />
        </FIELD>
        ...
      </FIELDS>
    </STATE>
  </STATES>
  <TRANSITIONS>
    <TRANSITION from="" to="To Do">
      <REASONS>
        <DEFAULTREASON value="New task" />
      </REASONS>
    </TRANSITION>
    <TRANSITION from="To Do" to="In Progress">
      <ACTIONS>
        <ACTION value="Microsoft.VSTS.Actions.StartWork" />
      </ACTIONS>
      <REASONS>
        <DEFAULTREASON value="Work started" />
      </REASONS>
    </TRANSITION>
    ...
  </TRANSITIONS>
</WORKFLOW>
```

zum Auslesen von Gruppenidentitäten innerhalb der jeweiligen Projektidentität über die SID verwendet. Daraufhin wird wiederum über die Menge dieser Gruppenidentitäten iteriert und entsprechende Einzelidentitäten herausgefiltert. Eine wichtige Filterung stellt hierbei die Überprüfung auf *SecurityGroup* dar, denn Microsoft hält innerhalb des TFS eine Vielzahl von virtuellen Nutzern, die mittels dieses Attributs herausgefiltert werden, sodass in der resultierenden Liste nur noch real angelegte Nutzer enthalten sind.

Da dieser kleine Exkurs in die C#-Welt unserer Meinung nach nicht notwendig sein sollte, um diese Funktionalität umzusetzen, haben wir eine Anfrage in einem offiziellen Microsoft-Forum zum Thema TFS SDK für Java gestellt und hoffen auf ein baldiges Statement seitens MS. Dort werden dann auch ggf. weitere Informationen über eine Umsetzung mithilfe des TFS SDKs für Java beschrieben [3].

Wie sieht eigentlich mein Workflow aus?

Von großer Bedeutung ist das Auslesen von Status und Statusübergängen für den Taskmonitor. Hierbei musste wiederum der Custom Service mit etwas C#-Code erweitert werden, um eine Transitions-XML-Datei aus den jeweiligen *WorkItem*-Typen („Bug“ und „Task“) zu erhalten. Die fehlende Funktion im TFS SDK für Java war hierbei schlichtweg eine „Export“-Funktion des *WorkItemType*-Objekts. Auch hier haben wir im Forum von Microsoft nachgefragt, wann und ob diese Funktionalität im TFS SDK für Java bereitgestellt wird [4].

Ungeachtet der Problematik zum Erhalt der XML-Daten stellt der Aufbau der Datei selbst einige Herausforderungen bereit, auf die im Folgenden eingegangen wird. Informationen, die der Taskmonitor benötigt:

1. Liste der Statusnamen
2. Reihenfolge der Status
3. Mögliche Statusübergänge (Transitionen)
4. Bugpriorisierung (Severity)
   a) Reihenfolge
   b) Namen
   c) Standardwert

In Listing 6 sieht man die TFS-XML-Struktur. Hierbei erkennt man die für Punkt 1, 2 und 3 relevanten Informationen: Innerhalb von <*WORKFLOW*> befindet sich eine Auflistung der <*STATES*> wiederum in einzelnen <*STATE*>-Nodes. Hier erhält man durch Parsen der XML-Struktur über das *value*-Attribut die Namen der Status (Punkt 1).

Um die Reihenfolge der Status ausfindig zu machen, muss man den „Start“-Status ausfindig machen.

Innerhalb des <*WORKFLOW*> befindet sich eine weitere Auflistung von <*TRANSITIONS*>, die wiederum einzelne <*TRANSITION*>-Nodes enthält. Der <*TRANSITION*>-Node, der ein leeres *from*-Attribut hat, enthält in seinem *to*-Attribut den *Start*-Status. Von diesem aus kann nun ein Transitionsgraph in Form einer Liste erstellt werden (Punkt 3). Zusätzlich ergibt sich dadurch auch die Reihenfolge der Status (Punkt 2).

Punkt 4 ist in dem von uns verwendeten Process-Template „Microsoft Visual Studio Scrum 2.1“ nur über den Work-Item-Typ *Bugs* erreichbar, da normale „Tasks“ keine Severity als Eigenschaft besitzen (Listing 7).

„Bugs“ enthalten ein zusätzliches <*FIELD*>-Node mit dem *name*-Attributewert *Severity*. Über dieses Attribut ist es leicht möglich, das Element heraus zu parsen und die geforderten Werte aus 4a, 4b und 4c auszulesen.

Wer kommuniziert mit wem?

In einer solchen Umgebung mit drei Hauptbestandteilen (TFS, Custom Service und Taskmonitor) ist es wichtig zu klären, wer eigentlich mit wem wie kommuniziert. Wir haben bereits gesehen, dass der Taskmonitor über das SDK und teilweise auch über den Custom-Service die angezeigten Informationen bezieht. Zwei wichtige

Bestandteile der Kommunikation fehlen jedoch noch. Zum einen muss der Taskmonitor über Änderungen an den Daten auf dem Laufenden gehalten werden. Zum anderen können die Benutzer des Taskmonitors Aktionen am Board ausführen, die natürlich mit dem TFS synchronisiert werden müssen.

Ersteres erledigt das Plug-in im TFS, mit Umweg über den Custom-Service. Das Plug-in registriert Listener im TFS, die auf jede relevante Veränderung reagieren und die entsprechende Logik im Plug-in auslösen. Feuert ein solcher Listener, trifft das Plug-in als Erstes eine Vorauswahl, ob die Änderung relevant ist. Sollte es sich um nicht relevante Änderungen handeln, wird keine Aktion ausgeführt. Zu den nicht relevanten Änderungen zählen zum Beispiel Änderungen an *WorkItems*, die im Taskmonitor nicht angezeigt werden, weil sie im Backlog liegen. Bei relevanten Änderungen meldet das Plug-in diese mit folgenden Informationen an den Custom-Service: ID des *WorkItems*, Eventtyp, auslösender Nutzer, Projekt. Der Custom-Service wiederum filtert die Events je nach Bedarf der angemeldeten Taskmonitore. Ein Taskmonitor meldet sich am Custom-Service mit dem Namen des Projekts an, für das er Informationen erhalten möchte und bekommt in der Folge nur Änderungsmeldungen dieses Projekts. Zusätzlich wird geprüft, welcher Nutzer die Änderung ausgelöst hat. Handelt es sich um den Servicenutzer, mit dem sich die Taskmonitore am TFS anmelden, wurde diese Änderung von einem Taskmonitor aus getätigt. Eine solche Änderung wurde natürlich als Erstes im Model des auslösenden Taskmonitors eingetragen, sodass eine Meldung darüber hinfällig ist. Würde diese Sperre nicht bestehen, käme es zu einer Schleife, da jede Änderungsmeldung vom Custom-Service an den Taskmonitor wiederum eine Änderung am TFS auslösen würde, wodurch die Listener erneut feuern und das Spiel von vorn beginnen würde. Dieser einfache Sperrmechanismus beschränkt die Zahl der Taskmonitore pro Projekt auf eins. Sollten mehr Taskmonitore sich an ein Projekt binden dürfen, käme es zu Inkonsistenzen zwischen den Taskmonitoren, da Änderungen eines Monitors nicht an die anderen gemeldet würden. Sollte Bedarf bestehen, dies anzupassen, müsste die Entscheidungslogik über Senden oder Nichtsenden von Änderungsmeldungen dahingehend erweitert werden, dass auch die auslösende Monitoradresse einbezogen wird (Alternativ kann die Entscheidung auch in die Taskmonitore verlagert werden).

Listing 7

```
<FIELD name="Severity" refname="Microsoft.
            VSTS.Common.Severity" type="String"
                        reportable="dimension">
  <DEFAULT from="value" value="3 - Medium" />
  <ALLOWEDVALUES expanditems="true">
    <LISTITEM value="1 - Critical" />
    <LISTITEM value="2 - High" />
    <LISTITEM value="3 - Medium" />
    <LISTITEM value="4 - Low" />
  </ALLOWEDVALUES>
</FIELD>
```

Listing 8

```
WorkItem task = retrieveTask();
task.open();
task.getFields().getField("Assigned To").
                                setValue("Bob");
task.save();
```

Der zweite Fall der Kommunikation, bei dem Änderungen durch Nutzer des Taskmonitors erfolgen, wird im Taskmonitor wieder durch das TFS SDK abgebildet. Diesmal mit den Schreibfunktionen der *WorkItems*, wobei das Öffnen des *WorkItems* (*WorkItem#open()*) sowie das anschließende Speichern (*WorkItem#save()*) die Aufgabe übernehmen. Zu beachten ist bei solchen Änderungen, dass es sich um „echte" Änderungen handelt. Stellt der TFS fest, dass ein Speicherkommando ausgelöst wurde, obwohl sich kein Wert geändert hat, antwortet er mit einer Exception. Listing 8 zeigt das Setzen des Bearbeiters an einem Task.

Fazit

Microsoft verspricht mit dem TFS SDK für Java die Anbindung von Java-Anwendungen an einen TFS und somit die vollständige Nutzung der .NET-TFS-Funktionalität. In großen Teilen ist dies auch gelungen. Die Anbindung und der Zugriff auf die Daten gelingen recht einfach. Dennoch gibt es ein paar Fallstricke, die es zu umgehen gilt. Manche unklaren Reaktionen des SDK waren zu untersuchen und auszugleichen.

Viele Funktionalitäten des SDK sind sehr bequem, da sie nur mit großem Aufwand selbst implementiert werden können. Dazu zählt insbesondere die Authentifizierung und Sicherung der Verbindung.

Dennoch ist das TFS SDK für Java die Komponente der Wahl, wenn es darum geht, von Java aus mit dem TFS zu arbeiten, auch für Verwendungen, die über unseren Bedarf hinausgehen. Erwähnt sei hier die Ansteuerung des TFS als SCM oder Build-Server. Als Alternative bietet sich noch das Team Foundation Server OData API an [5]. Dieses greift über das .NET-API auf den TFS zu und stellt die Informationen über eine REST-Schnittstelle zur Verfügung. Der Funktionsumfang erreicht jedoch noch nicht die Möglichkeiten einer direkten Nutzung des .NET-SDK.

Thomas Wilk ist Consultant der Saxonia Systems AG mit dem Schwerpunkt auf Client-side Softwareentwicklung, basierend auf Microsoft-Technologien. Er zeichnet sich verantwortlich für die TFS-Seite bei der Anbindung des Taskmonitors.

Denny Israel ist Consultant der Saxonia Systems AG mit dem Schwerpunkt auf Java-basierende Softwareentwicklung. Sein Aufgabenbereich umfasste in diesem Beispiel die Java-seitige Implementierung des vorgestellten Taskmonitors.

Links & Literatur

[1] http://www.microsoft.com/en-us/download/details.aspx?id=22616

[2] http://tinyurl.com/AttachmentsAndLinks

[3] http://tinyurl.com/readUsers

[4] http://tinyurl.com/StatesAndTransitions

[5] https://tfsodata.visualstudio.com/

Kolumne: Knigge für Softwarearchitekten

von Peter Hruschka und Gernot Starke

Der Fahnder – Teil 1

Willkommen zur zweiten Folge unserer Kolumne über Änderung, Evolution und Sanierung bestehender Software. Diesmal fahnden wir nach Softwareverbrechen, Codesünden oder risikoträchtigen Teilen der Software.

Fahndung: Suche mehr als nur Motiv und Gelegenheit

Bei klassischen Delikten haben Fahnder es meistens mit der Einzahl zu tun: ein Täter, ein einziges Verbrechen [1]. Nehmen wir als Beispiel die mutwillig zerschlagene Fensterscheibe: ein Täter, ein Tatwerkzeug, ein beschädigtes Fenster. Bei Software verursachen meistens viele Täter jeweils nur Teile des Gesamtschadens. An einer einzigen Sache werden sozusagen beliebig viele Verbrechen unterschiedlicher Arten begangen. Etwa so, als würde unser Steinewerfer auch noch falsch parken, das Haus mit Graffiti beschmieren und die Haustüre mit Sekundenkleber an ihrem Rahmen fixieren. Statt also klassisch nur nach Motiv und Gelegenheit zu suchen, müssen wir als IT-Fahnder zuerst einmal die begangenen Verbrechen untersuchen:

- **Qualitative Sünden**: Wo werden welche Qualitätsanforderungen verletzt?
- **Funktionale Sünden**: Wo verhält sich ein System fehlerhaft, stellt nicht die notwendige Funktionalität bereit oder arbeitet falsch?
- **Kostensünden**: Wo wird Geld verschwendet – durch überteuerten Betrieb oder aufwändige Wartung/Erweiterung?
- **Codesünden**: Wo steckt unverständlicher, schlechter Code?
- **Architektursünden**: Welche Entscheidungen, Frameworks, Technologien, Schnittstellen oder Strukturen erscheinen mangelhaft?
- **Prozesssünden**: Wo behindern starre oder aufgeblähte Prozesse die effektive Arbeit am System? Welche notwendigen Aufgaben werden vernachlässigt?
- **Managementsünden**: Wo fehlt es an notwendiger Unterstützung?

Wir haben schon Systeme erlebt (ähm – erlitten), an denen sämtliche dieser Sünden und Vergehen in wechselnden Mengenverhältnissen begangen wurden. Übrigens haben wir auch (aber nur sehr wenige) Systeme erlebt, an denen keine relevanten Verbrechen verübt wurden.

Suchen Sie mittels zweier verschiedener Ansätze nach den begangenen Softwareverbrechen eines Systems (in IT-Speak: Probleme, Risiken und technische Schulden): Durch Vernehmung von Opfern und relevanten Zeugen, oder über Spurensicherung am echten System.

Opfer und Zeugen vernehmen

Den Tatort, oder besser den „Gegenstand der Verbrechen“, inspizieren Sie am besten aus unterschiedlichen

Porträt

Peter Hruschka (www.systemsguild.com) und Gernot Starke (innoQ-Fellow) haben vor einigen Jahren www.arc42.de gegründet, das freie Portal für Softwarearchitekten. Sie sind Gründungsmitglieder des International Software Architecture Qualification Board (www.iSAQB.org) sowie Autoren mehrerer Bücher rund um Softwarearchitektur und Entwicklungsprozesse.

@gernotstarke

Perspektiven. Zuerst sollten Sie sich live ein Bild von seinem Zustand machen – am besten haben Sie vorher aus Berichten der Beteiligten den ursprünglichen Zweck des Systems verstanden.

Jetzt sind Sie gut gerüstet für die Vernehmung der ersten Opfer und Zeugen: Befragen Sie Anwender, Betreiber und Auftraggeber. Führen Sie Interviews mit Entwicklern und Testern des Systems bzw. direkter Nachbarsysteme. Fragen Sie nach deren Einschätzung der „schlimmsten Sünden", ob Technologie und Organisation überhaupt zu den Anforderungen passen.

Konzentrieren Sie sich dabei auf die Suche nach Problemen, aber sammeln Sie auch positive Aspekte. Falls Ihre Interviewpartner schon Lösungsansätze vorschlagen, nehmen Sie diese unkommentiert zur Kenntnis – aber verlieren Sie sich jetzt nicht in Lösungsdiskussionen. Es geht um eine Bestandsaufnahme bekannter Schäden, Schwachstellen und Stärken des Systems, nicht um Tatverdächtige. Wir wollen ein möglichst umfassendes Bild von relevanten Stakeholdern, bevor wir später zu Abhilfe oder Therapie kommen.

Spurensicherung

Mit den gezielten Hinweisen der betroffenen Stakeholder können Sie sich an die Spurensuche am System selbst begeben. Wir empfehlen Ihnen dringend, auch hier unterschiedliche Fahndungsansätze zu verfolgen. Zuerst untersuchen Sie die qualitativen Sünden: Die pragmatische Anwendung der erprobten ATAM-Methode [2] hilft Ihnen, die geforderten und real vorhandenen Qualitätseigenschaften des Systems sehr feingranular zu untersuchen. Besonders mögen wir an der Methode, dass sie ohne besondere Werkzeuge auf praktisch alle Arten von Systemen anwendbar ist. Sie lernen daraus viel über die Architektur des Systems. Als Nebeneffekt hilft sie, die Qualitätsanforderungen an Systeme sehr präzise zu formulieren. Wesentliche Voraussetzung für ATAM ist allerdings, dass Sie Architekten oder technische Experten des betroffenen Systems beteiligen können.

Zusätzlich zur qualitativen Betrachtung sollten Sie nun einen intensiven Blick auf den Quellcode werfen: Untersuchen Sie Kopplung, Kohäsion und Komplexität, finden Sie Ausreißer nach oben und unten. Korrelieren Sie die Ergebnisse unbedingt mit organisatorischen Metriken, beispielsweise Wartungs- oder Änderungsaufwänden, der Anzahl gefundener Fehler pro Komponente bzw. Subsystem oder der Anzahl vorhandener Know-how-Träger pro Komponente. Durch diese Untersuchung finden Sie kritischen oder riskanten Code.

Follow the Money

Krimiautoren erklären ihren Lesern oftmals, dass die Fahnder und Kommissare „der Spur des Geldes folgen sollen". Diesen Aspekt der Fahndung sollten Sie auf jeden Fall berücksichtigen. Stellen Sie fest, in welchen Bereichen von Konzeption, Entwicklung, Test und Betrieb für das System welche Kosten entstehen – und wodurch.

Vergleichen Sie dies auch mit dem durch das System erzeugten (finanziellen) Nutzen.

Am besten vergleichen Sie diese Ergebnisse mit anderen Systemen oder Projekten („Benchmarking"), sofern Sie diese Möglichkeit besitzen.

Fazit

Falls Sie auch nur im Entferntesten mit Änderungen, Erweiterung, Sanierung oder Evolution bestehender Software zu tun haben, sollte die Fahndung nach „Softwareverbrechen" (landläufig: „Bewertung", „Audits") zu Ihren regulären Tätigkeiten gehören.

Lassen Sie sich nicht von Gangstern einschüchtern – die meisten begehen in der IT-Branche ihre Sünden ja zum Glück unblutig. Aus den Fahndungsergebnissen können Sie effektive Handlungsempfehlungen ableiten und damit konstruktiv für zielgerichtete Verbesserung sorgen.

Links & Literatur

[1] Clages, Horst (Hrs): „Grundzüge der Kriminalpraxis"

[2] ATAM: Qualitative Bewertungsmethode für Softwarearchitektur, kompakt erklärt in „Effektive Softwarearchitekturen" von Gernot Starke, Hanser-Verlag. Kurzfassung: http://esabuch.de/downloads/download.html, ausführliche Erläuterungen: http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=5177

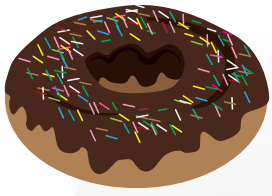
Android wird fünf Jahre alt. Wir sagen „Happy Birthday“ mit einer kleinen Rückschau auf die Android-Evolution.

1.0 Das Betriebssystem Android feierte sein Debüt offiziell im Oktober 2008 auf dem T-Mobile G1. Entwickelt von Andy Rubin kam die erste Version noch ohne viele der Features daher, die inzwischen selbstverständlich geworden sind: Ein On-Screen-Keyboard gab es genauso wenig wie Multitouch und durch den App Store fegte die Steppenhexe.

1.1 Die Plattform wurde erstmals bereits im Februar 2009 aktualisiert. Neben ein paar Bugfixes wurde das Installieren von Updates vereinfacht. Als offenes Betriebssystem war es auf Android von Anfang an möglich, Apps und Widgets jeglicher Art zu nutzen.

1.5 Android 1.5 markierte im April 2009 den ersten Meilenstein. Mit dem Codenamen „Cupcake“ läutete Google die süße Tradition ein, neue Major-Releases des OS in alphabetischer Reihenfolge nach Süßigkeiten zu benennen. Wie bei den kleinen zarten Kuchen, drehte sich auch bei Android Cupcake alles um die schöne Erscheinung: Das User Interface wurde aufpoliert. Hinzu kam das On-Screen-Keyboard und damit das erste Touchscreen-only-Phone (HTC Magic) auf dem Markt.

1.6 Android 1.6 Donut führte den CDMA-Support und die Unabhängigkeit in puncto Auflösung ein. Außerdem wurde die Quick Search Box eingeführt, die es von nun an möglich machte, eine Vielzahl an lokalem Content sowie das Internet gleichzeitig zu durchsuchen. Hinzu kamen Funktionen für Developer, die es ihnen erlaubten, sich einzupluggen, sodass auch ihre Applikationen durchsucht werden konnten.

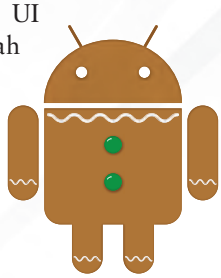


2.0 Im November 2009 kam Android 2.0 Eclair. Multiple-Account-Support, Google Maps Navigation, **2.1** der personalisierte Startbildschirm sowie Livehintergründe und viele weitere Features machten es zum bis dahin spektakulärsten Release seit der Einführung des OS. 2.1 hingegen sollte nicht wegen seiner Updates in die Geschichte eingehen, sondern als die Geburtsstunde des Nexus One.

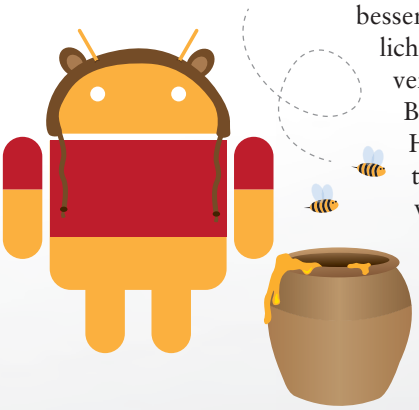
2.2 Dank Mobile-Hotspot-Support konnten Android-Geräte mit Froyo für bis zu acht externe Geräte als mobiler Hotspot fungieren. Den Zugang zum Web stellte das Android-2.2-Smartphone über seine UMTS-Verbindung her und ließ diese von anderen Geräten per USB- oder WiFi-Verbindung mitnutzen.

Erstmalig wird *Macromedia Flash auf Smartphones möglich:* Damit lassen sich jetzt beispielsweise Flash-Videos im Browser ansehen und Flash-Spiele spielen. Auch für Entwickler von Android-Apps entstehen dadurch neue Optionen. Mit Android Cloud 2 Device Messaging Service ließen sich nun Daten oder Hinweise vom PC an das Android-Smartphone senden und eine Zweiwegesynchronisation wurde möglich.

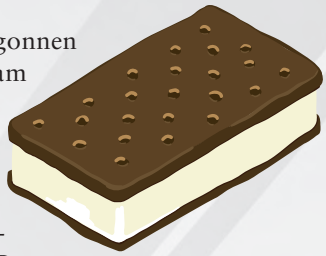
2.3 Ein halbes Jahr nach Froyo kam mit Gingerbread ein eher kleineres Release. Google beauftragte Samsung mit der Produktion des Nexus S, die Stock Widgets wurden aufgefrischt, das UI wurde wieder leicht angepasst und sah nun moderner aus. Mit Voice over IP per SIP wurde Videotelefonie erleichtert. Davon profitierten auch Entwickler, die eine Chatfunktion in ihre Apps integrieren wollten. Die NFC-Unterstützung erlaubte nun erste Datentransaktionen.



3.0 Honeycomb wurde speziell für den Einsatz auf Tablets optimiert und offenbarte ebenfalls eher kleine Updates, die sich als eine Vorarbeit für das folgende Release entpuppen sollten: Im UI wurden verstärkt blaue statt grüne Akzente gesetzt, Widget-Previews verbesserten die Nutzerfreundlichkeit, Multitasking wurde vereinfacht. Physikalische Buttons, wie Back und Home, wurden durch virtuelle ersetzt. Außerdem wurde die Action Bar eingeführt, die Entwickler dazu nutzen konnten, relevante Optionen, Kontextmenüs etc. in ihren Apps anzuzeigen.



4.0 Was in Honeycomb begonnen wurde, fand in Ice Cream Sandwich seine Vervollständigung: einfaches Multitasking, effektive Benachrichtigungen, anpassbare Homescreens, skalierbare Widgets und weitgreifende Interaktivität. Android Beam machte es auf Basis von NFC möglich, Daten auszutauschen, indem man einfach die entsprechenden Phones aneinander hält. Face Unlook war eine nette Spielerei, die Data-Usage-Analyse machte den Datenverbrauch einsehbar.

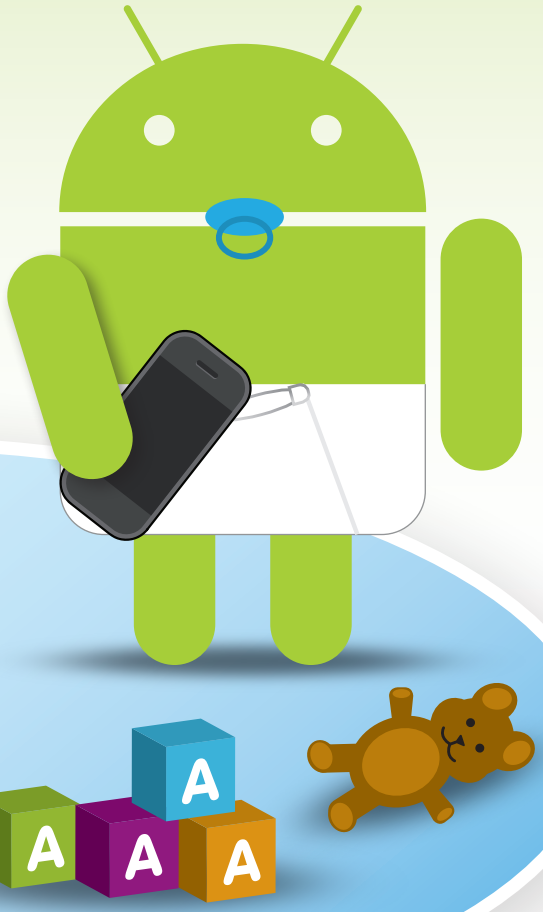


4.X *Project Butter* sorgte in Jelly Bean dafür, dass Android nun unter anderem dank Triple Buffering und VSync eine bessere Performance hatte und Apps deutlich schneller starten ließ. Das Platzieren von Widgets auf dem Homescreen wurde noch intuitiver. Auch die Benachrichtigungsleiste wurde um neue Funktionen wie erweiterte Gestensteuerung und direkte Rückrufe erweitert. Google Now hieß der persönliche Assistent, der von nun an nicht nur an bevorstehende Termine erinnerte, sondern auf Basis des ermittelten Standorts und der in Maps ermittelten Verkehrssituation zum Beispiel auch darüber informierte, wann man am besten losfährt, wie das Wetter am Zielort wird und was es dort sonst noch gibt.

4.4 Über die Features, die mit Android KitKat ausgerollt werden, ist bisher nichts Offizielles bekannt. Es soll im Oktober released werden, so viel steht zumindest fest. Der Rest bleibt Spekulation: KitKat soll niedrigere Spezifikationsanforderungen haben und damit all den Usern Rechnung tragen, die mit ihren älteren Devices noch auf Version 2.3 festsitzen. Weiter wird vermutet, Google wolle ein Pendant zu Apples iCloud etablieren und mehr Nutzen aus Multi-Core-CPUs schlagen, um so die Anforderungen an die Hardware, speziell die Batterie, zu senken. Fraglich ist, ob sich im Google Play etwas ändern wird, sprich, ob Google den kritischen Rufen nach besseren Sicherheitschecks von Apps folgen wird.

Evolution einer 7-Zoll-Ikone

Nexus 7 – Die nächste Generation



Mit dem Nexus 7 ist es Google und Asus 2012 gelungen, das Taschenbuchformat (7 Zoll) für Tablets erfolgreich zu etablieren. Galt der 7-Zoll-Formfaktor zuvor als „zu klein", um als ernstzunehmendes Tablet wahrgenommen zu werden, zogen etliche Hersteller aufgrund des Erfolgs des Nexus 7 bald nach. Im Juli hat der 7-Zoll-Pionier nun eine neue Version des Nexus 7 vorgestellt – bloße Modellpflege oder gibt es wirklich Neues zu vermelden? Im Folgenden möchte ich einen Überblick und Erfahrungsbericht über das Nexus 7 (2013) und das zugehörige Android-4.3-Release nach einer ersten vierwöchigen Nutzungsphase geben.

von Christian Meder

Im Laufe des vergangenen Jahres habe ich den 7-Zoll-Formfaktor immer stärker als meine persönliche Tabletgröße schätzen gelernt. Nach meinem ersten Erfahrungsbericht [1] wurde das Nexus 7 zum regelmäßigen Begleiter in Besprechungen, auf Reisen im öffentlichen Personennahverkehr und in Hotels, auf Konferenzen sowie beim Lesen auf der Couch. Auch die Nutzungsszenarien wurden im Laufe des Jahres vielfältiger: das Aufzeichnen von Notizen in Besprechungen und Vorträgen, die Pflege der ständig präsenten To-do-Liste, die schnelle Möglichkeit, E-Mails und Chatnachrichten nahezu jederzeit zu lesen und kurz zu beantworten, die permanente Verfügbarkeit meiner wichtigsten Nachrichtenquellen und der relativ großen Sammlung an IT-Büchern, der Zugriff auf unsere Musik- und Videosammlung auf unserem Familienfestplattenspeicher (NAS) daheim, das schnelle SSH-Terminal für Notfälle, das spontane Fotoalbum oder auch das Entertainmentsystem für die Kinder.

Zugegeben, die meisten beschriebenen Szenarien sind auch im Zusammenhang mit Smartphones, Phablets und 10-Zoll-Tablets denkbar, aber eben nicht ganz so uneingeschränkt nutzbar und ermüdungsfrei auch bei längerer Nutzung.

Als zufriedener Nutzer eines Nexus 7 Jahrgang 2012 empfand ich das neue Nexus 7 aufgrund der veröffentlichten Spezifikationen zwar als willkommenes Hardwareupgrade, aber nicht als weltbewegendes Must-have, da das Android-4.3-Upgrade ja auch bereits auf dem alten Nexus 7 verfügbar ist.

Harte Fakten

Die erste Neuerung nach dem Auspacken des Nexus 7 ist sofort spürbar: Es ist leichter, dünner und schmäler geworden. Immerhin 50 g weniger wiegt das neue Nexus 7 mit seinen 290 g im Vergleich zum Vorgänger. Die Breite des neuen Geräts hat sich um 0,6 cm verringert. Es ist minimal (1,5 mm) länger und 2 mm dünner als das erste Modell. Obwohl diese Änderungen auf dem Papier gering wirken, sind sie gerade im direkten Vergleich sehr deutlich wahrnehmbar, und das neue Nexus 7 wirkt dadurch weniger klobig und eleganter [2], [3].

Insgesamt ist das Tablet etwas gestreckter, was durch den breiten Rand am oberen und unteren Ende unterstrichen wird. Die Rückseite besitzt nicht mehr die geriffelte Textur des Vorgängers, sondern besteht aus einer matten, glatten, aber sicher zu greifenden Kunststoffoberfläche.

Durch dieses neue Profil lässt sich das Tablet sowohl in vertikaler Ausrichtung gut mit einer Hand umfassen als auch in horizontaler Ausrichtung mit beiden Händen aufgrund des breiten Randes als Daumenauflagefläche leicht halten.

Weiterhin besitzt das neue Nexus 7 eine LED für Benachrichtigungen, ein Vibrationsmotor ist allerdings auch in der neuen Version nicht enthalten. Das Tablet ist sehr gut verarbeitet, der einzige Kritikpunkt sind die etwas unpräzisen Tasten fürs Ein-/Ausschalten und die Lautstärke, aber das ist Jammern auf sehr hohem Niveau.

Die Prozessorsituation ist etwas verwirrend: Der Prozessor ist zwar mit Snapdragon S4 Pro bezeichnet, es handelt sich aber um einen modifizierten und heruntergetakteten Snapdragon 600 mit vier auf 1,5 GHz getakteten Krait-300-Kernen von Qualcomm. Kombiniert wird das Ganze mit 2 GB PCDDR3L-Hauptspeicher und je nach Ausstattung 16 oder 32 GB Flash-Speicher. Bei der verwendeten Grafikeinheit handelt es sich um eine Adreno 320 GPU. Nachgemessen ist die Geschwindigkeit des Prozessors damit um den Faktor 1,8 verbessert im Vergleich zum alten Nexus 7, die Geschwindigkeit der Grafikeinheit sogar um den Faktor 4 (ausführliche Benchmarks hierzu finden sich in [4] und [5]).

Was bedeutet das alles nun konkret für den Nutzer? Das Nexus 7 ist damit aktuell das schnellste 7-Zoll-Tablet und kommt in einigen 3-D-Benchmarks sogar an die Werte des Apple iPad 4 heran.

Schaltet man das Tablet ein, erkennt man sofort, warum der Bildschirm als weiteres Highlight des neuen Nexus 7 gepriesen wird: Farben, Helligkeit und Auflösung beeindrucken auf den ersten Blick. Betrachtet man die zugehörigen Spezifikationen und Messwerte des Bildschirms [4], zeigen sich auch sofort die Gründe für den ersten Eindruck. Der Bildschirm besitzt eine HD-Auflösung mit 1 200 x 1 920 Bildpunkten, das macht 323 ppi (pixel per inch) und ist damit der aktuell höchstauflösende Bildschirm in einem 7-Zoll-Tablet. Die Leuchtdichte ist stark verbessert im Vergleich zum Nexus 7 von 2012 und gehört mit zu den leuchtstärksten Bildschirmen im Vergleich mit allen anderen aktuellen Tablets. Auch die Farbtreue des Bildschirms ist in den Messungen hervorragend. Der erste Eindruck trügt daher in diesem Fall nicht: Es steckt ein hochauflösender, leuchtstarker und farbtreuer Bildschirm im neuen Nexus 7, der aktuell beste verfügbare Bildschirm in dieser Größe.

Abb. 1: Nexus 7 (2013) (Bild: Pressefoto von Google)

Hatte das Nexus 7 2012 noch keine Möglichkeiten, einen externen Bildschirm zusätzlich anzuschließen, gibt es mit der

Abb. 2: Screenshot Java-Magazin-App auf Nexus 7

neuen Version gleich zwei Optionen. Zum einen kann über einen Slimport Adapter der MicroUSB-Port per HDMI mit einem Ausgabegerät verbunden werden. Zum anderen kann man per Wireless Display/Miracast ein Ausgabegerät drahtlos anbinden [6]. Neu ist auch, dass im Querformat auf beiden Seiten Stereolautsprecher zum Einsatz kommen statt des sonst üblichen einzelnen Monolautsprechers.

Die Batterie ist zwar kleiner dimensioniert als beim alten Nexus 7, bietet aber trotzdem im Praxistest sogar etwas längere Laufzeiten im Vergleich zum 2012er Modell. Generell hält das neue Nexus 7 bei durchschnittlicher Nutzung einen kompletten Tag durch mit Laufzeiten jenseits von elf Stunden. Aufwändige 3-D-Spiele sorgen allerdings dafür, dass der Akku schon nach etwas mehr als vier Stunden schlapp macht.

Im Gegensatz zum Vorjahresmodell ist das Nexus 7, wie bereits andere 7-Zoll-Tablets, jetzt auch mit einer 5-MP-Kamera ausgestattet. Neu ist auch, dass der Qi-Standard für das drahtlose Aufladen des Akkus unterstützt wird.

In Sachen Konnektivität wird jetzt im WLAN-Bereich auch das 5-GHz-Frequenzband unterstützt. Und das Modell mit Mobilfunkunterstützung wurde um die Verbindungsmöglichkeiten per LTE erweitert. Die LTE-Ausführung unterstützt alle Netze und Frequenzbänder, die in Deutschland verwendet werden [7].

Die Software

Auf der Android-Seite wird das neue Nexus 7, wie alle Geräte der Nexus-Reihe, mit einem nativen und unveränderten Android ausgeliefert, in diesem Fall mit Version 4.3. Diese Version wurde auch bereits im Juli als Update auf die 2012er-Geräte Nexus 7, Nexus 4 und Nexus 10 ausgeliefert.

Im Vorfeld wurde bereits über einige Neuerungen spekuliert [8], Bluetooth Low Energy vorgestellt [9] und das Release aus Enterprise-Sicht unter die Lupe genommen [10]. Daher möchte ich hier noch kurz einige Neuerungen aus Nutzersicht zusammenfassen [11].

Die Telefon-App unterstützt in 4.3 nun die Autovervollständigung von Telefonnummern und Kontakten während der Eingabe beim Wählen. Diese Funktion muss allerdings explizit in den Einstellungen aktiviert werden.

Für die vereinfachte Standorterkennung gibt es mit 4.3 die zusätzliche Möglichkeit, auch bei abgeschaltetem WLAN selbiges kurzzeitig batterieschonend zur Positionsbestimmung zu aktivieren. Neben dem erwähnten Bluetooth Low Energy auf passender Hardware wird auch Bluetooth AVRCP 1.3 unterstützt. Damit wird beispielsweise in Autoradios der aktuell abgespielte Musiktitel eines per Bluetooth verbundenen Android-4.3-Geräts angezeigt.

Ebenfalls eingeführt wurde ein Virtual Surround Sound mithilfe der Fraunhofer *Cingo*-Technologie bei Stereowiedergabe über Lautsprecher und Kopfhörer. Hierbei wird durch das Anwenden digitaler Filter virtuell ein räumliches Tiefengefühl erzeugt. Voraussetzung ist allerdings ein vorhandener Mehrkanalton in den Audioquellen.

Wie in [11] bereits aus Enterprise-Sicht erläutert, werden mit 4.3 eingeschränkte Nutzerkonten eingeführt. Häufig wird diese Funktionalität auch mit einer typischen Mehrbenutzer- bzw. Eltern-/Kind-Situation in Verbindung gebracht. Aus aktueller Sicht bieten eingeschränkte Nutzerkonten allerdings noch Schlupflöcher, vor allem im Konfigurationsbereich, und sind daher eher als Technologievorschau für Entwickler zu empfehlen, aber nicht als komplett abgesicherte Konten.

Indirekt ist die Verwendung von *fstrim* im Dateisystem eine willkommene Verbesserung. Mit einem *trim*-Befehl wird bei modernem Flash-Speicher der Controller dabei unterstützt, freigegebene Speicherbereiche auch als ungenutzt zu markieren, damit die Inhalte bei Überschreiben nicht unnötig gesichert und umkopiert werden. Vor allem bei einem vollen oder nahezu vollen Dateisystem bricht ohne *trim*-Unterstützung die Schreibgeschwindigkeit massiv ein. Zu beobachten war dies in der Vergangenheit beim Nexus 7 mit 8 GB Kapazität, das schnell mit Daten gefüllt war, aber anfangs ohne *trim*-Unterstützung in der Schreibrate sehr stark einbrach und sich auch bei Freigabe von

Speicher nicht wieder beschleunigte. Android 4.3 sendet unter bestimmten Rahmenbedingungen nun *fstrim*-Befehle an den Controller und gibt damit alle nicht verwendeten Bereiche wieder zur weiteren Verwendung frei: quasi eine Garbage Collection auf dem Flash-Speicher einmal pro Tag in einem normalen Nutzungszyklus. Die Benchmarks [5] stützen den Erfolg dieses Ansatzes.

Der Grafikstack beherrscht mit 4.3 das Umsortieren und Zusammenführen von OpenGL-Operationen sowie das parallele Abarbeiten mehrerer Grafikoperationen (*Multi-Threading*) bei passender Hardware [8].

Abb. 3: Nexus 7 per Miracast verbunden (Empfänger unten rechts)

Zusätzlich erlaubt der Grafikstack nun optional OpenGL ES 3.0 auf passender Hardware für Spiele und fordernde grafische Anwendungen. Allerdings sind, nach den Erfahrungen aus der Community zu urteilen, noch einige Bugs in der Treiberunterstützung zu umgehen [12].

Für Entwickler gibt es nun einen offiziellen Weg, über den *NotificationListenerService* an alle Benachrichtigungen zu gelangen, um diese beispielsweise auf einer Smart Watch darzustellen. Bisher gab es nur den Workaround über Barrierefreiheitfunktionalität an die Benachrichtigungen zentral anzudocken. Weitere Erweiterungen aus Entwicklersicht sind in [13] beschrieben.

Eindrücke

Obwohl ich mit meinem 2012er Nexus 7 sehr zufrieden bin, wurde ich trotz meiner Skepsis positiv überrascht von der neuen Version des Nexus 7. Der Bildschirm allein liefert im direkten Vergleich einen plausiblen Grund für ein Update. Die hohe Auflösung des Bildschirms wird vor allem bei Texten außerordentlich präsent, und die gestochen scharfe Schrift macht das Lesen von elektronischen Büchern zu einem Vergnügen auf dem Gerät.

Bei Filmen und Fotos muss man schon etwas genauer hinschauen, um die Nuancen zu bemerken, aber bei Schriften wird der Unterschied sofort deutlich. Daneben erleichtert die Leuchtdichte des neuen Bildschirms die Nutzung im Freien erheblich. Nach der Verwendung des neuen Nexus 7 hatte ich häufiger das Gefühl, dass bei anderen Tablets ein Schatten über dem Bildschirm liegt.

Durch die etwas schmälere Bauform kann ich das 2013er Gerät noch besser mit einer Hand im Porträtmodus halten und in die Gesäßtasche passt es jetzt immer problemlos. Das geringere Gewicht ist deutlich zu spüren und ein angenehmer Nebeneffekt, wenn auch nicht ausschlaggebend.

Die CPU- und GPU-Upgrades führen dazu, dass selbst unter der Last von Hintergrundtasks keine Leistungseinbrüche zu bemerken sind.

Viel wichtiger für mich waren allerdings die neuen Anschlussmöglichkeiten an externe Bildschirme und die Verbesserungen der Konnektivität. Die Verbindung des Nexus 7 mithilfe des Miracast-Empfängers hat auf Anhieb funktioniert, war problemlos und auch im Betrieb stabil.

Allerdings war die Bildqualität auf einem HD-Fernseher im Vergleich zu einer Verbindung mit dem Slimport-Adapter schlechter, verständlich vor dem Hintergrund, dass bei Miracast das Signal in der Regel vom Tablet nochmals neu kodiert und dann erst an den Fernseher übertragen wird [6].

Beim Anschluss per Slimport-Adapter an einen HD-Fernseher gab es ebenfalls keine Probleme: brilliantes Bild, eine einfache und problemlose Kopplung.

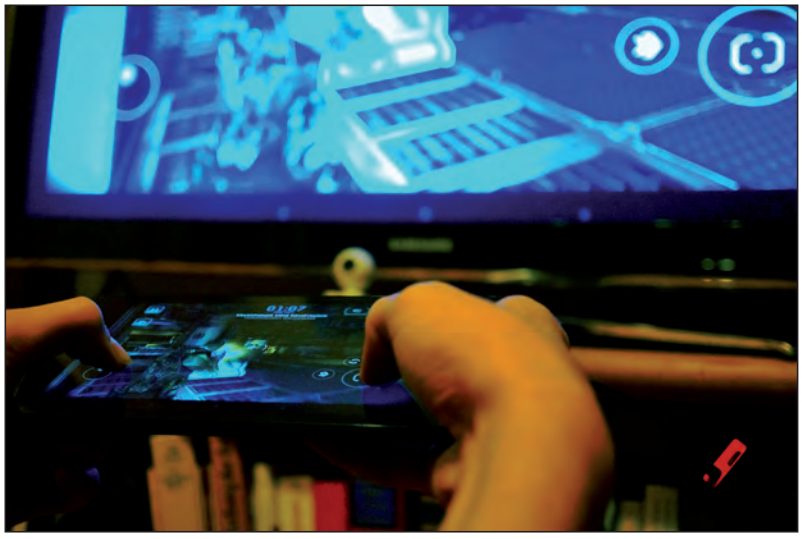


Abb. 4: Nexus 7 per Slimport-Adapter verbunden

Abb. 5: Bild mit Kamera des Nexus 7 erstellt

Vor dem Hintergrund, dass das 2012er Nexus 7 gar keine Anschlussmöglichkeiten für externe Bildschirme bietet, ist das neue Nexus 7 hier ein großer Fortschritt.

Sehr praktisch ist auch die Nutzung des 5-GHz-Frequenzbands im WLAN-Bereich. Auch wenn objektive Messungen hier schwierig sind, ist in städtischen WLAN-Szenarien jede weitere verfügbare Frequenz von Vorteil.

Viel interessanter war natürlich ein erster Test des Nexus 7 im LTE-Netzwerk von Vodafone. Nach der Aktivierung des LTE-Tarifs funktionierte die Verbindung auch auf Anhieb. Die Datenraten lagen in der Regel etwa im Bereich um 25 Mbit in Empfangs- und Senderichtung. Auf jeden Fall hoch genug, sodass bei normalem Browsen oder Downloads kein Unterschied zu einem guten WLAN wahrnehmbar ist. Eher im Gegenteil: In den meisten Hotel-WLANs ist man geneigt, das WLAN im Nexus 7 zu deaktivieren und nur LTE zu nutzen.

Aus der Handvoll Reisen mit LTE auf dem Nexus 7 stellten sich zwei Dinge heraus: Oft ist weit auf dem Land, fernab der Ballungszentren nur EDGE, aber auch LTE vorhanden. Eine durchaus interessante Erfahrung, mitten im Nirgendwo auf einmal eine Breitbandverbindung zu haben.

Zweitens ist in den Städten in der Regel auch überall LTE in guter Qualität vorhanden. Vor allem auch auf Konferenzen in Gebäuden war der Empfang ausgezeichnet. Hier hat sich auch das Tethering des Laptops über LTE bewährt.

Zusammenfassend lässt sich meine Erfahrung folgendermaßen: Die Netzabdeckung mit einer Breitbandverbindung (LTE, HSDPA oder UMTS) hat sich durch die Nutzung von LTE erheblich erhöht. Lücken im Netz bleiben aber selbstverständlich (leider) nach wie vor.

Weitere neue Funktionen des Nexus 7 sind eine nette Ergänzung, aber meiner Meinung nach kein Grund für ein Upgrade oder eine Kaufentscheidung. Kameras in Tablets sind in der Regel unhandlich und nur für Schnappschüsse zu gebrauchen, auch wenn die Bildqualität des Nexus 7 dem ersten Eindruck nach in Ordnung geht. Und auch Virtual Surround oder das Laden per Qi-Standard sind für sich allein gesehen nicht entscheidend.

Fazit

Während das Nexus 7 von 2012 eine neue Gerätekategorie mit einem sehr attraktiven Preis erschlossen hat, positioniert sich das neue Nexus 7 in mehrfacher Hinsicht an der Spitze: Der Bildschirm ist der beste im 7-Zoll-Bereich in puncto Leuchtdichte, Auflösung und Farbtreue, die Kombination aus Prozessor und Grafikeinheit gehört ebenfalls zur Spitze in diesem Segment, und WLAN-Durchsatz und LTE bewegen sich auch im High-End-Bereich. Trotz dieser Spitzenwerte hat das Nexus 7 aber wichtige Eigenschaften beibehalten: den attraktiven Preis, den weiter optimierten Formfaktor, die lange Batterielaufzeit und die unmodifizierte, aktuelle Android-Plattform.

Da die Tabletverkaufszahlen der letzten Quartale einen klaren Trend zu kleineren Tablets ausweisen, ist das Nexus 7 damit aktuell bestens aufgestellt, die Referenz für das derzeit wichtigste Tabletsegment zu bilden. Und es wird nun sehr schwer für mich werden, von einem neuen Nexus 7 wieder zu einem anderen Tablet zu wechseln.

Christian Meder ist CTO bei der inovex GmbH in Pforzheim. Dort beschäftigt er sich vor allem mit leichtgewichtigen Java- und Open-Source-Technologien sowie skalierbaren Linux-basierten Architekturen. Seit mehr als einer Dekade ist er in der Open-Source-Community aktiv.

Links & Literatur

[1] Meder, Christian: „Nexus 7: Das Taschenbuch unter den Tablets“, in Mobile Technology 4.2012

[2] http://www.google.de/nexus/7/

[3] http://www.androidnext.de/tests/nexus-7-2013/

[4] http://www.anandtech.com/show/7176/nexus-7-2013-mini-review

[5] http://www.anandtech.com/show/7231/the-nexus-7-2013-review

[6] Bälz, Daniel; Meder, Christian: „Android für Couch Potatoes“, in Java Magazin 7.2013

[7] http://www.androidmag.de/news/technik-news/lte-modell-des-neuen-nexus-7-unterstutzt-frequenzen-der-deutschen-netzanbieter/

[8] Meder, Christian: „Android ist anders“, in Java Magazin 9.2013

[9] Röwekamp, Lars; Limburg, Arne: „Bluetooth revisited“, in Java Magazin 10.2013

[10] Peuker, Jan: „Android 4.3 – jetzt aber sicher“, in Java Magazin 10.2013

[11] http://www.android.com/about/jelly-bean/

[12] https://dolphin-emu.org/blog/2013/09/26/dolphin-emulator-and-opengl-drivers-hall-fameshame/

[13] http://developer.android.com/about/versions/jelly-bean.html

Was Android-Entwickler darüber wissen sollten

Ressourcen und andere XML-Files

Wer mit der Android-Entwicklung startet, wird zunächst von einer Flut an XML-Dateien erschlagen. Dabei bietet diese Form der Konfiguration bei näherer Betrachtung und bei richtiger Anwendung eine gute Mischung aus Übersichtlichkeit und Struktur. Macht man sich allerdings über dieses generelle Vorgehen einmal näher Gedanken, kommen sofort einige Fragen auf: Vergrößert diese Flut an XML-Dateien das fertige APK nicht unnötig und ist XML-Processing nicht unglaublich langsam? Welche XML-Ressourcen gibt es überhaupt? Und wie kann ich eigene XML-Dateien definieren? Auf diese und weitere Fragen liefert diese Kolumne Antworten.

von Arne Limburg

Um die brennendsten Fragen zuerst zu beantworten: Das viele XML macht Android-Anwendungen weder langsam noch führt es zu überhöhtem Speicherverbrauch. Dies liegt daran, dass alle XML-Dateien, die sich im Ordner *res* eines Android-Projekts befinden, bereits beim Kompilieren in Binärdateien umgewandelt werden. Auf diese Weise gelingt es, die Übersichtlichkeit von XML-Files zur Entwicklungszeit mit der hohen Performance und dem geringen Speicherverbrauch von Binärdateien zur Laufzeit zu kombinieren.

Beim Erzeugen der Binärdateien entsteht im Übrigen ein weiteres Artefakt, das Android-Entwicklern schon bekannt sein dürfte, nämlich die Klasse *R*. Diese enthält für jeden Unterordner des Ordners „res“ eine innere Klasse, die wiederum Konstanten für den Zugriff auf die Binärinformationen enthalten, die beim Kompilieren entstanden sind. So kann man z. B. im Java-Code auf das Element *app_name* aus einer XML-Datei in dem Ordner *res/drawable* zugreifen, indem man die Konstante *R.drawable.app_name* verwendet. Zusätzlich wird in *R* eine innere Klasse namens *id* generiert, die alle IDs aus den XML-Dateien enthält. Auf die ID *my_id* kann folglich über die Konstante *R.id.my_id* zugegriffen werden.

Die einzigen Dateien, die beim Kompilieren nicht in Binärdateien umgewandelt werden, sind alle Dateien, die sich in den Ordnern *res/raw* und *res/xml* befinden. Während *res/raw* für Dateien jeglichen Typs gedacht ist, ist der Ordner *res/xml*, wie der Name schon sagt, speziell für XML-Dateien gedacht, die nicht von Android in Binärdateien umgewandelt werden sollen. Für diese Dateien bietet Android XML-Parsing Out of the Box. Für beide Ordner erzeugt Android auch eine Unterklasse in der Klasse *R*, sodass man z. B. auf eine Datei namens *myResource.xml*, die im Ordner *res/xml* liegt, im Code über die Konstante *R.xml.myResource* zugegriffen werden kann.

Android-Resource-XML-Files

Für jede Art von Ressource bietet Android ein eigenes XML-Schema [1]. Dies bietet den Vorteil, dass die XML-Dateien auf den jeweiligen Anwendungsfall direkt zugeschnitten sind. Ein Beispiel dazu ist, dass das XML-Schema für Styleinformationen dann auch nur stylespezifische Tags enthält.

Im Sinne von Convention over Configuration sind im Ordner *res* die Unterordner *anim*, *color*, *drawable*, *layout*, *menu* und *values* definiert, in denen die jeweiligen Ressourcen abgelegt werden können und dann von Android automatisch gefunden werden. Dabei ist für die Strukturierung der jeweiligen Ressourcen wichtig, dass die Dateinamen beliebig gewählt werden können.

Die jeweiligen Ordnernamen können durch so genannte Qualifier ergänzt werden, um sprach-, display-, größen- oder orientierungsspezifische Ressourcen abzulegen (z. B. *res/values-de* oder *res/drawable-hdpi*). Qualifier können dabei kombiniert werden [2].

Im Ordner *anim* werden dabei Animationskonfigurationen abgelegt. Eine Ausnahme bilden hier Frame-Animationen, also eine Animation, die einfach aus einer Liste von Drawables besteht, die der Reihe nach abgespielt werden. Diese Listen sind selbst wieder Drawables und müssen daher im Ordner *drawable* abgelegt werden.

Letzterer kann eine Vielzahl von Ressourcen (z. B. JPEG-, PNG- oder XML-Dateien) enthalten, für Animationen, Transformationen oder Listen von Dateien für verschiedene Zustände (Knopf gedrückt oder losgelassen). Eine komplette Liste bietet [3].

Die Ordner *layout*, *menu* und *color* enthalten jeweils wieder nur eine Art von Ressourcen, nämlich, wie der Name schon sagt, Layout-, Menü- und Farb-XML-Konfigurationen. Generell gilt hier, dass eine Layoutdatei immer einer Activity zugeordnet ist. Zum Zweck der Strukturierung und Wiederverwendung können aber Teile eines Layouts in separate XML-Dateien ausgelagert und via <*include*>-Tag eingebunden werden.

Der Ordner *values* enthält wiederum eine Sammlung aller weiteren benötigten Ressourcen, wie String-Ressourcen, Style-Ressourcen, Color-Ressourcen, Dimension-Ressourcen, Boolean-Ressourcen, Integer-Ressourcen und weitere [4].

Eigene XML-Definitionen verwenden

Möchte man ein eigenes XML-Schema verwenden, um z. B. applikationsspezifische Konfigurationen vornehmen zu können, muss man selbst Hand anlegen. Android bietet hier natürlich nicht die Möglichkeit, diese in Binärdateien umzuwandeln. Damit Android die XML-Datei dann auch als XML in das APK packt, muss die Datei im Ordner *res/xml* abgelegt werden. Diese kann dann über *Resources.getXML* ausgelesen werden. Man erhält so direkt einen *XmlResourceParser*. Hierbei handelt es sich um eine Unterklasse des *XmlPullParsers* [5], einen eventbasierten Parser, mit dem man die Datei auslesen kann. Möchte man eine eigene Bibliothek zum Parsen von XML verwenden, muss die Datei in *res/raw* abgelegt und dann mit *Resources.openRawResource* ausgelesen werden.

Eine XML-Konfiguration für Changelog-Einträge

Ein Beispiel für eine eigene Viewkomponente, die über ein spezifisches XML konfiguriert werden kann, hat Gabriele Mariotti [6] geschrieben und Open Source veröffentlicht. Die Idee dabei ist, dass für jede neue Version einer App eine Auflistung der Änderungen sinnvoll ist. Daher hat Mariotti ein spezifisches XML-Schema definiert, um solche Changelog-Einträge zu konfigurieren. Diese werden dann automatisch ausgelesen und in einer Ansicht dargestellt.

Diese kann dann auch über verschiedene Einstellungsmöglichkeiten noch angepasst werden. Da es sich bei der Komponente um eine eigene View handelt, kann sie sowohl in Activities als auch Fragments eingesetzt und natürlich auch in ein eigenes Layout eingebunden werden, falls sie nicht den gesamten Bildschirminhalt ausfüllen soll. Zusätzlich unterstützt sie Mehrsprachigkeit und auch HTML-Text Markup.

Eine Changelog-View kann dabei über <*changelog*>-Tags definiert werden (Listing 1). Darin gibt es dann pro Version einen <*changelogversion*>-Tag, in dem dann wiederum <*changelogtext*>-Tags geschachtelt werden können.

Zur Unterstützung von Mehrsprachigkeit genügt es, die Changelog-XML-Datei in jeder Sprache anzulegen und in dem jeweils zugehörigen Ordner (z. B. *res/raw-de*) abzulegen.

Listing 1

```
<?xml version="1.0" encoding="utf-8"?>
<changelog bulletList="true">
  <changelogversion versionName="1.1" changeDate="Oct 20,2013">
    <changelogtext>[b]New[/b] stunning features</changelogtext>
    <changelogtext>minor bugfixes</changelogtext>
  </changelogversion>
  <changelogversion versionName="1.0" changeDate="Oct 06,2013">
    <changelogtext>Initial release.</changelogtext>
  </changelogversion>
</changelog>
```

Fazit

XML-Dateien eignen sich gut, um gewisse Konfigurationen vorzunehmen. Dem Nachteil, dass sie verhältnismäßig groß sind und dass das Parsen lange dauert, begegnet Android damit, dass die Standardkonfigurationsdateien beim Kompilieren in Binärdateien übersetzt werden. Dadurch wird die Übersichtlichkeit von XML-Files mit der Performance und dem geringen Speicherplatz von binären Konfigurationsdateien kombiniert.

Durch die Möglichkeit einer beliebigen Wahl des Dateinamens bei z. B. String-Ressourcen und des Auslagerns von Layoutteilen ist außerdem die Chance einer guten Strukturierung der XML-Dateien gegeben.

Benötigt man für die eigene Anwendung plain XML, kann man dieses in *res/xml* ablegen. Es wird zwar leider nicht in eine Binärdatei kompiliert, dennoch bietet es eine gute Möglichkeit, eigene Konfigurationen in XML unterzubringen. Diesen Mechanismus macht sich das vorgestellte Projekt von Mariotti zunutze. Es bietet eine sinnvolle Komponente, mit der ein Changelog angezeigt werden kann, das über eine XML-Konfiguration befüllt wird.

Arne Limburg ist Softwarearchitekt bei der open knowledge GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

@ArneLimburg

Links & Literatur

[1] http://developer.android.com/guide/topics/resources/available-resources.html

[2] http://developer.android.com/guide/topics/resources/providing-resources.html

[3] http://developer.android.com/guide/topics/resources/drawable-resource.html

[4] http://developer.android.com/guide/topics/resources/more-resources.html

[5] http://developer.android.com/reference/org/xmlpull/v1/XmlPullParser.html

[6] http://gmariotti.blogspot.de/2013/08/an-android-library-to-display-your.html

Mobile und Desktopanwendungen mit der ArcGIS Runtime entwickeln

Die Welt entdecken

Geoinformationssysteme haben in den letzten Jahren, gerade im mobilen Umfeld, sehr an Bedeutung gewonnen. Beispielhaft sei hier nur Google Maps erwähnt, das die Darstellung von georeferenzierten Daten enorm vorangetrieben hat. Eine Vielzahl mobiler Anwendungen wäre ohne einfach zu benutzende und zuverlässige Schnittstellen zu GIS-Funktionen undenkbar. Doch nicht immer reicht der Funktionsumfang dieser Bibliotheken aus oder es stehen ihrer Benutzung technische Hindernisse entgegen. Wie lässt sich zum Beispiel GIS-Funktionalität in Desktopanwendungen integrieren oder wie können GIS-Informationen in einem Intranet dargestellt werden? Wie genau arbeiten Google Maps und Co., und welches Kartenmaterial liegt ihnen zugrunde? Ob das nächste Restaurant 50 Meter weiter links oder rechts auf dem Smartphone dargestellt wird, ist sicher nicht so entscheidend. Wesentlich präziser hingegen sollte z. B. die Darstellung der aktuellen Minenlage in militärischen Szenarien sein.

von Marco Hüther

Der Begriff ArcGIS ist der Oberbegriff für alle Produkte des kommerziellen Geoinformationssystems (GIS) des Unternehmens Esri. Die Produktpalette umfasst dabei mehrere Systeme zur Erfassung, Bearbeitung, Organisation, Analyse und Visualisierung raumbezogener Daten und stellt faktisch einen Quasistandard im professionellen Einsatz von GIS-Systemen dar. Auch für uns Entwickler werden einige SDKs angeboten, die die Realisierung von GIS-Anwendungen sowohl im mobilen Bereich als auch im Desktopumfeld erheblich erleichtern.

Durch die Firma Esri wurde im letzten Jahr die Entwicklung der so genannten ArcGIS Runtime vorangetrieben. Dieser Begriff steht hierbei für eine Reihe nativer SDKs zur Erstellung von leichtgewichtigen und mobilen GIS-Anwendungen. Im Rahmen der Java-Entwicklung lassen sich Anwendungen im mobilen Bereich mit dem ArcGIS Runtime SDK for Android entwickeln sowie im Desktopbereich mit dem ArcGIS Runtime SDK for Java, das auf Swing-Komponenten basiert. Entscheidend dabei ist, dass der darunterliegende Code nahezu identisch ist.

Für die Umsetzung von Desktopanwendungen und mobilen Applikationen eröffnet dies ganz neue Möglichkeiten, da insbesondere die Integration von Desktopapplikationen enorm erleichtert wird. Deshalb möchte ich die beiden SDKs im Folgenden vorstellen.

„Hello World"-Map

Das ArcGIS Runtime SDK for Android erfordert ein paar Installationsschritte, die unter [1] ausführlich beschrieben werden. Voraussetzung ist hierbei das Anlegen eines Esri-Global-Accounts. Die Anmeldung ist kostenlos, allerdings für den Download des SDKs zwingend erforderlich.

Nach der erfolgreichen Einrichtung der Entwicklungsumgebung kann bereits die erste Karte mit entsprechenden Grundfunktionalitäten auf dem mobilen Endgerät dargestellt werden. Hierzu stellt das ArcGIS Runtime SDK entsprechende Wizards zur Verfügung, um Projekte zu erzeugen. Auch zahlreiche Beispiele werden bereitgestellt, die ebenfalls über die Wizards des mitgelieferten Eclipse-Plug-ins erzeugt und als Vorlage genutzt werden können.

Um das SDK besser kennenzulernen, bietet es sich an, zuerst einmal mit einem leeren ArcGIS-Projekt zu starten. Dazu ruft man den Wizard, wie in Eclipse bekannt, mit File | New | Other | ArcGIS for Android | ArcGIS Project for Android auf. Im eigentlichen Wizard sind dann nur noch ein Name für das Projekt – im vorliegenden Fall also *HelloWorldMapMobile* – und die Package-Bezeichnung zu vergeben.

Nachdem das Eclipse-Projekt angelegt und die benötigten Bibliotheken automatisch hinzugefügt wurden, muss ich in meiner Entwicklungsumgebung nur noch eine kleine Anpassung in der *project.properties* vornehmen, da der Wizard von einem Android-API-Level in der Version 10 ausgeht, ich aber die Version 17 installiert habe. Hierzu habe ich das Target wie folgt angepasst: *target =android-17*. Darüber hinaus ist es natürlich auch möglich, die entsprechende Version über den Android SDK Manager nachzuinstallieren.

Startet man jetzt das Projekt auf dem Mobilgerät oder auf dem Emulator, wird noch keine Karte dargestellt, sondern nur eine *TextView* mit dem altbekannten „Hello World"-Text angezeigt. Um dies zu ändern, muss die *TextView* durch eine entsprechende *MapView* ersetzt werden. Dazu tauscht man innerhalb der *main.xml* das *TextView*-Element mit dem *MapView*-Layout, das im ArcGIS Runtime SDK enthalten ist, aus. Das ent-

sprechende XML-Element ist in Listing 1 dargestellt. Die *MapView* bietet nun die Möglichkeit, mehrere Schichten an Informationen – so genannte Layer – auf ihr zu platzieren. Wie in Listing 2 zu sehen, wird hierfür in der *onCreate*-Methode der *HelloWorldMapActivity* ein so genannter *ArcGISTiledMapServiceLayer* auf die *MapView* gelegt. Diese Klasse ermöglicht den Zugriff auf einen ArcGIS-Online-Service, dessen Karteninformationen in Form von einzelnen Kacheln vorliegen. Diese Bilder werden über einen REST-Service geladen, der über ArcGIS Online zur Verfügung steht. Im Anschluss kann die „Hello World"-Applikation auf dem Mobile-Gerät gestartet werden. Der hinzugefügte Layer ist sofort zu sehen und bestimmte Grundfunktionalitäten wie das Zoomen über Gesten stehen direkt zur Verfügung. Im Folgenden wird der Funktionsumfang der App noch erweitert.

Die „Hello World"-Map steht neben weiteren Beispielen innerhalb des ArcGIS Runtime SDK über dem oben genannten Eclipse-Wizard zur Verfügung. Die manuelle Entwicklung der ersten App sollte allerdings einmal aufzeigen, wie man mit einfachen Mitteln ansprechende Karten auf dem Mobile visualisieren kann.

Zur Verteilung der fertigen Anwendung sind noch einige Vorgaben zu beachten, die unter [2] beschrieben sind. Unter anderem ist ein Esri-Logo mit der Methode *setEsriLogoVisible* auf der *MapView* einzublenden und ein Copyright-Text in den About-Dialog der Anwendung zu integrieren. Danach steht dem Deployment der Map-Applikation nichts mehr im Wege.

Listing 1

```
<!-- MapView layout -->
 <com.esri.android.map.MapView
  android:id="@+id/map"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
 </com.esri.android.map.MapView>
```

Listing 2

```
@Override
public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);

 mapView = (MapView) findViewById(R.id.map);

 // Layer über REST Service von ArcGIS Online hinzufügen
 mapView.addLayer(new ArcGISTiledMapServiceLayer(
  "http://services.arcgisonline.com/ArcGIS/rest/services/World_Street_Map/
                                                          MapServer"));
}
```

Onlinedienste

Neben dem Zugriff auf ArcGIS Online über den *ArcGISTiledMapServiceLayer* bietet das SDK allerdings auch noch mehr Optionen, um auf unterschiedliche Informationsquellen und Dienste zuzugreifen. Folgende Layertypen stehen derzeit noch zur Verfügung:

- Der *GraphicsLayer* bietet die Möglichkeit, Informationen auf der Karte zu zeichnen, die georeferenziert sind. So können verschiedene Geometrien (Kreise, Linien oder Punkte) mit entsprechender Symbolik durch Methodenaufrufe auf dem Layer erstellt werden.
- *ArcGISFeatureLayer:* Ein *FeatureLayer* basiert intern auf einem *GraphicsLayer*, stellt aber im Gegensatz hierzu eher einen abstrakten Zugriff auf die Daten bereit. Die Informationen für den Layer werden in der Regel über einen Dienst auf einem ArcGIS-Server zur Verfügung gestellt und mittels JSON-Objekten

Listing 3

```
private void reverseGeocoding() {
 mapView.setOnLongPressListener(new OnLongPressListener() {
  public void onLongPress(float xCoordinateOnScreen, float CoordinateOnScreen)
  {
   // Koordinate ermitteln
   final Point mapLocation = mapView.toMapPoint(xCoordinateOnScreen,
   yCoordinateOnScreen);

   final Locator locator = new Locator(
   "http://geocode.arcgis.com/arcgis/rest/services/World/GeocodeServer");

   new AsyncTask<Void, Void, Map<String, String>>() {
    @Override
    protected Map<String, String> doInBackground(Void... param) {
     try {
      // Rest Service verwenden, um die Adresse zu ermitteln
      LocatorReverseGeocodeResult reverseGeocode =
      locator.reverseGeocode(mapLocation, 100,
      mapView.getSpatialReference(), mapView.getSpatialReference());
      return reverseGeocode.getAddressFields();
     } catch (Exception e) {
      return Collections.emptyMap();
     }
    }

    @Override
    protected void onPostExecute(Map<String, String> addressFields) {
     TextView textView = new TextView(HelloWorldMapActivity.this);

     textView.setText(addressFields.get("Address") + "\n" +
     addressFields.get("Postal") + " " + addressFields.get("City"));
     // Ergebniss anzeigen
     mapView.getCallout().show(mapLocation, textView);
    }
   }.execute();
  }
 });
}
```

übertragen. So reicht oft eine einzige Zeile Code, um so genannte Features auf der Karte anzuzeigen. Das folgende Beispiel fügt der Karte einen Featureservice hinzu, der zum Beispiel Städte gruppiert nach deren Bevölkerungsdichte anzeigt:

```
mapView.addLayer(new ArcGISFeatureServiceLayer(" http://sampleserver6.
    arcgisonline.com/arcgis/rest/services/SampleWorldCities/MapServer "));
```

- Die *TiledLayer*, die bereits in dem vorherigen Beispiel verwendet wurden, bieten die beste Performance, da die Informationen in Form von gecachten Rasterdaten bereits auf dem Server vorliegen und nicht zur Laufzeit aufgebaut bzw. berechnet werden müssen. Sollte einmal kein Netzwerkzugriff vorhanden sein, ist es auch möglich, über den *ArcGISLocalTiledLayer* mit lokalen Caches zu arbeiten. Neben dem Zugriff auf ArcGIS-Rasterdaten bietet sich auch der Zugriff auf Bing- oder OpenStreetMap-Karten an.
- Eine *WebMap* ist kein einzelner Layer, der auf der *MapView* platziert werden kann, sondern eine Konfiguration von Layern und Diensten, die über ArcGIS Online zur Verfügung gestellt werden. Gerade ArcGIS Online sollte einen erneuten Blick wert sein. Das Portal bietet neben bestimmten GIS-Diensten auch viele weitere Grundkarten, so genannte Basemaps an, die im Vergleich zu anderen Quellen wesentlich genauer und aktueller sind. Des Weiteren lassen sich über ArcGIS Online auch individuelle Karten für das eigene Unternehmen erstellen. Dies ist allerdings kostenpflichtig und der Funktionsumfang bedarf eines eigenen Artikels.

Die richtige Adresse finden

Weitere Dienste lassen sich problemlos über das ArcGIS-REST-API in unsere „Hello Word"- Map integrieren. Das API ermöglicht die Arbeit mit Services, die von ArcGIS Online gehostet werden, einschließlich der von Esri gehosteten, sofort einsatzfähigen Services. Statt direkt mit dem REST-API zu arbeiten, bietet das ArcGIS Runtime SDK verschiedene Klassen und Methoden an, die den Zugriff erheblich erleichtern. Beispielhaft soll im Folgenden ein Locator-Service zum Einsatz kommen, der es unter anderem zulässt, ein so genanntes Reverse Geocoding durchzuführen. Damit kann man mittels Angabe der aktuellen Koordinaten als Position die Adresse des Standorts feststellen. Die aktuellen Koordinaten sollen durch ein längeres Berühren der *MapView* an einer bestimmten Stelle ermittelt werden.

In Listing 3 ist hierzu die Methode *reverseGeocoding* dargestellt, die in der *onCreate*-Methode der *HelloWorldMapActivity* nach dem Erstellen der *MapView* aufgerufen wird. Zu Beginn der Methode wird der *MapView* ein *OnLongPressListener* hinzugefügt, um eine entsprechende Position auf der Karte zu ermitteln. Da es sich nur um die Bildschirmposition handelt, bietet die *MapView* auch hier eine entsprechende Konvertierungsfunktion an. Bei der Locator-Klasse handelt es sich um die eben angesprochene Wrapper-Klasse, die den Zugriff auf den REST-Service kapselt. Es wird lediglich der URL des entsprechenden Service mitgegeben. Esri stellt auch hierzu eine frei zugängliche Schnittstelle zur Verfügung. Über einen von der Android-Entwicklung bekannten *AsyncTask* wird nun der eigentliche Serviceaufruf durchgeführt. Die *reverseGeoCode*-Methode bekommt die zuvor ermittelte Koordinate übergeben und liefert als Ergebnis die Adresse, die sich an dieser Position befindet. Im Anschluss an den Serviceaufruf wird im GUI-Thread die Methode *onPostExecute* aufgerufen, die das Resultat in einem so genannten *Callout* anzeigt. Ein Callout stellt eine Art Sprechblase dar, die auf der Karte angezeigt wird und deren Hinweisstrich auf eine bestimmte Koordinate zeigt.

Abb. 1: Mobile Anwendung mit Reverse Geocoding

Im vorliegenden Fall ist dies die Position, für die das Reverse Geocoding durchgeführt wurde. Wird die „Hello World"-Map auf dem Gerät gestartet, kann man direkt sehen, wie einfach es war, mit wenigen Zeilen Code relativ umfangreiche GIS-Dienste zu verwenden. Das Ergebnis eines Reverse Geocoding ist in **Abbildung 1** zu sehen. Natürlich sollte man sich jetzt noch um ein entsprechendes Exception Handling oder auch um die Anzeige eines Progressdialogs beim Aufruf des Service kümmern. Da es sich hierbei allerdings nicht um Runtime-Funktionalitäten handelt und da das Beispiel so kompakt wie möglich sein sollte, wurden diese Anteile weggelassen.

Die Runtime auf dem Desktop verwenden

Wie bereits zu Beginn des Artikels angesprochen, liegt ebenfalls ein ArcGIS Runtime SDK für die Verwendung auf dem Desktop vor. Im Bereich der Java-Entwicklung ist dies in Form von entsprechenden Swing-Komponenten umgesetzt. Nach der erfolgreichen Einrichtung der Entwicklungsumgebung, wie unter [3] ausführlich beschrieben, kann man sich in Eclipse wieder eines Wizards bedienen, der unter File | New | Other | ArcGIS Runtime Java Application zu erreichen ist. Der Wizard sorgt dafür, dass alle für ein Runtime-Projekt erforderlichen Bibliotheken automatisch importiert werden. Nachdem man eine kleine Anwendung erstellt hat, die einen *JFrame* anzeigt, kann man sich mit ähnlichen Mitteln wie bei der mobilen Entwicklung eine *HelloWorldMap* erstellen. Wie in Listing 4 zu sehen, reicht das Erzeugen einer Swing-Komponente und das Hinzufügen eines *ArcGISTiledMapServiceLayer* aus, um schon ähnliche Ergebnissen zu erhalten. Als Basemap kommen allerdings statt einer Straßenkarte Satelli-

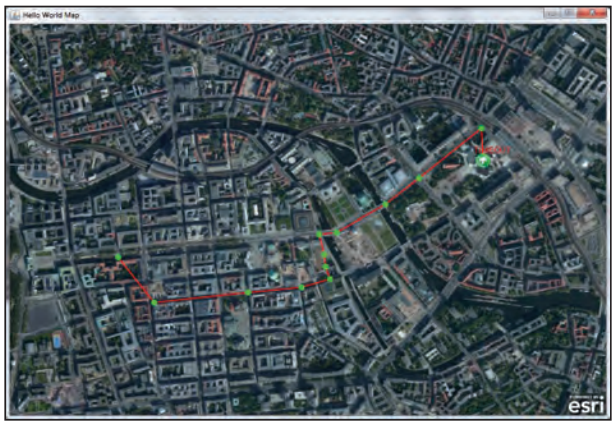


Abb. 2: Desktopanwendung mit „GPS-Layer“

tenbilder zum Einsatz, die ebenfalls über ArcGIS Online zur Verfügung gestellt werden. Das zugrunde liegende Material ist sehr detailliert und aktuell.

Diese grundlegende Funktionalität lässt sich erweitern, indem man beispielsweise dynamische Informationen anhand eines *GPSLayer* in die Desktopanwendung integriert. Der *GPSLayer* sorgt dafür, dass die aktuelle GPS-Position, wie in **Abbildung 2** zu sehen, sowie die vorherigen Track-Positionen auf der Karte angezeigt werden. Auch das Zoomen der Karte auf die aktuelle Position wird ermöglicht. Die GPS-Informationen erhält der Layer entweder über einen seriellen Empfänger oder wie im vorliegenden Fall über eine Textdatei, die NMEA-Daten beinhaltet. Bei NMEA handelt es sich um einen Standard, der bei der Kommunikation zwischen GPS-Empfänger und PCs sowie mobilen Endgeräten genutzt wird. Um die eben beschriebenen Features umzusetzen, reichen bei der ArcGIS Runtime zwei Zeilen Code völlig aus, siehe **Abbildung 2**. Natürlich lassen sich die Symbolik und die Anbindung über einen *SerialPortGPSWatcher* jederzeit über den *GPSLayer* einstellen.

Das Deployment der Anwendung gestaltet sich im Vergleich zur mobilen Variante etwas aufwändiger. Der Download des SDKs sowie der Vertrieb der fertigen Anwendung erfordern einen kostenpflichtigen Zugriff auf das Esri Developer Network. Verwendet man wie in unserem Beispiel nur so genannte Basic-Features, fallen keine weiteren Lizenzkosten an. Möchte man hingegen Standard-Features wie zum Beispiel das Starten eines lokalen Servers verwenden, fallen weitere Kosten an, die davon abhängen, wie oft man die entwickelte Software ausliefern möchte.

Listing 4

```
public class HelloWorldMap {
  private JFrame frame;
  private JMap map;

  public HelloWorldMap() {
    createApplicationFrame();
    addBaseMap();
    frame.setVisible(true);
  }

  private void createApplicationFrame() {
    frame = new JFrame("Hello World Map");
    frame.setSize(1024, 768);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  private void addBaseMap() {
    map = new JMap();

    map.getLayers().add(new ArcGISTiledMapServiceLayer(
    "http://services.arcgisonline.com/ArcGIS/rest/services/" +
    " World_Imagery/MapServer"));

    frame.add(map);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
        public void run() {
          new HelloWorldMap();
        }
    });
  }
}
```

Zusammenfassung

Die ArcGIS Runtime SDKs stellen umfangreiche Funktionalitäten bereit, um schlanke und leistungsstarke Anwendungen für verschiedene Plattformen zu entwickeln. Für Entwickler, die ähnliche Funktionalitäten sowohl für den Desktop als auch im mobilen Bereich anbieten möchten und die nicht komplett unterschiedliche APIs hierfür einsetzen wollen, bieten sich mit den ArcGIS Runtime SDKs entsprechende Möglichkeiten. Ich hoffe, ich konnte einen ersten Eindruck vermitteln und wünsche viel Spaß beim Implementieren der eigenen Weltsicht.

Marco Hüther ist Softwareentwickler im Bereich Military & Defense. Er verfügt über langjährige Erfahrungen als Entwickler und Architekt im Java-Umfeld sowie bei der Umsetzung von GIS-Anwendungen.

Links & Literatur

[1] https://developers.arcgis.com/en/android/install.html

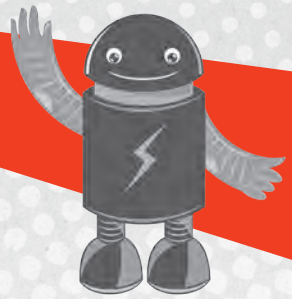
[2] https://developers.arcgis.com/en/android/guide/attributing-your-application.htm

[3] http://bit.ly/15A6VnB

Listing 5

```
private void addGPSLayer() {
  IGPSWatcher watcher = new FileGPSWatcher("d:/temp/route.nmea",
                                                    3000, false);
  GPSLayer gpsLayer = new GPSLayer(watcher);
  map.getLayers().add(gpsLayer);
}
```

m2m minutes



Nicht nur auf der JavaOne spielte das Internet der Dinge eine herausragende Rolle. Auch zahlreiche Fachveranstaltungen wie der M2M Summit, die World Maker Faire oder die World Smart Week kündigten bereits einen heißen Herbst für die Branche mit all ihren Technologien, Bereichen und Communitys an. In dieser neuen Rubrik präsentieren wir wichtige Meldungen des Monats aus der M2M-Welt.

Google Coder bringt kleine Raspberry-Pi-Fans ins Netz

Auch andere Unternehmen haben die Welt der Elektronikbastler und Zwergcomputer längst für sich entdeckt: Gerade hat Oracle auf der JavaOne angekündigt, dass der Einplatinencomputer Raspberry Pi künftig standardmäßig mit Java SE ausgeliefert wird. Und auch in Mountain View bleibt der Pi-Hype nicht aus: Die Idee des Google-Entwicklers Jason Striegel, den Raspberry Pi mithilfe eines Open-Source-Tools in einen kleinen Webserver samt Entwicklungsumgebung zu verwandeln, erntete in den vergangenen Wochen begeisterten Zuruf. Das Tool, speziell für den programmierenden Nachwuchs entwickelt, wurde auf den bescheidenen Namen „Coder“ getauft und kann auf der Projektseite heruntergeladen werden.

http://bit.ly/1aPduCJ

openHAB mit Duke’s Choice Award ausgezeichnet

Das Java-Projekt openHAB ist auf der JavaOne in San Francisco mit dem Duke’s Choice Award ausgezeichnet worden. openHAB stellt ein Framework zur Verfügung, mit dem sich heterogene Protokolle und Standards integrieren lassen. Ziel ist es, den Zugriff auf diese Geräte zu vereinfachen. Mit der wachsenden Community um openHAB stehen auch immer mehr Bindings an diverse Technologien zur Verfügung.

https://www.java.net/dukeschoice

Intel goes Arduino

Immer mehr große IT-Konzerne bauen Brücken zur Maker-Szene. Der Elektronikhersteller Intel, der neulich einen Roboter aus dem 3-D-Drucker angekündigt hat, stellte jüngst den Mikrocontroller „Galileo“ vor, das erste Board, das sowohl hardware- als auch softwareseitig voll mit den Arduino-Erweiterungen (*Shields*) kompatibel ist. Es ist mit Intels neuem Prozessor Quark SoC X1000 ausgestattet und verfügt wie der Arduino Uno über vierzehn digitale Ein- und Ausgabekanäle (sechs davon mit PWM-Ausgang) sowie über sechs analoge Eingabekanäle. Der Preis wurde noch nicht bekannt gegeben. In den nächsten achtzehn Monaten möchte Intel 1 000 Universitäten weltweit insgesamt 50 000 Galileo-Boards spendieren.

http://arduino.cc/en/Arduino Certified/IntelGalileo

Eurotech kündigt Everyware-Framework 2.0 an

Eurotech, ein auf Embedded-Lösungen spezialisiertes Unternehmen, hat ein neues Major-Release seines Everyware Software Frameworks (ESF) angekündigt. Das Framework wird zur Lebenszyklusverwaltung von Geräten in Machine-to-Machine-Anwendungen entwickelt. Die neue Version bietet die Möglichkeit, Geräte per Fernzugriff zu bedienen und zu verwalten. Hinzugekommen sind u. a. ein Anwendungscontainer mit Standardservices, um die Anwendungsentwicklung zu beschleunigen, sowie eine grafische Nutzeroberfläche, die die lokale Konfiguration von Services und Anwendungen vereinfachen soll.

http://bit.ly/16PdjKk

Vorschau auf die Ausgabe 1.2014

Alles aus einem Guss: Die neue Spring-IO-Plattform

Seit Kurzem erstrahlt die Website des Spring-Ökosystems in neuem Gewand und präsentiert einen neuen Aufbau der Plattform und damit des kompletten Spring-Stacks. Ziel ist es, orchestrierte Releases der Foundation anzubieten, in denen Versionskompatibilität der unterschiedlichen Module sichergestellt wird. Mit diesem Schritt rücken Spring-Technologien künftig näher zusammen. Was genau hinter dem neuen Aufbau steckt, werden wir kommenden Monat erklären. Es soll dann auch um das neue Spring-Tooling und jungen Spring-Familienzuwachs gehen.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 4. Dezember 2013

Querschau

eclipse MAGAZIN — **Ausgabe 6.2013** | www.eclipse-magazin.de

- **Orion: Softwareentwicklung im Browser**
- **MoDisco: Modellbasierte IT-Modernisierung**
- **Ein Pläuschchen mit MQTT: Chatclient mit Eclipse Paho und Vaadin**

entwickler magazin — **Ausgabe 6.2013** | www.entwickler-magazin.de

- **Objektorientierte Programmierung in ANSI-C**
- **Webentwicklung mit Saxon-CE und XSLT 2.0**
- **Theorie und Praxis der Softwaremessung**

MOBILE TECHNOLOGY — **Ausgabe 4.2013** | www.mobiletechmag.de

- **Dependency Management in iOS-Projekten leicht gemacht**
- **Entwicklung der Android-Spiele-App Hungry Archie**
- **Wie man mit Mobile Advertising seine App zu Geld macht**

Inserenten

Verlag:
Software & Support Media GmbH

Anschrift der Redaktion:
Java Magazin
Software & Support Media GmbH
Darmstädter Landstraße 108
D-60598 Frankfurt am Main
Tel. +49 (0) 69 630089-0
Fax. +49 (0) 69 630089-89
redaktion@javamagazin.de
www.javamagazin.de

Chefredakteur: Sebastian Meyen
Redaktion: Claudia Fröhling, Diana Kupfer, Hartmut Schlosser
Chefin vom Dienst/Leitung Schlussredaktion: Nicole Bechtel
Schlussredaktion: Jennifer Diener, Frauke Pesch
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Flora Feher, Karolina Gaspar, Dominique Kalbassi, Laura Keßler, Nadja Kesser, Maria Rudi, Petra Rüth, Franziska Sponer

Autoren dieser Ausgabe:
Sven Efftinge, Andreas Feldschmid, Tobias Flohre, Uwe Friedrichsen, Lars George, Florian Heidenreich, Peter Hruschka, Michael Hunger, Marco Hüther, Denny Israel, Klaus Kreft, Heiner Kücker, Angelika Langer, Arne Limburg, Sebastian Mancke, Christian Meder, Michael Müller, Tobias Nestler, Dennis Nobel, Lars Pfannenschmidt, Florian Pirchner, Lars Röwekamp, Dr. Mirko Seifert, Gernot Starke, Christian Wende, Thomas Wilk, Eberhard Wolff

Anzeigenverkauf:
Software & Support Media GmbH
Patrik Baumann
Tel. +49 (0) 69 630089-20
Fax. +49 (0) 69 630089-89
pbaumann@sandsmedia.com

Es gilt die Anzeigenpreisliste Mediadaten 2013

Pressevertrieb:
DPV Network
Tel.+49 (0) 40 378456261
www.dpv-network.de

Druck: PVA Landau
ISSN: 1619-795X

Abonnement und Betreuung:
Leserservice Java Magazin
65341 Eltville
Tel.: +49 (0) 6123 9238-239
Fax: +49 (0) 6123 9238-244
javamagazin@vuservice.de

Abonnementpreise der Zeitschrift:

Inland:	12 Ausgaben	€ 118,80
Europ. Ausland:	12 Ausgaben	€ 134,80
Studentenpreis (Inland)	12 Ausgaben	€ 95,00
Studentenpreis (Ausland):	12 Ausgaben	€ 105,30

Einzelverkaufspreis:

Deutschland:	€ 9,80
Österreich:	€ 10,80
Schweiz:	sFr 19,50
Luxemburg:	€ 11,15

Erscheinungsweise: monatlich

Bildnachweis: Der Android Robot steht unter der Creative-Commons-Lizenz.