Schneller in die Produktion! Mit Puppet und RPM ▸81

S&S

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 7.2013

JAVA Mag

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Java Embedded

BeagleBone: Ran an den Knochen ▸86

Java EE 6

Migration: Ein Praxisbericht ▸59

Memory Leaks

Weak References unter der Lupe ▸20

W-JAX 13

Alle Infos hier im Heft! ▸51

JVM ROCKS!

JVM und die Sprachen

Gemeinsam sind wir stark ▸33

„Der erste Schritt ist, polyglott zu werden“

Interviews rund um die JVM ▸ab 40

DIE JVM IST SO ATTRAKTIV WIE NIE

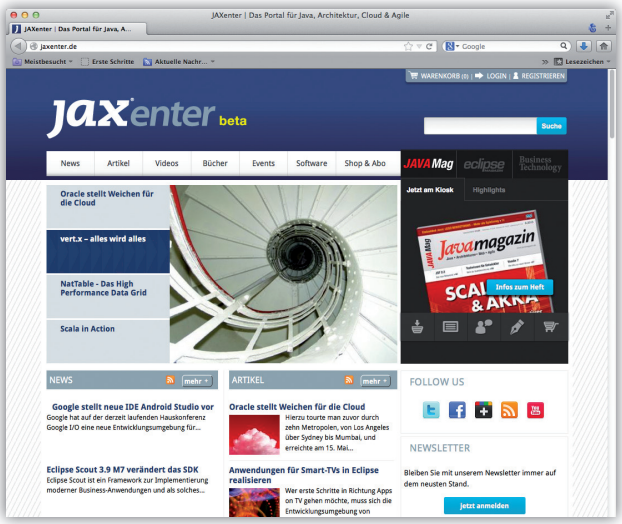
4 194586 709801 07

Anzeige

And … Action!

Alles neu macht der Mai – so lautet nicht nur der Titel eines bekannten deutschen Volkslieds, sondern auch unser Motto bei der diesjährigen Planung unserer Portale. Dem einen oder anderen ist es sicherlich schon aufgefallen: Wir haben Anfang des Monats einen Relaunch unserer Seiten www.javamagazin.de, www.eclipse-magazin.de und www.JAXenter.de vorgenommen.

Schon lange ist es unser Ziel, Magazin und Newsportal enger miteinander zu verbinden, mit dem Relaunch haben wir es nun geschafft, die Magazine eng in JAXenter als Dachportal zu integrieren. News aus dem Java-Universum, Kommentare zu wichtigen Meilensteinen, Interviews mit den führenden Experten der Branche – das alles werden wir Ihnen weiterhin auf JAXenter präsentieren und haben gleichzeitig die Chance, die Vertiefung dieser Inhalte im Java Magazin und im Eclipse Magazin prominent auf dem Portal zu ergänzen. Denn die Magazine sind nun integraler Bestandteil, mit allen Infos zu aktuellen und älteren Ausgaben inklusive Inhaltsangaben, Autorenverzeichnissen, Editorials, Link-Tipps und Quellcodes zum freien Download. Das neue JAXenter präsentiert sich dabei im zeitgemäßen Gewand, wie Sie im Screenshot sehen können.



Neu ist außerdem der verstärkte Fokus auf Interaktivität der Leser – ab sofort gibt es nämlich die Möglichkeit, einen eigenen Account anzulegen und so mehr Einfluss auf Ihr Portal zu nehmen. Probieren Sie es aus!

Apropos Interaktivität: Wir haben ein neues Poster für Sie entworfen, das erste Poster, das Sie selbst erweitern können! Aber kommen wir erst zum Inhalt des neuen Wandschmucks.

Git steht für zeitgemäße Versionsverwaltung. Ganz egal, für welche Plattformen man Anwendungen entwickelt – Git wird immer beliebter und damit unentbehrlich. Den kometenhaften Aufstieg des dezentralen Codeverwaltungssystems haben wir zum Anlass genommen, gemeinsam mit zehn Experten ein Git-Poster im A0-Format zu erstellen. „Git – The Big Picture" zeigt alles Wissenswerte über Git auf einen Blick. Es ist ein Dankeschön an unsere Abonnenten, die das Poster im Innenteil des Magazins finden. Sie sind noch kein Abonnent und möchten das Poster erhalten? Dann bestellen Sie einfach das Premiumabonnement des Java Magazins. Neuabonnenten erhalten das Poster gratis, solange der Vorrat reicht. Weitere Informationen unter http://www.javamagazin.de/abo.

Und was hat es jetzt mit dem interaktiven Inhalt auf sich? Ganz einfach: Auf dem Poster finden Sie einen A4-großen Bereich, der frei gelassen wurde, damit Sie ihn füllen können! Zu diesem Zweck haben wir spannende Add-ons entwickelt, die genau auf dieses Feld passen. Das erste finden Sie hier im Heft auf Seite 46 und online zum freien Download unter http://jaxenter.de/specials. Weitere Add-ons folgen in Kürze, bleiben Sie up to date und folgen uns auf Twitter oder Google+, um informiert zu werden, sobald ein neues Add-on verfügbar ist. Oder haben Sie selbst eine Idee, wie der freie Platz gefüllt werden sollte? Dann schicken Sie uns Ihre Anregungen an redaktion@javamagazin.de. Wir freuen uns auch über Beweisbilder, wo das Poster bei Ihnen im Büro hängt :-)

In diesem Sinne wünsche ich Ihnen viel Spaß bei der Lektüre dieser Ausgabe und beim Aufhängen des Posters!

Claudia Fröhling, Redakteurin

@JavaMagazin

gplus.to/JavaMagazin

JVM rocks!

Die Java Virtual Machine ist nicht nur einfach eine VM, sie ist das Zuhause vieler spannender Sprachen und ein Garant für Robustheit und Skalierbarkeit. Kein Wunder also, dass immer mehr Firmen zur JVM wechseln, sobald sie „Serious Business" machen müssen. Wir fragen diesen Monat, was genau hinter der Attraktivität der JVM steckt und sprechen dafür mit ausgewiesenen Experten des Java-Ökosystems. Außerdem kommentiert Peter Roßbach für uns die Vor- und Nachteile von Java und wagt einen Blick in die Zukunft. Alle sind sich in einem Punkt einig: Die Java Virtual Machine rockt!

Memory Leaks

In diesem Teil der Java-Core-Serie wollen wir uns Weak References genauer ansehen. Mit ihrer Hilfe ist es möglich, Situationen zu vermeiden, in denen Memory Leaks entstehen können. Denn Vorbeugung ist die halbe Miete!

Magazin

Java Core

Titelthema

Agile

Enterprise

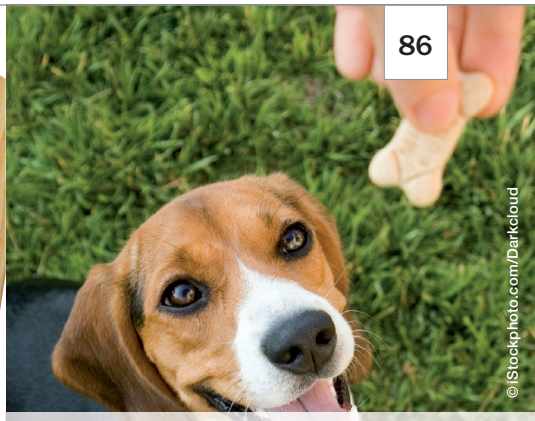
Leserbriefe und Feedback zu den Artikeln des Java Magazins bitte an redaktion@javamagazin.de.

Java-EE-6-Praxis

Das Kundenportal eines Telekommunikationsanbieters basiert heute auf dem Java EE 6 Web Profile. Im Rahmen eines Migrationsprojekts wurde es von teilweise zehn Jahre alten Frameworks und Technologien innerhalb weniger Monate auf einen modernen Technologiestack überführt, um so Wettbewerbsfähigkeit und Time to Market zu gewährleisten.

JSF 2.2

Der Sprung von JSF 2.1 auf JSF 2.2 ist zunächst ein Minor-Update. Dennoch sind in diesem vermeintlich kleinen Versionssprung sehr viele Neuerungen enthalten, die beinahe schon ein Major-Update gerechtfertigt hätten. Im Tutorial gehen wir dem auf den Grund.



Java Embedded

Der BeagleBone ist eine vielfältig einsetzbare und erweiterbare Variante des BeagleBoards von der Größe einer Kreditkarte. Die Anwendungsmöglichkeiten erstrecken sich von der Visualisierung von 3-D-Grafiken und der Anzeige von HD-Videos bis hin zu energieeffizient arbeitenden Robotersteuerungen und Webservern. Ein Erfahrungsbericht.

Tutorial

Web

Embedded

Tools

Architektur

Android360

Standards

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

Java 8: Neuer Termin, neues Featureset

Die Verschiebung von Java 8 auf den 18. März 2014 ist beschlossene Sache. Die verlorenen sechs Monate, die vor allem auf die Verzögerungen im Projekt Lambda zurückzuführen sind, bedeuten für manch anderes Teilprojekt indes gewonnene Zeit. So schaffen es einige JEPs (Java Enhancement Proposals), die bis September 2013 nicht fertig geworden wären, nun doch noch in das Java-8-Boot. Java-Plattform-Chefentwickler Mark Reinhold teilt auf der OpenJDK-Mailingliste mit, dass die folgenden JEPs in Java 8 enthalten sein werden:

- 176 Mechanical Checking of Caller-Sensitive Methods
- 178 Statically-Linked JNI Libraries
- 179 Document JDK API Support and Stability
- 180 Handle Frequent HashMap Collisions with Balanced Trees
- 184 HTTP URL Permissions

Aufhorchen lässt hier die Nr. 178: Die Möglichkeit, statische Libraries im Java Native Interface zu unterstützen, zielt auf einen flexibleren Einsatz von Java SE 8 in Embedded-Szenarien ab. Richtig spannend wird es erst, wenn man hinzufügt, dass mit diesem JEP die bisherigen Hürden abgeräumt werden, um Java-Programme direkt auf iOS-Geräten zum Laufen zu bringen. Java-Vater James Gosling hatte dies als „Big Deal" bezeichnet, da es gelingen könnte, so den Richtlinien des App Stores gerecht zu werden. Wird Java also bald Objective-C Konkurrenz machen?

Drei der anderen JEPs beziehen sich auf die Verbesserung der Plattformsicherheit. Wie wir auf der JAX von Wolfgang Weigend, Master Principal Sales Consultant bei der Oracle Deutschland GmbH, hören konnten, wurde der Fokus der Oracle-Entwicklung stark in Richtung Security verschoben, was den Einbezug dieser JEPs erklären dürfte. Einmal geht es um die automatische Identifikation „Caller-sensitiver" Methoden. JEP 184 bezieht sich auf das Einrichten von Netzwerkautorisierungen auf Basis von URLs statt IP-Adressen. Und in JEP 180 sollen Bäume statt verlinkte Listen zur Datenspeicherung zugelassen werden.

Doch so wie er gibt, so nimmt er auch: Mark Reinhold teilt im gleichen Zuge mit, dass es die folgenden JEPs nicht in Java 8 machen werden:

- 143 Improve Contended Locking
- 165 Compiler Control

Diese beiden Verbesserungen seien in Verzug geraten und nicht weit genug gediehen, um in Java 8 berücksichtigt werden zu können.

Ein weiterer JEP wurde von Reinhold zusätzlich vorgeschlagen: JEP 185 „JAXP 1.5: Restrict Fetching of External Resources". Diese ebenfalls sicherheitsbezogene Änderung wird „wahrscheinlich" ebenfalls noch in den Meilenstein 7 integriert werden.

Das kommende Java-8-M7-Release, in dem alle Features für Java 8 enthalten sein sollen, wird am 23. Mai fällig. Danach soll es nur noch um Stabilität und Sicherheit gehen. Es folgen noch der M8 als Developer-Preview (05.09.2013) und der M9 als Final Release Candidate (23.01.2014) – und wenn dieses Mal alles gut geht, halten wir am 18.03.2014 das GA-Release von Java 8 in der Hand.

▶ http://openjdk.java.net/projects/jdk8/features

Links und Downloadempfehlungen zu den Artikeln im Heft

- Sourcecode für das Memory-Leak-Beispiel aus dem Artikel „Effective Java: Weak References": http://www.angelikalanger.com/Articles/EffectiveJava/66.Mem.Analysis/66.Mem.Analysis.zip
- File I/O Tutorial: http://docs.oracle.com/javase/tutorial/essential/io/fileio.html
- Processing: http://processing.org
- Mockdemo zum Artikel „Doppelt hält besser: Teststile: Schwierige Tests mit Doubles": https://github.com/kspichale/mock-demo

Git Forking im Enterprise mit Stash 2.4

Stash ist ein Tool zum Verwalten von Git Repositories, die sich hinter einer Firewall befinden. So eignet sich die Software des Herstellers Atlassian besonders für den Unternehmenseinsatz und zur Verwendung in Teams.

Das neue Release Stash 2.4 bringt die in der Softwareentwicklung schon weit verbreitete Option des Forkings nun auch in den Enterprise-Bereich. Durch Forks können Entwickler auch Code zu Repositories beitragen, für die sie keinen Schreibzugriff haben. Dabei lassen sich Repositories in jedes andere Stash-Projekt forken, für das man Adminrechte besitzt.

Ebenfalls neu sind Repository Permissions, die den Zugang zu den einzelnen Verzeichnissen kontrollieren. Diese werden vor allem dann nützlich, wenn man eine weitere Neuheit von Stash 2.4 nutzt: Ab sofort ist es nämlich möglich, persönliche Repositories zu erstellen, die in keinerlei Zusammenhang zu anderen Projekten stehen. Dort kann man etwa private Snippets lagern oder auch sein eigenes Projekt starten. Per Default sind diese privaten Verzeichnisse für andere Nutzer nicht sichtbar, mithilfe der neuen Repository Permissions lassen sie sich aber gezielt auch für andere öffnen.

Für alle Stash-Nutzer, die von einer früheren Version kommen, empfiehlt sich der Upgrade Guide. Stash-Neulinge sollten das kostenpflichtige Atlassian-Produkt zuerst einmal anhand der kostenlosen Probeversion ausprobieren.

Zum Abschluss noch eine Warnung: In den Release Notes zu Stash 2.4 kündigen die Entwickler an, ab Version 3.0 nur noch Java 7 und höher zu unterstützen. Stash 3.0 soll noch dieses Jahr erscheinen.

▶ http://bit.ly/ZMGOa5

Eclipse 4.3 M7 Kepler

Nicht mehr lange hin ist es bis zur Veröffentlichung der neuen Eclipse-Version 4.3. Am 26. Juni soll der Release-Train namens Kepler vollendet werden, der an die 70 auf Eclipse 4.3 abgestimmte Projekte enthalten wird. Mit dem jetzt erreichten Meilenstein 7 sind die Neuerungen der Eclipse-Plattform 4.3 bereits abgeschlossen.

Bemerkenswert an Eclipse 4.3 M7 ist vor allem das neue, offizielle API auf Basis der Eclipse-4-Architektur. Mit dem API ist ein erster Schritt getan, um das Programmiermodell der neuen Plattform direkt für eigene Anwendungen und Plug-ins nutzen zu können. Eclipse 4 führt hier Dependency Injection und Services ein. Außerdem ist es mittels eines UI-Modells möglich, mit relativ einfachen Mitteln die Struktur des User Interface zu manipulieren. Das komplette Modell ist ab sofort über das neue API ansprechbar.

In den Java Developement Tools (JDT) wurden einige Detailverbesserungen vorgenommen. So lassen sich unerwünschte Typenargumente, die vom Content-Assistenten vorgeschlagen werden, durch das Löschen von „<“ komplett entfernen. Ein neuer Quickfix „Create loop variable“ korrigiert unvollständige *For*-Schleifen, Templates und Keywords werden jetzt ohne Präfix vorgeschlagen, und bei den JUnit-Test-Templates steht nun das JUnit-4-Template an erster Stelle.

Im Plug-in Developement Environment (PDE) ist es nun möglich, JUnit-Plug-in-Test-Launch-Konfigurationen für das Plug-in-Testen zu nutzen. Ein neuer p2 Remediation Wizard hilft dabei, p2-Fehler zu beheben.

Der siebente Meilenstein ist der letzte seiner Art – nun geht es weiter mit den Releasekandidaten. Eine Übersicht aller wichtigen Neuerungen in Eclipse 4.3 M7 geben die Release Notes.

▶ http://bit.ly/113Qq33

Autor des Monats

Martin Künkele ist Inhaber der Firma SMK Software Management Kommunikation GmbH. Mit seiner Firma entwickelt er Webapplikationen auf der Basis von JDeveloper/ADF. Er ist zertifizierter Projektmanager nach P.M.I. und Certified Scrum Master der Scrum Alliance. Er beschäftigt sich mit der Frage, wie man ADF-Projekte agil durchführen kann und welche Aspekte aus Sicht des Projektmanagements zusätzlich zu berücksichtigen sind, um ein Projekt erfolgreich abzuschließen. Sein Blog befindet sich auf http://martinkuenkele.blogspot.de.

Wie bist du zur Softwareentwicklung gekommen?

Während des Informatikstudiums an der Uni Karlsruhe habe ich nebenbei als Programmierer gearbeitet und kleinere Anpassungen an einem umfangreichen Programmsystem in COBOL vorgenommen. Als Selbstständiger Anfang der 90er Jahre entwickelte ich Programme in C/C++ und später in Java für den Zahlungsverkehr zwischen Banken. Ein USP war die Implementierung des Triple-DES, also eines Algorithmus, der mit einem geheimen Schlüssel versorgt aus dem Inhalt einer Datei einen Wert berechnet. Die Dateien wurden verwendet, um Überweisungen zwischen Finanzinstituten zu übertragen, und anhand des Wertes, der an die Datei angehängt wurde, konnte der Empfänger feststellen, ob die Datei unversehrt die Übertragung überstanden hatte, was ja ziemlich wichtig war.

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Die Diskussion im Team über den besten Lösungsweg halte ich für absolut wichtig, und das begeistert mich immer wieder aufs Neue, wenn so ein Teamgedanke entsteht, wenn man spürt, dass alle an einem Strang ziehen. Das konnte ich als Projektleiter eines virtuellen Teams, bestehend aus fünfzehn hochkarätigen Spezialisten, erleben. Zuvor hatte ich bereits in einem XP-Projekt mitgearbeitet, und seit dieser sehr positiven Erfahrung hat mich die agile Vorgehensweise nicht mehr losgelassen. In diesem Projekt lernte ich auch Frameworks kennen. Das hat mich ebenfalls stark beeinflusst.

Was ist für dich ein weniger schöner Aspekt?

Wenn in einem Entwicklungsteam Egoismen vorherrschen und nach Partikularinteressen gehandelt wird. Das ist auch einer der wichtigsten Gründe, warum Projekte scheitern.

Wie und wann bist du auf Java gestoßen?

Als Alternative zur CGI-Programmierung in C++. Ich begann mit Applets und beschäftigte mich immer mehr mit der Programmiersprache. Ich arbeitete mich in die objektorientierte Programmierung mit Java ein, entwarf und implementierte die ersten Klassen und setzte das eine oder andere Entwurfsmuster intuitiv um, wie ich erst später bemerkte. Das war eine aufregende Zeit, und ich kann mich noch erinnern, wie ich ganze Abende und Wochenenden damit verbracht hatte. In diese Zeit fällt auch ein Versuch, mit einem Team in Nepal zusammenzuarbeiten, um dort die Softwareentwicklung durchführen zu lassen. Ich bin deshalb einmal für ein Wochenende nach Kathmandu geflogen, um das Team kennen zu lernen. Aus verschiedenen Gründen ist aber nach einem guten Anfang aus der Zusammenarbeit nichts geworden. Es war trotzdem eine wertvolle Erfahrung zum Thema Offshoreentwicklung.

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

Java ist inzwischen so mächtig, dass es schwierig ist, den Überblick zu behalten. Deshalb würde ich eine Art Gelbe Seiten für Java einrichten lassen.

Was ist zurzeit dein Lieblingsbuch?

Mal wieder „Die unerträgliche Leichtigkeit des Seins“ von Milan Kundera. Ich lese sehr gerne tschechische Literatur.

Was machst du in deinem anderen Leben?

Ich reise gerne mit meiner Frau. Einige Zeit verbringe ich damit, E-Gitarre zu spielen, am liebsten Fusion.

JUG-Kalender* Neues aus den User Groups

WER?	WAS?	WO?
JUG **Schweiz**	05.06.2013 – jOOQ: A peace treaty between SQL and Java	http://jug.ch
JUG **Stuttgart**	10.06.2013 – Objektforum: Schätzung von IT-Risiken: Schwierigkeiten und Best Practices	http://jugs.org
JUG **Darmstadt**	13.06.2013 – JAX-RS 2.0	http://jugda.wordpress.com
JUG **Düsseldorf**	13.06.2013 – Java in der Ausbildung	http://rheinjug.de
JUG **Ostfalen**	13.06.2013 – Java 8 – Feature	http://jug-ostfalen.de
eJUG **Austria**	25.06.2013 – „Spring 4“ und „Warum ich meinen Kunden Spring noch immer empfehle“	http://community.ejug.at
JUG **Frankfurt**	26.06.2013 – Schnelle und leichtgewichtige Anwendungsentwicklung mit HTML5 und JEE/REST	http://jugf.de
JUG **Stuttgart**	03.07.2013 – Workshop „Java für Entscheider“	http://jugs.org

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

Einführung in Node.js

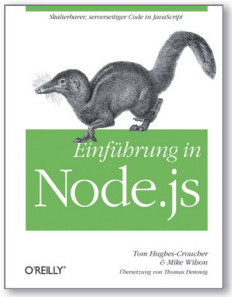
von Tom Hughes-Croucher und Mike Wilson

JavaScript, vor einigen Jahren noch gerne als „suspekt“ im Browser abgeschaltet, ist inzwischen nicht mehr fortzudenken. Spätestens mit Ajax ist diese Sprache ein wesentlicher Bestandteil der browserseitigen Webentwicklung. Doch warum nur im Browser einsetzen? Node.js ist eine JavaScript-Implementierung für den Server. Oder passender, eine JavaScript-Umgebung zum Erstellen von Servern. Bereits im Java Magazin 4.2013 wurde Node.js thematisiert. Zeit also, auch ein passendes Buch zu besprechen: Es handelt sich um die deutschsprachige Ausgabe von „Node.js up and running“ aus demselben Verlag. Es startet mit einer – so bezeichneten – „sehr kurzen Einführung in Node.js“. Dabei geht es auch um die Installation, ansonsten aber gleich um die Programmierung eines HTTP-Servers – mit weniger Zeilen, als der gesunde Mensch Finger hat. So zeigen die Autoren gleich, was für eine Leistungsfähigkeit Node.js entwickeln kann. In diesem Zusammenhang erläutern sie auch die Hintergründe der Performanz, das nicht blockierende I/O als wesentliches Schlüsselkonzept, dessen Komplexität bei der Programmierung mit Node.js aber weitgehend verborgen bleibt. Und dann gibt es weitere Appetithappen: kleine Projekte zum Start. Wenige Zeilen reichen für einen Chatserver oder einen „Twitter“-Nachbau. Danach heißt es „Robuste Node-Anwendungen bauen“. In diesem Kapitel erläutern die Autoren Hintergründe wie Eventschleife oder typische Lösungsmuster.

Teil 2 des Buchs widmet sich Details und der API-Referenz. Zumindest lautet die Überschrift derart. Wer nun mit der Erwartungshaltung einer typischen Referenz, z. B. in thematischer und alphabetischer Reihenfolge mit detaillierter Syntax, weiterliest, sollte nicht enttäuscht sein, wenn er (oder sie) etwas anderes vorfindet. Es werden zwar die Themen Core-API, Hilfs-APIs und mehr angesprochen, doch es handelt sich weitgehend um einen Prosatext. Die wichtigen Funktionen werden erläutert, die Syntax des API jedoch nicht gezeigt, zumindest nicht direkt in formaler Darstellung. Dafür bieten die Autoren Codebeispiele und Übungen an. Und über den Index, in dem auch Events und Methoden aufgelistet sind, lassen sich die entsprechenden Textstellen recht gut finden. So lässt sich das API zumindest für Einsteiger leichter lesen und verstehen. Ob aber über die Beispiele hinausgehend andere Parameter bei den Aufrufen zulässig sind oder optionale entfallen können, erfährt der Leser so nur eingeschränkt. Nun, die Syntax (nebst einiger Beispiele) ist im Web unter http://node.org nachzulesen – mit knappen Erklärungen. Diese liefert das Buch dafür ausführlicher.

Es handelt sich bei diesem Band nicht um einen wissenschaftlichen Text, sondern vielmehr um locker geschriebene Fachliteratur mit vielen Codebeispielen. Diese heißen nicht „Listing“, sondern „Übung“. Die Autoren scheinen den Leser zum direkten Nachvollziehen animieren zu wollen. Aber keine Sorge, das Buch lässt sich auch bestens fernab vom PC lesen. Es deutet an, was alles in diesem System steckt – und weckt Lust, die Erstellung einer interaktiven Webapplikation mit Node.js in Angriff zu nehmen.

Michael Müller



Tom Huges-Croucher, Mike Wilson
(deutsche Übersetzung von Thomas Demmig)

Einführung in Node.js

Skalierbarer, serverseitiger Code in JavaScript

216 Seiten, 34,90 Euro
O'Reilly, 2012
ISBN 978-3-86899-797-2

Experten teilen ihr Wissen auf Deutschlands führender Java-Konferenz

Alive and kickin'

Von bewährten Standards bis zu den „New Kids on the Block" war wieder alles dabei auf der JAX 2013 im sonnigen Mainz. Über 2000 Menschen pilgerten in die Rheingoldhalle, um eine Java-Plattform zu feiern, die erfolgreicher ist denn je.

von Claudia Fröhling, Diana Kupfer und Judith Lungstraß

Knapp 200 Speaker präsentierten dieses Jahr insgesamt 203 Talks, acht Keynotes und dreizehn Workshops. Nicht nur bewährte Standards wie Java EE, das Spring Framework und JavaServer Faces waren dabei, sondern auch neue Themen wie JavaFX, New School Enterprise IT und Embedded Technology. Nachfolgend wollen wir einige Highlights der JAX hervorheben.

The Developer Strikes Back: Spannende Eröffnungs-Keynote

Mit einem wahren Paukenschlag startete das Programm der diesjährigen und bislang größten JAX: Zur Auftakt-Keynote betrat James Governor (s. Interview auf S. 40) die Bühne, um den Wandel der IT-Landschaft aus Sicht der Entwickler zu beleuchten. „Developers are the New Kingmakers" lautete der Titel der angekündigten Keynote. Und genauso unmissverständlich war die Botschaft: In Zeiten von Open Source, Kollaboration, Diversifizierung und erhöhter Entwicklerpartizipation auf allen Ebenen hat die Emanzipation der Entwickler, in der Governor nichts Geringeres als eine „soziale Revolution" sieht, gerade erst begonnen.

In wenigen Schritten vollzog Governor einen Streifzug durch 38 Jahre Softwareentwicklung, von traditionellen, proprietären Enterprise-IT-Lösungen („Software used to be something that was given away") über Technologien und Organisationen der Neunziger Jahre, die aus seiner Sicht Game Changer darstellten (Linux, Netscape, PHP, Apache, Google), bis hin zu aktuellen Phänomenen wie Kollaboration, Social Media, Quantified Self und dem wachsenden Softwareanteil in allen erdenklichen Industriebereichen – was er mit Marc Andreessens Diktum „Software is eating the World" auf den Punkt brachte. Leitmotive der Ausführungen Governors waren erstens die unaufhaltsame Fragmentierung des Technologiemarkts und zweitens die Vorzüge von Open-Source-Entwicklung: Kollaboration, Transparenz, Effizienz und Entscheidungen, die „bottom-up", also von Entwicklern selbst, nicht in der Managementetage getroffen werden. „Open Source was beginning to change the game even as the web just took off", so der Speaker. GitHub, Eclipse, Apache, SourceForge und Co. seien letztendlich nur „the consumer version of what open source had been doing all along", also Institutionalisierungen einer Entwickler- und Arbeitskultur, deren Ausprägung bereits viel früher, in den Neunziger Jahren, begonnen hatte.

Der Managermäzen

Welche Art des Umdenkens erfordern diese Entwicklungen – die Dezentralisierung des Markts, die Konjunktur von Open Source und die Emanzipation der Entwickler – nun von Unternehmen? Hier bediente Governor die Metapher „Farmers versus Foragers", also Landwirte versus Wildbeuter. Statt bequem auf den Ertrag der eigenen Saat zu setzen und das ewig gleiche Land zu kultivieren, sollten Unternehmer auf Talentfang gehen. Und innerhalb der Unternehmen? Aufgabe des Managements, so Governor, sei es nicht länger, Entwicklern seine oft realitätsfernen Entscheidungen zu oktroyieren, sondern Letztere dazu zu befähigen, „Großartiges" zu erreichen. Um es auf den Punkt zu bringen, empfahl der Analyst IT-Managern, nicht als autoritäre Vorgesetzte, sondern als „Mäzene" (Patrons) zu wirken, ihnen den Spielraum zu gewähren und das Vertrauen zu schenken, die Innovation heutzutage unabdingbar voraussetzt.

BigDataCon

Zum dritten Mal fand dieses Jahr die BigDataCon parallel zur JAX statt. Für das Buzzword Big Data sind spezielle Technologien nötig. Und wer sonst könnte Urheber der

ersten Big-Data-Technologien sein als das Unternehmen Google, das täglich wahre Unmengen an Daten generiert und diese natürlich auch auswerten muss? So ist es kein Wunder, dass hier das Google File System, kurz GFS, und der dazugehörige MapReduce-Algorithmus ihren Ursprung haben. Der Zugriff auf die gespeicherten Daten erfolgt schließlich mittels der BigTable-Technologie.

Wie uns Lars George (Cloudera) in seiner Session mit dem Titel „HBase, die Hadoop-Datenbank – eine Einführung“ erklärte, kommt auf genau dieser Entwicklungsstufe das bekannte Big-Data-Framework Apache Hadoop ins Spiel. Dieses besteht grundlegend aus zwei Komponenten, auf der einen Seite ist da das Hadoop Distributed File System (HDFS), eine offene Implementierung des Google File Systems, auf der anderen Seite MapReduce zur Verarbeitung der erfassten Daten.

So effizient das Hadoop Distributed File System auch arbeitet, kommt es dennoch mit einer großen Einschränkung daher: Hadoop erlaubt lediglich den Schreibe- und Lesezugriff auf die erfassten Daten – modifizierbar sind diese nicht. Einmal geschrieben und abgespeichert, sind sie geschlossen, für wichtige Aktualisierungen kann man sie höchstens löschen und neu anlegen. Bei dieser Einschränkung handelt es sich aber keinesfalls um ein Manko, das den Hadoop-Verantwortlichen vorzuwerfen wäre, sondern um eine bewusste, durchaus sinnvolle Entscheidung. Schließlich vereinfacht diese das Zugriffsmodell erheblich und sorgt folglich dafür, dass Hadoop überhaupt erst mit großen Datenmengen umgehen kann.

Nichtsdestotrotz müssen Daten eben manchmal auch bearbeitet werden. Dafür gibt es HBase, kurz für Hadoop Database, eine freie Implementierung von Googles BigTable. Im Februar 2007 als Prototyp auf dem Markt, seit Oktober desselben Jahres auch als nutzbare Version verfügbar, schließt sie genau die Lücken, die Hadoop hinterlässt.

Nach einer ausführlichen Einführung in das Thema Big Data allgemein und Hadoop im Speziellen gab Lars George schließlich einen Einblick in die Architektur der Datenbank HBase.

Programmierkunst zum Anfassen: Embedded Experience Day

Software is eating the World – Mit dieser Aussage von Marc Andreessen [1] war die JAX Keynote von Mirko Novacovic betitelt. Und auch sonst zog sich das Bonmot wie ein roter Faden durch das JAX-Programm. Doch wer bei der neuen Embedded-Werkstatt der JAX vorbeischaute oder die eine oder andere Session des Embedded Experience Days verfolgte, der musste den Eindruck gewinnen, dass Hardware die Welt mindestens genauso für sich einnimmt: Wo das Internet der Dinge sich Bahn bricht und alle erdenklichen Objekte miteinander vernetzt werden, ist Software eben nicht alles, sondern nur so wichtig wie die physischen Objekte, die sie steuert. Insofern: Hardware is eating the World, too!

Dem Trend M2M im Rahmen der JAX Rechnung zu tragen, war den Konferenzorganisatoren dieses Jahr ein besonderes Anliegen. Umgesetzt wurde es in einem Dreiklang aus einem Embedded Day, einer Embedded-Werkstatt am Rande des Expobereichs sowie einer Keynote von Benjamin Cabé, der zwei der drei Eclipse-Projekte im Bereich M2M betreut.

Natürlich durfte Embedded Java dabei nicht fehlen: Terrence Barr (Oracle) erklärte in seinem Talk und an seinem Werkstatttisch die Möglichkeiten von Oracles „Device to Datacenter“-Lösung und die Rolle, die Java auf dem M2M-Markt einnehmen könnte. JavaFX für Embedded Systems, von Oracle auf der JavaOne 2012 vorgestellt, thematisierte Gerrit Grunwald. Auf die Frage „Warum Java“? hatte er klare Antworten parat: die hervorragende Toolunterstützung, die weite Verbreitung (man denke an die 10 Millionen Entwickler), die Tatsache, dass es sich um eine reife und bewährte Technologie handelt und tausende Libraries. Ähnliche Argumente führte auch Benjamin Cabé im JAX-TV-Interview ins Feld [2].

JavaFX: Warten auf die große Acht

„Bezeichnend ist, dass die größten Skeptiker nicht von den Mitbewerbern, sondern aus dem eigenen Lager – der Welt der Java-Entwickler – zu kommen scheinen.“ Dieses Zitat stammt von unserem JAX Speaker Lars Röwekamp und ist knapp fünf Jahre alt. Ahnen Sie, worum es geht? Richtig, die UI-Technologie JavaFX. Die JAX präsentiert dieses Jahr erstmals einen ganzen Special Day zu JavaFX – und das ist auch gut so.

In den letzten fünf Jahren hat sich viel getan. Oracle hat JavaFX von Sun Microsystems übernommen und (überraschend für viele) nicht direkt begraben, sondern weitergeführt, umgekrempelt, mehr Ressourcen dafür abgestellt. Das Ergebnis ist eine Technologie, die mit dem Ursprungsprojekt von 2005 nicht mehr viel zu tun hat und die sich als ernst zu nehmende Alternative zu HTML5, Swing und Co. präsentiert.

Aber warum avanciert eine schon totgeglaubte Technologie plötzlich zu einer richtigen Alternative, wenn es um User Interface Design geht? Wir sprachen auf der letzten W-JAX mit Gerrit Grunwald, Skeptiker der ersten Stunde und mittlerweile großer Fan. Er ist überzeugt, dass Oracles Commitment die Technologie so voranbringt: Milestones werden eingehalten, Roadmaps

sind klar einsehbar und werden befolgt, das Team ist, so Gerrit, „sehr kontaktfreudig“ geworden. Das komplette Interview mit Gerrit gibt es unter [3].

Zur W-JAX testeten wir erstmals einen JavaFX-Workshop, nach dessen riesigem Erfolg haben wir das Programm zur diesjährigen JAX erweitert und einen ganzen Special Day rund um die UI-Technologie geplant. Moderiert von CaptainCasa-Chef Björn Müller ging es am ersten Hauptkonferenztag um Pro und Contra von JavaFX. Adam Bien schilderte zum Auftakt in seinem Talk „Enterprise JavaFX 8“ seine ersten Gehversuche mit der Technologie. „Das geniale an JavaFX ist, wie es in XML und Inversion of Control genutzt wird“, so Adam. Zuerst habe er nicht verstanden, warum man plötzlich wieder alte Konzepte rauskrame, aber nach dem Bau der ersten eigenen Anwendungen habe er es verstanden und für gut befunden.

Inversion of Control ist ein altbekanntes Paradigma für jeden, der sich mit Objektorientierung beschäftigt: es geht um die Steuerung der Ausführung von Unterprogrammen. Im Falle von JavaFX sei genial, dass man so gut wie nie hard coden müsse, um etwas zu verändern, so Adam. Der Body wird gebaut, der Rest lässt sich durch CSS-Dateien etc. on the fly regeln. Ein Interview mit Adam finden Sie unter [4].

Unbrauchbar für Geschäftsanwendungen?

Einen sehr erfrischenden und auch überraschenden Talk präsentierte Karsten Lentzsch: er kommt aus der Swing-Welt und schult Firmen bei der Umsetzung von Geschäftsanwendungen. Zu Beginn stellte er gleich klar: Swing mag er so wenig, wie ein Zahnarzt schlechte Zähne mag. Aber es war immer sein Job, in Swing-Projekten Feuer zu löschen, und so hatte er gehofft, mit JavaFX endlich keine schlaflosen Nächte mehr zu haben. „Da wurde ich enttäuscht“, so Karsten in seinem Talk.

Seine Kritik: Effekte sind ein zentrales Element in JavaFX, nur braucht man diese für Geschäftsanwendungen so gut wie gar nicht. Außerdem sei der Komponentensatz im Vergleich zu anderen Toolkits wie beispielsweise Sencha viel zu klein für zeitgemäße Anwendungen. Es gäbe nur Basislayouts, die für Profis unbrauchbar seien.

Positiv in JavaFX bewertet er dagegen die einfachen Konzepte, das pure Toolkit und die zeitgemäße Rendering Engine. In den großen Geschäftsanwendungen seiner Kunden sieht er JavaFX dennoch nicht.

Moderator Björn Müller schließlich stellte in seinem Talk eine wichtige Frage: JavaFX oder HTML5 – was soll ich verwenden, was ist für mein Projekt die richtige Wahl? Hier ist natürlich als Erstes die Frage zu stellen, was gebraucht wird und dann die richtige Technologie für den richtigen Job zu wählen. Und wer sagt, dass man nicht beides machen kann? Sollten Sie den Talk verpasst haben, empfehlen wir Björns Artikel im Java Magazin 5.2013, in dem er diese Fragen noch einmal detailliert erläutert.

Still not there yet

Und jetzt? Wird es Zeit für Sie, alles stehen und liegen zu lassen und Ihre Anwendung ASAP auf JavaFX zu migrieren oder mit JavaFX zu konzipieren? Immerhin ist die offizielle Mitteilung von Oracle, dass Swing nicht mehr weiter entwickelt wird. Swing wird deprecated und jeder, der diese Technologie in seinem Unternehmen einsetzt, muss sich über kurz oder lang Gedanken über die Zukunft machen. Migriere ich zu JavaFX, baue ich eine neue JavaFX-Anwendung oder wähle ich einen ganz anderen Weg, zum Beispiel in Richtung HTML5.

Wer sich mit dieser Frage auseinandersetzen muss, dem muss auch klar sein, dass JavaFX aktuell noch nicht für jeden Anwendungsfall geeignet ist. Das bestätigen auch unsere Experten: Anfang 2014 wird Java 8 (hoffentlich) kommen und damit auch JavaFX 8, und diese Version wird viele Änderungen mit sich bringen und voraussichtlich abwärtskompatibel sein. Bis dahin bleibt JavaFX das, was es die letzten acht Jahre auch schon war: eine Spielerei, ein Nice-to-have. Ignorieren kann man es aber sicher nicht, das beweisen nicht zuletzt die 50+ Teilnehmer des JavaFX-Workshops am JAX-Montag und die überfüllten Räume beim JavaFX Day.

Total Eclipse of the JAX

Auf der diesjährigen JAX gab es zwar wieder speziell für Eclipse-Themen reservierte Slots und Tage, doch war Eclipse – ähnlich wie JavaFX – noch in zahlreichen weiteren Panels und Sessions vertreten. Auf der Eclipse-Agenda der JAX standen neben traditionellen Themen wie Modeling und Plug-in-Entwicklung auch neuere Bereiche wie der Brückenschlag zwischen Eclipse und JavaFX, M2M oder Entwicklung im Browser.

Wieder einmal wurde deutlich, dass Projekte aus der Mitte der Eclipse-Community sich „all over the place“ ansiedeln. Man muss Lars Vogel also respektieren für das Wunderwerk, in seiner Traditions-Session „What is hot in Eclipse?“ einen Abriss über das zurückliegende Eclipse-Jahr zu präsentieren, Kurzporträts neuer Projekte inklusive. Eine ausführliche Berichterstattung zum Thema Eclipse @ JAX gibt es im Eclipse Magazin 4.2013 und online auf JAXenter.de [5].

Project Nashorn: Zweite Chance für JavaScript auf der JVM

Viele Wege führen nach Rom und ebenso viele Wege führen auch auf die Java Virtual Machine. Das ist einer der Gründe, warum ihre Attraktivität ungebrochen ist und warum wir diesen Monat das passende Titelthema auf dem Cover haben.

Anzeige

Seit den Anfängen der JVM Mitte der Neunziger wurde eine Vielzahl von Sprachen für die Plattform implementiert, die nicht zwingend mit Java verwandt waren. Genau das war der Sinn hinter Write Once/Run Anywhere: das unabhängige Bytecode-Format lässt freie Hand, was kompiliert wird und auf der JVM laufen soll. So hat auch JavaScript seinen Weg hierher gefunden.

Aber wie genau passen nun JavaScript und die JVM zusammen und wie können Entwickler hier Code zum Laufen bringen? Eine bekannte und schon in die Jahre gekommene Möglichkeit ist die Open Source verfügbare JavaScript-Engine Rhino, die seinerzeit von Netscape ins Leben gerufen wurde und heute von der Mozilla Foundation verwaltet wird. Rhino zielt auf serverseitige Anwendungen ab und wird in selbige embedded. Rhino ist übrigens komplett in Java geschrieben. So wie das Rhinozerus ein Tier mit Millionen Jahre alter Geschichte ist, so ist auch Rhino ein betagtes Projekt.

Im Gegensatz zu Rhinos langjähriger Geschichte ist dynjs ein ganz jungfräuliches Projekt, das von Douglas Campos ins Leben gerufen wurde. Douglas arbeitet bei Red Hat und ist Project Lead der Mobile-Library AeroGear. dynjs entstand vor dem Hintergrund, Code bauen zu können, der neue Werkzeuge wie ASM und invokedynamic nutzen kann, erzählt Douglas im Interview mit dem Java Magazin, das Sie auf Seite 41 finden. Da Rhinos Code nicht aktuell genug war, hat Douglas kurzerhand sein eigenes Projekt gestartet. Mittlerweile ist dynjs bereits in vert.x integriert, der nächste Schritt ist mit nodej ein Kompatibilitäts-Layer für Node.js.

Ein zweiter Anlauf

Fassen wir noch mal zusammen: Auf der einen Seite gab es eine offizielle JavaScript-Engine, die mit Java ausgeliefert, aber nicht mit hoher Priorität weiterentwickelt wurde. Eine Engine, die zwar im offiziellen Oracle-Package enthalten, aber nicht bei Oracle gebaut wurde. Auf der anderen Seite haben wir ein junges Projekt, das als Hobby begann und vor allem dank Red Hat zur starken Alternative aufgebaut wird. Diese Situation sorgte für ein Vakuum, das gefüllt werden musste. Das Ergebnis: Project Nashorn.

Nashorn wurde 2011 auf dem JVM Language Summit von Oracle erstmals angekündigt und stellt wie Rhino und dynjs eine JavaScript-Implementierung für die JVM dar. Als Teil von Java 8 wird Nashorn nächstes Jahr verfügbar sein, der Code wurde bereits in das OpenJDK Repository überführt.

Die Namensgebung des Projekts war natürlich kein Zufall, „Nashorn“ ist das deutsche Wort für Rhinocerus und macht deutlich, dass Oracles Projekt die alte Engine von Mozilla beerbt. Es ist sozusagen JavaScripts zweite Chance für die JVM.

Auf der JAX hatten wir das Vergnügen, mit Marcus Lagergren zu sprechen, der bei Oracle im Java-Language-Team arbeitet und am Donnerstag die Engine in seiner Keynote vorstellte [6].

Marcus' Keynote war eine Liebeserklärung an die JVM. Für ihn begann alles mit Sprachen wie Lisp und Smalltalk, die Grundsteine der JVM wie Class Libraries und Garbage Collection eingeführt hatten. Heute ist die JVM das Zuhause vieler Bibliotheken, Frameworks und Sprachen, vor allem der dynamischen Sprachen. „And dynamic languages are hot!“, so Marcus. Und JavaScript im Speziellen erlebt eine wahre Renaissance. „It's spreading like a forest fire right now!“

Wie Marcus auch im Interview oben bestätigt, ist invokedynamic ein enorm wichtiger Meilenstein für die polyglotte Entwicklung und Nashorn ist sozusagen der Proof of Concept von invokedynamic. „We knew we must become the ultimate invokedynamic consumer“, resümierte Marcus in seiner Keynote. Das macht Lust auf mehr und bestätigt ein weiteres Mal, dass das Java-Ökosystem alive and kickin' ist!

JVM prominent auf der JAX

Nicht nur das Projekt Nashorn zeigt, dass das Ökosystem rund um die Java Virtual Machine „alive and kicking“ ist. Noch viel mehr spannende JVM-Themen gab es am Donnerstag im Rahmen des „Around the JVM“ Special Days, moderiert von Arno Haase. Ziel des Tages war es, das technische Zusammenspiel der JVM mit der Umgebung, in der sie läuft – seien es Betriebssysteme, Netzwerkumgebungen, Prozessoren oder Virtualisierungsplattformen – zu beleuchten. Auch mit Arno hatten wir Gelegenheit, vorab zu sprechen [7].

Mit all diesen frischen Erfahrungen von der JAX freuen wir uns jetzt auf die Planung der zweiten großen Java-Konferenz dieses Jahres: die W-JAX wird vom 4. bis 8. November in München stattfinden. Alle Updates zum Programm finden Sie wie gewohnt auf www.jax.de.

Links & Literatur

[1] http://bit.ly/16glsp2
[2] http://bit.ly/12WtVe9
[3] http://www.youtube.com/watch?v=HWuiZyMFiDc
[4] http://www.youtube.com/watch?v=ynjxEFtKoO0
[5] http://jaxenter.de/artikel/Total-Eclipse-of-the-JAX
[6] http://www.youtube.com/watch?v=vkU8D0AP6tc
[7] http://www.youtube.com/watch?v=oQF8UrXPpiw

Zeitgemäßes Arbeiten mit Dateien

java.nio.file: höher, weiter, schneller

Mit der Version 7 wurde in Java die Art und Weise, Dateien zu verwalten und Dateiinhalte zu bearbeiten, von Grund auf neu implementiert. Zwar existiert auch weiterhin die altbekannte Klasse *java.io.File*, zusätzlich gibt es nun jedoch im Package *java.nio.file* ein komplett neues und von *java.io.File* losgelöstes API zum Zugriff auf das Dateisystem. Ein erster Überblick über dieses neue API und die sich daraus ergebenden neuen Möglichkeiten im Vergleich zu *java.io.File*.

von Christian Robert

Um in Java-Versionen vor Java SE 7 mit Dateien zu arbeiten, muss die Klasse *java.io.File* genutzt werden. Exemplare dieser Klasse bilden jeweils den Java-Repräsentanten einer Datei im lokalen Dateisystem. Das eigentliche Konzept eines Dateisystems hingegen ist im klassischen *java.io.File*-API so gut wie unbekannt bzw. für den Entwickler nicht zugänglich. Zwar existiert eine Klasse namens *java.io.FileSystem*, diese ist jedoch *package-private* innerhalb von *java.io* und damit für eine Nutzung durch Clientcode nicht vorgesehen.

Implizit ist mit „Dateisystem" vor Java SE 7 immer das lokale Dateisystem gemeint. Anders ausgedrückt: Ein *java.io.File*-Objekt repräsentiert immer eine Datei im lokalen Dateisystem. Eine Möglichkeit, den Dateibegriff abstrakter und weiter zu interpretieren, ist nicht vorgesehen. Ein Dateizugriff auf ein entferntes System (beispielsweise per FTP) lässt sich mit *java.io.File* nicht realisieren und muss zwangsläufig mit externen Frameworks umgesetzt werden.

Java SE 7 hingegen macht mit *java.nio.file.FileSystem* das Konzept eines Dateisystems auch für den Entwickler explizit verfügbar. Das lokale Dateisystem ist nur eine mögliche Implementierung; beliebige andere lassen sich vom Entwickler jederzeit dem System bekannt machen und nutzen.

Es bietet sich damit eine ganz neue Möglichkeit, Abstraktionen zur Speicherung von Inhalten zu entwickeln und zu nutzen, da nicht mehr auf jeder Ebene zwischen interner Speicherung (auf dem lokalen Dateisystem) und externer Speicherung (auf einem Ziel außerhalb des lokalen Dateisystems) unterschieden werden muss.

Zugegeben: Wirklich revolutionär ist diese Idee nicht – Frameworks wie Apache Commons VFS bieten eine entsprechende Abstraktion bereits seit einigen Jahren an. Mit *java.nio.file* bietet sich jedoch erstmals die Möglichkeit, dies direkt mit Java-Bordmitteln zu lösen.

Path

Das Pendant zu *java.io.File* im *java.nio.file* Package stellt das Interface *java.nio.file.Path* dar. Im Gegensatz zu *File* stellt jedoch *Path* zunächst keinen direkten Zugriff auf die Inhalte (bzw. Metadaten) einer Datei bereit, sondern enthält lediglich Informationen zur Lokalisierung der eigentlichen Datei im Dateisystem.

FileSystemProvider

Wie bereits erwähnt, bietet *java.nio.file* eine komplette und offene Verwaltung für Dateisysteme. Einstiegspunkt ist hierbei die Klasse *FileSystemProvider*. Implementierungen dieser Klasse sind für die tatsächliche Ausführung der I/O-Operationen auf Dateien, die durch *Path*-Objekte eindeutig identifiziert werden können, verantwortlich.

FileSystem

Während *FileSystemProvider* für die tatsächliche Implementierung der I/O-Operationen verantwortlich ist, ist es die Aufgabe des *FileSystem*, die dateisystemspezifische Hierarchie zu verwalten und über *Path*-Objekte zurückzugeben. Ein *Path* ist daher immer explizit einem *FileSystem* zugeordnet und wird von diesem (bzw. seinem *FileSystemProvider*) erstellt und verwaltet.

FileStore

Als Gruppierung innerhalb eines *FileSystem* liefert *FileStore* Informationen über Devices, Partitionen oder andere Arten der *FileSystem*-Aufteilung. Im lokalen

Windows-Dateisystem existiert jeweils ein *FileStore* für jedes lokal eingebundene Laufwerk (C:\, D:\ etc.).

URIs

Bereits Java SE 1.4 erlaubte es erstmals, ein *java.io.File*-Objekt aus einem URI zu erzeugen. Akzeptiert werden allerdings nur URIs aus dem *file*-Schema. Mit *java.nio.file* wird diese Möglichkeit weiter ausgebaut, und es werden weitere Attribute des URI genutzt.

So kann direkt aus dem Schema des URI auf das zu verwendende Dateisystem geschlossen werden (Beispiele hierfür wären *file*, *ftp* oder *jar*). Ein URI erlaubt daher einen Dateinamen nicht nur relativ zu seinem Dateisystem, sondern absolut zur gesamten Hierarchie aller Dateisysteme eindeutig zu identifizieren.

Erste Zugriffe

Ein direktes Erzeugen eines *Path*-Interface ist nicht möglich. Wir erinnern uns: Ein *Path* stellt (anders als ein *java.io.File)* lediglich eine abstrakte Repräsentation dar, die erst im Zusammenspiel mit dem zugehörigen *FileSystem* bzw. *FileSystemProvider* sinnvoll verwendet werden kann. Ein *Path* benötigt (und bietet) daher immer eine Referenz auf das *FileSystem*, an das die eigentlichen I/O-Operationen delegiert werden können. Um einen *Path* einfach zu halten, können wir uns der Utility-Klasse *java.nio.file.Paths* bedienen:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
System.out.println("1 "+path.getClass());
System.out.println("2 "+path.getFileSystem().getClass());
```

Zu vergleichen ist diese *Path*-Erzeugung mit:

```
File file = new File("C:/test.txt");
```

Führen wir die oberen Codezeilen aus, so erhalten wir als Ausgabe:

```
1 class sun.nio.fs.WindowsPath
2 class sun.nio.fs.WindowsFileSystem
```

Wir sehen, dass als konkrete Ausprägungen des *Path*-Interface und der abstrakten Klasse *FileSystem* die entsprechenden Windows-Implementierungen zurückgegeben werden.

Hinter den Kulissen führt der Aufruf von *Paths.get* dazu, dass alle im System vorhandenen und registrierten Implementierungen von *FileSystemProvider* durchlaufen werden. Jede dieser Implementierungen ist für genau ein Schema verantwortlich. Das bedeutet, dass eine Implementierung von *FileSystemProvider* genau dann ausgewählt werden kann, wenn ihre Schemaangaben mit dem Schema des übergebenen URI übereinstimmen. Auf dem so ermittelten *FileSystemProvider* wird nun die Methode *getPath* aufgerufen, die die tatsächliche, zum URI passende *Path*-Implementierung zurückliefert.

Dateiinhalte lesen und verändern

Vor Java SE 7 konnte auf Dateiinhalte mittels *FileInputStream* bzw. *FileOutputStream* zugegriffen werden. Mit *java.nio.file* bietet sich eine Reihe von Möglichkeiten, die je nach Anwendungsfall gewählt werden können.

Der „klassische“ Weg, die Inhalte über *Input-* bzw. *OutputStreams* byteweise zu lesen bzw. zu schreiben, lässt sich mit *java.nio.file* wie folgt abbilden:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
InputStream inStream = Files.newInputStream(path);
```

Das hier gezeigte Codebeispiel verhält sich analog zu folgendem Beispiel nach alter Vorgehensweise mit *java.io.File*:

```
File file = new File(C:/test.txt");
InputStream inStream = new FileInputStream(file);
```

Analog zur bereits oben gezeigten Verwendung von *Paths.get* führt auch die Verwendung der Methode *Files.newInputStream* zunächst einen Look-up auf den zum *Path* (bzw. dem ihm zugeordneten *FileSystem*) gehörigen *FileSystemProvider durch*. Die Erzeugung des *InputStreams*, von dem die Dateiinhalte gelesen werden können, übernimmt dann eben jener *FileSystemProvider*. Wir hätten daher auch schreiben können:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
FileSystem fileSystem = path.getFileSystem();
FileSystemProvider provider = fileSystem.provider();
InputStream inStream = provider.newInputStream(path);
```

Eine weitere Möglichkeit, Dateiinhalte zu bearbeiten, bieten weitere Hilfsmethoden aus *java.nio.files.Files*. Wollen wir beispielsweise den Inhalt einer Datei als Array von Bytes weiterverwenden, so können wir die *Files.copy*-Methode nutzen:

```
Path path = Paths.get(URI.create("file:/C:/ test.txt"));
ByteArrayOutputStream out = new ByteArrayOutputStream();
Files.copy(path, out);
byte[] fileBytes = out.toByteArray();
```

Sehen wir uns eine vergleichbare Logik an, die das klassische *java.io.File*-API verwendet:

```
File file = new File("C:/test.txt");
FileInputStream in = new FileInputStream(file);
ByteArrayOutputStream out = new ByteArrayOutputStream();
for(int b = in.read(); b > -1; b = in.read()) {
  out.write(b);
}
byte[] fileBytes = out.toByteArray();
```

Es finden sich noch weitere Hilfsmethoden zum Lesen und Schreiben von Daten in der *Files*-Utility-Klasse, die typische I/O Use Cases abbilden.

Erweiterte Funktionalitäten

Bisher haben wir uns hauptsächlich mit den Funktionalitäten von *java.nio.file.Path* beschäftigt, die so oder ähnlich auch bereits in *java.io.File* vorhanden sind. Neben der Unterstützung für unterschiedliche Dateisysteme bietet das neue API jedoch auch weitere Funktionalitäten, die in dieser Art und Weise erstmals direkt in Java zur Verfügung gestellt werden.

Links

Links, also Verknüpfungen zwischen Dateien auf Dateisystemebene, können erst mit der Einführung von *java.nio.file* sinnvoll erkannt und bearbeitet werden. Das API unterstützt hierbei sowohl echte Links (auch *hard links* genannt) als auch symbolische Links (auch *soft links* genannt). Erstellt werden beide Arten von Links analog zum Auslesen von Dateien über die Utility-Klasse *java.nio.file.Files*:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));

Path slink = Paths.get(URI.create("file:/C:/slink.txt"));
Files.createSymbolicLink(slink, path);

Path hlink = Paths.get(URI.create("file:/C:/hlink.txt"));
Files.createLink(hlink, path);
```

Ebenfalls existiert die Möglichkeit, symbolische Links aufzulösen und auf die „echte“ Datei zu gelangen:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
try {
  Path resolvedPath = Files.readSymbolicLink(path);
} catch(NotLinkException e) {
  System.err.println("Path is not a link");
}
```

Um die hier gezeigte Exception-Behandlung zu umgehen, lässt sich auch vor der Auflösung des Links eines *Path* überprüfen, ob dieser überhaupt ein Link ist:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
if(Files.isSymbolicLink(path)) {
  doStuffWithRealFile(Files.readSymbolicLink(path));
} else {
  doStuffWithRealFile(path);
}
```

Benutzerberechtigungen

Unter Betriebssystemen, die POSIX-Dateiberechtigungen verwenden, unterstützt *java.nio.file* erstmals auch das Abfragen bzw. Setzen von Dateiberechtigungen. Der

folgende Code zeigt das Abfragen einer Berechtigung einer Datei:

```
Path path = Paths.get(URI.create("file:/tmp/file.txt"));
Set<PosixFilePermission> permissions = Files.getPosixFilePermissions(path);
boolean groupHasPermission = permissions.contains(PosixFilePermission.
                                                    GROUP_READ);
```

Auch das Setzen von Berechtigungen ist entweder programmatisch oder durch Umwandlung eines Berechtigungsstrings möglich:

```
Path path = Paths.get(URI.create("file:/tmp/file.txt"));
Set<PosixFilePermission> permissions = new HashSet<>();
permissions.add(PosixFilePermission.OWNER_READ);
permissions.add(PosixFilePermission.OWNER_WRITE);
permissions.add(PosixFilePermission.GROUP_READ);
Files.setPosixFilePermissions(path, permissions);
```

Setzen von POSIX-Dateiberechtigungen nach Umwandlung eines Berechtigungsstrings:

```
Path path = Paths.get(URI.create("file:/tmp/file.txt"));
Set<PosixFilePermission> permissions = PosixFilePermissions.
                                                fromString("rw-r-----");
Files.setPosixFilePermissions(path, permissions);
```

Werden POSIX-Dateiberechtigungen vom entsprechenden Dateisystem nicht unterstützt, so wird eine *UnsupportedOperationException* der *getPosixFilePermissions*- bzw. *setPosixFilePermissions*-Methode geworfen.

Notification

Für bestimmte Einsatzzwecke kann es notwendig sein, zu erfahren, wann eine Datei innerhalb eines Verzeichnisses geändert, neu erstellt oder gelöscht wurde. Mit Java-Bordmitteln und ohne *java.nio.file* lässt sich dies nur durch ständiges Pollen eines Verzeichnisses und Vergleich mit einem vorherigen Zustand erreichen.

Mit *java.nio.file* bietet uns das API hierfür explizite Unterstützung. Wollen wir beispielsweise erfahren, wenn innerhalb eines Verzeichnisses eine neue Datei erstellt wurde, so können wir dies mit einem einfachen Codeschnipsel (Listing 1) erreichen.

Listing 1: Notification für neue Dateien einrichten

```
Path path = Paths.get(URI.create("file:/C:/Temp/"));
WatchService watchService = path.getFileSystem().newWatchService();
WatchKey watchKey = path.register(watchService, StandardWatchEventKinds.ENTRY_CREATE);
while(true) {
  for(WatchEvent<?> event : watchKey.pollEvents()) {
    Path newPath = (Path)event.context();
    System.out.println("New file: " + newPath);
  }
}
```

Interoperabilität

Auch wenn *java.nio.file* als kompletter Ersatz für die klassische Dateibehandlung mit *java.io.File* dienen kann, so wird es immer wieder Situationen geben, wo beide APIs parallel zum Einsatz kommen müssen. Der typische Anwendungsfall hierfür dürfte bestehender Code sein, für den es keinen Grund zum Refactoring gibt oder die Verwendung externer Frameworks, die noch das alte File-API nutzen. Sowohl *java.io.File* als auch *java.nio.file.Path* bieten hierzu entsprechende Konvertierungsmethoden an:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
File file = path.toFile();
Path pathFromFile = file.toPath();
```

Zu beachten ist hierbei jedoch, dass nicht jedes *Path*-Objekt automatisch in ein *File*-Objekt umwandelbar ist (nicht jedes *Path*-Objekt repräsentiert schließlich eine Datei im lokalen Dateisystem). Im entsprechenden Fall wird beim Aufruf von *Path#toFile* eine *UnsupportedOperationException* geworfen.

Fazit

Mit dem neuen *java.nio.file*-API führt Java SE 7 ein mächtiges und leistungsfähiges neues API zur Dateiverwaltung und Dateibearbeitung ein. Wir haben einige der grundlegenden Funktionalitäten gezeigt und die hierdurch gebotenen Möglichkeiten gesehen. Es existieren allerdings noch eine Reihe weiterer interessanter Optionen (wie Traversierung), auf die wir hier nicht näher eingegangen sind.

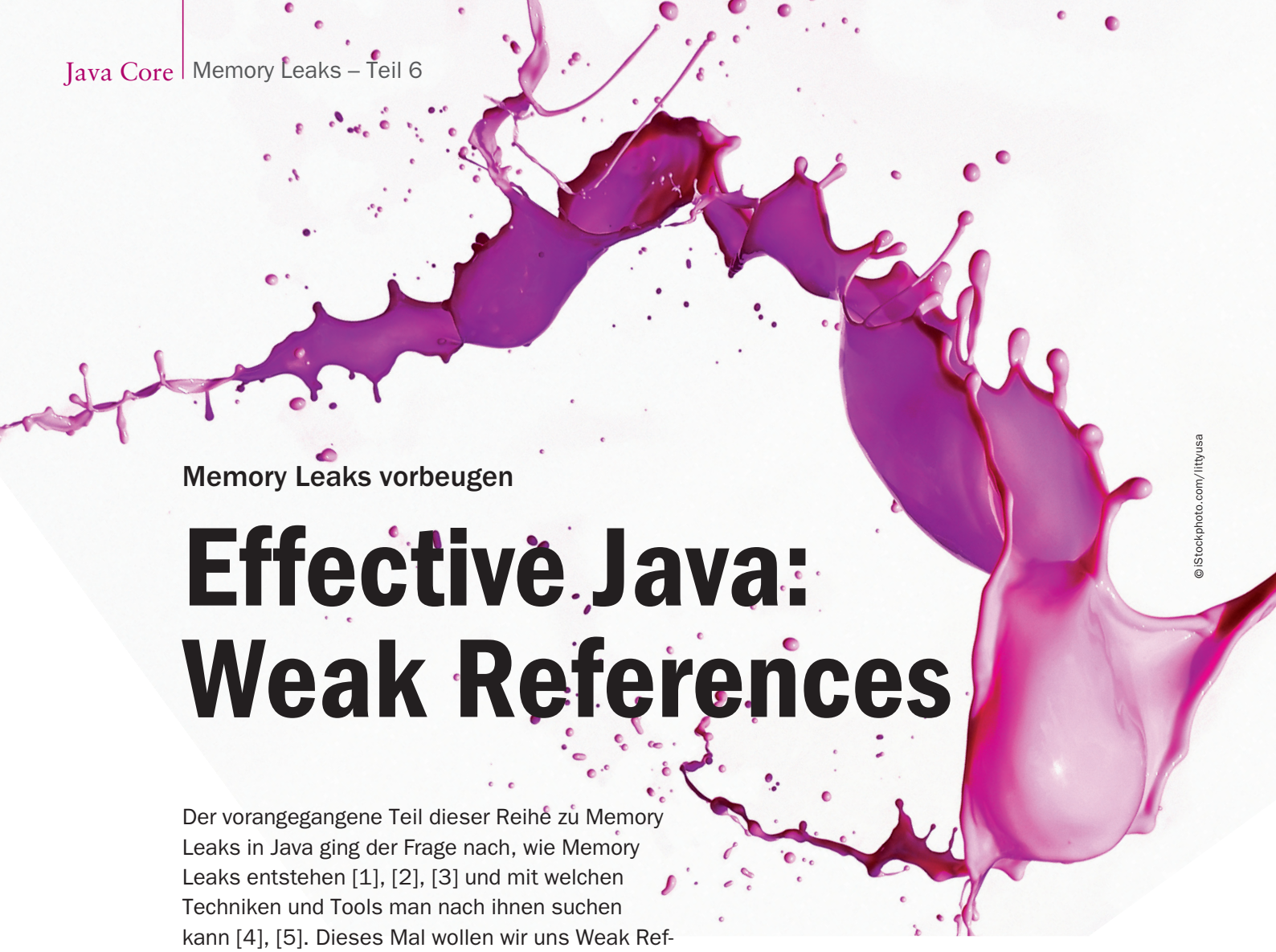
Durch die zusätzlichen Abstraktionsschichten wirkt manches auf den ersten Blick im Vergleich zum *java.io.File*-API noch ungewohnt, aber nach kurzer Einarbeitungszeit wird man die neuen Möglichkeiten schätzen und nicht mehr missen wollen. Ein kleiner Wermutstropfen bleibt die Tatsache, dass nun zwei konkurrierende APIs existieren, die Dateiverwaltung und -bearbeitung erlauben. Da jedoch *java.nio.file* alle bisherigen Anwendungsfälle – und einiges darüber hinaus – abbilden kann, ist bei neuen Projekten ein Wechsel mehr als anzuraten.

Christian Robert ist Senior Software Engineer bei der anderScore GmbH in Köln. Seit über zehn Jahren beschäftigt er sich mit der Entwicklung von Individualsoftware im Java-Umfeld. Seine aktuellen Schwerpunkte liegen in der Entwicklung von pragmatischen und dennoch (oder gerade deswegen) effizienten Architekturen sowie der Implementierung flexibler Build- und Deployment-Konzepte.

Links

http://docs.oracle.com/javase/tutorial/essential/io/fileio.html

?lxcgec



Memory Leaks vorbeugen

Effective Java: Weak References

Der vorangegangene Teil dieser Reihe zu Memory Leaks in Java ging der Frage nach, wie Memory Leaks entstehen [1], [2], [3] und mit welchen Techniken und Tools man nach ihnen suchen kann [4], [5]. Dieses Mal wollen wir uns Weak References genauer ansehen. Mit ihrer Hilfe ist es möglich, Situationen zu vermeiden, in denen Memory Leaks entstehen können.

von Klaus Kreft und Angelika Langer

Gehen wir noch einmal zurück zu dem Beispiel eines Memory Leaks, das wir ausführlich im ersten Artikel zum Thema Memory Leaks diskutiert haben [1]. In dem Beispiel haben wir einen rudimentären Server auf Basis der mit Java 7 eingeführten *AsynchronousSocketChannels* implementiert. Das Memory Leak entstand dadurch, dass wir im Server für jeden neuen Client Verwaltungsinformationen in einer Map gespeichert und diese clientspezifischen Map-Einträge nicht nach der Beendigung der Kommunikation mit den jeweiligen Clients wieder gelöscht haben. Es ist offensichtlich, dass sich die Map mit jedem neuen Client vergrößert. Erzeugt man also in einem Testprogramm viele Clients hintereinander, so stürzt der Server irgendwann mit einem *OutOfMemoryError* ab. Der Sourcecode zu dem Beispiel sowie verschiedene alternative Korrekturen finden sich unter [6].

Wie wir in dem darauf folgenden Artikel [2] diskutiert haben, ist diese Situation relativ typisch für ein Memory Leak, das zu einem *OutOfMemoryError* führt. Es gibt in Java häufig Verwaltungen, in die man Objekte eintragen und aus denen man letztere, wenn sie nicht mehr benötigt werden, auch wieder löschen muss. Ein weiteres Beispiel, das wir uns dazu angesehen haben, ist die Verwaltung der Callbacks, die man als AWT Event Listener einhängt.

In unserer Artikelserie sind wir bisher so verblieben, dass es Aufgabe des Benutzers ist, solche Verwaltungen korrekt zu verwenden. Das heißt, er muss darauf achten, dass er jedes Objekt, das er in eine solche Verwaltung eingetragen hat, auch zum angemessenen Zeit-

punkt wieder löscht. Wenn er sich nicht daran hält, hat er als Konsequenz das entstandene Memory Leak und den eventuell folgenden Programmabsturz aufgrund eines *OutOfMemoryError* zu verantworten.

Ungewollte Referenzen und Weak References

In diesem Artikel wollen wir nun sehen, ob derjenige, der die Verwaltung implementiert und bereitstellt, nicht auch etwas tun kann, was dazu führt, dass es erst gar nicht zu Memory Leaks kommen kann. Schön wäre es doch, wenn der Benutzer sich gar nicht um das Löschen kümmern müsste, weil dies automatisch erledigt wird.

Um zu verstehen, wie so etwas gehen könnte, rufen wir uns noch einmal in Erinnerung, wie es genau zu einem Memory Leak kommt. Der Garbage Collector ermittelt ausgehend von so genannten *Root References*, welche Objekte in einem Java-Programm referenziert und damit erreichbar sind. Alle nicht erreichbaren Objekte räumt der Garbage Collector bei der Garbage Collection weg und gibt ihren Speicher frei. Wenn wir nun auf ein Objekt verweisen, von dem wir sicher sagen können, dass wir es im weiteren Kontext unseres Programms gar nicht mehr benutzen werden, haben wir ein Memory Leak, denn das nicht mehr benötigte Objekt wird vom Garbage Collector nicht weggeräumt, weil es noch referenziert wird. Diese Referenz wird in der englischsprachigen Fachliteratur *unwanted reference* (also: ungewollte Referenz) genannt.

Übertragen wir diese Beschreibung auf das Memory-Leaks-Beispiel aus dem ersten Artikel: Die clientspezifischen Verwaltungsdaten werden auch nach der Beendigung der Kommunikation mit dem Client weiter über die Map referenziert, sodass der Garbage Collector sie nicht freigeben kann. Die Map mit ihrer internen Datenstruktur bildet also unsere ungewollte Referenz auf die clientspezifischen Daten.

Wie wäre es denn nun, wenn man dem Garbage Collector explizit sagen könnte, dass die ungewollten Referenzen genau das sind: ungewollt. Man möchte ausdrücken, dass diese Referenzen zwar für die Navigation im Objektgraphen der Applikation zur Verfügung stehen, damit man die Objekte erreichen kann. Aber für den Garbage Collector soll es nicht bedeuten, dass die Objekte, auf die sie verweisen, am Leben gehalten werden sollen. Genau zu diesem Zweck gibt es seit dem JDK 1.2 das Package *java.lang.ref* im JDK, mit dem neben einigen weiteren Referenztypen (Kasten: „Referenztypen im JDK“) die so genannten Weak References eingeführt wurden.

Die Idee ist, dass man ungewollte Referenzen innerhalb einer Datenverwaltung als Weak References implementiert. Damit braucht sich der Benutzer der Datenverwaltung nicht mehr um das Löschen der Objekte, die er in die Datenverwaltung eingetragen hat, zu kümmern. Das Löschen wird stattdessen automatisch vom Garbage Collector erledigt. Wie das Ganze im Detail geht, wollen wir weiter unten am Beispiel des Servers aus dem ersten Artikel [1] diskutieren. Als Erstes sehen wir uns aber die *WeakReference* selbst etwas genauer an.

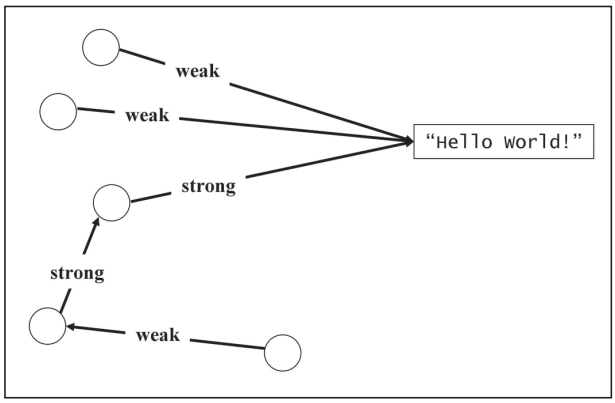


Abb. 1: „weak“-erreichbarer String

Die WeakReference im Detail

Fangen wir zuerst mit der Namensgebung an. Da man auf ein Objekt über eine normale Referenz und über eine *WeakReference* verweisen kann, reicht ein Verb wie „referenzieren“ nicht mehr aus, um die spezifische Art der Referenz zu beschreiben. In der englischen Fachliteratur haben sich deshalb die Ausdrücke *strongly referenced* für das Referenzieren über eine normale Referenz und *weakly referenced* für das Referenzieren über eine *WeakReference* gebildet. In Anlehnung daran werden wir die Ausdrücke *strong*-referenziert bzw. *weak*-referenziert verwenden. Entsprechend ergibt sich:

Ein Objekt, das *strong*-erreichbar (also über normale Referenzen erreichbar) ist, wird nicht vom Garbage Collector freigegeben; ein Objekt das *weak*-erreichbar ist, wird hingegen trotz der Erreichbarkeit vom Garbage Collector freigegeben.

Dabei gilt ein Objekt als *weak*-erreichbar, wenn alle Referenzketten von den Root-Referenzen auf das fragliche Objekt mindestens eine *weak*-Referenz enthalten. In der Verweiskette können durchaus auch normale Referenzen vorkommen, aber die Verweiskette muss mindestens eine *weak*-Referenz enthalten, damit das referenzierte Objekt als *weak*-erreichbar gilt. **Abbildung 1** zeigt eine solche Situation, in der ein String nur noch *weak*-erreichbar ist, obwohl es eine direkte Strong-Referenz auf den String gibt.

Sehen wir uns nun an, wie man eine *WeakReference* benutzt. Beginnen wir mit einer Klasse *MyClass*, die ein Feld vom Typ String hat. Das bedeutet in der Terminologie von oben, dass eine Instanz von *MyClass* das enthaltene String-Objekt *strong*-referenziert. Die dazu gehörende, recht rudimentäre Implementierung unserer Beispielklasse sieht aus wie in Listing 1.

Listing 1

```
public class MyClass {
  private String myString;

  public MyClass(String s) {
    myString = s;
  }

  public boolean doSomething(String o) {
    return myString.equalsIgnoreCase(o);
  }
}
```

Listing 2

```
public class MyClass {
  private WeakReference<String> myWeakString;

  public MyClass(String s) {
    myWeakString = new WeakReference<String>(s);
  }

  public boolean doSomething(String o) {
    String s = myWeakString.get();

    if (s == null)
    return false;
    else
    return s.equalsIgnoreCase(o);
  }
}
```

Wie sieht die Implementierung von *MyClass* nun aus, wenn wir den String nicht mehr *strong*- sondern *weak*-referenzieren wollen? Die *WeakReference* ist eine generische Klasse, die als Weak-Referenz-Wrapper um das Originalobjekt fungiert, sodass wir dann die in Listing 2 gezeigte Implementierung bekommen.

Um auf das eigentliche Objekt, das *weak*-referenziert wird, zugreifen zu können, muss temporär eine *strong*-Referenz genutzt werden. In unserem Beispiel ist dies die Stack-Variable *s* vom Typ *String* in der Methode *doSomething()*. Die *WeakReference* liefert die *Strong*-Referenz beim Aufruf von *get()* zurück, wenn das referenzierte Objekt noch existiert. *get()* liefert *null* zurück, wenn das Objekt irgendwann vorher nur noch *weak*-erreichbar war und deshalb bereits vom Garbage Collector weggeräumt worden ist. Beim Zugriff auf das referenzierte Objekt muss man also immer darauf vorbereitet sein, dass das Objekt gar nicht mehr existiert, und entsprechend im Programm mit dieser Situation umgehen können.

Es gibt noch ein weiteres interessantes Features von Weak References. Es ist möglich, bei der Konstruktion einer *WeakReference* anzugeben, dass man informiert werden möchte, wenn das Objekt, auf das diese Referenz verweist, vom Garbage Collector weggeräumt worden ist. Zu diesem Zweck muss man als zusätzliches Konstruktorargument eine Queue vom Typ *ReferenceQueue* aus dem Package *java.lang.ref* mitgeben. Der Garbage Collector stellt eine so konstruierte *WeakReference*-Instanz in die Queue, sobald er festgestellt hat, dass das referenzierte Objekt nur noch *weak*-erreichbar ist. In der *ReferenceQueue* findet man also all die Weak References, deren Objekte der Garbage Collector demnächst wegräumen wird oder bereits weggeräumt hat. Dieser Mechanismus erlaubt es dem Benutzer der *WeakReference* nun, auf diese Situation zu reagieren. Wie das konkret aussehen kann, werden wir weiter unten sehen.

Die WeakHashMap als Lösung für das Memory-Leak-Beispiel

Kommen wir wieder zurück zu unserm Memory-Leak-Beispiel aus dem ersten Artikel der Serie [1]. Wie oben bereits erwähnt, entstand das Memory Leak dadurch, dass wir für jeden neuen Client Verwaltungsinformationen in einer Map gespeichert und diese clientspezifischen Map-Einträge nicht nach der Beendigung der Kommunikation mit den jeweiligen Clients wieder gelöscht haben. Der Key des Map-Eintrags war die *ClientSession* und der Value der *AsynchronousSocketChannel* des jeweiligen Clients. Die Map, die verwendet wurde, ist als Feld der Serverklasse folgendermaßen definiert:

```
private final Map<ClientSession, AsynchronousSocketChannel>
                                                    sessionToChannel =
        new ConcurrentHashMap<ClientSession,
        AsynchronousSocketChannel>();
```

Die Map ist vom Typ *ConcurrentHashMap*, weil die *completed()*-Methoden der *read()*-Callbacks konkurrierend auf die Map zugreifen. Wie oben bereits gesagt, kann man den vollständigen Code unter [6] herunterladen.

Die Idee ist nun, die ungewollten Referenzen, die unsere Map auf die *ClientSession* und den *AsynchronousSocketChannel* hält, geschickt durch Weak References zu ersetzen und so das Memory Leak zu vermeiden. Die Sourcecode-Änderung für diese Korrektur ist minimal. Die Erklärung, warum sie funktioniert, ist erheblich länger. Also schauen wir uns zuerst die Änderung an, dann kommen wir zu den ausführlichen Erläuterungen. Die Änderung besteht darin, dass wir die Map nun folgendermaßen definieren:

```
private final Map<ClientSession, AsynchronousSocketChannel>
                                                    sessionToChannel =
        Collections.synchronizedMap(new WeakHashMap<ClientSession,
          AsynchronousSocketChannel>());
```

Die primäre Idee ist, eine *WeakHashMap* zu verwenden. Da diese aber nicht Thread-sicher bei konkurrierendem Zugriff ist, müssen wir zusätzlich einen *synchronizedMap* Wrapper verwenden. Zugegeben, bei hoher Last wird die *synchronizedMap* nicht so gut skalieren wie eine *ConcurrentHashMap*. Aber mit diesem Nachteil wollen wir leben, denn immerhin haben wir auf diese Weise mit minimalem Aufwand das Memory Leak behoben.

Warum das so ist und wie die *WeakHashMap* im Detail funktioniert, wollen wir uns nun ansehen. Wie die *WeakReference* kam auch die *WeakHashMap* mit der Version 1.2 zum JDK dazu. Die *WeakHashMap* implementiert eine Hash Map, bei der der Key nur *weak*-referenziert wird. Das heißt, wenn das Key-Objekt eines Eintrags nicht von anderer Stelle im Programm *strong*-referenziert wird, räumt der Garbage Collector das Key-Objekt weg. Die *WeakHashMap* lässt sich darüber vom Garbage Collector informieren. Dies geht über den Mechanismus mit der *ReferenceQueue*, den wir oben bereits beschrieben haben. Als Reaktion auf diese Notifikation löscht die *WeakHashMap* den gesamten noch verbliebenen Eintrag, bestehend aus der (bereits entleerten) *WeakReference* als Key und dem dazugehörigen Value.

In unserem Beispiel löst die *WeakHashMap* das Memory-Leak-Problem. Entscheidend dafür ist, dass der Key (d. h. die *ClientSession*) genauso lange von außerhalb der Map *strong*-referenziert wird, wie die Verbindung mit dem Client besteht. Dies ist bei uns gegeben, wie ein Blick auf die *completed()*-Methode des *read()* Callbacks bestätigt (Listing 3).

Wie man sehen kann, schleusen wir die *ClientSession* immer wieder als zweiten Parameter des *read()* zum

nächsten Callback-Aufruf. Das heißt, die notwendige *Strong*-Referenz auf das *ClientSession*-Objekt kommt aus dem *AsynchronousSocketChannel*-Framework im JDK, wo das Objekt für den nächsten Callback-Aufruf über eine *Strong*-Referenz zwischengespeichert wird. Offensichtlich ist auch, dass diese *Strong*-Referenz nicht mehr existiert, wenn die Kommunikation mit dem Client beendet ist. Dann landen wir nämlich im *else*-Zweig der *completed()*-Methode, und es wird kein erneuter *read()*-Aufruf (mit der *ClientSession* als Parameter) gemacht.

Auf den ersten Blick mag es so aussehen, als wäre es allein ein glücklicher Zufall, dass die *WeakHashMap* mit so geringem Aufwand unser Memory-Leak-Problem löst. Dem ist aber nicht so. Die Situation ist sogar eher typisch. Denn im Allgemeinen hat man eine *Strong*-Referenz auf das Key-Objekt, mit dem man auf die Map zugreift, bis zu dem Zeitpunkt, wo dann die Map zur ungewollten Referenz auf das Key-Objekt und das Value-Objekt wird. Genau da hilft einem dann das spezifische Verhalten der *WeakHashMap*, denn Key- und Value-Objekt werden nun von der *WeakHashMap* in Zusammenarbeit mit dem Garbage Collector freigegeben.

Die WeakHashMap im Detail

Werfen wir am Ende noch einen Blick auf einige Implementierungsdetails der *WeakHashMap*. Dies hilft auch zu verstehen, wie man Weak References in eigenen Implementierungen sinnvoll einsetzt.

Typisch ist, dass man meist nicht die *WeakReference* direkt nutzt, sondern eine Klasse davon ableitet, die zusätzliche Attribute und Funktionalität enthält. Bei der *WeakHashMap* ist diese von *WeakReference* abgeleitete Klasse die private Inner Class Entry, die zusätzlich das Interface *Map.Entry* implementiert:

```
private static class Entry<K,V> extends WeakReference<Object> implements
                                                    Map.Entry<K,V> { ... }
```

Ein Entry besteht aus dem eigentlichen Key-Value-Paar und einer Reihe weiterer Attribute. Dabei besteht das

Referenztypen im JDK

Neben der *WeakReference* sind mit Java 1.2 die *SoftReference* und die *PhantomReference* zum JDK hinzugekommen. Die Soft References haben dasselbe API wie Weak References. Charakteristisch ist für sie, dass in ihrer JavaDoc relativ explizite Aussagen gemacht werden, wann und wie sie vom Garbage Collector weggeräumt werden:

„All soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an OutOfMemoryError. Otherwise no constraints are placed upon the time at which a soft reference will be cleared or the order in which a set of such references to different objects will be cleared. Virtual machine implementations are, however, encouraged to bias against clearing recently-created or recently-used soft references.“

Die Idee ist, dass man Soft References dazu nutzt, um eigene Caches zu implementieren. Die gecachten Objekte werden im Cache über Soft References referenziert. So wird die Cachegröße über den Garbage Collector kontrolliert. Bevor der Cache zu groß wird und die JVM mit *OutOfMemoryError* abstürzt, werden die *soft*-referenzierten Objekte im Cache vom Garbage Collector weggeräumt. Das Ganze klingt erst einmal nach einem ziemlich schlüssigen Konzept. In der Praxis hat sich der Ansatz aber als nicht gerade problemlos erwiesen. Das liegt zum einen daran, dass die Behandlung der Soft References und somit auch das Cache-Verhalten vom konkret genutzten Garbage-Collection-Algorithmus abhängt. Das ist eine Abhängigkeit, der man sich häufig nicht bewusst ist und die man meist auch nicht haben möchte. Zum anderen bedeutet das Wegräumen der *SoftReference* zusätzliche Arbeit für den Garbage Collector. Wenn nun beim Programmablauf möglichst kurze Garbage-Collection-Pausen erreicht werden sollen, ist die Benutzung von Soft References für den Cache kontraproduktiv.

Unterm Strich bleibt festzustellen, dass es meist keine gute Idee ist, die Cachegröße allein mithilfe von Soft References über den Garbage Collector zu kontrollieren. Es ist aber durchaus sinnvoll, eine eigene Cachegrößenkontrolle zu implementieren und die Soft References als zusätzliches Sicherheitsnetz zu nutzen, damit der Cache die JVM nicht mit *OutOfMemoryError* abstürzen lässt.

Die Phantom References sind völlig anders als die *Weak*- und *Soft*-Referenzen. Wie wir in diesem Beitrag erläutert haben, werden *weak*- oder *soft*-erreichbare Objekte vom Garbage Collector nach unterschiedlichen Strategien weggeräumt, und es bleiben leere Referenzen übrig. Die *Phantom*-Referenzen bewirken das Gegenteil: Sie verhindern die Speicherfreigabe der referenzierten Objekte. Eine *PhantomReference* auf ein Objekt besteht sogar dann noch, wenn der Garbage Collector den Finalizer für das referenzierte Objekt bereits ausgeführt hat. Es bleibt eine *Phantom*-Referenz auf ein finalisiertes und damit unbrauchbares Objekt zurück, das bereits alle seine Ressourcen abgeräumt hat. Deshalb werden die Phantom References auch als *Post-mortem*-Referenzen bezeichnet: Eigentlich ist das referenzierte Objekt schon tot, und die Referenz verweist nur noch auf einen nicht mehr genutzten Speicherbereich ohne sinnvollen Inhalt.

Phantom References waren für die Implementierung eigener Memory Pools gedacht. Mithilfe der Phantom References könnte man die Pool-Elemente am Memory Management des Garbage Collectors vorbei recyceln und wiederbenutzen. Da aber die heutigen Garbage-Kollektoren ihre Aufgabe so gut erledigen, dass es in der Praxis wenig Sinn macht, einen eigenen Memory Pool in der Anwendung zu implementieren, sind Phantom References in der Praxis bedeutungslos und hier nur der Vollständigkeit halber erwähnt.

Key-Value-Paar aus einer *weak*-Referenz auf den Key und einer *strong*-Referenz auf den Value. Die Bestandteile eines *Entry* kann man gut an der Implementierung des Konstruktors von Entry sehen:

```
Entry(Object key, V value, ReferenceQueue<Object> queue, int hash,
                                                    Entry<K,V> next) {
  super(key, queue);
  this.value = value;
  this.hash  = hash;
  this.next  = next;
}
```

Der Key wird als erstes Argument an den Konstruktor der Superklasse *WeakReference* übergeben. Damit ist der Key nicht über eine normale *Strong*-Referenz im Entry abgelegt, sondern in eine *Weak*-Referenz verpackt. Der Value hingegen ist über eine normale Referenz (als Attribut *value*) im *Entry* abgelegt.

An den Konstruktor der Superklasse *WeakReference* wird neben dem Key als zweites Argument die Queue übergeben, in der die *WeakReference*-Instanzen (oder genauer: die *Entry*-Instanzen) landen, deren *weak*-referenzierte Objekte vom Garbage Collector weggeräumt werden. Diese Queue wird später von der privaten Methode *expungeStaleEntries()* verwendet, um Key-Value-Paare mit nur noch *weak*-erreichbarem Key aus der Map zu entfernen (siehe unten).

Listing 3

```
public void completed(Integer len, ClientSession clSession) {
  AsynchronousSocketChannel channel = sessionToChannel.get(clSession);
  if (clSession.handleInput(buf, len)) {
    buf.clear();
    channel.read(buf, clSession, this);
  }
  else {
    try {
      channel.close();
    } catch (IOException e) { /* ignore */ }
  }
}
```

Ganz wichtig ist der Hashcode, der im *Entry* abgelegt ist. Er wird von der privaten Methode *expungeStaleEntries()* benötigt, um die Key-Value-Paare mit *weak*-erreichbarem Key in der Map zu finden und zu entfernen. Der Key ist ja nur noch *weak*-erreichbar (oder schon ganz weggeräumt) und kann deshalb nicht mehr verwendet werden. Es wird dabei ausgenutzt, dass die Positionen der Entries in der Map vom Hashcode der Entries

abhängen. Für die Berechnung des Hashcodes eines *Entry*s wird der Hashcode des Keys benötigt. Wie bereits erwähnt, ist der Key aber nur noch *weak*-erreichbar und möglicherweise sogar bereits vom Garbage Collector weggeräumt worden. Genau deshalb wird der Hashcode des Keys im *Entry* als eigenes Feld gespeichert. Auf diese Weise ist der Hashcode des Keys noch verfügbar, wenn der Key selbst bereits verschwunden ist. So kann das *Entry* dann in der Map gelöscht werden.

Die private Methode *expungeStaleEntries()* wird in den *public*-Methoden der *WeakHashMap* aufgerufen. Wir haben die *WeakHashMap* in einen *synchronizedMap* Wrapper verpackt; der *synchronizedMap* Wrapper sorgt dafür, dass alle *public*-Methoden synchronisiert sind. Damit ist auch das interne Löschen der *Entry*s aus der *WeakHashMap* gegen konkurrierende Zugriffe von außen auf die *WeakHashMap* geschützt, und unsere Lösung ist Thread-sicher.

Zusammenfassung und Ausblick

In diesem Artikel haben wir uns angesehen, wie Weak References helfen können, Memory Leaks zu vermeiden. Die zentrale Idee dabei ist, die ungewollten Referenzen in Verwaltungen, die zu Memory Leaks führen, als Weak References zu implementieren. Handelt es sich bei der Verwaltung um eine Map, so kann man in vielen Fällen gleich die *WeakHashMap* benutzen. Die eigentliche Änderung im Code ist dann mit sehr geringem Aufwand verbunden, wie wir in unserem Beispiel gesehen haben.

Mit diesem Artikel schließen wir unsere Reihe über Memory Leaks in Java. Das nächste Thema, dem wir uns widmen werden, ist Java 8 und die dazugehörigen Neuerungen. Den ersten Artikel dazu lesen Sie in der Ausgabe 9.2013 des Java Magazins.

Angelika Langer arbeitet selbstständig als Trainer mit einem eigenen Curriculum von Java- und C++-Kursen. Kontakt: http://www.AngelikaLanger.com.

Klaus Kreft arbeitet selbstständig als Consultant und Trainer. Kontakt: http://www.AngelikaLanger.com.

Links & Literatur

[1] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 1: Ein Beispiel“, in Java Magazin 9.2012, S. 12–18

[2] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 2: Akkumulation“, in Java Magazin 11.2012, S. 30–33

[3] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 3: Ausgenullt“, in Java Magazin 1.2013, S. 16–21

[4] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 4: Tool Time“, in Java Magazin 3.2013, S. 48-52

[5] Kreft, Klaus, Angelika Langer: „Memory Leaks, Teil 5: post mortem“, in Java Magazin 5.2013, S. 12-15

[6] Sourcecode für das Memory-Leak-Beispiel aus diesem Artikel: http://www.angelikalanger.com/Articles/EffectiveJava/66.Mem.Analysis/66.Mem.Analysis.zip

„SegmentedArrayList“ implements „java.util.List“

Mit Erfolg durch die Krise

Die Klasse *SegmentedArrayList* ist eine Implementierung von *java.util.List*, optimiert für die effiziente Bearbeitung einer sehr großen Anzahl von Elementen. Das folgende Fallbeispiel schildert, wie diese Liste in einem Release erfolgreich eingeführt wurde, und erläutert die Struktur und Funktionsweise der Klasse *SegmentedArrayList*.

von Robert Bruckbauer und Zoran Savić

Viele Softwareprojekte scheitern, und die Gründe dafür sind so unterschiedlich wie die Software selbst. Es gibt aber auch Softwareprojekte, die jahrelang erfolgreich in die Produktion eingeführt werden. Die Gründe für den Erfolg sind ebenso vielfältig. Der Erfolg wird oft anhand der Menge umgesetzter Anforderungen oder der Anzahl behobener Defekte definiert. Er wird auch an der Zufriedenheit der Benutzer gemessen. Und Benutzer sind vor allem dann zufrieden, wenn Anforderungen rasch umgesetzt werden. Mit jeder Einführung erwarten Manager und Benutzer erfolgreicher Softwareprojekte daher neue Funktionen. Rein technische Änderungen zur Stabilisierung der Software lassen sich dann nur noch unter der Bedingung umsetzen, dass mit der Einführung zumindest eine Leistungssteigerung verbunden ist.

Im konkreten Fall möchten wir ein Softwareprojekt eines deutschen Logistikunternehmens beleuchten, das seit dem Jahr 2006 insgesamt zwölf erfolgreiche Einführungen in die Produktion geschafft hat. Gemäß der Definition am Beginn des Artikels ist dieses Projekt erfolgreich. Der funktionale Umfang (bewer-

Das System

FABL (Anm.: Name wurde geändert) ist ein rechnergestütztes Betriebsleitsystem. Das System ist mehrbenutzer- und mandantenfähig. Es gibt sehr hohe Anforderungen an die Verfügbarkeit (24/7, Wartungsfenster nur am Wochenende) und an die Wiederherstellung im Katastrophenfall. Kurz gesagt: Ein Stillstand oder Ausfall des Systems gefährdet den Erfolg des Unternehmens.

Das System

- hat eine grafische Benutzeroberfläche, mit der Disponenten schnell und effektiv auf Störungen im Betriebsablauf reagieren können. Mit diesem Werkzeug können sich Disponenten Übersicht über den aktuellen Fahrplan, die Betriebslage (aktuelle Position der Fahrzeuge) und die bereits durchgeführten und dokumentierten Maßnahmen (Änderungen aufgrund von Störungen) ansehen. Der Client wird laufend mit aktuellen Daten aus dem Betrieb versorgt.
- besitzt eine Schnittstelle zur Anwendung auf mobilen Endgeräten, die Mitarbeiter auf den Fahrzeugen und in den Niederlassungen automatisch über Abweichungen vom Fahrplan oder über Dienständerungen informiert.
- stellt besonders berechtigten Disponenten ein Werkzeug zur Stammdatenpflege zur Verfügung, um Stammdaten zur Infrastruktur (Niederlassungen, Strecken etc), der Fahrzeuge (z. B. Kennzeichen) und Personale (z. B. Kontaktdaten) zu pflegen.
- sammelt eine Fülle von Daten aus dem Fahrbetrieb in einem Langzeitdatenarchiv und stellt dem Management ein Berichtswesen zur Verfügung, das die Erstellung tabellarischer und grafischer Kennzahlenberichte und Auswertungen erlaubt (Business Intelligence).
- sendet laufend Statusmeldungen zur Verfügbarkeit von Komponenten und Schnittstellen. Ein Werkzeug zur Überwachung aggregiert diese Statusmeldungen zu einer Übersicht über den Betriebszustand des Systems (Business Activity Monitoring).

Abb. 1: Interne Struktur der Klasse „ArrayList“ [7]

tet durch Function Points) verdoppelte sich in dieser Zeit. Die technische Architektur blieb in dieser Zeit nahezu unverändert, sieht man von den im Lauf der Zeit notwendigen Aktualisierungen von Plattform (Java, aktuell Version 6), Middleware (JBoss, aktuell Version 5.1) und Datenbank (Oracle, aktuell Version 11.2) ab.

Die technische Architektur

FABL besitzt eine Mehrschichtarchitektur, bestehend aus einem Verband (kein Cluster) mehrerer JBoss-Applikationsserver (Version 5.1) und einem Oracle-Datenbankserver, Rich Clients auf Basis von Swing/AWT und Fremdprodukten für Business Intelligence und Business Application Monitoring. Das Grundkonzept der Architektur basiert auf den Entwurfsmustern:

- MVC [1] (Model View Controller) als Grundkonzept für die Gestaltung der Clients
- CQRS [2] (Command Query Responsibility Segregation) als Konzept für die Bereitstellung und Änderung von Daten
- EDA [3] (Event-driven Architecture) als Konzept für eine effektive Umsetzung der internen Prozesse
- CEP [4] (Complex Event Processing) als Konzept für eine optimale und rasche Verarbeitung der Daten

Die Rich Clients laden und speichern Daten mit Enterprise Java Beans und empfangen automatisch verteilte Ereignisse direkt vom Applikationsserver mit dem Messaging Service von JBoss. Als Werkzeuge für Business Intelligence und Business Acitivity Monitoring werden Standardprodukte eingesetzt.

Property „SortsOnUpdates“ der Klasse „DefaultRowSorter“ [9]

```
/**
 * If true, specifies that a sort should happen when the underlying
      model is updated (<code>rowsUpdated</code> is invoked). For
   example, if this is true and the user edits an entry the location of
              that item in the view may change. The default is false.
 *
 * @param sortsOnUpdates whether or not to sort on update events
 */
public void setSortsOnUpdates(boolean sortsOnUpdates) {
  this.sortsOnUpdates = sortsOnUpdates;
}
```

Das dreizehnte Release von FABL enthielt eine Reihe von funktionalen Änderungen, für die neue Objekte eingeführt und eine Reihe von existierenden Objekten verändert wurden. Zum Laden der neuen Objekte gab es eine neue technische Schnittstelle. Zusätzlich hatte sich die Häufigkeit und Menge der Ereignisse innerhalb des Applikationsservers und damit auch das Volumen der an die Clients zu verteilenden Daten erhöht. In der Benutzeroberfläche wurden zwei neue Ansichten zur Darstellung der neuen Objekte eingeführt. Die Mengengerüste der bereits vorhandenen Objekte veränderten sich nicht. Die Anzahl der Benutzer blieb auch gleich.

Das technische Design für die fachlichen Anforderungen erforderte keine Änderungen an der technischen Architektur und Plattform. Middleware und Datenbank waren bereits ausreichend aktuell.

In Anbetracht des Reifegrads der Software zeigte das Ergebnis des Systemtests ein unerwartet instabiles System. Unsere erste Analyse zeigte, dass der Client mehr Speicher verbrauchte, als anhand der Menge der neuen Objekte zu erwarten war. Außerdem zeigte der Client vorerst nicht erklärbare Stillstände: Die Anwendung reagierte nicht mehr auf Eingaben von Maus und Tastatur. Die Dauer der Stillstände bewegte sich zwischen einer Sekunde und länger als eine Minute – auch abhängig von der Hardware des Arbeitsplatzes.

Listing 1: Pseudocode „ArrayList“

```
class  ArrayList<E>
{

  E[] data;
  int size;

  boolean add(E element) {
    data[size++] = element;
    return true;
  }

  E get(int idx) {
    return data[idx];
  }

  int indexOf(E element) {
    for (int i = 0; i < size; i++)
      if (element.equals(data[i]))
        return i;
  }

  E remove(int idx) {
    numMoved = size - idx - 1;
    arraycopy (data, idx+1, data, idx, numMoved);
  }

}
```

Löschen wir das letzte Element der Liste, wird einfach der Eintrag aus dem Index entfernt.

Der Client wurde mit mehr Speicher ausgestattet (ca. 30 Prozent) – eine Maßnahme, die durch die Einführung der neuen Objekte gut begründet werden konnte. Leider zeigte diese Maßnahme nicht die gewünschte Wirkung: Die Stillstände traten in unveränderter Häufigkeit auf. Eine weitere Erhöhung der Speichergrenze war nicht möglich, weil die Ausstattung der Arbeitsplätze dies nicht zuließ. Neue Lösungsansätze waren nun erforderlich. Wir analysierten parallel den Speicher- und CPU-Verbrauch des Clients, mit dem Ziel, beides zu verringern:

- Wir versuchten, Anzahl und Größe von Objekten im Client zu reduzieren. Für die Analyse verwendeten wir die Kennwerte aus dem Business Application Monitoring, die im Zuge eines Lasttests erhoben wurden. Da der Systemtest aber schon abgeschlossen war, konnten wir mit vernünftigem Risiko keine wirksamen Maßnahmen mehr umsetzen.
- Wir waren auf der Suche nach einem Speicherleck und verwendeten Profiler (VisualVM, JProfiler, YourKit, dynaTrace), um die Speicherbelegung des Clients zu analysieren. Wir fanden keine Fehler, nur durch fachliche Einschränkungen konnten wir eine leichte Verbesserung erreichen.
- Wir überprüften die Einstellungen der Garbage Collection der VM. Wir experimentierten mit *DisableExplicitGC*, *PermSize*, *NewSize* und *SurvivorRatio,* wie im Technology Network [5] beschrieben. Wir erreichten eine Einstellung, die die Notwendigkeit einer Full GC massiv reduzierte und gleichzeitig die Dauer einer Full GC auf maximal zwei Sekunden reduzierte.
- Wir analysierten jeden CPU-Hotspot, den uns die Profiler zeigten. Hier hatten wir endlich Erfolge.

Es zeigte sich, dass bei Systemen, die in hohem Maß mit Ereignissen arbeiten, welche in aktualisierenden Ansichten angezeigt werden, auch besondere Anforderungen an die Implementierung mit Swing/AWT gestellt werden. Der Wert *true* für die Property *SortsOnUpdates* der Implementierung eines *RowSorter* zeigte Wirkung in einer Ansicht, die sehr häufig durch Ereignisse aktualisiert wird. Der CPU-Verbrauch des Clients konnte mit dieser Maßnahme spürbar verringert werden.

Durch die besondere Eigenschaft des von uns eingesetzten Profilers dynaTrace [6] konnten wir Ereignisse von der Quelle (in unserem Fall die Datenbank) über den Applikationsserver bis zum Client verfolgen. Dabei konnten wir überprüfen, ob alle Objekte, die den Client erreichen und in Ansichten angezeigt werden, nach Ablauf ihrer Gültigkeit bzw. Sichtbarkeit auch wieder aus den Ansichten gelöscht werden. Wir fanden keine wirklich gravierenden Fehler, ermittelten in diesem Zusammenhang aber die exakten Mengengerüste der Objekte auf dem Client. Wir fanden eine Ansicht, in der bis zu 30 000 Objekte in einem *TableModel* [8] eingetragen und laufend aktualisiert wurden. Dabei stellte sich heraus, dass eine *ArrayList* [7] hier enorme Schwächen hat.

Die interne Struktur der Klasse *ArrayList*, wie in **Abbildung 1** dargestellt, ist ziemlich einfach. Es handelt sich um ein Array (*E[]*), dessen Größe bei der Initialisierung der Liste festgelegt wird. Wenn die Kapazität dieses Arrays für die neuen Elemente nicht reicht, wird ein größeres Array erzeugt. Die existierenden Elemente werden vom alten Array in das neue kopiert.

In Listing 1 werden die Methoden der *ArrayList* gezeigt, die im CPU-Hotspot unserer Anwendung verwendet werden. Dabei ist zu beachten, dass hier keine vollständige Implementierung dargestellt wird, sondern eine vereinfachte Variante (Pseudocode), um die grundsätzliche Funktionsweise der Methoden zu erklären.

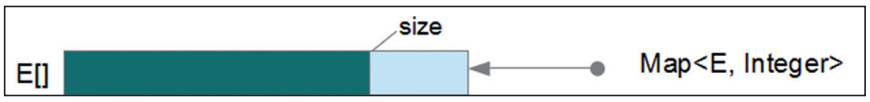


Abb. 2: Zuordnung des Index zu „ArrayList“ als Verbesserungsmaßnahme

In den Methoden *add* und *get* wird auf die Elemente direkt zugegriffen, was dazu führt, dass die Methoden immer schnell sind, unabhängig davon, wie groß die Liste ist. In den Methoden *indexOf* und *remove* wird jedoch auf die Elemente sequenziell zugegriffen. Die Laufzeit steigt dadurch proportional mit der Anzahl der Elemente in der Liste.

Die Laufzeit einer Methode werden wir in diesem Artikel mit dem Ausdruck *t(method, n)* beschreiben, wobei *method* der Name der Methode und *n* die Anzahl der Elemente ist. Die Abhängigkeit selbst wird durch die asymptotische Komplexität in der O-Notation dargestellt. Für unsere Methoden aus Listing 1 sehen die Laufzeiten folgendermaßen aus:

- *t(add, n) = O(1)*
- *t(get, n) = O(1)*
- *t(indexOf, n) = O(n)*
- *t(remove, n) = O(n)*

Die erste Verbesserungsmaßnahme ist die Optimierung der Methode *indexOf* durch die Einführung eines Index. Dafür wird eine *HashMap<E, Integer>* als Index verwendet. Beim Hinzufügen eines Elements in der Liste wird gleichzeitig der Index aktualisiert, das Element ist der Key und die Position des Elements in der Liste ist der Value (**Abb. 2**).

In der Methode *indexOf* muss das Element nicht mehr sequenziell gesucht werden, weil die Position des Elements im Index nachgeschlagen wird. Die Laufzeitkomplexität dieser Abfrage beträgt *O(1)*. Diese Lösung hat aber zur Konsequenz, dass die Aktualisierung des Index bei einer Änderung der Liste länger dauert als bisher. Dabei ist die Methode *add* nicht kritisch, es muss schließlich nur ein Eintrag dem Index hinzugefügt werden. Tatsächlich liegt das Problem in der Methode *remove*. Um das Verhalten dieser Methode zu erklären, betrachten wir das Löschen des letzten und ersten Elements der Liste.

Löschen wir das letzte Element der Liste, wird einfach der Eintrag aus dem Index entfernt. Löschen wir jedoch das erste Element der Liste, ist die Aktualisierung des Index sehr viel aufwändiger. Es reicht nicht nur, den Eintrag aus dem Index zu entfernen. In allen verbleibenden Einträgen muss der Value auch um eins reduziert werden, da die Position des entsprechenden Elements in der Liste um eine Position verschoben wurde. Die

Listing 2: Pseudocode „SegmentedArrayList“

```
/**
 * <ul>
 * <li>no constructor</li>
 * <li>no duplicate elements</li>
 * <li>not null safe</li>
 * <li>no validation code</li>
 * </ul>
 */
class  SegmentedArrayList<E>
{

  int[] segmentSize;                // array of segment sizes
  E[][] segmentData;                // array of segment data
  Map<E, Integer>[] segmentIndex; // array of segment indexes
  int segmentCursor;                // segment to add elements

  boolean add(E element) {
    segmentData[segmentCursor][segmentSize[segmentCursor]] = element;
    segmentIndex[segmentCursor].put(element,
                                    segmentSize[segmentCursor]);
    segmentSize[segmentCursor]++;
    return true;
  }

  E get(int idx) {
    int segment = calculateSegment(idx);
    int position = calculatePosition(idx);
    return segmentData[segment][position];
  }

  int indexOf(E element) {
    for (int segment = 0; segment <= segmentCursor; segment++) {
      Integer position = segmentIndex[segment].get(element);
      if (position != null) {
        return  position + calculatePreviousSegmentsSize(segment);
      }
    }
    return -1;
  }

  E remove(int idx) {
    int segment = calculateSegment(idx);
    int position = calculatePosition(idx);
    E removedElement = segmentData[segment][position];
    int numMoved = segmentSize[segment] - position - 1;
    arraycopy (segmentData[segment], position+1, segmentData[segment],
                                                    position, numMoved);
    segmentIndex[segment].remove(removedElement);
    applySegmentIndex(segment, position);
    return removedElement;
  }
}
```

durchschnittliche Laufzeit beim Löschen eines beliebigen Elements aus der Liste liegt zwischen diesen beiden extremen Fällen.

Jetzt sind wir wieder am Anfang. Die Methode *indexOf* ist zwar schneller geworden, da dort keine sequenzielle Suche mehr notwendig ist. Gleichzeitig hat sich die Methode *remove* verlangsamt, weil eine sequenzielle und aufwändige Bearbeitung des Index durchzuführen ist. Diese Laufzeit ist nun *O(n)*, mit der ursprünglichen Implementierung bekommen wir nun *2*O(n)*. Für unsere Methoden aus Listing 1 sehen die Laufzeiten nun so aus:

- *t(add, n) = O(1)*
- *t(get, n) = O(1)*
- *t(indexOf, n) = O(1)*
- *t(remove, n) = 2*O(n)*

Es stellt sich nun die Frage: Können wir die *remove*-Methode beschleunigen? Als Maßnahme teilen wir das große Array *E[]* in mehrere kleine Arrays, so genannte *Segmente*. Eine sequenzielle Verarbeitung, die bei einem *remove* notwendig ist, lokalisieren wir dann nur auf ein Segment, in dem sich das zu löschende Element befindet. Die anderen Segmente bleiben dabei unverändert. Einen großen Index brauchen wir auch nicht, wir können an jedem Segment einen eigenen lokalen Index

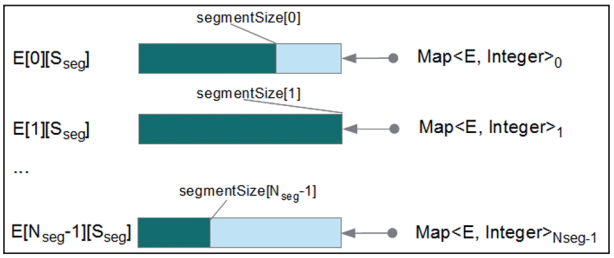


Abb. 3: Interne Struktur der Klasse „SegmentedArrayList“

verwenden. Die Aktualisierung eines solchen Index erfolgt dann ebenfalls effizienter. Diese Struktur ist in der **Abbildung 3** dargestellt.

Auf Basis der bisherigen Erklärungen wird die Klasse *SegmentedArrayList* gebaut. Der Pseudocode der Klasse wird in Listing 2 dargestellt.

In der Methode *add* ist ein bisschen mehr als in der entsprechenden Methode der Klasse *ArrayList* zu tun. Da es sich grundsätzlich nur um einfache Aktualisierung in einem Array und einer *HashMap* handelt, bleibt die Komplexität dieser Methode unverändert. In der Methode *get* sind zwei Berechnungen durch die Methoden *calculateSegment* und *calculatePosition* durchzuführen. Die Methode *calculateSegment* findet das Segment, in dem sich das gesuchte Element befindet, die Methode *calculatePosition* findet die Position des Elements in die-

Diese Probleme sind bei hochwertiger Architektur und gutem technischem Design lösbar.

sem Segment. Da beide Berechnungen nicht aufwändig sind, bleibt die gesamte asymptotische Komplexität der Methode *get* praktisch unverändert.

In der Methode *indexOf* wird durch die Indexe aller Segmente iteriert, bis das gesuchte Element gefunden wird. Wenn wir die Anzahl von Segmenten als *Nseg* bezeichnen und wenn wir berücksichtigen, dass die asymptotische Komplexität bei einer Index-Abfrage *O(1)* beträgt, können wir die Laufzeit dieser Methode mit *Nseg*O(1)* beschreiben. Die Laufzeit der Methode *remove* bleibt bei *2*O(n)*. Da die Methode *remove* nur die Elemente eines Segments aktualisiert, ist die Zahl *n* nun begrenzt, sie überschreitet nicht die Segmentgröße *Sseg*. In der Methode *applySegmentIndex* wird nur durch den Index des Segments iteriert, das durch die Löschung eines Elements betroffen ist. Der Value des Index wird jedoch nur für die Elemente aktualisiert, die sich nach dem gelöschten Element befinden. Schließlich sehen die Laufzeiten der Methoden aus Listing 1 folgendermaßen aus:

- *t(add, n) = O(1)*
- *t(get, n) = O(1)*
- *t(indexOf, n) = Nseg*O(1)*
- *t(remove, n) = 2*O(n), n< Sseg*

Auf den ersten Blick haben die Änderungen in den Methoden *indexOf* und *remove* nicht viel bewirkt. Die Praxis sieht aber ganz anders aus. Mit einer vernünftigen Auswahl der Größe der Segmente können wir die Geschwindigkeit aller betrachteten Methoden gut steuern. In FABL wurde zum Beispiel durch die Ausführung mehrerer Tests festgestellt, dass eine Segmentgröße von 1 000 Elementen (*Sseg* = 1000) in einem Array von 30 000 Elementen (*Nseg* = 30) die optimalen Laufzeiten der Methoden bringt. Zusätzliche Tests haben gezeigt, dass die Implementierung bis zu einer Listengröße von 500 000 Elementen sehr effizient arbeitet.

Fazit

Lange Zeit waren wir mit unserer Software erfolgreich. Zwar wurden wir laufend mit neuen Anforderungen konfrontiert – durch Änderungen der Geschäftsprozesse, der rechtlichen Grundlagen (z. B. Tarifverträge, Arbeitsrecht) oder der Wartungsverträge für Plattformen, Middleware oder Datenbanken. Doch durch eine geringe Fluktuation im Entwicklerteam war es uns möglich, über einen langen Zeitraum eine hohe Qualität in Sachen technisches Design und Implementierung zu bewahren. Trotzdem traten die in diesem Artikel beschriebenen Probleme unerwartet auf und stellten für uns eine große Herausforderung dar. Die gute Nachricht: Diese Probleme sind bei hochwertiger Architektur und gutem technischem Design ohne Weiteres lösbar. Die Optimierungsmaßnahmen beschränken sich nicht nur auf Klassen, die im Projekt erstellt wurden, sondern erstrecken sich auch auf etablierten Lösungen aus dem JDK. Eine Klasse wie die *ArrayList* kann durchaus auch in Frage gestellt werden.

Robert Bruckbauer ist Berater bei der eMundo GmbH in München (www.emundo.de). Seit 1990 ist er in der Softwareentwicklung tätig und entwickelte anfangs mit Fortran, C und C++; mittlerweile verfügt er über zehn Jahre Erfahrung in der Programmierung von Anwendungen mit Java und Java EE. Der Schwerpunkt der Tätigkeit in Projekten ist mittlerweile der Entwurf und die Umsetzung technischer Architekturen, das technische Design von Anwendungskomponenten und das Coaching von Entwicklerteams. Aktuell ist er im Bereich der technischen Architektur und Entwicklung für einen weltweit tätigen Mobilitätskonzern tätig.

Zoran Savić ist Berater bei der BLUECARAT AG in Köln (www.bluecarat.de). Er konnte in den letzten zwanzig Jahren vielfältige Erfahrungen im Bereich Softwareentwicklung machen. Er entwickelte mit C, C++ und Visual Basic, mittlerweile verfügt er über zehn Jahre Erfahrung in der Programmierung von Anwendungen mit Java und Java EE. Parallel hierzu vertiefte er seine Kenntnisse im Bereich Datenbanken und konnte dieses zur praktischen Anwendung bringen. Hierbei konnte er auch entsprechende Branchenkenntnisse in den Bereichen Automotive, Banking und Transport sammeln. Er zog seine Motivation zur selbstständigen Weiterentwicklung aus dem Wunsch, die zu betreuenden Systeme immer weiter zu optimieren. Aktuell ist er im Bereich der technischen Architektur und Entwicklung für einen weltweit tätigen Mobilitätskonzern tätig.

Links & Literatur

[1] Software Architecture Pattern MVC (Model View Controller): http://en.wikipedia.org/wiki/model-view-controller

[2] Software Architecture Pattern CQRS (Command Query Responsibility Segregation): http://martinfowler.com/bliki/CQRS.html

[3] Software Architecture Pattern EDA (Event-driven Architecture): http://en.wikipedia.org/wiki/event-driven_architecture

[4] Software Architecture Pattern CEP (Complex Event Processing): http://en.wikipedia.org/wiki/complex_event_processing_(CEP)

[5] Virtual Machine Garbage Collection Tuning: http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

[6] The Magic of PurePath Technology: http://www.compuware.com/application-performance-management/apm-dynatrace-purepath-technology.html

[7] java.util.ArrayList: http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

[8] javax.swing.table.TableModel: http://docs.oracle.com/javase/6/docs/api/javax/swing/table/TableModel.html

[9] javax.swing.DefaultRowSorter: http://docs.oracle.com/javase/6/docs/api/javax/swing/DefaultRowSorter.html

Java, die neuen JVM-Sprachen und andere Herausforderungen

TREFFEN DER GENERATIONEN

Gehört die Zukunft weiterhin Java? Diese Frage haben wir uns sicherlich alle in den letzten Jahren häufiger gestellt. Der Boom der Java-Plattform [1] scheint ungebrochen, aber neue Wege werden beschritten. Der TIOBE Index [2] weist auf eine etwa gleichbleibend hohe Verwendung der Programmiersprache Java hin. Weitere Programmiersprachen wie Groovy, Ruby, Clojure oder JavaScript auf der JVM werden indes immer beliebter.

von Peter Roßbach

Als Nachfolger der Sprache Java steht Scala hoch im Kurs und findet täglich neue Anhänger. Mit Play und Akka werden heute sehr effektive, nicht blockierende, skalierbare und nachrichtenorientierte Anwendungen realisiert [3], [4]. Eine neue Art der Modellierung der Probleme durch massive Parallelität und Funktionsorientierung in den Algorithmen findet Einzug in den Lösungskatalog vieler Entwickler. Die Effektivität und Änderbarkeit von Software sind für viele Unternehmen von großem Wert. Hier versprechen die neuen Programmiersprachen ihre größten Vorteile: weniger Code, klarere Konventionen, schlankere Syntax und Anwendung von funktionaler Programmierung.

Hybride Systeme

Die Risiken eines kompletten Wechsels können auf der JVM durch den Einsatz mehrerer Sprachen in einem Projekt minimiert werden. Aktuell wird die Verwendung verschiedener Programmiersprachen zur angemessenen Implementierung verschiedener Aufgaben in einem Projekt erfolgreich erprobt. Mit diesem Schritt werden an die Entwickler und die Organisationen neue Herausforderungen gestellt: Ein Entwickler muss gleichzeitig mehrere Programmierstile, -methoden und -sprachen gut beherrschen.

Besonders gefordert sind hier die Firmen in Sachen Auswahl und Schulung der Mitarbeiter. Nicht jedes Unternehmen findet eine Lösung für die Bereitstellung der Wartungsverpflichtungen solch hybrider Produkte. Die Komponenten solcher Systeme werden in übersichtliche Teilsysteme mit abgegrenztem Funktionsumfang aufgeteilt. So hofft man, dass jeder Teil langfristig beherrschbar bleibt. Eine Umgebung, die eine hybride Entwicklung in besonderer Weise fördert, ist vert.x. [5]. Hier werden moderne asynchrone Programmierung und die Verwendung von verschiedenen Programmiersprachen in einer einheitlichen Ablaufumgebung geboten.

Java: die Nachteile

Was hingegen die ausschließliche Programmierung mit der Sprache Java angeht, so machen der langsame Innovationszyklus und der nicht immer erkennbare letzte Wille an der Weiterentwicklung nachdenklich. Java bekommt immer mehr Eigenschaften. Damit wird die Sprache scheibchenweise komplexer, und die Attraktivität für bestehende und neue Entwickler hält sich in Grenzen, da die Erwartungen nicht zeitnah erfüllt werden. Gerade die hitzigen Diskussionen um Modularisierung, Cloud-Fähigkeiten oder die Einführung von funktionaler Programmierung in Form von Lambda-Ausdrücken in Java 8 zeigen dieses Problem deutlich. Die Kompatibilität zu bestehendem Code zu bewahren ist sicherlich ein hohes Gut der Plattform Java, aber offensichtlich auch die größte Einschränkung für Erneuerung und Erweiterung. Die geschwätzige Syntax von Java kann da dem Vergleich mit einigen Alternativen immer schwerer standhalten. Allerdings ist die Syntax von Java vielen vertraut. Missverständnisse in der Interpretation sind die Ausnahme. Fehlende Kenntnisse der Konventionen der neuen Sprachen sind hier leider eher eine Quelle für Irritationen. Die Implementierung in Java ist sehr explizit, aber der Code ist bestimmt nicht immer elegant.

Java: die Vorteile

Trotzdem setzen Teams weiter auf Java als Programmiersprache. Einer der Gründe liegt sicherlich darin, dass Unternehmen in den letzten fünfzehn Jahren erfolgreich Systeme mit Java umgesetzt haben. In und mit Java gibt es fast für alles ein Framework oder eine adäquate Ablaufplattform wie OSGi, JEE, Web oder ESB Container. Das ist für viele von uns die schönste Verführung, die Java realisiert hat. Vielfach genutzte, wiederverwendbare und ausgereifte Lösungen in einer nahezu perfekten Umgebung mit reichhaltiger Werkzeugunterstützung sind so entstanden. Diesen Luxus wollen wir nicht ohne langfristige Verbesserung aufgeben. Die letzten Jahre haben gezeigt, dass auf der Plattform Java ebenfalls neue Impulse zeitverzögert realisiert werden. Anderen Umgebungen fällt es in Teilen schwer, mit den vorhandenen Lösungen der Java-Plattform Schritt zu halten. Die vielen Frameworks und Lösungen bereitzustellen, ist eine zeitraubende Beschäftigung. Die gleichen

Position Apr 2013	Position Apr 2012	Delta in Position	Programming Language	Ratings Apr 2013	Delta Apr 2012	Status
1	1	=	C	17.862%	+0.31%	A
2	2	=	Java	17.681%	+0.65%	A
3	3	=	C++	9.714%	+0.82%	A
4	4	=	Objective-C	9.598%	+1.36%	A
5	5	=	C#	6.150%	–1.20%	A
6	6	=	PHP	5.428%	+0.14%	A
7	7	=	(Visual) Basic	4.699%	–0.26%	A
8	8	=	Python	4.442%	+0.78%	A
9	10	↑	Perl	2.335%	–0.05%	A
10	11	↑	Ruby	1.972%	+0.46%	A

Abb. 1: TIOBE Index (Quelle: [2])

Konzepte nur in einer anderen Programmiersprachenvariation bereitzustellen, überzeugt längst nicht jedes Team.

Gemeinsam sind wir stark

So ist ein gutes Nebeneinander von Programmiersprachen auf einer leistungsfähigen JVM von entscheidender Bedeutung für die Zukunft der Plattform Java. Von dieser Stärke profitieren natürlich alle Programmiersprachen. Welche Herausforderungen stellen sich da für die Zukunft eigentlich noch und wohin wird uns die Reise führen?

Die zunehmende Verwendung der NoSQL-Lösungen in Java, angefangen bei Hadoop über CouchDB, Cassandra, ElasticSearch, Solr und Lucene bis hin zu Neo4j ist beeindruckend und anhaltend. Hier wird es für Java immer leistungsfähigere Anbindungen und Verwendungen in Lösungen geben. JavaFX ist ein aussichtsreicher Kandidat für den Einsatz in der Welt der Embedded Systems: Für lokale Informationssysteme in der Industrie und POS-Systeme ist es eine große Bereicherung. Schon heute kann man so leicht mit einem „Raspberry Pi“ [6] kostengünstige Embedded Systems herstellen. Die Dalvik Virtual Machine ist die Basis der Android-Plattform. Die Anwendungen werden mit der Programmiersprache Java implementiert. Hier profitiert der Markt von der Kenntnis der vielen Java-Entwickler und der Portierung existierender Frameworks. Die Übertragung von Lösungen ist hier in vollem Gange, um immer besser persönliche Informationen zwischen Nutzern auszutauschen. Die Plattform Java wird von vielen Cloud-Anbietern für die individuellen Lösungen im Netz bereitgestellt. Die Google App Engine, Jelastic, oder Heroku erfreuen sich großer Beliebtheit bei den Start-ups und als Testumgebungen. In der Zukunft wird hier ein großes Potenzial an Lösungen existieren und die Standardisierung wird voranschreiten. Die Industrialisierung von Ablaufumgebungen und Services ist eine große Chance für die IT und birgt ein signifikantes Zukunftspotenzial für Java. Der Support der großen Open-Source-Communitys Apache und Eclipse Foundation verstärkt den Trend, Java in der Zukunft weiterhin intensiv zu nutzen. Der neue und alte Trend wird sein, mehr auf und mit der Java-Plattform zu produzieren. Für den eigenen Erfolg wird entscheidend sein, die richtige Programmiersprache für seine Aufgaben zu wählen.

Peter Roßbach ist freiberuflicher Systemarchitekt und Coach zahlreicher Java-EE-Anwendungen. Sein besonderes Interesse liegt in der Entwicklung von komplexen Informationssystemen, einschließlich der Gestaltung und Realisierung von testgetriebenen Softwareprozessen. Seit 1997 wirkt Peter Roßbach im Bereich „Java Server und Servlets“ und veröffentlichte das gleichnamige Buch und zahlreiche Fachartikel. Mit dem Buch „Tomcat 4x“ und als Autor der TomC@-Kolumne möchte er Sie für das Apache-Tomcat-Projekt begeistern. Er ist aktiver Entwickler des Tomcats und Mitglied der Apache Software Foundation (tomcat.apache.org).

pr@objektpark.de

Links & Literatur

[1] http://www.oracle.com/de/technologies/java/overview/index.html

[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

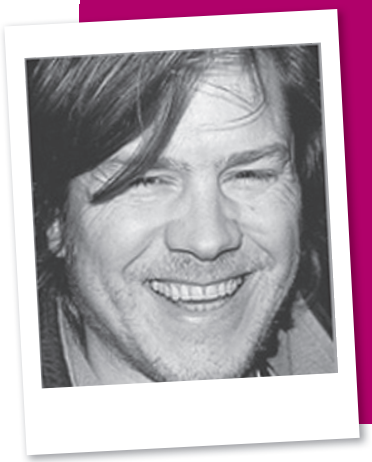
[3] http://akka.io

[4] http://www.playframework.com

[5] http://vertx.io

[6] http://www.raspberrypi.org

Anzeige



James Governor zur neuen Freiheit der Entwickler

„Developers are the new kingmakers."

Java Magazin: Hallo James. Auf der JAX 2013 hast du die Keynote „Developers are the new kingmakers" präsentiert. Entwickler sind diejenigen, die auf einer gewählten Plattform arbeiten müssen, damit das Geschäft läuft. Ist es dieser Tage schwer, zu wählen?
James Governor: Es ist schwieriger denn je, oder leichter – das hängt davon ab, aus welchem Blickwinkel man es betrachtet. Wir beobachten derzeit eine steil ansteigende Fragmentierung in den Bereichen Sprachen, Frameworks und Datenspeicherung. Aber die Frage verfehlt etwas das Thema der New Kingmakers. Entwickler müssen nicht machen, was ihnen gesagt wird, und sie müssen keine Sprachen nutzen, die sie nicht wollen. Wenn dir die Technologien, für die sich dein Arbeitgeber entscheidet, nicht gefallen, dann kündige! Es gibt eine Vielzahl von Entwicklerstellen da draußen. Wir stehen am Beginn einer neuen Epoche der Freiheit für Softwareentwickler, besonders für die talentierten unter ihnen.

JM: Es ist schwierig, eine Wahl fürs Leben zu treffen. Vielleicht ist es eher ein evolutionärer Prozess: als Start-up bzw. noch sehr junge Firma muss man schnell agieren können, sonst verliert man das Rennen. Kein Start-up würde eine schwergewichtige Technologie zu Beginn wählen. Aber wenn das ernste Geschäft beginnt, benötigt man auch ernsthafte Technologien. Was denkst du darüber?
Governor: Es ist kein Geheimnis, dass viele Webunternehmen, die ursprünglich Java und ähnliche Technologien abgelehnt haben, heute JVM-Läden sind und oft Java als Sprache nutzen, zum Beispiel Twitter oder Facebook. Aber ich bin immer vorsichtig damit, bestimmte Technologien im Vergleich zu anderen als „schwergewichtig" oder „ernsthaft" abzustempeln. Wir bei RedMonk haben Themen wie Open Source, Cloud Computing, NoSQL, den LAMP Stack und andere Technologien genau verfolgt, die ursprünglich abgelehnt wurden mit der Begründung, dass sie „nicht skalieren". Offen gesagt machen gute Entwickler Technologie skalierbar.

JM: SoundCloud hat kürzlich bekannt gegeben, dass sie Ruby on Rails aufgeben und zur JVM wechseln. RoR skaliere nicht gut, auch wenn die Arbeit damit viel Spaß mache. Man kann aber sagen, dass RoR hoch produktiv ist, wenn es um einfache Aufgaben geht. Provokant gefragt: Geht es etwa um Spaß versus Erfolg?
Governor: Ganz so einfach ist das nicht. Es hängt davon ab, wie man Skalierbarkeit definiert. Ich störe mich auch an der Unterscheidung zwischen Spaß und Erfolg. Spaß *ist* Erfolg. Siehe GitHub oder Etsy. Wir haben kürzlich erfahren, dass der National Health Service in seinem Spine-2-Projekt RIAK verstärkt einsetzen wird und dafür weniger Oracle nutzen wird – aber das bedeutet ja nicht, dass Oracle nicht ordentlich skaliert, oder?

JM: Um zurück zur ersten Frage zu kommen: Wie schwer fällt es Entwicklern, die richtige Sprache für die JVM zu wählen?
Governor: Es ist schwierig, eine Sprache zu wählen, weil es einfach so viel Innovation in Sachen JVM-Sprachen gibt. Aber meiner Meinung nach ist der erste Schritt, polyglott zu werden, denn dann bringt die Wahl mehr Spaß mit sich und der Entwickler kann die richtige Sprache für den jeweiligen Kontext wählen und das richtige Werkzeug für die (jeweilige) Aufgabe.

Porträt

James Governor ist Mitbegründer von RedMonk, einer Open-Source-Analystenfirma, die sich auf die Interessensgruppe der Entwickler spezialisiert hat. Er berät Unternehmen, Start-ups und große Konzerne wie IBM und Microsoft in Sachen entwicklergesteuerte Innovation, Community- und Technologiestrategien. James ist Koautor des O'Reilly-Buchs „Web 2.0 Design Patterns: What Architects and Entrepreneurs need to know". RedMonk hat Social-Media-Tools umfassend im Einsatz – James (@monkchips) hat beispielsweise mehr als 9 000 Follower auf Twitter. Er wird außerdem vom Institute of Analyst Relations in den Top 5 der Analysten weltweit gelistet. Als Vorsitzender des „External Panel for Stakeholder Aussurance in Sustainability Strategy and Reporting" 2009 leitete er die Bildung des Greenmonk Sustainability Advisory Services, einer RedMonk-Tochterfirma.

Interview mit Douglas Campos zu seinem Projekt dynjs und der Attraktivität der Java Virtual Machine

„Die JVM ist ein Meisterwerk der Ingenieurskunst.“

Java Magazin: Douglas, erzähl uns bitte etwas über dich: Du hast dynjs gegründet und arbeitest an Red Hats Open-Source-Bibliothek für Mobile-Apps, AeroGear. Habe ich etwas vergessen?
Douglas Campos: Genau, ich bin der Erfinder für dynjs, außerdem JRuby-Committer, seit langer Zeit OSS-Contributor und Project Lead von AeroGear.

JM: In deinem Blog steht, dass du dynjs ins Leben gerufen hast, weil es so schwer war, JavaScript auf der JVM laufen zu lassen. Kannst du das genauer erläutern?
Campos: Naja, JavaScript auf der JVN laufen zu lassen ist schwierig, aber das war eigentlich nicht der Grund, warum ich das Projekt gestartet habe. Ich wollte einen Talk über JavaScript auf der JVM halten und habe im Zuge dessen angefangen, die Sourcen von Rhino zu durchforsten, aber die Codebasis war anscheinend in die Jahre gekommen. Versteht mich nicht falsch: Die Rhino-Jungs sind super, es ist unglaublich, wie weit sie damit gekommen sind, dass Rhino so schnell läuft.

Ich wollte Code haben, der so lesbar wie irgend möglich für Neulinge ist; Code, der neue Tools nutzt, wie ASM, und der die Vorteile von invokedynamic nutzt. Da das nach einem spaßigen Projekt klang, habe ich mich direkt draufgestürzt.

JM: Da du invokedynamic erwähnst: Das kam mit Java 7, gemeinsam mit Method Handles. Beides wird gemeinhin als großer Schritt in Richtung einer polyglotten JVM verstanden. Was hältst du von invokedynamic?
Campos: Das ist die beste Erfindung seit geschnitten Brot und Bier! invokedynamic eröffnet uns eine Unmenge an Möglichkeiten, dynamische Sprachen auf der JVM schnell zu gestalten.

JM: Wie unterscheidet sich dynjs vom Nashorn-Projekt, das bei Oracle entwickelt wird und ebenfalls ein Versuch ist, JavaScript auf die JVM zu bringen?
Campos: Ich habe dynjs auf der brasilianischen Konferenz namens BrazilJS im Mai 2011 bekannt gegeben, Oracle kündigte Nashorn drei Monate später auf dem JVM Language Summit an, mit dem Hinweis „open source: TBD“ auf den Slides. Das war für mich das K.O.-Kriterium, denn ich wollte offen entwickeln, sobald es mir möglich war, mit Nashorn ein „Hello World“ zum Laufen zu bringen.

Beide Projekte haben dasselbe Ziel, aber dynjs ist komplett unabhängig vom OpenJDK-Entwicklungszyklus.

JM: SoundCloud hat kürzlich bekannt gegeben, zur JVM zu wechseln und ist damit in die Fußstapfen von Firmen wie Netflix, Twitter & Co. getreten. Ist die JVM plötzlich wieder „hip“ oder ist es die Vielfalt an Sprachen auf der JVM, die für ihre Attraktivität sorgen? Oder gar ein ganz anderer Grund?
Campos: Ich wage zu behaupten, dass es nicht die Sprachen auf der JVM sind, die sie so attraktiv machen, sondern die Tatsache, dass das Java-Ökosystem sehr solide ist. Die Java Virtual Machine ist ein Meisterwerk der Ingenieurskunst, in das Tausende Stunden Denkarbeit investiert wurden. Viele Sprachen zur Auswahl zu haben, ist wichtig, aber das Tüpfelchen auf dem i ist das Verknüpfen und das Nutzen all des Wissens, das bereits existiert.

JM: Was sind die nächsten Schritte für dynjs?
Campos: Wir haben dynjs schon in vert.x integriert und arbeiten gerade an nodej, das ist ein Node.js-Kompatibilitäts-Layer. Parallel dazu wird an der Performance gearbeitet.

Hinweis: Das Interview wurde von der Redaktion aus dem Englischen übersetzt.

Porträt

Douglas Campos ist Entwickler, Klarinettist, Vater, Christ, Stadt-Radler, Geek und Lehrer. Seine Leidenschaft gilt JRuby und anderen JVM-basierten Sprachen. Er ist der Begründer des dynjs-Projekts und arbeitet bei Red Hat als AeroGear Project Lead.

Interview mit Arno Haase, Moderator des Around the JVM Days auf der JAX

„invokedynamic war ein politischer Push."

Java Magazin: Arno, du bist hier auf der JAX Moderator des Around the JVM Days. In der Beschreibung des Tages heißt es „Vorträge von Geeks für Geeks". Was kann man sich darunter vorstellen?

Arno Haase: Ich habe mir einfach überlegt, was ich gern selbst für Vorträge auf der JAX hören würde. Wo finde ich eine Ergänzung des Angebots noch spannender. Und dabei kamen dann Talks zustande, die da anfangen, wo andere aufhören, und die ganz in die Tiefe gehen. Das habe ich dann vorgeschlagen und es ist auf Resonanz gestoßen und so ist dann der Day entstanden. Als thematischer Fokus hat sich herauskristallisiert, wie die JVM mit ihrer Umgebung interagiert. Also: mit dem Betriebssystem, mit dem Speicher, mit dem Netzwerk. Wie sie Dinge in den Orkus zieht, oder mit in den Orkus gezogen wird und sich eventuell auch wieder erholt.

JM: Stichwort JVM. Man hat das Gefühl, sie ist so attraktiv wie nie zuvor. Wo siehst du persönlich den Grund dafür?

Porträt

Arno Haase ist selbstständiger Softwarearchitekt aus Braunschweig. Neben seiner Beratungstätigkeit krempelt er gerne die Ärmel hoch und schreibt selbst Software, sowohl in Kundenprojekten als auch Open Source. Er ist eines der Gründungsmitglieder von www.se-radio.net, dem Software Engineering Podcast. Außerdem spricht er regelmäßig auf einschlägigen Konferenzen und ist Autor diverser Artikel und Patterns sowie Koautor von Büchern über JPA und modellgetriebene Softwareentwicklung.

Haase: Der Hauptgrund für die Attraktivität der JVM ist in meinen Augen, dass sie unglaublich gut optimiert. Sie nimmt einem wahnsinnig viel Arbeit bei der Optimierung ab, der Just-in-Time-Compiler wird fast von Monat zu Monat besser. Mit jedem Build sind Optimierungen enthalten, die besser funktionieren. Es gibt noch mehr Einstell- und Tuning-Möglichkeiten. Das ist ein wichtiger Grund. Sie ist inzwischen einfach sehr schnell, ohne dass man viel dafür tun muss. Ein anderer Grund ist, dass es auf der JVM eine riesige Fülle an tollen Frameworks und Bibliotheken gibt, wo man dann nicht mal an eine Programmiersprache gebunden ist, weil die JVM-Sprachen im Wesentlichen ja gut interoperabel sind. Davon kann man natürlich profitieren. Genau wie von der riesigen Open-Source-Community. Und nicht zuletzt läuft die JVM überall, also auf allen gängigen Serverplattformen. Schritt für Schritt sind die bekannten Versprechen Realität geworden und man profitiert davon.

JM: Da hat man es ja umgekehrt eher schwer, sich bei der Auswahl zu entscheiden. Du selbst kommst ja aus der Scala-Ecke, aber im Grunde kann man ja je nach Anwendungsfall entscheiden: Jetzt nehmen wir Scala, für den nächsten Punkt ist aber etwas anderes besser geeignet.

Haase: Im Prinzip ja, soweit die Theorie (*lacht*). Ganz praktisch gibt es dann natürlich politische und projektbezogene Contraints, aber ich sehe die Stärke darin, dass man Bibliotheken aus der Scala-Ecke mit Bibliotheken aus der Groovy-Ecke kombinieren kann. Und diese wiederum mit Bibliotheken aus der Clojure-Ecke oder mit Sachen, die einfach in Java geschrieben sind. Ich glaube

eher nicht, dass ein und dasselbe Projektteam oder ein und dieselbe Firma für eng verwandte Dinge auf eine Vielfalt an Sprachen setzt. Aber das stimmt, theoretisch wäre so etwas möglich.

JM: Naja, man muss die Sprachen ja auch erst einmal lernen, das geht ja auch nicht über Nacht!
Haase: Ja genau, und zwar nicht nur man selbst, sondern das Management muss überzeugt sein, dass es möglich ist, Entwickler zu finden. Das halte ich persönlich für ein großes Problem, weil gerade gute Entwickler gern neue Sprachen lernen, recht gut darin sind, auch mit neuen Sprachen schnell produktiv zu werden. Aber es gibt natürlich eine konservative Haltung des Managements, die natürlich auch nicht ganz aus der Luft gegriffen ist.

JM: Mit Java 7 kam invokedynamic und das hat der JVM noch mal einen ordentlichen Push gegeben. Würdest du das auch so sehen?
Haase: invokedynamic war technisch ein Push, wegen der vielen Sprachen auf der JVM, aber es war vor allem politisch ein Push, weil es ein Committment war, dass die JVM nicht nur für Java gedacht ist. Es war die erste Erweiterung an der virtuellen Maschine, die überhaupt nichts mit der Sprache Java zu tun hatte und von der die Sprache Java auch überhaupt nicht profitiert. Sondern die ausschließlich für dynamische Sprachen wie Groovy gedacht ist und dort effizienter Umsetzungen erlaubt. Ein weiterer Meilenstein daran war, dass invokedynamic ganz stark in Kooperation mit der Open-Source-Community entstanden ist. Das heißt, es wurde Feedback aus der Python-, aus der Ruby-Community und so weiter geholt, um zu lernen, was passieren müsste, damit diese Sprachen gut auf die JVM portiert werden können. Also kulturell ein wichtiger Meilenstein und technisch gesehen ein sehr nützliches Feature.

JM: Also ein polyglotter Meilenstein für die Java-Welt.
Haase: Das hast du schön gesagt (*lacht*).

JM: Vielen Dank für deine Zeit, Arno!

Dieses Interview wurde ursprünglich auf der JAX 2013 geführt und ist auch als Video unter www.youtube.com/VideosJAXenter *verfügbar.*

Anzeige

Einführung in den Git- und Mercurial-Client SmartGit/Hg

Git it on!

SmartGit/Hg ist ein Git- und Mercurial-Client, der unter Windows, Mac OS X und Linux läuft. Er präsentiert alle relevanten Inhalte eines Repositorys in übersichtlicher Form und spricht diejenigen Benutzer an, die eine workfloworientierte Umsetzung der Git-Funktionalität der puren Kommandozeile von Git vorziehen.

von Thomas Singer und Marc Strapetz

SmartGit/Hg kann direkt von [1] heruntergeladen werden. Die Windows- und Mac-Downloadpakete beinhalten bereits die notwendigen Git-Kommandozeilentools, unter Linux müssen sie (wie auch das Java Runtime Environment) vom Nutzer, z. B. mithilfe des jeweiligen Paketmanagers, installiert werden. Beim ersten Programmstart durchsucht SmartGit/Hg den Pfad nach den Git- und Mercurial-Kommandozeilentools und greift nur dann auf die beiliegenden zurück, wenn keine entsprechenden auf dem System gefunden wurden. Im Folgenden gehen wir nur auf Git ein, wobei viele Aussagen dazu in ähnlicher Weise auch auf Mercurial zutreffen. Für nicht kommerzielle Nutzung ist SmartGit/Hg kostenfrei, kommerzielle Nutzer müssen eine Lizenz pro Nutzer erwerben, mit der man alle neuen Versionen nutzen kann, die innerhalb eines Jahres nach dem Kaufzeitpunkt releast wurden (inkl. aller auch später veröffentlichter Bugfix-Releases).

Set-up Repository, öffnen Repository, Clone

Um lokal mit Git zu arbeiten, gibt es grundsätzlich drei Möglichkeiten: man kann ein neues Repository anlegen, ein schon lokal vorhandenes öffnen oder ein anderes Repository klonen.

Zum Anlegen eines neuen oder Öffnen eines schon lokal vorhandenen Repositorys wählt man Open an existing local or create a new repository im Welcome-Dialog oder den Menüpunkt Project | Open Repository und gibt den Pfad an, wo das Repository auf der Festplatte zu finden ist oder angelegt werden soll. Wenn SmartGit/Hg kein bestehendes Repository findet, fragt es nach, ob es ein solches anlegen soll. Repositories werden in so genannten „Projekten“ verwaltet. Ein Projekt kann mehrere Repositories enthalten, hat einen optionalen Namen und beinhaltet diverse Einstellungen, die SmartGit/Hg zusätzlich zu den bereits bestehenden Konfigurationsmöglichkeiten von Git bietet.

Zum Klonen wählt man Clone existing repository im Welcome-Dialog oder den Menüpunkt Project | Clone. Je nachdem, ob man von einem lokalen oder auf einem Server befindlichen Repository klonen möchte, wählt man die entsprechende Option aus und gibt entweder den lokalen Pfad oder den Git-Repository-URL ein. Sollte das Repository auf einer bekannten Plattform wie Assembla oder GitHub gehostet sein, kann man nach Eingabe der jeweiligen Accountdaten direkt das gewünschte Repository auswählen. Im nächsten Schritt wird festgelegt, ob evtl. vorhandene Submodules (verschachtelte Repositories) ebenfalls gleich geklont und ob nur eine bestimmte Branch geklont werden soll. Abschließend gibt man den lokalen Pfad, wo das Repository auf der Festplatte abgelegt werden soll, und den Namen des neu erzeugten Projekts an.

SmartGit/Hg fragt bei Zugriffen auf das auf einem Server befindliche Repository nach erforderlichem Nutzernamen, Passwort oder Private Key und kann diese abspeichern.

Lokale Änderungen

Nachdem lokale Änderungen vorgenommen wurden, kann man diese entweder Git-typisch erst mit „Stage“ im Index speichern, um danach alle im Index gespeicherten Änderungen zu committen, oder man wählt die jeweiligen geänderten Dateien aus und committet sie direkt ohne Nutzung des Index.

Sollen nur einzelne Teile geänderter Textdateien committet werden, kann man diese im Changes-Bereich durch Klick auf [<<] oder mithilfe des Kontextmenüs im Index speichern („stagen“). Sind noch kleinere Überarbeitungen der zu stagenden Änderungen notwendig, kann man diese im Indexeditor vornehmen. Dieser bietet eine Repository- (HEAD), Index- und Working-Tree-Ansicht, wobei die letzteren beiden frei editierbar sind. Zum Committen von

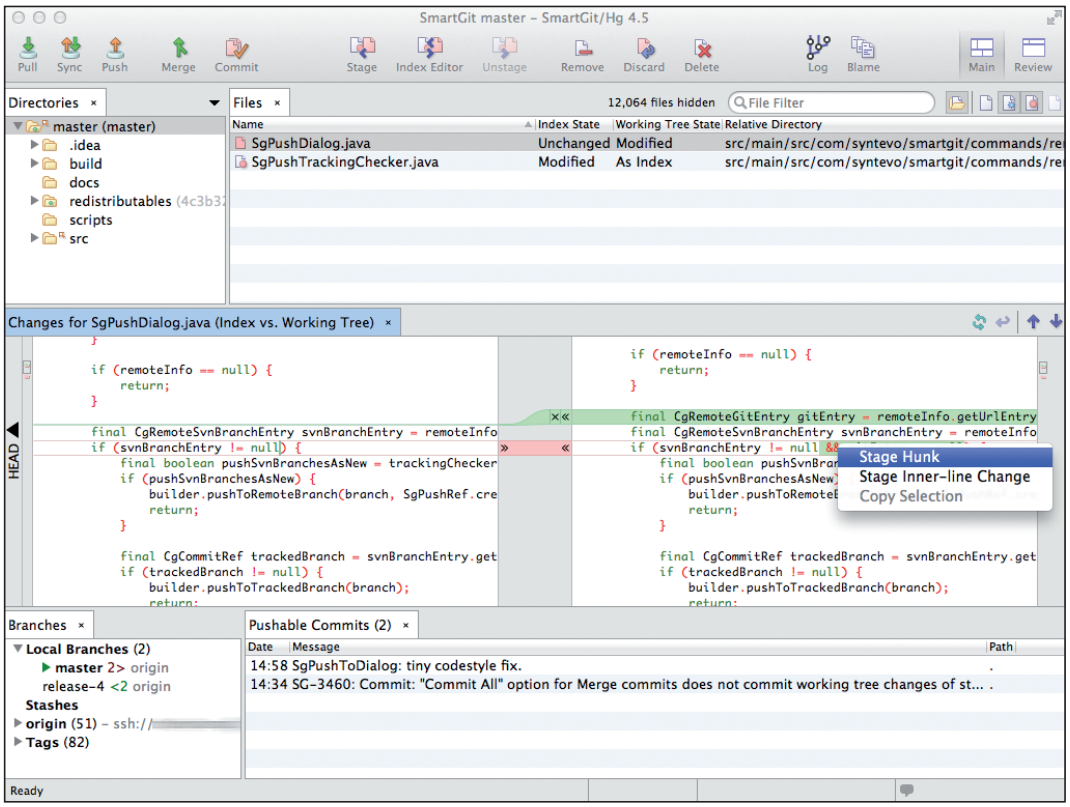


Abb. 1: Das Projektfenster gibt einen Überblick über die lokalen Änderungen, den Branch-Zustand und lokale (zu pushende) Commits, es erlaubt auch, Änderungen zeilenweise zu committen

Indexänderungen kann man entweder im Directories-Bereich den Repository-Knoten oder eine Datei mit Indexänderungen anwählen und Commit aufrufen. Hat man für ein bereits erfolgtes Commit die eine oder andere Änderung übersehen, kann man diese mit der Amend-Option des Commit-Dialogs zum letzten Commit hinzufügen.

Noch nicht gepushte Commits werden im Pushable-Commits-Bereich angezeigt. Man kann deren Reihenfolge einfach durch Drag and Drop ändern, Commit Messages korrigieren oder benachbarte Commits zusammenfassen. Durch Kombination dieser Aktionen lassen sie sich neu arrangieren, was dem „interactive rebase" der Git-Kommandozeile entspricht. Damit können aus einer Menge von „vorläufigen" Commits abschließend „saubere" Commits erstellt werden und eigene Fehler lassen sich so nachträglich bereinigen. Einschränkungen gibt es nur für Merge Commits und in jenen Fällen, wo das Umordnen zu Konflikten führen würde.

Zum Verwerfen von lokalen Änderungen kann man Discard aufrufen und entweder auf den Index- oder Repository-(HEAD-)Zustand zurücksetzen.

Umbenannte oder verschobene Dateien werden mit dem Untracked- bzw. Missing-Status angezeigt. Findet sowohl die Löschung als auch die Hinzufügung beider Dateien im gleichen Commit statt, erkennt Git die Ähnlichkeit der Dateien und im Log werden sie mit verbundener Geschichte angezeigt.

Log

Zum Ansehen der Commit-Geschichte des Repositorys wählt man im Directories-Bereich den Repository-Knoten und dann Log. Wenn man dies nur für eine bestimmte Datei sehen will, kann man diese mit Edit | Filter Files durch Eingabe ihres Namens schnell finden und auswählen (auch wenn sie aufgrund ihres Änderungszustands nicht den im View-Menü eingestellten Filtern entspricht und folglich ohne Suche nicht angezeigt werden würde), bevor man Log aufruft.

Im Log kann man im Branches-Bereich festlegen, für welche Branches die Versionsgeschichte angezeigt werden soll. Bei großen Repositories kann das Ausblenden von aktuell uninteressanten Branches helfen, sich auf das Wesentliche zu konzentrieren (und „den Wald vor lauter Bäumen wiederzuerkennen"). Durch Auswahl eines Commits sieht man im Details- und Filesbereich die entsprechenden Detailinformationen sowie die betroffenen Dateien. Zum Vergleich zweier Repository-Zustände wählt man einfach diese zwei Commits gleichzeitig aus. Nach Auswahl einer Datei werden im Changes-Bereich die Unterschiede zwischen beiden Commits angezeigt. Über das Kontextmenü kann man einen alten Dateizustand wiederherstellen oder diesen mit dem Working Tree vergleichen.

Verschiedene Kommandos kann man direkt im Log ausführen, was besonders für das Arbeiten mit Branches (z. B. Merge, Rebase) sinnvoll ist.

Commits gehen im Repository nicht verloren, selbst wenn man die zugehörige Branch gelöscht hat oder einen Rebase macht. Im Branches-Bereich des Logs gibt es dazu die Option „Lost Heads", die dabei hilft, scheinbar verlorene Commits wiederzufinden: Hier greift SmartGit/Hg auf die so genannten „ref-logs" von Git zurück und es scheinen alle Commits auf, die jemals von einer Branch (oder irgendeiner anderen „Ref") referenziert wurden.

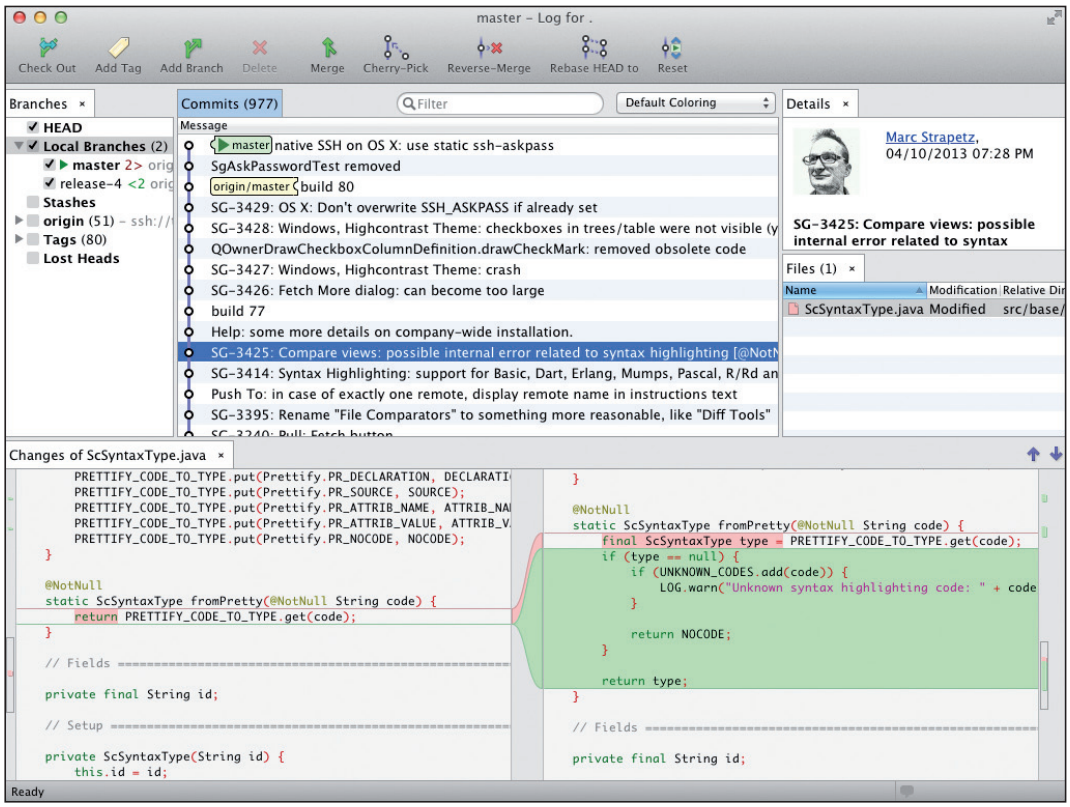


Abb. 2: Das Log-Fenster zeigt die Versionsgeschichte des Repositorys, Branches und Tags können zur besseren Übersicht ein-/ausgeblendet, lose Branch-Enden („lost heads“) angezeigt oder verschiedene Kommandos ausgeführt werden

Pull + Push

Falls man ein bestehendes Repository geklont hat, kann man die lokalen Commits mit Push in das ursprüngliche Repository zurückspielen. Sollte jemand anderes bereits seine Änderungen dorthin gepusht haben, wird Git das eigene Pushen verweigern. Dann muss man die fremden Änderungen mit Pull holen und in die eigenen integrieren. Git bietet dafür zwei Möglichkeiten: Merge und Rebase. Merge erzeugt einen Merge Commit aus den eigenen und den fremden Commits, während Rebase die eigenen Commits auf die fremden Commits „draufsetzt“ (ähnlich zu einem Cherry Pick). Da Rebase keine Merge Commits erzeugt, bleibt die Geschichte linear und optisch sauberer.

Mehr zum Thema Rebase (unabhängig von SmartGit/Hg) finden Sie übrigens auch im beiliegenden Poster, im Profitipp von Martin Dilger.

Merge + Rebase

Beim Mergen hat man die Wahl, ob Git automatisch ein Merge Commit anlegen soll oder man die Änderungen erst im Working Tree anschauen will. Hat man sich für Letzteres entschieden, wird das Merge Commit erst beim folgenden Commit erzeugt. Will man dagegen zwar die Änderungen übernehmen, aber kein Merge Commit anlegen („squash merge“), wählt man die entsprechende Option im Commit-Dialog.

Sollten bei dem Merge oder Rebase Konflikte auftreten, kann man für die konfliktbehafteten Dateien durch Doppelklick den Conflict Solver öffnen. Dieser zeigt die beiden individuellen Änderungen links und rechts an, während der teilgemergte Zustand in der Mitte editierbar ist. Man kann nun manuell die Konflikte beheben und beim Schließen des Conflict Solvers wird man nach dem Speichern gefragt, ob man die Änderungen „stagen“ will. Falls der Konflikt komplizierter ist, kann man auf der konfliktbehafteten Datei das Log aufrufen und sich die Commits, die zu dem Konflikt geführt haben, mit ihren jeweiligen Änderungen ansehen. Nach dem Beheben aller Konflikte und dem Stagen der jeweils notwendigen Änderungen, kann man mit Commit den Merge Commit erzeugen oder das Rebase fortsetzen. Sollte man unglücklicherweise den roten Faden verloren haben, kann man mithilfe von Discard das Merge bzw. das Rebase abbrechen, um es danach erneut zu starten.

Mit Branches arbeiten

Git ist prädestiniert für die Arbeit mit mehreren Branches. Man kann sie einfach erzeugen und später wieder löschen und so ist es empfehlenswert, diese auch schon für kleinere Änderungen zu verwenden. Beginnt man beispielsweise im „master“ mit der Arbeit an einem Feature und die Implementierung stellt sich doch als aufwändiger heraus, kann man mit Add Branch rasch eine neue Branch erzeugen und seine Änderungen in diese committen. Diese lokale Branch kann bei Bedarf auf den Server gepusht werden, z. B. damit ein Kollege die Änderungen begutachten oder fortsetzen kann. Zum schnellen Umschalten zwischen Branches klickt man doppelt auf den jeweiligen Eintrag im Branches-Bereich. So lässt sich die Arbeit an einem Feature unterbrechen und zu einem anderen Arbeitspaket wechseln. Sollte die Ziel-Branch hinter ihrer Remote Branch liegen, fragt SmartGit/Hg, ob man diese gleich aktualisieren möchte („fast-forward merge“). Später kann man auf diesel-

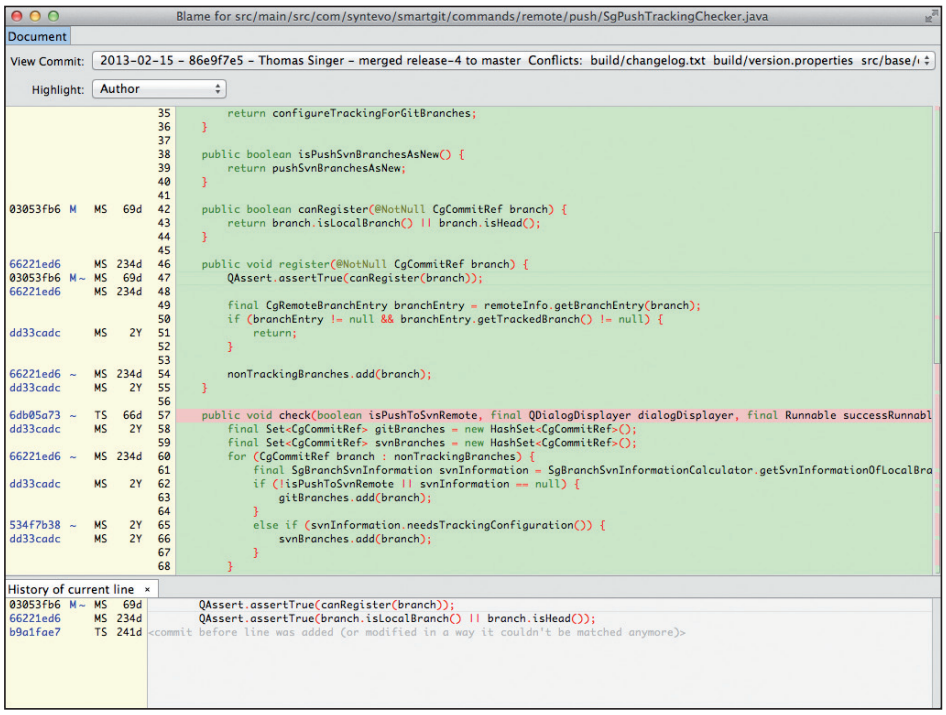


Abb. 3: Das Blame-Fenster zeigt die Entstehungsgeschichte einer Datei auf Zeilenebene, um zu beantworten, wann ein bestimmter Codeblock eingebaut oder entfernt wurde, es erlaubt die Navigation zwischen verschiedenen Versionen einer Datei

be Weise zu der Branch mit dem begonnenen Feature zurückwechseln, das Feature fertigstellen und die Änderungen zurück in den „master“ oder eine beliebige andere Branch mergen. Üblicherweise verwendet man hier beim Commit die „Squash“-Option, um ein einfaches Commit zu erzeugen und die einzelnen Änderungen aus der Feature-Branch nicht als Merge Source miteinzubinden. Abschließend wird die Branch wieder entfernt.

Stash

Um vorübergehend einen sauberen Working Tree zu erhalten, z. B. um einen soeben entdeckten Fehler zu beheben, kann man Save Stash nutzen und später die lokalen Änderungen mit Apply Stash wieder herstellen. SmartGit/Hg verwendet Stashes bei Bedarf auch selbstständig, wenn beispielsweise beim Pullen aus dem Remote Repository (oder beim Wechsel zu einer anderen Branch oder einem anderen Commit) lokale Änderungen im Working Tree das erfolgreiche Ausführen des Kommandos verhindern.

Blame

Um herauszufinden, in welchem Commit eine bestimmte Codezeile einer Datei eingefügt wurde, ruft man auf dieser Datei Blame auf. Das Blame-Fenster zeigt im Document-Bereich den Inhalt der Datei zum Zeitpunkt des aktuell gewählten Commits. Man kann nun über direkte Auswahl des Commits oder durch Navigation über die Hyperlinks zu vorherigen Inhalten der Datei wechseln, um so Änderungen in einem bestimmten Bereich zu erkennen und zu verstehen.

Für die aktuell gewählte Zeile zeigt der History-Bereich alle jemals existenten Versionen dieser Zeile an. Dies ist hilfreich, um sich bei der Suche von komplexeren Fehlern möglichst rasch einen Eindruck über jene Änderungen zu verschaffen, die mit dem Problem in Zusammenhang stehen könnten.

SVN

SmartGit/Hg kann auch als interessante Alternative zu einem klassischen SVN-Client genutzt werden. Statt ein SVN Repository in eine lokale SVN Working Copy auszuchecken, wird es als lokales Git Repository geklont. SmartGit/Hg holt zuerst nur die neueste SVN-Revision (wie ein svn checkout), sodass man schon nach kurzer Zeit mit dem Git Repository arbeiten kann. Alle älteren Revisions werden danach im Hintergrund heruntergeladen und später in das lokale Git Repository integriert.

SmartGit/Hg bildet die Tag- und Branch-Struktur des SVN Repositorys im lokalen Git Repository ab. Im Gegensatz zum Kommandozeilentool „git-svn“ werden alle SVN Properties auf entsprechende Git-Pendants abgebildet, dies gilt insbesondere auch für die Abbildung von SVN Externals auf SVN Submodules im Git Repository.

Wie bei Git üblich, kann man in dem erzeugten Klon lokal seine Commits erstellen, eine Verbindung zum SVN Repository ist nicht erforderlich. Wenn man mit seinen lokalen Commits zufrieden ist und sie anderen Nutzern des SVN Repositorys zur Verfügung stellen möchte, pusht man sie zum SVN Repository zurück. Erst jetzt wird eine Verbindung zum Server aufgebaut und die erzeugten SVN Revisions werden für die anderen Nutzer sichtbar. Auch beim Push findet eine Abbildung aller Git-Attribute auf entsprechende SVN-Pendants statt. Insbesondere werden Merge Commits bzw. Cherry Picks in der *svn:mergeinfo* Property vermerkt und auch die *svn:externals* Property wird angepasst, sollten sich die SVN Submodules im Git Repository geändert haben. Neu erzeugte, lokale Git Branches werden auf Nachfrage entsprechend im *branches/*-Verzeichnis des SVN Repositories erzeugt. Sinngemäßes gilt für lokal erzeugte Tags, die entweder direkt bei deren Erzeugung gepusht oder später über das Kontextmenü im Branches-Bereich gepusht werden.

Thomas Singer und **Marc Strapetz** sind Geschäftsführer der syntevo GmbH mit Sitz in Freilassing (Nähe Salzburg). Unter ihrer Leitung entstanden im Laufe der Jahre die erfolgreichen VCS-Clients SmartGit/Hg, SmartCVS und SmartSVN. Letzteres wurde im September 2012 von WANdisco International Ltd. übernommen.

Links und Literatur

[1] www.syntevo.com/smartgithg

Teststile: Schwierige Tests mit Doubles

Doppelt hält besser

Fake it till you make it: So könnte man den zweiten Teil dieser Serie auf den Punkt bringen. Die Rede ist von Testdoubles, die als Platzhalter für echte Objekte in Unit Tests verwendet werden können, um bestimmte Programmeinheiten isoliert zu testen oder die Interaktion zwischen Objekten in die Verifikation einzubeziehen. Den Einsatz dieser Testdoubles und unterschiedliche Teststile beschreibt der folgende Beitrag.

von Kai Spichale

Ein Unit Test sollte jeweils nur einen Aspekt in Isolation testen und sich auf kleine Programmeinheiten beziehen. Diese Programmeinheiten reichen im Idealfall von einzelnen Methoden bis hin zu wenigen zusammenhängenden Klassen. Doch leider existieren in einer durchschnittlich komplexen Anwendung so viele Abhängigkeiten, dass kaum ein Objekt ohne Mitwirkung anderer Objekte auskommt.

Bei einem pragmatischen Bottom-up-Ansatz könnte man zunächst Objekte der untersten Schicht testen. Die getesteten Objekte werden anschließend mit der nächsthöheren Schicht integriert, um auch diese zu testen. Der Prozess wird wiederholt, bis schließlich auch die oberste Schicht der Objekthierarchie an der Reihe war. Bei einem Top-down-Vorgehen – geläufig ist auch die Bezeichnung *outside-in* – würde man die Entwicklung mit dem Test des UI beginnen, eventuell Benutzerinteraktionen einbeziehen und sich Schicht für Schicht nach unten arbeiten. Dieses Vorgehen setzt in Tests das Vorhandensein von Objekten in unteren Schichten voraus, die noch nicht entwickelt wurden und deren Schnittstellen erst durch Tests identifiziert werden sollen. Diese Objekte können durch *Testdoubles* ersetzt werden. Der Begriff Testdoubles [1] ist die allgemeine Bezeichnung für Objekte, die zu Testzwecken Objekte des produktiven Codes ersetzen. Mit dem Bottom-up-Vorgehen könnte man theoretisch auf Testdoubles verzichten. Praktisch werden jedoch in beiden Varianten der testgetriebenen Entwicklung (Test-driven Development, TDD) Doubles eingesetzt.

Die wohl geläufigsten Varianten dieser Testdoubles sind Test-Stubs und Mock-Objekte. Die Unterschiede dieser Testdoubletypen werden in den nachfolgenden Beispielen dieses Artikels erläutert. Ebenfalls werden die Eigenschaften von Verhaltens- und Zustandsverifikation zweier unterschiedlicher Teststile beschrieben.

TDD mit Zustandsverifikation

Listing 1 zeigt einen Unit Test auf Basis von JUnit und FEST Assert. In diesem Beispiel soll eine Wartung für ein Kraftfahrzeug eines Fuhrparks angesetzt werden. Fahrzeuge in Wartung sind im Fuhrpark nicht mehr frei verfügbar. Fahrzeuge werden durch Objekte vom Typ *Vehicle* abgebildet. Das Interface *VehicleFleet* beschreibt den Fuhrpark. Das Objekt *Maintenance* repräsentiert eine Wartung. Ziel ist es, das Objekt *Maintenance* zu testen. Doch dieses ist abhängig vom Interface *VehicleFleet*, das von *PersistentVehicleFleet* implementiert wird.

PersistentVehicleFleet, dessen Quellcode hier nicht abgebildet ist, speichert alle Zustandsänderungen in einer Datenbank. Der vollständige Quellcode ist in einem öffentlichen Git Repository [3] einsehbar. Der Datenbankzugriff ist mit Spring Data implementiert, sodass zur Testausführung der Applikationskontext von Spring gestartet werden muss. Dies geschieht durch Angabe der Annotation *@RunWith(SpringJUnit4ClassRunner.*

Listing 1

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:context.xml")
@Transactional
public class MaintenanceStateTest {
  private Vehicle vehicle = new Vehicle();
  @Autowire private VehicleFleet fleet;

  @Before public void setUp() {
    fleet.add(vehicle);
  }
  @Test public void vehicleIsNotAvailableWhenScheduledForMaintenance() {
    Maintenance maintenance = new Maintenance(vehicle);
    assertThat(maintenance.schedule(fleet)).isTrue();
    assertThat(fleet.getAvailableVehicles()).isEmpty();
  }
}
```

Artikelserie

Teil 1: Eigenschaften von TDD und BDD, Best Practices für Unit Tests
Teil 2: Teststile
Teil 3: Test- und Build-Automatisierung

class). Der Pfad zu den Metadaten des Applikationskontexts wird mit *@ContextConfiguration* angegeben. Dieser JUnit-Test und andere vergleichbare Tests durchlaufen die allgemeinen Phasen Set-up, Exercise, Verify und Teardown [1]:

- Ein Test beginnt mit der Set-up-Phase, in der die getesteten Objekte und deren Umgebung vorbereitet werden. Die Menge dieser für den Test vorbereiteten Objekte wird als Test-Fixture bezeichnet. In unserem Beispiel besteht das Test-Fixture aus dem Spring-Kontext, dem injizierten Objekt *fleet*, dem Objekt *vehicle* und der Ausführung der Methode *setUp*.
- Der Methodenaufruf *maintenance.schedule(fleet)* entspricht der Exercise-Phase. In dieser Phase muss das getestete Objekt all das tun, was getestet werden soll.
- Die Assertions gehören zur Verify-Phase und stellen sicher, dass die in der Exercise-Phase aufgerufenen Methoden korrekt ausgeführt wurden.
- Die abschließende Tear-down-Phase wird in diesem Beispiel vollständig vom Garbage Collector abgedeckt.

Das getestete Objekt *Maintenance* wird auch als SUT (*system under test*) bezeichnet. Das am Test beteiligte Objekt *fleet* ist ein Collaborator. Der Collaborator hat zwei Aufgaben während des Tests: Zum einen ist er notwendig, um den Test überhaupt zum Laufen zu bringen. Und zum anderen wird er benötigt, um das getestete Verhalten zu überprüfen. Denn das Einstellen der Fahrzeugwartung könnte auch den Zustand des Fuhrparks geändert haben. Der in diesem Beispiel angewandte Teststil wird nach Martin Fowler [4] als Zustandsverifikation (*state verification*) bezeichnet, weil sowohl der Zustand des SUT als auch der des Collaborators zur Verifikation des Testergebnisses verwendet wird.

Zustandsverifikation mit Test-Stub

Der Gebrauch des *PersistentVehicleFleet* im Test von Listing 1 hat den Vorteil, dass dessen Persistenzfunktion indirekt mitgetestet wird. Allerdings sollte ein Unit Test jeweils nur einen Aspekt eines SUT testen, um den Test einfach und schnell zu halten und um im Fehlerfall leicht die Ursache identifizieren zu können. Anstelle des *PersistentVehicleFleet*, das eine Datenbankverbindung benötigt, könnte zu Testzwecken auch ein Testdouble eingesetzt werden. Die Persistenzfunktion von *PersistentVehicleFleet* kann mit einem dedizierten Integrationstest überprüft werden.

Ein spezielles Testdouble ist der Test-Stub, der den echten Collaborator simuliert, aber eine einfachere Implementierung besitzt. Ein Test-Stub ist ideal geeignet, um den Aufwand für die Erzeugung des Test-Fixtures zu reduzieren. Listing 2 zeigt einen Test-Stub für *PersistentVehicleFleet*.

Zur Speicherung des Zustands verwendet der Test-Stub keine Datenbank, sondern zwei einfache Collections. Das ist für den Test von *Maintenance* von Vorteil. Der echte Collaborator sollte separat getestet werden. Der Teststil hat sich durch den Einsatz des Test-Stubs nicht verändert. Nach wie vor basiert der Test auf der Zustandsüberprüfung von SUT und Collaborator. Der JUnit-Test für *Maintenance* kann getreu dem Prinzip „Keep it simple, stupid“ deutlich vereinfacht werden, weil der Spring-Kontext nicht mehr benötigt wird. Testdoubles werden darüber hinaus auch aus folgenden Gründen eingesetzt:

Typen von Testdoubles

Für automatisierte Unit Tests werden häufig Testdoubles eingesetzt. Das sind Objekte, die Teile der produktiven Software ersetzen. Testdoubles verhalten sich wie die echten Objekte des produktiven Codes, basieren jedoch auf einer anderen, i. d. R. einfacheren Implementierung, um die Komplexität von Tests zu reduzieren. Gerald Meszaros [1] unterscheidet folgende Testdoubles:

- Ein **Dummy** entspricht dem primitivsten Typ der Testdoubles. Er enthält keine Implementierung und ist vergleichbar mit einem Nullwert, der hauptsächlich zum Füllen von Parameterlisten verwendet wird.
- Ein **Fake-Objekt** hat eine funktionierende Implementierung, die jedoch nicht für den produktiven Gebrauch bestimmt ist. Ein Fake-Objekt könnte beispielsweise eine leichtgewichtige In-Memory-Datenbank sein, um den Zugriff auf die tatsächliche Datenbank bei Tests zu vermeiden.
- Ein **Test-Stub** reagiert auf Methodenaufrufe mit einem vordefinierten Verhalten bzw. beantwortet diese mit vordefinierten Rückgabewerten. Der getestet Code erhält vom Test-Stub indirekt Testeingaben.
- **Mock-Objekte** unterscheiden sich von einfachen Testdoubles durch ihr Vermögen, das Testergebnis zu verifizieren. Das bedeutet, sie führen intern Assertions durch. Mock-Frameworks bieten aus diesem Grund die Möglichkeit, Erwartungen zu spezifizieren, um zu überprüfen, ob die Methoden der Mock-Objekte wie erwartet während des Tests aufgerufen werden.
- Ein **Test-Spy** kann wie ein Mock-Objekt den getesteten Quellcode überprüfen. Während die Erwartungen für Mock-Objekte vor der Interaktion mit dem zu testenden Code definiert werden, erfolgt die Definition der Erwartungen bei Test-Spys nachträglich.

Die Verwendung dieser Begriffe ist in der Literatur leider nicht einheitlich. Meszaros Begriffsdefinition verwendet beispielsweise auch Martin Fowler, allerdings verzichtet er auf die Unterscheidung zwischen Mock-Objekten und Test-Spys. Michael Feathers unterscheidet nur zwischen Fake- und Mock-Objekten. Laut Feathers [2] Definition implementieren Fake- und Mock-Objekte die gleiche Schnittstelle wie die Objekte, die sie simulieren. Auf diese Weise ist die Verwendung eines Testdoubles für alle anderen Objekte, die mit diesem interagieren, transparent.

Fazit: Mock-Objekte sind Testdoubles, die im Gegensatz zu den einfacheren Fake- und Stub-Objekten das Testergebnis überprüfen können.

- Bei nichtdeterministischen Rückgabewerten können Testdoubles als Ersatz dienen, um die Stabilität von Tests zu verbessern. Dieses Problem kann beispielsweise bei Objekten, die Messwerte von Sensoren zurückgeben, auftreten. Dazu zählt auch Programmcode, der auf die Systemzeit zurückgreift.
- Ein anderer Grund sind schwer künstlich herstellbare Zustände, wie etwa Netzwerkfehler. Der Aufwand für die Entwicklung eines Test-Stubs kann in einem solchen Fall geringer sein als der für die tatsächliche Herstellung des Testkontexts.
- Unit Tests müssen schnell sein, um möglichst häufig ausgeführt werden zu können. Durch die Vermeidung von zeitraubenden Operationen wie Dateizugriffen, Netzwerkkommunikation und dem Einspielen von Testdaten in Datenbanken können Unit Tests beschleunigt werden. Die Vermeidung dieser Operationen verbessert außerdem die Umgebungsunabhängigkeit der Tests.
- Ein Testdouble kann auch notwendig sein, um ein Objekt zu ersetzen, das noch nicht implementiert wurde oder das das notwendige Verhalten noch nicht besitzt.

Listing 2

```
public class VehicleFleetStub implements VehicleFleet {
  private Collection<Vehicle> availableVehicles = new HashSet<Vehicle>();
  private Collection<Vehicle> notAvailableVehicles = new HashSet<Vehicle>();

  @Override public void add(Vehicle vehicle) {
    availableVehicles.add(vehicle);
  }
  @Override public Collection<Vehicle> getAvailableVehicles() {
    return Collections.unmodifiableCollection(availableVehicles);
  }
  @Override public void setAvailability(Vehicle vehicle, boolean available) {
    if (available) {
      availableVehicles.add(vehicle);
      notAvailableVehicles.remove(vehicle);
    } else {
      availableVehicles.remove(vehicle);
      notAvailableVehicles.add(vehicle);
    }
  }
  @Override public boolean isAvailable(Vehicle vehicle) {
    return availableVehicles.contains(vehicle);
  }
}
```

TDD mit Verhaltensverifikation

In Listing 3 testen wir *Maintenance* ein weiteres Mal mithilfe von jMock [5]. In der Phase Set-up werden notwendige Testdaten und Erwartungen definiert. Damit unterscheidet sich diese Phase grundlegend von den bisherigen Tests. Als Collaborator wird ein Mock-Objekt auf Basis des Interface *VehicleFleet* erzeugt. Die definierten Erwartungen beziehen sich auf das Mock-Objekt. Erwartet wird, dass die Methode *isAvailable* des Mock-Objekts mit dem Parameter *vehicle* einmal aufgerufen wird. Der Rückgabewert sei *true*. Erwartet wird auch, dass *setAvailability* mit den Parametern *vehicle* und *false* auf dem Mock-Objekt aufgerufen wird. Diese Erwartungen müssen während der Phase Exercise, die sich nicht von den anderen Tests unterscheidet, erfüllt werden. Das Mock-Objekt beteiligt sich, im Gegensatz zum Test-Stub, an der Verifikation des Tests. Es überprüft die Interaktion mit dem SUT und stellt auch indirekt Testdaten bereit, wie etwa den Rückgabewert *true* beim Aufruf der Methode *isAvailable*. Falls diese Methode mit anderen Parametern als angegeben oder überhaupt nicht aufgerufen wird, schlägt der Test fehl.

Mock-Objekte verwenden einen anderen Teststil. Der wesentliche Unterschied zu den beiden ersten zustandsbasierten Tests ist die Überprüfung der Interaktionen. Deswegen ist die Bezeichnung *Verhaltensüberprüfung* (Behavior Verification) [3] üblich.

Wie Mock-Objekte die Entwicklung treiben

Test-driven Development versucht, das Design eines Systems iterativ zu entwerfen, durch Schreiben von Tests noch vor dem produktiven Code. Beginnend mit der Entwicklung einer Story in Form eines ersten Tests wird ein Teil der Schnittstelle des SUT entworfen. Durch die Definition der Erwartungen an die Collaborators wird die Interaktion mit anderen beteiligten Objekten definiert. Sobald der erste Test erfolgreich durchläuft, können die definierten Erwartungen als Ausgangspunkt für die nächsten Schritte verwendet werden. Für jede Erwartung sollte ein Test für den jeweiligen Collaborator entwickelt werden.

Ein kritischer Blick

Ein Nachteil der Erstellung eines Testdoubles kann darin bestehen, dass dies zu erheblichem Mehraufwand führen kann. Ziel sollte es sein, auch komplexen Code mit einfachem Testcode abzudecken. Falls das Testdouble sehr kompliziert wird, ist man wahrscheinlich über das Ziel hinausgeschossen. Falls für einen Test sehr viele Erwartungen an ein Mock-Objekt definiert werden müssen, deutet das darauf hin, dass entweder versucht wird, mehr als einen Aspekt pro Test abzudecken oder dass es eine starke strukturelle Kopplung zwischen SUT und Collaborator gibt, die überarbeitet werden sollte.

Durch den Einsatz von Mock-Objekten können potenziell Tests entstehen, die stark an die Implementierung der verwendeten SUT gebunden sind. Man sollte aus diesem Grund Mock-Objekte nicht beliebig einsetzen, sondern gezielt zur Überprüfung des Verhaltens der Objekte. Einfache Value-Objekte sollten beispielsweise niemals gemockt werden, denn diese sollten sowieso unveränderlich (immutable) sein. Im Allgemeinen sind Objekte, die kein interessantes Verhalten besitzen, keine guten Kandidaten für Mocking.

Freeman und Pryce [6] empfehlen, das Mocken von Third-Party-Libaries zu vermeiden. Hintergrund ist, dass diese Tests keinen Beitrag zum Entwurf des Designs liefern können, da das verwendete API in der Regel unveränderlich ist. Außerdem besteht die Gefahr, dass sich das Verhalten des Testdoubles vom Verhalten des tatsächlichen Objekts in der Third-Party-Libary unterscheidet. Ein externes API kann durch eine dünne Schicht gekapselt werden, die nach innen eine für die Applikation passende Schnittstelle mit den Begriffen des Domänenmodells anbietet. Diese Schnittstelle sollte durch TDD und Mocking entworfen werden.

Manche Fehler werden erst durch Integrations- oder durch Unit Tests, bei denen mehrere Objekte beteiligt sind, aufgedeckt, denn das Fehlverhalten einzelner Objekte im Zusammenspiel mit anderen kann durch den intensiven Einsatz von Testdoubles auch mit hoher lokaler Testabdeckung unbemerkt bleiben.

Zusammenfassung

Es gibt Situationen, in denen ein Objekt nur schwer getestet werden kann, weil es beispielsweise Abhängigkeiten zu anderen Objekten hat, die nicht in der Testumgebung verwendet werden können. Dies kann dann der Fall sein, wenn das abhängige Objekt zum Zeitpunkt der Testausführung nicht zur Verfügung steht, wenn es nicht die erforderlichen Ergebnisse zurückliefert oder wenn die Ausführung unerwünschte Seiteneffekte hat. Ein Platzhalter ist auch dann notwendig, wenn der Test mehr Kontrolle oder eine andere Sichtbarkeit des internen Verhaltens der beteiligten Objekte voraussetzt. Falls der echte Collaborator nicht im Test verwendet werden kann, kann dieser durch ein Testdouble ersetzt werden. Dieses muss die gleiche Schnittstelle bereitstellen, sich aber nicht wie das ersetzte Objekt verhalten. Wichtig ist, dass die Verwendung des Testdoubles für das SUT transparent ist. Mock-Objekte sind besondere Testdoubles, die auch an der Verifikation des Tests beteiligt sind. Mock-Objekte sollten dann eingesetzt werden, wenn sie den Entwurf des Systems unterstützen können.

Listing 3

```
public class MaintenanceBehaviorTest {
  @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

  @Test public void vehicleIsNotAvailableWhenScheduledForMaintenance() {
    // setup data
    final Vehicle vehicle = new Vehicle();
    final Maintenance maintenance = new Maintenance(vehicle);
    final VehicleFleet fleet = context.mock(VehicleFleet.class);
    // setup expectations
    context.checking(new Expectations() {{
      oneOf(fleet).isAvailable(vehicle);
      will(returnValue(true));
      oneOf(fleet).setAvailability(vehicle, false);
    }});
    // exercise and verify
    assertThat(maintenance.schedule(fleet)).isTrue();
  }
}
```

Kai Spichale ist Senior Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG. Sein Tätigkeitsschwerpunkt liegt in der Konzeption und Implementierung von Softwaresystemen auf Basis von Java EE und Spring.

Links & Literatur

[1] Meszaros, Gerard: „xUnit Test Patterns: Refactoring Test Code“, Addison-Wesley, 2007

[2] Feathers, Michael: „CWorking Effectively with Legacy Code“, Pearson Education, 2004

[3] https://github.com/kspichale/mock-demo

[4] Fowler, Martin: „Mocks Aren't Stubs”, 2007: http://www.martinfowler.com/articles/mocksArentStubs.html

[5] http://jmock.org/cookbook.html

[6] Freeman, Steve; Pryce, Nat: „Growing Object-Oriented Software, Guided by Tests“, Addison-Wesley Professional, 2009

Anzeige

Migration eines TK-Kundenportals auf Java EE 6

Neu ist immer besser

Das Kundenportal eines norddeutschen Telekommunikationsanbieters basiert heute auf dem Java EE 6 Web Profile. Im Rahmen eines Migrationsprojekts wurde das Kundenportal von teilweise zehn Jahre alten Frameworks und Technologien innerhalb weniger Monate auf einen modernen und standardisierten Technologie-Stack überführt, um so Wettbewerbsfähigkeit und Time to Market zu gewährleisten. Dieser Artikel beschreibt Erfahrungen aus der Praxis und gibt Einblicke in technische und fachliche Aspekte der Migration und des neuen Java-EE-6-basierten Kundenportals.

von Thilo Focke, Marc Petersen und Christian Zillmann

Java-Entwicklung am Anfang dieses Jahrhunderts war sehr geprägt von technologischen Aspekten. Das Problem lag darin, dass die Aufsplittung in J2ME, J2SE und J2EE eigentlich zielgerichtete Frameworks liefern sollte, dies aber nicht tat. J2EE ist angetreten, um dem Entwickler die technologischen Aspekte abzunehmen, damit er sich überwiegend auf die Umsetzung der Businesslogik konzentrieren kann. Dieses Ziel wurde aber deutlich verfehlt [1]. Als logische Konsequenz waren Anwendungen aus dieser Zeit weniger geprägt von Fachlichkeit als vielmehr von technologischer Komplexität. Als Softwareentwickler konnte man vor allem eher mit der Kenntnis möglichst vieler Design-Patterns glänzen als mit branchenspezifischem Fachwissen. Zurückblickend betrachtet war Java-Entwicklung – zumindest in der Anwendungsentwicklung – zu dieser Zeit mit wenig Freude verbunden.

Wie in vielen anderen Unternehmen entstanden in dieser Zeit auch bei einem im norddeutschen Raum tätigen Telekommunikationsunternehmen zahlreiche, meist webbasierte J2EE-Anwendungen. In diesem Artikel wird aufgezeigt, wie eine Anfang des Jahrhunderts entstandene, sehr technologisch fokussierte Portalanwendung zu einer State-of-the-Art-Anwendung überführt wurde. Aspekte wie Fachlichkeit, Time to Market, Verständlichkeit und Entwicklungsgeschwindigkeit stehen dabei im Vordergrund; Technologie wird wieder Mittel zum Zweck – nicht zum Selbstzweck.

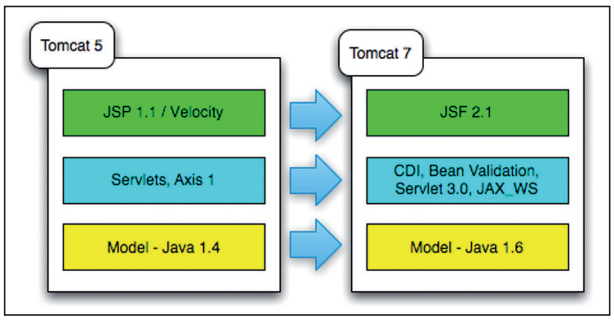


Abb. 1: Schichten und Technologien

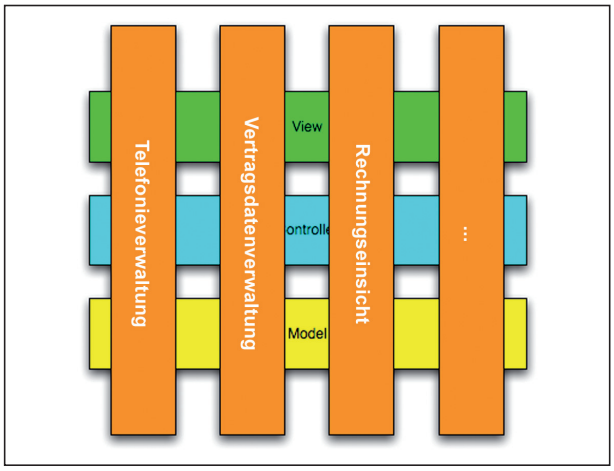


Abb. 2: Fachliche Schichten

Always change a running System – Challenge accepted!

Nicht selten wird versucht, durch Technologie organisatorische Probleme zu kompensieren, was selten zu Erfolg führt. In diesem Fall führten technologische Aspekte allerdings dazu, dass Unternehmensziele nicht in geforderter Zeit und Qualität umgesetzt werden konnten. Wie konnte es dazu kommen? Ein System über Jahre hinweg technologisch am Puls der Zeit zu halten ist durchaus eine Herausforderung. Gemäß der *Broken-Windows-Theorie* beginnt der Verfall von Qualität im allgemeinen Sinn mit Kleinigkeiten, die nur lange genug unbeachtet bleiben. Die Initiative *Clean Code Developer* hat hier bereits zahlreiche Prinzipien und Praktiken vorgeschlagen, wie Software unter anderem evolvierbar bleibt [2]. Eine einfache, klare und leicht verständliche Lösung sollte daher immer bevorzugt werden. Nur: Wo fängt man an aufzuräumen?

Erstes Opfer aus der Istwelt

Im Rahmen eines großen Migrationsverfahrens sollen verschiedene Systeme, die teilweise vor bis zu zehn Jahren entwickelt wurden, Schritt für Schritt auf eine unternehmensweite Java-EE-Strategie migriert werden. Die Machbarkeit dieses Migrationsprojekts soll am Beispiel einer unternehmenskritischen Anwendung evaluiert werden. Das so genannte Customer-Self-Service-Portal (CSS) dient Kunden zur Verwaltung ihrer Telefonie- und Internetprodukte sowie zahlreicher zusätzlicher Leistungsmerkmale, wie zum Beispiel Webhosting. Das System wird täglich von mehreren tausend Kunden genutzt und setzt eine 24/7-Verfügbarkeit voraus.

Istarchitektur

Die Istarchitektur basiert auf einem Model View Controller Pattern, um eine strikte Trennung von Präsentation, Programmlogik und Datenschicht zu gewährleisten. MVC-Frameworks wie Jakarta Struts steckten damals noch in den Kinderschuhen, und entsprechendes Know-how war schwer am Markt zu bekommen. Daher wurde eine eigene Lösung eines MVC-Modells implementiert. Das Model wurde mithilfe von Java POJOs abgebildet. Für die View waren Apache-Velocity-basierte Templates verantwortlich. Jede Aktion im Portal wurde entsprechend mit einem Servlet Request realisiert. Bei der Umsetzung neuer Anforderungen war stets viel Handarbeit notwendig. Viele Aspekte wie Security mussten explizit durch den Entwickler berücksichtigt werden. Es war sicherzustellen, dass kein Request manipuliert werden konnte, und Fehlermeldungen mussten sowohl in den Templates als auch in den Controller-Klassen explizit behandelt werden.

Migrationsverfahren

Für eine anstehende Migration einer klassischen MVC-basierten Schichtenarchitektur bieten sich generell zwei unterschiedliche Ansätze an: Zum einen kann Schicht für Schicht migriert werden, zum anderen schichtenübergreifend in fachlichen Tranchen. Beide Ansätze setzen allerdings voraus, dass eine saubere Trennung der Schichten bzw. der Fachlichkeit vorliegt.

Da das Team historisch bedingt anfänglich weiterhin stark in Schichten gedacht hat, lag zunächst die Überlegung nahe, die in **Abbildung 1** dargestellte MVC-Architektur Schicht für Schicht und weitestgehend automatisiert in die Java-EE-6-Zielarchitektur zu überführen. Dabei wurde evaluiert, wie JSPs auf JSF-Seiten und wie Servlets auf CDI Managed Beans abgebildet werden können.

Insbesondere die Migration von View-Technologien stellt eine äußerst komplexe Aufgabe dar [3], und eine zumindest teilweise automatisierbare Abbildung von JSP (mit Velocity als Template Engine) auf JSF war im Rahmen des Migrationsprojekts nicht sinnvoll realisierbar, da es eine komplexe und zeitaufwändige Entwicklung von entsprechenden Tools bedeutet hätte. Es wurde daher bewusst die Entscheidung getroffen, die Anforderungen der einzelnen Domänen anhand des Legacy-Codes einem Reverse Engineering zu unterziehen und mit den Fachabteilungen zu validieren. Auch die Migration der verschiedenen Servlets in der Anwendung auf CDI Managed Beans war nicht direkt abbildbar. Nachdem der Ansatz der Schichtenmigration sich als nicht praktikabel herausstellte, wurde begonnen, das Altsystem in seine fachlichen Domänen zu zerlegen. Es wurde vertikal in fachliche Schichten aufgeteilt. Die so entstandenen kleinen und fachlich klar voneinander abgegrenzten Domänen erlaubten zwar weiterhin keine

automatische Migration, dafür aber eine risikolose und vor allem testbare manuelle Überführung in den Java-EE-6-Stack.

Fachliche Schichtenmigration

Nachdem die zu migrierenden fachlichen Domänen in der Anwendung identifiziert worden waren (**Abb. 2**), wurde in Zusammenarbeit mit der Qualitätssicherung und den zu den Domänen gehörenden Fachabteilungen ein Migrationsvorgehen entwickelt (**Abb. 3**), das die Migrationsschritte bei der Migration einzelner Schichten beschreibt. Ein Problem, das sich hierbei herausstellte, war die ungenügende fachliche Dokumentation des Systems. Durch den Reverse-Engineering-Ansatz und in Abstimmung mit den Fachabteilungen wurden veraltete Anforderungen gestrichen und natürlich – wie soll es auch anders sein? – kam dabei auch eine ganze Zahl neuer Anforderungen auf, die in einem Backlog aufgenommen wurden, aber für die Migration außen vor gelassen wurden.

Migration in einer „serviceorientierten Landschaft“

Das Portal bettet sich in eine sehr heterogene Systemlandschaft ein. Schon frühzeitig wurde bei der Gestaltung der Systemlandschaft auf eine serviceorientierte Architektur gesetzt. Die einzelnen Systeme kommunizieren über einen Protocol-driven Enterprise Service Bus (ESB) [4] (**Abb. 4**). Für die geplante Migration hin zu einer Java-EE-6-Anwendung (Web Profile) bietet diese „Istarchitektur“ Vor- und Nachteile.

Die relative lose Kopplung der einzelnen Systeme über den ESB ermöglicht zum einen, ein System relativ unabhängig von den anderen austauschen zu können. Auf der anderen Seite ergeben sich aber auch eine Reihe von Nachteilen bzw. Schwierigkeiten bei der Migration einer Anwendung innerhalb einer SOA. Die Entwicklung ist in großen Unternehmen oft sehr verteilt, und die Kommunikation der einzelnen Entwicklungsteams gestaltet sich zuweilen schwierig. Weiterhin ist die gesamte Systemlandschaft stets volatil, befindet sich also im Wandel und in der Weiterentwicklung. Das führt dazu, dass Systeme in der Testlandschaft nicht verfügbar sind oder sich Schnittstellen ändern.

Die Migration bzw. die Entwicklung sollte möglichst unabhängig von den Umsystemen durchgeführt werden. Die frühzeitige Ausarbeitung einer Strategie, wie man Services in einer SOA effektiv für die Entwicklung mocken kann, ist sinnvoll, um auch bei Systemausfällen die Entwicklung fortführen zu können. Natürlich ist die Entwicklung von Mocks zeitaufwändig. Deren Nutzen zeigt sich aber nicht nur bei Systemausfällen, sondern auch bei UI-Tests. Letztere werden nicht durch Systemausfälle beeinträchtigt, und zum anderen entfällt in der Regel das Aufräumen von Testdaten in den Testsystemen.

Die Mock-Einspritzung

Das Legacy-System kommuniziert ausschließlich über Web Services mit dem unternehmensweiten ESB. Die

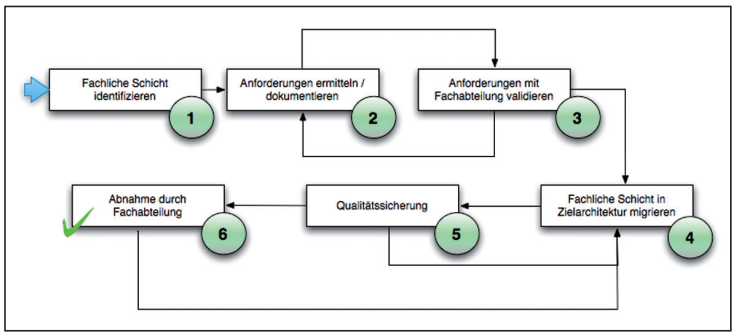


Abb. 3: Migrationsvorgehen

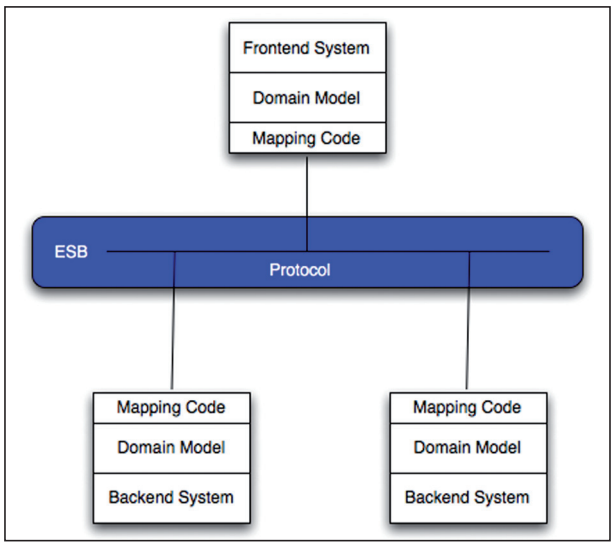


Abb. 4: Protocol-driven Enterprise Service Bus

dafür verwendeten Web-Service-Clients wurden in der neuen Welt ausschließlich mit JAX-WS und CDI Managed Beans implementiert. Dabei wurden für jeden Service zwei Implementierungen bereitgestellt. Die Implementierung, die direkt gegen den ESB geht, wurde mit der selbstgeschriebenen Annotation *@ESBService* markiert. Eine zweite alternative Implementierung wurde mit *@MockService* annotiert. Die Serviceklassen werden mit CDI in die Controller injiziert. Das Default-Verhalten ist das Injizieren der *@ESBService*-Implementierungen. Bei Ausfällen von nachgelagerten Systemen auf der Test-Stage, zu expliziten Testzwecken oder auch zur schnellen Entwicklung, ist es möglich, die Implementierung, die injiziert werden soll, in der *beans.xml* des entsprechenden Projekts zu ändern. Dadurch kann schnell auf die Mock-Implementierung gewechselt werden.

„doAll“-Methoden vs. Fachlichkeit

Der Quellcode des Altsystems bestand häufig aus so genannten *doAll*-Methoden. Bei diesen handelte es sich um Methoden zwischen 110 und 830 Zeilen Code, die direkt Eingabevalidierung, Fehlerhandling, Transformationen und Serviceaufrufe implementieren. Dem Zielsystem wurde also als erste Maßnahme Checkstyle mit einer Begrenzung von 100 Zeilen pro Methode konfiguriert – eine wichtige Grundregel, mit deren

Hilfe die initial an die Migration gestellten Anforderungen, wie verbesserte Lesbarkeit und Wartbarkeit, realisiert wurden.

Die *doAll*-Methoden haben stets einen ähnlichen Ablauf: Zunächst wird die Berechtigung des angemeldeten Nutzers geprüft, um anschließend die Werte aus dem Request in lokale Variablen zu schreiben. Die vom Nutzer vorgenommenen Eingaben werden direkt in der Methode mittels Hilfsklassen formatiert und validiert. Hierbei handelt es sich um ein klassisches Anti-Pattern, das häufig auf mangelndes Seperation of Concerns hinweist [5]. Es ist erkennbar an Klassen, die z. B. auf *Util* oder *Manager* enden.

Neben den Vorteilen, die eine Anwendung durch CDI, JSF und Bean Validation geschenkt bekommt, wurde im Zielsystem auf den Domain-driven-Design-Ansatz gesetzt, um so eine stärkere Fokussierung auf die Fachlichkeit zu erreichen [6]. Ein Ausschnitt aus einem typischen Controller, wie er im Zielsystem umgesetzt wurde, ist in Listing 1 dargestellt. Dazu sei kurz erwähnt, dass der in diesem Artikel aufgelistete Quellcode aus Platzgründen leicht modifiziert wurde und z. B. hart kodierte Strings normalerweise nichts im Quellcode verloren haben!

Zunächst wird hier ein Service injiziert, der fachliche Exceptions wirft und die Web-Service-Aufrufe kapselt. Grundsätzliche Validierung und Formatierung erfolgen in so genannten *SimpleValueObjects*. Listing 2 zeigt dies am Beispiel von *FtpDirectory*. Die abgeleitete abstrakte Klasse *SimpleValueObject*, eines der Herzstücke der Domäne, ist in Listing 3 vollständig dargestellt. Die wichtigste Regel hierbei ist, dass *SimpleValueObjects immutable* sein müssen. Daher finden sich in den entsprechenden Implementierungen auch keine Setter-Methoden.

Ein weiteres wichtiges Konzept zur Verbesserung der Codequalität ist die konsequente Verwendung von Buildern. Bei der Interaktion mit dem ESB werden häufig generierte *DataTransferObjects* (DTO) durchgereicht. Die DTOs werden dabei niemals außerhalb von Services verwendet, sondern innerhalb des Service mithilfe von Buildern in das Domänenmodell überführt. Aus drei einfachen Gründen:

1. Die DTOs ändern sich, sobald die Schnittstelle geändert wird, was im schlimmsten Fall Änderungen in allen Schichten bedeuten würde.
2. Die vom ESB bereitgestellten DTOs stimmen fachlich nicht immer mit dem überein, was in dem jeweiligen Anwendungsfall benötigt wird. Somit ist es manchmal sinnvoll, mehrere DTOs zu einer Entity zu mappen oder auch aus mehreren Entities ein DTO zu machen.

DTOs bestehen nur aus Attributen und Gettern/Settern, weswegen es nicht möglich ist, Aussagen darüber zu treffen, ob es sich dabei um ein valides Objekt handelt. Genau dieses Problem wird mit Buildern gelöst. Ein Entity Builder leitet von

```
public abstract class AbstractEntityBuilder<T> {
  public abstract T build();
}
```

Listing 1

```
@Named @RequestScoped
public class FtpAccountController {

  @Inject
  private FtpService ftpService;
  @Inject
  private CustomFacesContext context;

  @NotNull
  private FtpUsername username;
  @NotNull
  private FtpDirectory directory;
  ...

  @RolesAllowed({"owner"})
  @LogStatistics
  public Outcome createFtpAccount() {
    try {
      ftpService.createFtpAccount(username, directory, ...);
    } catch (UserAlreadyExistsException e) {
      context.addMessage("Username already exists");
      return Outcome.FAILURE;
    }
    return Outcome.SUCCESS;
  }

  // getter and setter for JSF
}
```

Listing 2

```
public class FtpDirectory extends SimpleValueObject<String> {

  public FtpDirectory(String aValue) {
    super(aValue);
  }

  @Override
  protected String validateAndNormalize(final String aValue) {
    String result = super.validateAndNormalize(aValue);
    return result.startsWith("/") ? result.substring(1) : result;
  }

  public String getPath() {
    return super.getValue();
  }

  public String getFullPath() {
    return "/html/" + super.getValue();
  }
}
```

ab, wobei *build()* sicherstellt, dass nur valide Entitäten erzeugt werden können. Listing 4 zeigt beispielhaft den *FtpProductBuilder*, der wie folgt benutzt wird:

```
FtpProductBuilder.newInstance(new FtpUserName("john"))
                                    .at(new FtpDirectory("foo/bar").build();
```

Auf diese Weise lässt sich sprechender Code schreiben, der sehr schnell von anderen Entwicklern verstanden wird. Insgesamt hat der konsequente Einsatz von Immutable Value Objects, Services und Buildern entscheidend dazu beigetragen, dass der Quellcode im Zielsystem verständlich, übersichtlich und nachhaltig ist.

Fabrikarbeit

Die über Services bereitgestellten Objekte werden in verschiedensten Controllern genutzt. Dabei gilt es, die „technische Infrastruktur" für den Zugriff auf diese Objekte möglichst gering zu halten.

In der Praxis bedeutet dies, dass für den angemeldeten Benutzer eine Reihe von fachlichen Objekten an verschiedenen Stellen vorgehalten werden müssen. Ein Benutzer hat zum Beispiel Rechnungen, Internetverträge, Telefonverträge, Webhostingprodukte oder Domains, die er verwalten möchte. Hierfür wurde vereinbart, dass solche Objekte über CDI Producer produziert und an den entsprechenden Stellen injiziert werden. Die spezifische Fachlichkeit wurde dabei durch CDI Qualifier wie folgt zum Ausdruck gebracht:

```
@Produces @Latest
public Invoice getLatestInvoice() { .. }
```

Jeder Controller, der länger als einen Request lebt, hat nun allerdings das Problem, dass er möglicherweise veraltete Objekte vorhält. Die Lösung heißt hier: CDI Events.

Feuer frei!

Der ESB stellt für die verschiedenen fachlichen Domänen des Portals Web Services bereit. Diese unterscheiden sich stark in den Antwortzeiten. Es gibt viele Web Services, die im Millisekundenbereich antworten, leider aber auch einige, die komplexere Daten liefern und daher zwei oder drei Sekunden brauchen können. Ein typisches Beispiel für einen solchen Service ist das Laden der gesamten Vertragsstruktur eines Kunden mit seinen Produkten. Dies geschieht zum ersten Mal bei der Anmeldung des Kunden. Anschließend werden die Vertragsdaten in der Session des Kunden gehalten und fortan nur noch über die Session herangezogen. Es gibt allerdings verschiedene Stellen im Portal, an denen der Kunde neue Produkte bestellen oder bestehende Produkte verändern kann. Hierdurch ändert er die Vertragsstruktur im Backend, und die gecachten Daten sind nicht mehr synchron. An diesen Stellen wird ein CDI Event gefeuert.

Die Producer-Klasse für die Vertragsdaten reagiert auf das Event, lädt die Vertragsdaten anschließend aus

Listing 3

```
public abstract class SimpleValueObject <T extends Comparable> implements
                        Serializable, Comparable<SimpleValueObject<T>> {

  private T value;

  protected SimpleValueObject() { }

  public SimpleValueObject(T aValue) {
    this.value = this.validateAndNormalize(aValue);
  }

  protected T getValue() {
    return this.value;
  }

  protected T validateAndNormalize(T aValue) {
    Validate.notNull(aValue, "Value may not be null");
    return aValue;
  }

  public boolean equals(final Object o) {
    if (this == o) {
      return true;
    }
    if (o == null || this.getClass() != o.getClass()) {
      return false;
    }
    SimpleValueObject that = (SimpleValueObject) o;
    if (this.value != null ?
      !this.value.equals(that.value) : that.value != null) {
        return false;
      }
      return true;
    }

    public int hashCode() {
      return this.value != null ? this.value.hashCode() : 0;
    }

    public String toString() {
      return String.valueOf(this.value);
    }

    public int compareTo(final SimpleValueObject<T> o) {
      return this.value.compareTo(o.getValue());
    }
  }
```

dem Backend nach und speichert diese wieder in der Session. Damit sind Backend und Session wieder synchron. Durch diesen Mechanismus konnte die Anzahl der „teuren“ Web-Service-Aufrufe auf ein Minimum reduziert werden.

Authentication mit JAAS

Die Authentifizierung von Kunden wird im neuen System nicht mehr direkt von der Anwendung durchgeführt. Stattdessen wird der Java Authentication and Authorization Service (JAAS) genutzt. Hierfür wurden so genannte Providermodule geschrieben, die Kunden gegen einen Authentication Service des ESB am Portal authentifizieren. Im Code können Rechte sowohl in den Views über die *web.xml* als auch in den Java-Methoden mittels Annotationen vergeben werden.

Die Rechteprüfung basiert im Standard auf der Annotation *@RolesAllowed*. Im Portal wurde zusätzlich die selbst implementierte Annotation *@RolesNotAllowed* eingeführt. Gemäß Convention over Code bzw. over Configuration mussten so deutlich weniger Methoden annotiert werden, wobei das gleiche Ergebnis erzielt wurde wie bei der Verwendung von *@RolesAllowed*. In der *web.xml* ist es weiterhin möglich, Views auf bestimmte Rollen einzuschränken. Über den *FacesContext* lässt sich jederzeit ermitteln, welche Rollen der angemeldete User hat. Somit kann mittels JSF-Attribut *rendered* die Berechtigung auf View-Komponenten-Ebene vergeben werden.

Listing 4

```
public class FtpProductBuilder extends AbstractEntityBuilder<FtpProduct> {

  private FtpProduct product;

  public static CssFtpProductBuilder newInstance(FtpUserName userName) {
    notNull(userName, "userName may not be null");
    FtpProductBuilder builder = new FtpProductBuilder();
    builder.product = new FtpProduct();
    builder.product.setUsername(userName);
    return builder;
  }

  public FtpProductBuilder at(FtpDirectory directory) {
    notNull(directory, "directory may not be null");
    product.setDirectory(directory);
    return this;
  }

  @Override
  public CssFtpProduct build() {
    notNull(product.getDirectory(), "directory is missing");
    return product;
  }
}
```

Fazit

Die Automatisierung von Migrationsschritten hin zu Java EE ist mit den schnellen Entwicklungszyklen und den ausgereiften technologischen Aspekten der Java-EE-6-Plattform keine Notwendigkeit mehr, um eine Migration erfolgreich durchführen zu können. In diesem speziellen Fall war das Migrationskonzept „Neuentwicklung“ wirtschaftlich sinnvoller als Zeit und Geld in automatisierte Migrationsverfahren oder Tools zu stecken. Der Schritt hin zu einer standardisierten Plattform und weg von eigenentwickelten Lösungen hat dazu geführt, dass die „Ramp-up Time“ von neuen Entwicklern von mehreren Wochen auf wenige Tage heruntergeschraubt werden konnte. Weiterhin konnte viel Zeit durch die Verwendung von etablierten Komponentenbibliotheken wie *PrimeFaces* eingespart werden. CDI hilft an vielen Stellen, die Fachlichkeit in den Vordergrund zu stellen und technischen Code, zum Beispiel zum Laden von Daten, zu minimieren. Wichtig für die erfolgreiche Migration war zunächst zu überlegen, ob es schon eine Lösung für dieses Problem in Java EE 6 gibt. Diese Lösungen entsprechen zum einem dem Standard und sind zum anderen in der Regel besser dokumentiert. Sie führen dazu, dass Entwickler sich auf die Fachlichkeit konzentrieren können, statt sich um technische Infrastrukturaspekte der Anwendung kümmern zu müssen.

Thilo Focke ist Lead Software Engineer bei der EWE TEL GmbH. Er beschäftigt sich seit fünfzehn Jahren mit Softwareentwicklung und -architektur. Sein aktueller Fokus liegt auf mobiler Entwicklung und der Realisierung von TK-Konsolidierungsprojekten im Java-EE-Umfeld.

Marc Petersen ist Softwareentwickler bei der open knowledge GmbH in Oldenburg. Sein Schwerpunkt liegt auf der Entwicklung webbasierter Enterprise-Lösungen mittels Java EE. Darüber hinaus gilt sein Interesse der Realisierung von adaptiven Web- und Mobile-Anwendungen.

Christian Zillmann ist Software Engineer bei der EWE TEL GmbH und beschäftigt sich seit über zehn Jahren mit Java-EE-Entwicklung und -Betrieb. Ein besonderer Schwerpunkt liegt hierbei auf JSF, CDI und JAX-WS/RS.

Links & Literatur

[1] Shannon, Bill: „JavaTM Platform, Enterprise Edition (Java EE) Specification, v5“, Sun Microsystems, 2006

[2] http://www.clean-code-developer.de

[3] Schumann, Jens: „Generalüberholung Java EE 6 Style“, in Java Magazin 2.2013, S. 38–43

[4] Josuttis, Nicolai: „SOA in der Praxis – System-Design für verteilte Geschäftsprozesse“, dpunkt.verlag, 2008

[5] Laplante, Philip A.: „What every engineer should know about software engineering“, CRC, 2007

[6] Evans, Eric: „Domain-Driven Design: Tackling Complexity in the Heart of Software“, Addison-Wesley, 2003

Kolumne: EnterpriseTales

von Lars Röwekamp und Arne Limburg

Let it flow: Faces Flow in JSF 2.2

Der logische und visuelle Ablauf einer JSF-Anwendung lässt sich bekanntermaßen mittels expliziter Deklaration von Navigationsregeln innerhalb der JSF-Konfiguration oder aber alternativ über die Angabe konkreter Zielseiten als Rückgabewert von JSF-Action-Methoden steuern. Was aber ist, wenn das zugrunde liegende Regelwerk komplexer ausfällt und sich nicht mehr mit den bisher bekannten Bordmittel abbilden lässt? Ein (Aus-) Blick auf JSF 2.2 zeigt eine mögliche Lösung.

Die Umsetzung nicht trivialer Navigationspfade ist in JSF seit jeher nicht gerade ein Quell der Freude. Der Entwickler kann sich bisher lediglich zwischen zwei eher suboptimalen Alternativen entscheiden. Entweder erstellt er extrem komplexe und somit kaum wartbare, XML-basierte Navigationsregeln oder er verschleiert die vorhandene Komplexität der Navigation, indem er sie stark verteilt, über alle JSF-Action-Methoden hinweg, als deren berechneten Rückgabewerte implementiert. Richtig spaßig wird es dabei immer dann, wenn die Navigation zusätzlich bestimmten Bedingungen – zum Beispiel der Rolle des angemeldeten Benutzers – unterliegt. Abhilfe soll hier nun eines der „Big Tickets“ aus JSF 2.2 [1] schaffen, welches sich an bereits etablierten Lösungen, wie ADF Task Flow und Spring Web Flow orientiert – Faces Flows.

Der Weg ist das Ziel

Faces Flows erlauben laut Spezifikation die Kapselung von zusammenhängenden Views und Beans zu einem, innerhalb mehrerer Anwendungen widerverwendbaren Modul, mit wohldefiniertem Einstiegs- und Ausstiegspunkt. Zu abstrakt? Nehmen wir als Beispiel eine Webshopanwendung inkl. einem „Zur Kasse“-Wizard. Der gesamte Check-out-Prozess mit Bestellübersicht, Angabe der Zahlungsbedingungen – inkl. spezifischer Verzweigungen für Rechnung, Kreditkarte etc. – Eingabe von Liefer- und Rechnungsadresse und endgültiger Bestätigung der Bestellung, ließe sich mithilfe eines Faces Flows abbilden.

Der eigentliche Unterschied zu den bereits bekannten JSF-Mechanismen ist, dass sich die Navigation innerhalb einer Anwendung nicht mehr, wie bisher, auf die Navigation zwischen zwei Seiten beschränkt, sondern zukünftig die Übergänge zwischen *Flow Nodes* angegeben werden können. Bei den Flow Nodes wiederum kann dabei aus mehreren verschiedenen Typen gewählt werden:

- *View Node*: JSF-Seite innerhalb der Anwendung
- *Method Call Node*: Aufruf von Businesslogik (via Expression)
- *Switch Node*: Navigationslogik innerhalb des Flow-Graphen (via EL)

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

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

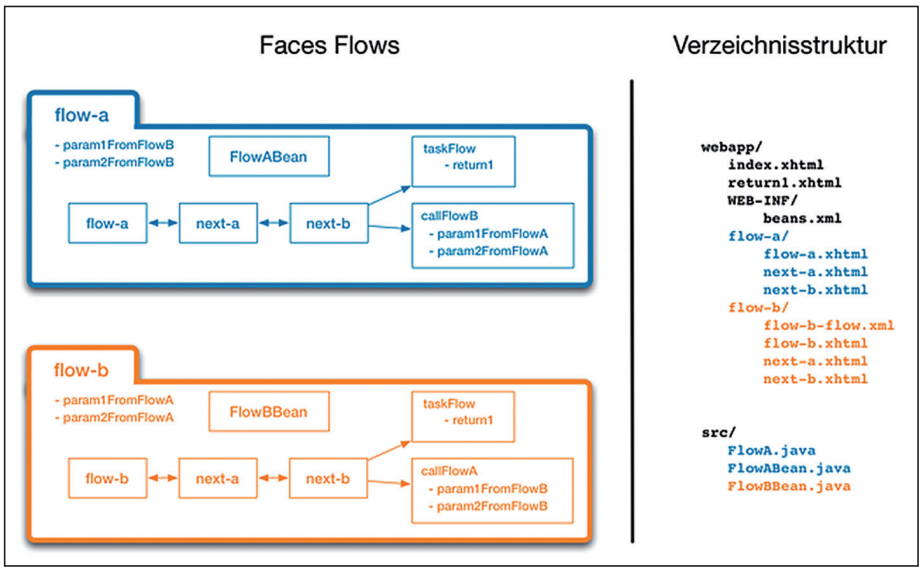


Abb. 1: Faces Flows im Einsatz

- *Flow Call Node*: Parametrisierter Aufruf eines anderen Flows
- *Flow Return Node*: Rücksprung aus einem Flow inkl. Outcome

Wie die oben aufgeführte Liste von Nodes bereits vermuten lässt, können mithilfe der fünf Node-Typen vollständige View-Workflows inkl. Business- und Navigationslogik definiert werden.

Möchte man die innerhalb eines Flows gesammelten Daten in einer speziellen Bean zusammenfassen, kann man diese mit dem neuen Scope *@FlowScoped* annotieren. Die oben bereits angesprochene Definition des Flows selbst kann via XML-Deklaration oder alternativ mithilfe des Fluent-FlowBuidler-API erfolgen. Im zweiten Fall wird die zugehörige Klasse, die den Faces Flow definiert, mit *@FlowDeclaration* annotiert.

Ordentlich verpackt

Wie spätestens seit Java EE 5 üblich, greifen auch bei den Faces Flows einige Konventionen und vereinfachen so das Leben des Entwicklers:

- Werden Faces Flows direkt in eine Anwendung eingebunden, bekommt jeder Flow sein eigenes Verzeichnis.
- Der Flow Node innerhalb des Verzeichnisses, der denselben Namen trägt, wie der Flow selbst, ist automatisch der Einstiegspunkt in den Faces Flow.
- Navigation aus dem Flow heraus definiert automatisch einen Ausstiegspunkt.
- Existiert eine XML-Deklaration, dann trägt sie den Namen *<flow>-flow.xml*.

Abbildung 1 zeigt ein leicht modifiziertes Beispiel aus der JSF-2.2-Spezifikation, welches zwei sich gegenseitig aufrufende Flows definiert. Während *flow-b* mithilfe von XML in *flow-b-flow.xml* deklarativ angegeben wurde, findet sich die Definition von *flow-a* in der mit *@FlowDeclaration* annotierten Klasse *FlowA.java*. Der initiale Einsprung in die Flows erfolgt über die Seite *index.xhtml*. Am Ende eines jeden Flows kann entweder in den jeweils anderen Flow – inkl. Übergabe von Parametern – gesprungen werden oder alternativ via gleichnamigen Outcome auf die Seite *return1.xhtml*.

Etwas komplizierter wird es, wenn ein oder mehrere Faces Flows als eigens JAR verpackt ausgeliefert werden sollen. In dem Fall muss jeder vorhandene Faces Flow explizit in der Konfiguration *META-INF/faces-config.xml* angegeben und die Nodes der Flows in gleichnamigen Unterverzeichnissen von *META-INF* abgelegt werden. Liegen darüber hinaus mit *@FlowScoped* oder *@FlowDeclaration* annotierte Beans in dem JAR, muss dies zusätzlich innerhalb der CDI-Konfiguration *beans.xml* signalisiert werden.

Fazit

Mithilfe von Faces Flows wird eine wichtige und vor allem nervige Lücke innerhalb der JSF-Spezifikation geschlossen. In der aktuellen Spezifikation 2.1 bisher nur sehr umständlich zu definierende, komplexe und durch Businesslogik beeinflussbare Navigationspfade können dank Faces Flow zukünftig modularisiert und widerverwendbar angegeben werden. Natürlich können Faces Flows keinen Ersatz für eine ausgereifte Workflow-Engine bieten. Dies ist aber auch nicht das Anliegen der Spezifikation. Ziel ist vielmehr die deutliche Vereinfachung der Definition von komplexen Webnavigationspfaden – nicht mehr und nicht weniger.

Links & Literatur

[1] JSF 2.2 Spezifikation: http://jcp.org/en/jsr/detail?id=344

Ausbau des Webshops mit JSF 2.2

T-Shirt-Shop de luxe

Der Sprung von JSF 2.1 auf JSF 2.2 ist zunächst ein Minor Update. Dennoch sind in diesem vermeintlich kleinen Versionssprung sehr viele Neuerungen enthalten, die beinahe schon ein Major Update gerechtfertigt hätten. Interessant an dieser neuen Version ist, dass die Java-EE-Welt näher zusammenrückt und nicht mehr jeder Standard ein separates Dasein führt. So integriert sich JSF zusehends mit CDI und lässt eigene Ansätze im @ManagedBean-Bereich auslaufen.

von Andy Bosch

Im ersten Teil dieses Tutorials haben wir einen T-Shirt-Webshop aufgebaut, der bereits viele neue Funktionen von JSF 2.2 einsetzte. Wir haben das Konzept von Resource Library Contracts kennen gelernt und verschiedene Templates als Module austauschbar in der Anwendung konfiguriert. Mittels *ViewActions* konnten wir speziell bei *Get*-Requests eine Aktion vor dem Darstellen der Seite aufrufen und ggf. sogar auf andere Seiten verweisen. Natürlich basiert unser Shop auf HTML5 und damit auch auf neuen Tags und Attributen, die wir mittels *PassThroughAttributes* und *PassThroughElements* auch problemlos verwenden konnten. Über den in JSF 2.2 standardisierten *FileUpload* waren wir in der Lage, Dateien mit JSF-Tags an den Server zu übertragen.

Das Ausbauvorhaben des Webshops

Im Webshop, den wir in Teil 1 begonnen haben, haben wir bereits etliche neue Funktionen aufgenommen. Dennoch gibt es ein paar Ecken, an denen wir noch feilen können und mittels weiterer neuer Funktion von JSF 2.2 Verbesserungen nutzen können. Ein großer Punkt wird das Thema **Faces Flows** sein. Mittels Faces Flows können (View-)Abläufe zusammengefasst werden und über fest definierte Ablauffolgen ausgeführt werden. Zudem können mit diesem Konzept wiederverwendbare Seitenfolgen als Module ausgelagert bzw. auch einer Anwendung als JAR-Bibliothek hinzugefügt werden.

Eine weitere wichtige Eigenschaft einer Webanwendung ist es häufig, Multi-Tab- bzw. Multi-Window-fähig zu sein. Das bedeutet, dass wir unsere Anwendung in mehreren Tabs oder Fenstern quasi parallel betreiben können. Hierzu konnte man in der Vergangenheit mittels CDI Conversations schon einiges erreichen, das JSF-Framework selbst hatte dazu jedoch nichts vorzuweisen. Neu hinzugekommen ist hier das **ClientWindow**, das es ermöglicht, neue Tab-/Fensterinstanzen über eine dynamisch erzeugte ID zu identifizieren.

Da unser Webshop auch sicher gegenüber Hackerangriffen sein soll, ist das Thema **Security** natürlich von großer Bedeutung. Ein in der Praxis häufig anzutreffender Angriffsvektor ist CSRF (Cross Site Request Forgery). Für eine ausführliche Definition verweise ich einfach auf den entsprechenden Wikipedia-Artikel [1]. Kurz zusammengefasst bedeutet CSRF, dass es nicht möglich sein soll, einen ungewollten Request eines Angreifers innerhalb der eigenen legitimierten Session ausführen lassen zu können.

Alles im Fluss

Beginnen wir zunächst mit Faces Flows. Faces Flows sind ein neues Konzept, um zusammenhängende Seitenflüsse inklusive der Navigationslogik modular zu verpacken. Diese Flows können damit wiederverwendbar gestaltet werden. Auch eine Bereitstellung von Flows über ein JAR-File ist angedacht und möglich. Die Idee dieser Flows ist nicht neu. Vielmehr haben Open-Source-Projekte wie Spring Web Flow, ADF Task Flows oder MyFaces CODI hier geholfen, die besten Ideen daraus in den JSF-Standard zu packen – getreu dem Motto, das Ed Burns einmal prägte: „Standards are for standardizing. Not for innovation."

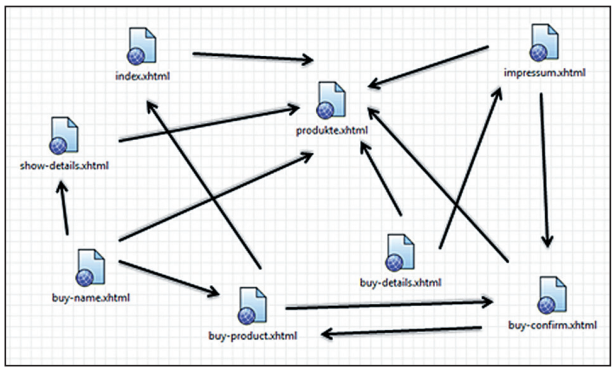


Abb. 1: Freie Navigationsmöglichkeiten

In **Abbildung 1** ist die übliche Navigationslogik in JSF visualisiert. Es ist in der Regel möglich, von jeder Seite beliebig auf eine andere Seite zu springen. Zwar bietet die JSF-Navigationslogik seit 1.0 bereits die Möglichkeit, dies ein wenig einzuschränken; in der Praxis wird dies nach meiner Beobachtung jedoch nicht allzu häufig verwendet. Man arbeitet eher mit globalen Navigationsregeln, die genau dieses „wilde" Navigieren zulassen. Mit Faces Flows wird hier ein stringenterer Mechanismus bereitgestellt, der einen klaren Pageflow bestimmt, mit einem Anfangspunkt und Ausstiegspunkten (**Abb. 2**).

Zusätzlich wurde ein eigener Scope dafür bereitgestellt, der *@FlowScoped*, der Beans genauso lange am Leben erhält, wie der Flow aktiv ist. Verlässt der Benutzer den Flow, wird die Bean zerstört.

Listing 1: Bestellprozess mit FacesFlows

```
<flow-definition id="orderflow">
  <initializer>#{shopCtrl.initializeFlow}</initializer>
    <start-node>product</start-node>
    <view id="product">
       <vdl-document>/flows/orderflow/buy-product.xhtml</vdl-document>
    </view>
    <view id="name">
      <vdl-document>/flows/orderflow/buy-name.xhtml</vdl-document>
    </view>
    <view id="details">
      <vdl-document>/flows/orderflow/buy-details.xhtml</vdl-document>
     </view>
    <view id="confirm">
      <vdl-document>/flows/orderflow/buy-confirm.xhtml</vdl-document>
    </view>

    <flow-return id="/index">
      <from-outcome>/index.xhtml</from-outcome>
     </flow-return>
     <flow-return id="/pages/produkte">
       <from-outcome>/pages/produkte.xhtml</from-outcome>
     </flow-return>
    <flow-return id="/pages/impressum">
       <from-outcome>/pages/impressum.xhtml</from-outcome>
     </flow-return>

   <finalizer>#{shopCtrl.cleanUpFlow}</finalizer>
</flow-definition>
```

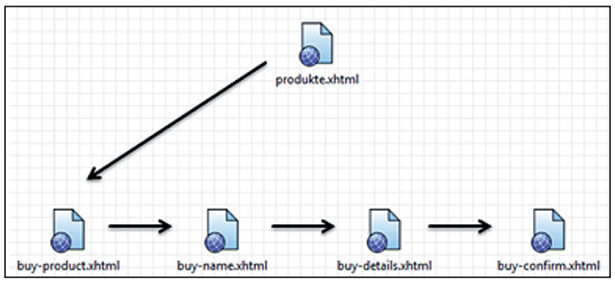


Abb. 2: Navigation über Flows

In Listing 1 sehen Sie ein umfangreicheres Beispiel für den Bestellprozess in unserer T-Shirt-Anwendung. Es werden die entsprechenden Views für die Bestellung als View Nodes definiert. Doch diese View Nodes sind nur ein kleiner Anteil dessen, was JSF 2.2 an dieser Stelle zu bieten hat. Konkret unterstützt die Spezifikation folgende Node Types:

- *View Node*: das sind die bereits erwähnten Views, sprich: die konkreten Seiten, die innerhalb des Flows dargestellt werden.
- *Method Call*: kann in einem Schritt während der Flow-Abarbeitung aufgerufen werden. Die Methode ist i. d. R. eine Method Expression, die bei Auftreten evaluiert wird. Die Rückgabe der Methode kann auf weitere Nodes verweisen, z. B. eine konkrete View Node für die Anzeige aktivieren.
- *Switch*: entspricht dem klassischen Verständnis einer Switch-Anweisung aus Java. Es wird eine Expression ausgewertet und je nach Rückgabe der Auswertung ein weiterer Node mit angezogen.
- *Flow Call* und *Flow Return*: Von einem Flow aus kann ein weiterer (Sub-)Flow aufgerufen und natürlich aus diesem Flow wieder zum rufenden Flow zurückgekehrt werden.

Zusätzlich zu den erwähnten Node Types unterstützt die Spezifikation auch so genannte *Initializer* und *Finalizer*. Dies sind Einsprungpunkte (meist Expressions), die bei Beginn respektive bei Verlassen eines Flows ausgeführt werden.

Eine Flow-Definition wird mit einer eindeutigen ID beschrieben, über die sie auch im Programm aufgerufen werden kann. Anschließend werden die verschiedenen Navigationsmöglichkeiten definiert. Erwähnenswert ist, dass ich die drei Hauptnavigationspunkte explizit als Flow-Return aufgenommen habe. Dies bewirkt, dass zu jedem Zeitpunkt auf die Hauptmenüeinträge geklickt werden kann und damit der Flow beendet wird.

Passend zum neu gestalteten Flow wurde die dahinter liegende Bean, die Klasse *ShirtOrder*, von *SessionScope* auf *FlowScope* geändert. Das ist wesentlich ressourcensparender und auch vom Design her klarer als alles immer in die Session zu packen.

Im *FlowScoped* Bean, das in Listing 2 abgebildet ist, wurde zunächst nur der Scope per Annotation ausgetauscht. Für erste Lernschritte habe ich zwei Methoden mit *@PostConstruct* und mit *@PreDestroy* annotiert, um genau verfolgen zu können, dass die Bean auch korrekt beim Betreten des Flows angelegt und beim Verlassen des Scopes zerstört wird.

Doch wo werden diese Flow-Definitionen abgespeichert? Die Spezifikation kennt hier wieder zwei Ablageorte: Zum einen kann im */flows*-Verzeichnis in der Web-App ein Flow mitsamt allen Views gespeichert werden; zum anderen können Flows auch in JAR-Files ausgelagert werden. Dort werden Flows im *META-INF/flows*-Verzeichnis gesucht. Es ist somit möglich, modulare Seitenabfolgen als separate Deployment Unit zur Verfügung zu stellen und zur Deployment-Zeit in einer Anwendung bereitzustellen.

Listing 2: FlowScoped Bean

```
@Named
@FlowScoped("orderflow")
public class ShirtOrder implements Serializable {

  @PostConstruct
  public void afterCreation() {
    System.out.println("## wurde erzeugt");
  }

  @PreDestroy
  public void beforeDelete() {
    System.out.println("## werde gelöscht");
  }
  ...
}
```

Mehr als ein Fenster

Eine Sache, die wir als Power-User im Internet immer wieder gerne nutzen, ist die Möglichkeit, in mehreren parallelen Fenstern oder Tabs zu arbeiten. Speziell bei Übersichtsseiten oder Suchergebnisseiten öffnen wir die in einer Liste dargestellten Informationen für die Detailansicht gerne in einem neuen Tab. Was aus Sicht des Users eine sehr praktische Lösung ist, stellt für den Anwendungsentwickler regelmäßig eine große Herausforderung dar. Das Problem ist, dass die genau gleiche Anwendung (inklusive der Anwendungssitzung) in mehreren Zuständen sein kann (verschiedene Seiten in verschiedenen Tabs geöffnet). Erfolgen jetzt Requests an den Server, muss dieser unterscheiden können, auf welches Fenster sich die Aktion bezieht. Da das HTTP-Protokoll diese Information nicht enthält, müssen Lösungen gebastelt werden, die eine Unterscheidung des

geöffneten Fensters oder Tabs ermöglichen. Die Lösung in JSF 2.2 lautet *ClientWindow*. Ein *ClientWindow* stellt serverseitig ein Fenster bzw. Tab dar, also eine Art eigenständige Arbeitsumgebung. Damit JSF hier serverseitig eine Unterscheidung machen kann, muss im *ClientRequest* eine Information mitgegeben werden. Bei jedem Request muss somit eine Identifikation enthalten sein, die es dem Server (bzw. JSF) ermöglicht, das konkrete Fenster bzw. Tab zu identifizieren. Da es hierfür keinen Königsweg gibt, wie man einem Request diese Informationen mitgeben kann, kennt die Spezifikation verschiedene Varianten. Über den Kontext-Parameter *javax.faces.CLIENT_WINDOW_MODE* kann die gewünschte Variante gesteuert werden. Die Spezifikation kennt die Werte *none* und *url*. Eigene Erweiterungen können hier eingehängt werden. Der Wert *none* bewirkt, dass das Feature deaktiviert wird und kein *ClientWindow* zur Verfügung steht. *url* bewirkt, dass über URL-Parameter und ggf. Hidden Fields eine *WindowId* mitgegeben wird. In Listing 3 ist der Eintrag der *web.xml* zu sehen. Fehlt dieser Parameter, wird *url* als Standard gesetzt.

Wie wirkt sich nun das Vorhandensein eines *ClientWindow* in JSF aus? Zunächst bewirkt es nichts Offensichtliches. Es gibt in der Spezifikation keinen Scope, der speziell an ein *ClientWindow* gekoppelt ist. Natürlich steht es den verschiedenen zusätzlichen Frameworks im JSF-Umfeld offen, sich genau an dieser Stelle einzuklinken und passende Scopes zu liefern. Für unser Shop-Beispiel habe ich einen sehr rudimentären Mechanismus geschrieben, in dem ich Daten pro *ClientWindow* ablege und somit einen eigenen Scope erzeugt habe (wie gesagt, nur sehr rudimentär zu Demonstrationszwecken). Damit ermögliche ich es, dass man sich die verschiedenen Shirts in separaten Tabs anschauen kann. Die verschiedenen Tabs werden im URL mit einem Parameter aufgerufen, der serverseitig zu verschiedenen *ClientWindows* führt. Die anzuzeigenden Werte mache ich von diesem Wert (bzw. von der *WindowId*) abhängig. Konkret habe ich eine Erweiterung auf der Produktseite eingebaut, mit der man auf eine Produktdetailseite navigiert. In dieser Detailseite wird eine Bean mitsamt ihren Properties angesprochen. Nun kann der Benutzer von der Produktseite mehrere Detailseiten in mehreren parallelen Tabs öffnen. In jedem Tab bzw. Fenster soll die passende Bean in einem „Tab-Scope“ gespeichert werden. Der Einstieg in die Detailseiten erfolgt über die Produktliste, in der eine weitere Spalte für Details abgebildet ist. Das relevante Codeschnipsel sieht folgendermaßen aus:

```
<h:link disableClientWindow="true" value="Details" outcome="show-details.
                                                              xhtml">
  <f:param name="prodName" value="#{current.name}" />
</h:link>
```

Ein *<h:link>*-Tag wird verwendet, um einen *GET*-Aufruf zu erzeugen, den man üblicherweise mit Open in new window bedienen kann. Da ich mich jedoch bereits in einer *ClientWindow*-Umgebung befinde, muss im neuen Tab sichergestellt sein, dass eine neue *WindowId* erzeugt und nicht die bestehende weitergereicht wird. Dies erfolgt über das Attribut *disableClientWindow*. Jetzt wird ein URL erzeugt, der als Parameter den Namen des Produkts übergeben bekommt (das ist jetzt nicht sonderlich elegant, hier sollte in der Praxis eine – verschlüsselte – ID verwendet werden). Auf der Zielseite muss dieser Parameter ausgewertet und die Produktdetails für diesen Scope (das Window bzw. Tab) müssen geladen werden. Hierzu verwende ich die ebenfalls in JSF 2.2 neu hinzugekommenen *Viewactions*. Auf diese sind wir schon im ersten Teil des Tutorials eingegangen.

Die Methode *loadShirtDetails* (Quellcode unten) wertet den über *ViewParameter* übergebenen Wert des URL aus, lädt das entsprechende Produkt und stellt es in den eigens geschaffenen Tab-Scope:

```
public void loadShirtDetails() {
  if ( urlProdName!=null && !"".equals( urlProdName ) ) {
    Shirt shirt = totalContainer.getShirtByName( urlProdName );
    detailContainer.setShirtInScope( shirt );
  }
}
```

Die Realisierung an dieser Stelle ist sehr einfach:

```
FacesContext jsfCtx = FacesContext.getCurrentInstance();
String wId = jsfCtx.getExternalContext().getClientWindow().getId();
storedShirtsInScope.put( wId, shirt );
```

Über den Aufruf *getClientWindow().getId()* wird der Identifier für das aktuelle Window gezogen. Dieser Wert wird verwendet, um das gerade geladene Shirt in einer Map zu speichern. Die Map liegt im *SessionScope*. So-

Listing 3: Kontextparameter für den ClientWindow Mode

```
<context-param>
  <param-name>javax.faces.CLIENT_WINDOW_MODE</param-name>
  <param-value>url</param-value>
</context-param>
```

Listing 4: Abfrage des eigenen Scopes

```
public Shirt getCurrentShirt() {
  FacesContext jsfCtx = FacesContext.getCurrentInstance();
  String wId = jsfCtx.getExternalContext().getClientWindow().getId();
  if ( storedShirtsInScope.containsKey( wId ) ) {
    return storedShirtsInScope.get( wId );
  }
  return null;
}
```

mit ist der neu geschaffene Tab-Scope direkt abhängig von der *HttpSession*. Eine elegante Lösung, wie die Daten des Tab-Scopes wieder entfernt werden, fehlt hier noch. Aber die Lösung zeigt sehr deutlich, wie mit wenig Aufwand ein clientseitiges Window oder Tab erkannt und serverseitig bearbeitet werden kann. Um die Lösung vollends abzurunden, zeige ich nun auf der Detailseite auf eine Bean, die die Daten für den aktuellen Tab bereitstellt:

```
<h:outputText id="col" value="#{shirtDetailContainer.currentShirt.name}" />
```

Die Implementierung der Methode *getCurrentShirt* ist in Listing 4 zu sehen.

Über die *WindowId* wird in der Map das dazugehörige Shirt geladen und zurückgeliefert. Die Auswertung erfolgt bei jedem Aufruf der *getter*-Methode, was generell in JSF nicht zu empfehlen ist. In diesem speziellen Fall sollte dies aber keine Performanceprobleme nach sich ziehen. Wir können jetzt Daten in verschiedenen Tabs parallel offen halten und bearbeiten – und das lediglich mithilfe der Möglichkeiten der Spezifikation. Wenn hier dann JSF-Komponentenbibliotheken weitere Scopes bauen, wird es sicherlich noch um einiges komfortabler.

Etwas Ajax mit Reset-Funktionalität

Ein bisschen Ajax und PPR (Partial Page Rendering) dürfen natürlich auch in diesem Webshop nicht fehlen. Um die neue Reset-Funktionalität zu demonstrieren, habe ich ein Formular mit einer kleinen Ajax-Erweiterung versehen. Bei Eingabe des Namens erfolgt per Ajax bei einem Treffer vorab die Befüllung der weiteren Felder wie Postleitzahl und Ort. Zugegeben, über die fachliche Korrektheit lässt sich streiten, sie zeigt jedoch ein neues Verhalten sehr schön:

```
Nachname:
<h:inputText value="#{customer.lastname}" id="last"
             valueChangeListener="#{shopCtrl.lastNameChanged}">
  <f:ajax render="myForm" execute="first last" />
</h:inputText>
<h:message for="last" />
```

In diesem Quellcodeausschnitt ist zu sehen, dass ein *ValueChangelistener* an dem Eingabefeld hängt, der nach Verlassen des Feldes aktiviert wird und ggf. weitere Werte in den Managed Bean Customer einstellt. Durch das *<f:ajax>*-Tag wird der Listener gleich nach Verlassen des Eingabefeldes angestoßen.

Da auf dem Eingabefeld für den Vornamen ein *Required-Validator* liegt, kann es jedoch zu Validierungsfehlern kommen (**Abb. 3**). Das alles funktioniert an sich recht gut. Nehmen wir jedoch einmal an, dass wir einen Reset-Button einbauen wollen, der die Maske in den Initialzustand zurücksetzt (alle Eingaben auf leer), hatten wir in JSF bislang ein Problem bei der Verwendung von Ajax. Zwar kann auf einem Reset-Button eine Aktionsmethode aufgerufen werden (die auch aktiviert wird, da *execute* nur auf *@this* beschränkt ist), aber das anschließende Re-Rendering zeigt evtl. immer noch einen Wert im Feld *Nachname* an:

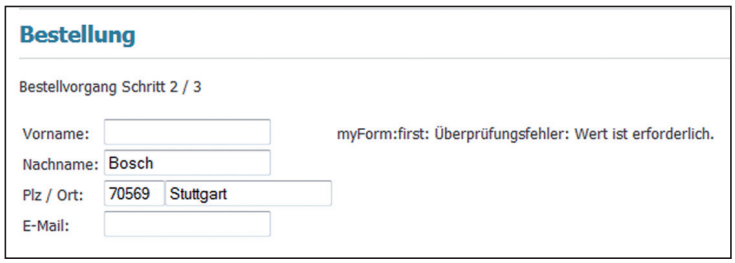


Abb. 3: Validierungsfehler bei Ajax-Verhalten

```
public void clearInputs() {
  customer.setFirstname( "" );
  customer.setLastname( "" );
  customer.setZip( "" );
  customer.setCity( "" );
}
```

Grund ist, dass die Komponente in JSF neben der Datenhaltung in der Managed Bean einen so genannten *Local Value* hat. Das ist der Wert, der in der Komponente zwischengespeichert wird, solange die *UpdateModel*-Phase noch nicht durchlaufen wurde. Dieses Verhalten ist aus Sicht des JSF-Lifecycles korrekt und in den allermeisten Fällen auch passend zur Fachlichkeit. Im konkreten Fall eines Resets stört dieses Verhalten jedoch. Zwar wurde die Managed Bean zurückgesetzt (Quellcode oben), aber da das Eingabefeld noch einen lokalen Wert hat, wird dieser zunächst visualisiert. Um dieses Verhalten zu umgehen, muss der lokale Wert in der Komponente zurückgesetzt werden. Erst danach greift der Renderer bei einem Render-Vorgang auf den Wert in der Bean zurück. Die Lösung mit JSF 2.2 lautet *resetValues*. Dies ist ein Attribut, das in einem *<f:ajax>*-Tag gesetzt werden kann. Es weist die Komponenten an, den lokal gespeicherten Wert zurückzusetzen – genau das Verhalten, das wir hierfür benötigen. Damit ist das leidige Reset-Problem, das in der Community schon länger für Unmut gesorgt hatte, endlich beseitigt.

CSRF lässt grüßen

Das Thema Security gehört seit einigen Jahren zu einem Standard-Feature bei Webanwendungen. So ziemlich jeder Webentwickler wird die OWASP (Open Web Application Security Project, [2]) Top 10 kennen (oder zumindest schon davon gehört haben). Auch viele Angriffsvektoren sind von der Begrifflichkeit vertraut. CSRF (Cross Site Request Forgery) ist ein solches Beispiel. Jeder kennt den Begriff, aber mancher wird sich sicher schon gedacht haben: Hoffentlich fragt mich niemand nach Details. Doch genau diese Herangehensweise ist falsch. Wir als Entwickler sollten uns sehr wohl bewusst machen, wie eine Webanwendung gehackt werden kann, damit wir im Vorfeld bereits Gegen-

maßnahmen ergreifen können. Speziell bei CSRF geht es darum, dass eine bösartige Anfrage (Request) einem ahnungslosen Opfer untergeschoben wird. Oder mit anderen Worten: Es wird innerhalb der Session eines Opfers ein *HttpRequest* abgesetzt (also im Namen des Opfers), ohne dass dieser die Aktion tatsächlich wollte. Das können einfach Anfragen nach Detailseiten sein, aber auch gravierendere Aktionen wie die Durchführung einer Überweisung. Wenn es einem Angreifer z. B. gelänge, dass folgender *GET-Url* ausgeführt wird, wird das Ausmaß von CSRF sehr deutlich: http://www.bankDesOpfers.com?transferMoney=1000&from=dasOpfer&To=Hacker.

Dies ist natürlich stark vereinfacht dargestellt, zeigt aber, dass wir hier Vorsicht bei unserer Webanwendung walten lassen sollten. Seit JSF 2.0 ist die GET-Unterstützung stark verbessert worden. Es können *ViewParameter* ausgewertet werden, und seit JSF 2.2 existieren *ViewActions*. *PreRenderView*-Events und *PhaseListener* können zudem auf URL-Parameter reagieren und ggf. Aktionen antriggern. Zur Vermeidung, dass ungewollte Requests abgesetzt werden und auf Serverseite verarbeitet werden, werden häufig so genannte *Page Tokens* eingesetzt. Ein Page Token ist eine willkürlich erzeugte Zufallszahl, die als Hidden Field oder URL-Parameter in den Requests eingebunden wird. Diese Zahl wird serverseitig ebenfalls gespeichert. Erfolgt nun ein Request vom Client an den Server, wird geprüft, ob in den Request-Parametern dieser Token vorhanden ist und dem aktuellen Wert im Server entspricht. Wenn ja, ist der Request gültig und wird verarbeitet. Es wird dann ein neuer Token generiert und an den Client zurückgesendet. JSF war hier in der Vergangenheit bereits recht gut aufgestellt. Über den *ViewState* musste (zumindest bei Post) ein String zur Identifizierung des *ServerStates* mitgesendet werden. Allerdings war dies kryptografisch nicht sehr sicher ausgearbeitet, konnte man doch mit ein wenig Aufwand diese ID berechnen. Zudem war bei GET-Requests dieser Schutz überhaupt nicht gegeben.

Mit JSF 2.2 halten so genannte **Protected Views** Einzug in die Spezifikation. Damit können Seiten als schützenswert deklariert werden, sodass diese nicht mehr direkt (z. B. per direkter URL-Eingabe) aufgerufen werden können. Das Aufrufen dieser geschützten Seiten ist nur noch mit gültigem Token möglich. Dieses Token wird durch JSF automatisch gesetzt, wenn über JSF-Aktionen auf diese Seite navigiert wird. CSRF wird somit praktisch unmöglich.

Im Folgenden ist ein Beispiel zu sehen, in dem lediglich die Detailseite eines Produktes geschützt wird:

```
<protected-views>
  <url-pattern>/pages/show-details.xhtml</url-pattern>
</protected-views>
```

Unsere übliche Anwendungslogik funktioniert weiter tadellos. Wir können über die Produktlistenseite auf die Detailseite navigieren. Wenn wir jedoch direkt den URL aufrufen (und alle eventuell vorhandenen URL-Parameter abtrennen), wird eine *ProtectedViewException* erzeugt. Diese könnte ggf. mittels entsprechender Fehlerseiten behandelt werden. Es zeigt jedoch vor allem, dass ohne gültigen Token die Seite nicht aufgerufen werden kann. Der Token wird bei POST-Requests als Parameter mitgesendet. Bei GET-Requests kann man diesen Parameter sehr deutlich im URL erkennen. In unserem Shop ist der URL, den man für die Navigation auf die Detailseite verwendet z. B. http://localhost:8080/ShirtApp/faces/pages/show-details.xhtml?javax.faces.Token=1366139142819. Der eigentliche Token-Wert ist nicht deterministisch. Somit kann ein CSRF-Angriff diesen nicht vorausberechnen und unberechtigterweise die Detailseite aufrufen (und ggf. Aktionen darin vornehmen).

Fazit

In diesem zweiten Teil des JSF 2.2 Tutorials haben wir weitere sehr hilfreiche und nützliche neue Features kennen gelernt. Faces Flows sind ein mächtiges Instrument. Sehr gut lassen sich diese mit Resource Library Contracts kombinieren, und man erhält eine sehr modulare und flexible Webanwendung. Das Thema Security wurde endlich auch in der Spec ein wenig mehr gewürdigt, und der CSRF-Schutz bringt einiges an Mehrwert. Es gibt jedoch noch zahlreiche weitere Features, auf die wir im Artikel nicht eingegangen sind, u. a. Ajax Delay, Composites und Non-Composites in einer *Taglib*, CDI-basierter *ViewScope* etc. Dies bleibt Ihrer Forscherfreude vorbehalten.

Alle Feature-Requests konnten natürlich auch mit dieser neuen Version von JSF 2.2 nicht berücksichtigt werden. Aber die Planungen für JSF 2.3 sind bereits im Gange. Es ist somit gut zu wissen, dass JSF nicht stehenbleibt und kontinuierlich weiterentwickelt wird.

Andy Bosch (andy.bosch@jsf-academy.com) ist Trainer und Berater im Umfeld von JSF und Portlets. Auf seiner Onlinetrainingsakademie www.jsf-academy.com stellt er regelmäßig Trainingsvideos zu JSF, CDI und Portlets bereit. Er ist Autor mehrerer Bücher zu JSF und hält regelmäßig Vorträge auf nationalen und internationalen Konferenzen.

Links & Literatur

[1] http://en.wikipedia.org/wiki/Cross-site_request_forgery

[2] https://www.owasp.org/index.php/Main_Page

[3] https://maven.java.net/content/groups/public/org/glassfish/javax.faces/2.2.0-SNAPSHOT/

[4] http://weblogs.java.net/blog/edburns/archive/2011/09/26/try-out-mojarra-220-snapshot

[5] http://www.jsf-academy.com/ShirtApp.zip, Demoanwendung zum Download

Mit ADF Task Flows Abläufe steuern

Alles fließt

Wie behält man den Überblick über den Ablauf einer Webapplikation? Mit ADF Task Flows steht ein leistungsfähiges Werkzeug zur Steuerung von Webseiten zur Verfügung. Dieses bietet darüber hinaus noch weitere nützliche Merkmale, wie z. B. Transaktionsmanagement und Exception Handling.

von Martin Künkele

Wenn es die ADF Task Flows nicht bereits gäbe, müsste man sie erfinden. Mit JSF Faces Flows bekommt JSF 2.2 eine Ablaufsteuerung, die sehr von ADF Task Flows bzw. Spring Web Flow beeinflusst ist ([1], Kap. 7.5).

Eine Webanwendung besteht aus Webseiten, die in einer durch eine Beschreibung festgelegten Reihenfolge aufgerufen werden sollen. Vor und nach dem Aufruf einer Seite müssen möglicherweise Prüfungen durchgeführt und/oder Daten gelesen, aufbereitet oder auch geschrieben werden. Daraus leitet sich die Anforderung ab, auch nicht visuelle Komponenten in den Ablauf einbetten zu können.

Ab dem Zeitpunkt aber, ab dem sich z. B. durch Benutzereingaben Daten verändern, ergibt sich die Frage, was passiert, wenn der Ablauf abbricht, also die Anforderung nach einem Transaktions-Handling. Ist es möglich, dass der Ablauf unter eine Transaktionskontrolle gestellt werden kann, mit dem Ziel des Alles-oder-Nichts? Also so, dass nach einem Abbruch der Zustand von vor dem Beginn des Ablaufs wiederhergestellt wird und erst, wenn er vollständig durchlaufen werden konnte, alles festgeschrieben wird? Wie steht es mit der Idee, einen einmal definierten Ablauf an verschiedenen Stellen in einer Webanwendung einsetzen zu können? Wie lässt sich dieser Ablauf dann noch beeinflussen, d. h. an leicht unterschiedliche Bedingungen anpassen? Fragen über Fragen. Mal sehen, ob und wenn ja, welche Antworten ADF Task Flows zu bieten hat.

Im Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development [2] wird ADF Task Flows ein eigener Abschnitt (Part IV) gewidmet. Außerhalb der sechs Kapitel dieses Abschnitts findet man an anderer Stelle des User Guides noch weitere Informationen zu ADF Task Flows, z. B. im Kapitel 34 über so genannte Contextual Events, auf die auch in diesem Artikel noch einzugehen sein wird.

JDeveloper

Wenn man mit ADF Task Flows arbeitet, kommt man am JDeveloper – Oracles IDE – nicht vorbei. Im JDeveloper ist es möglich, Task Flows grafisch zu erstellen. Man kann die einzelnen Komponenten aus der Component Palette auswählen und dann in den grafischen Editor ziehen. Ein so genannter einfacher Bounded Task Flow sieht dann wie in **Abbildung 1** aus.

Navigation

Ein Bounded Task Flow – BTF – hat genau einen Eintrittspunkt, in **Abbildung 1** eine View namens „Create", die direkt in einem Browser aufgerufen werden kann. Allgemeiner ist die View eine Activity, von der es mehrere Typen gibt (s. u.). Aktivitäten werden durch so genannte Control Flow Cases verknüpft. Das sind im Beispiel die Pfeile mit der Bezeichnung (das so genannte *from outcome*) *toCreate* bzw. *toDelete*.

Wenn zwei Activities durch ein Control Flow Case verknüpft werden, dann wird in die XML-Datei, in der die komplette Definition des BTFs gespeichert wird, eine Control Flow Rule eingefügt. Wichtig ist, dass die Case ein Bestandteil der Rule wird. Eine Regel kann mehrere Fälle enthalten. Nach einer priorisierten Prüfung ([2], Kap. 18.4), vergleichbar denen der JSF Navigation Rules, wird zur Laufzeit erkannt, welche Regel erfüllt ist, und entschieden, wie im Ablauf des Bounded Task

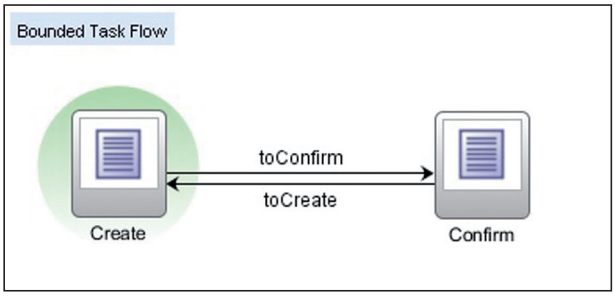


Abb. 1: Bounded Task Flow

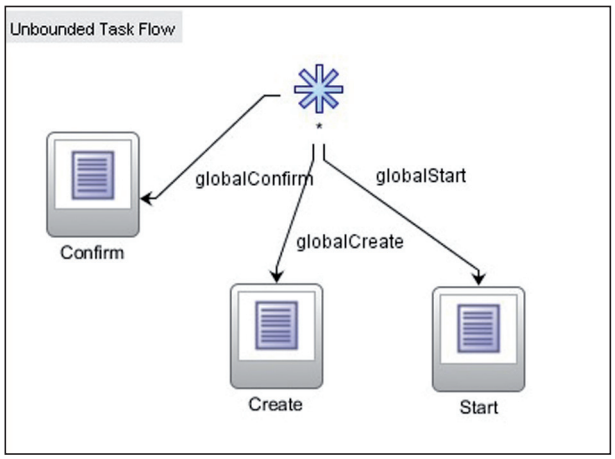


Abb. 2: Unbounded Task Flow

Flows weiter verfahren wird. Folgende Typen von Activities sind möglich:

- View: die Anzeige einer Seite oder eines Fragments – facelets
- Method Call: der Aufruf einer Methode, also einer nicht visuellen Komponente
- Task Flow: der Aufruf eines weiteren Bounded Task Flows, d. h. BTFs können weitere BTFs rufen
- Parent Action: die Rückkehr zum Aufrufer, z. B. zum rufenden BTF
- Router: eine Verzweigung

Alle Activities, Control Flow Rules und Aufrufe von Methoden in Managed Beans werden in einer XML-Datei gespeichert, deren Name frei gewählt werden kann.

Der zweite Typ von ADF Task Flows ist der Unbounded Task Flow oder UTF. Seine Definitionen werden in einer Datei *WEB-INF/adfc-config.xml* gespeichert. Ein UTF kann mehrere Eintrittspunkte in die Anwendung haben, wie in **Abbildung 2** gezeigt. Die Darstellung mit dem blauen Stern vermittelt, dass jede der Views ein Eintrittspunkt in die Anwendung sein kann. Es kann in einer Fusion-Webapplikation nur einen Unbounded Task Flow geben.

Die nachfolgend beschriebenen Konzepte sind meist nur mit Bounded Task Flows möglich.

Kapselung

Einen Bounded Task Flow kann man sich wie eine Methode vorstellen: Er hat eine eigene Bezeichnung, die durch das Attribut *id* des Tags *<Task-Flow-definition>* festgelegt ist. Diese *id* wird beim Anlegen auch für Namen der XML-Datei vergeben, in die die Definitionen des BTFs gespeichert werden. Es sind weitere Tags möglich, wie z. B. *<input-parameter-definition>* für die Definition eines oder mehrerer Eingabeparameter bzw. *<return-value-definition>* für die Definition eines Rückgabewerts.

Wiederverwendbarkeit

Ins Bild der Methode passt dann auch, dass ein BTF wiederverwendet werden kann. Das bedeutet, er kann von unterschiedlichen Stellen einer Webapplikation aus aufgerufen werden; und nicht nur innerhalb einer Webapplikation, sondern auch von einer anderen Webanwendung kann derselbe BTF aufgerufen werden.

Interessant ist die Wiederverwendbarkeit – Reuse – im Zusammenhang mit Regions. Diese werden durch das Tag

```
<af:Region value="#{bindings.TaskFlowdefinition1.RegionModel}" id="r1"/>
```

beschrieben und können so entweder in einer JSP-Seite oder einem Page Fragment definiert werden. Mithilfe des JDevelopers ist das Einbetten der Region sehr einfach: Man öffnet die Seite im Editor und zieht aus dem Application Navigator den Bounded Task Flow mit gedrückter linker Maustaste in die Seite. Nach der Auswahl „Region“ im Pop-up-Menü wird das oben beschriebene Tag und zusätzlich in das betreffende *PageDef*-File der Seite folgender Eintrag eingefügt:

```
<TaskFlow id="TaskFlowdefinition1" TaskFlowId="/WEB-INF/Task-Flow-
                definition.xml#Task-Flow-definition" activation="deferred"
                xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

Damit wird die Verbindung zwischen Task Flow und der Seite hergestellt. Alle Änderungen am Bounded Task Flow betreffen jetzt nur die Region, die anderen Komponenten auf der Seite bekommen davon nichts mit. Ein BTF als Bestandteil einer Region kann nur Page Fragments enthalten, keine kompletten Seiten.

Eine Erweiterung zu der eben gezeigten Art und Weise, wie eine Region in eine Seite eingebettet wird, ist die Verwendung einer Dynamic Region. Der kleine Unterschied ist, dass die *TaskFlowId* im Tag *<TaskFlow>* variabel ist und nicht fest „verdrahtet“ wie oben. Damit hat man zur Laufzeit die Möglichkeit, aus einer Reihe von Regions auf einer Seite eine Auswahl zu treffen – passend zu den Anforderungen aus der aktuellen Situation der Anwendung. Erreicht wird dies durch die Verwendung einer Managed Bean, in deren Attribut *TaskFlowId* der Name und die Location des Bounded Task Flows gespeichert werden. Die Flexibilität wird dadurch noch weiter gesteigert, dass man bei der Task-Flow-Definition im *pagedef*-File Parameter definieren kann. Unter [3] befindet sich ein Beispiel, das man sich herunterladen kann. Wichtig ist, dass der Scope der Managed Bean auf Page Flow definiert wird. Damit sind wir beim nächsten interessanten Aspekt von ADF Bounded Task Flow.

Lebenszyklus

Der Page Flow Scope garantiert, dass die Managed Bean so lange lebt wie der Bounded Task Flow. Der Page Flow Scope definiert einen Speicherbereich, der zum Austausch von Daten zwischen den einzelnen Activities innerhalb des BTFs verwendet wird. Wenn allerdings ein BTF einen anderen aufruft, dann kann er nicht auf dessen Page Flow Scope zugreifen, d. h. er sieht dessen Daten nicht. Was aber, wenn wir dem aufgerufenen BTF

Daten übermitteln müssen? ADF definiert verschiedene Scopes, die sich in der Lebensdauer unterscheiden:

- Application Scope: Bleibt während der gesamten Dauer der Anwendung bestehen, d. h. auf in ihm gespeicherte Werte kann während der gesamten Laufzeit der Applikation zugegriffen werden.
- Session Scope: Beginnt, wenn der Benutzer eine Seite der Anwendung aufruft, und endet entweder durch *timeout* oder die *session* wird *invalidated*.
- Page Flow Scope: Siehe oben.
- View Scope: Die Lebensdauer beginnt und endet, solange sich der Benutzer auf der Seite befindet. Wenn er sie verlässt, dann werden die Werte im Scope freigegeben und es kann nicht mehr auf sie zugegriffen werden.
- Request Scope: Die Lebensdauer ist nicht länger als der aktuelle Request.
- Backing Bean Scope: Der Vollständigkeit halber hier erwähnt.

Zurück zur Frage: Naheliegend ist der Session Scope, weil er länger andauert als der Page Flow Scope. In diesem Speicherbereich abgelegte Werte stehen also länger zur Verfügung, d. h. auf Werte im Session Scope kann auch der gerufene BTF zugreifen. Damit wird die Kommunikation zwischen Task Flows ermöglicht.

Eine andere Art der Kommunikation kann mit Contextual Events realisiert werden. Oft ergibt sich die Anforderung einer Kommunikation von der Parent Page zur Region. Liefert die Region ein Ergebnis, das für die Seite relevant ist, dann möchte die Region mit der Parent Page kommunizieren. Für diese beiden Fälle und den dritten, nämlich die Kommunikation zwischen Regions, sind Contextual Events ein probates Mittel, das immer funktioniert. Contextual Events folgen dem Producer-Consumer-Pattern, d. h. es gibt einen Producer (Publisher) der ein Event auslöst, und es gibt einen (oder mehrere) Consumer (Handler), der auf ein spezifisches Event lauscht. Für die Kommunikation interessant ist, dass einem Event eine Payload mitgegeben werden kann, und in diese Payload schreibt man eben die Daten, die man übermitteln will. In [4] ist beschrieben, wie man Contextual Events einsetzen kann.

Transaktionskontrolle

Eine der eingangs aufgeworfenen Fragen war, was passiert, wenn der Bounded Task Flow abbricht – sei es durch eine Fehlerkonstellation oder durch ein unvorhergesehenes Ereignis.

Um die Transaktionskontrolle von ADF Task Flows zu verstehen, muss zu ADF etwas weiter ausgeholt werden. Bislang haben wir uns nur bei dem View/Controller des MVC Patterns, das von ADF umgesetzt ist, aufgehalten. ADF kennt natürlich auch die Model-Seite, und dort liegen die so genannten Business Components, abgekürzt ADF BC. Die ADF BCs erleichtern den Zugriff auf die Daten, vor allem aber nicht nur in einer Datenbank. Dazu stellt ADF BC folgende Komponenten zur Verfügung:

- Entity Objects: Repräsentiert eine R*ow* in der Datenbank. Ein EO kann mit anderen Entity Objects verknüpft werden.
- View Objects: Repräsentiert eine SQL-Abfrage. Die VOs bilden eine sehr mächtige Schicht innerhalb der ADF BCs, erlauben sie doch den Zugriff auf Daten durch *joins*, *filters*, *sorts* und Aggregation. Ein Benutzer arbeitet mit den Daten, die ihm durch die VOs zur Verfügung gestellt werden. Die VOs arbeiten mit den EOs zusammen, um zu gewährleisten, dass die Daten nach Änderungen konsistent bleiben.
- Application Module: Repräsentiert das Data Model und bietet Service Methods nach außen an, die aus der View-/Controller-Schicht damit letztendlich vom Benutzer aufgerufen werden können und in einer Logical Unit of Work ablaufen. Hier kommt das Transaktionsverhalten ins Spiel. Eine Instanz eines Application Module entspricht einer Datenbanktransaktion, d. h. alle Änderungen, die in dieser Instanz durchgeführt werden, werden entweder gespeichert (*commit*) oder verworfen (*rollback*).

Mit Änderungen sind Updates auf die Datenbank gemeint. Wie kommen diese Updates in das Application Module? In den meisten Fällen dürften Änderungen auf Benutzeraktionen zurückzuführen sein, d. h. auf Operationen auf der View-/Controller-Seite. Welche Komponenten sind das, die es erlauben, aus der View-/Controller-Schicht Änderungen auf der Model-Seite zu bewirken?

Es sind die Data Controls – DC. Die sind bereits im JDeveloper verfügbar, wenn man auf der Model-Seite ein Application Module mit einem Data Model erstellt hat.

Einen großen Fundus an ADF-Mechanismen findet man, wenn man die Fusion-Order-Demosoftware von Oracle herunterlädt [5]. Die sich im Paket befindliche ADF-Anwendung „StoreFrontModule“ besteht aus zwei Projekten: eines für das Modell und das andere für den ViewController. Im Modell existieren zwei Application Modules – AM, die man in den Data Controls wiederfindet. Das AM *StoreServiceAMDataControl* ist aufgeklappt, und das DC für die Adressen ist selektiert (**Abb. 3**).

Um zur Frage zurückzukehren: Wie können wir jetzt dieses DC verwenden, um z. B. eine neue Adresse in die Datenbank zu schreiben oder Änderungen an einer bereits bestehenden Adresse durchzuführen?

Der JDeveloper unterstützt uns bei der Erstellung einer einfachen Seite, mit deren Hilfe wir eine Tabelle erzeugen können, in der die bereits bestehenden Adressen in der Datenbank aufgelistet werden. Man selektiert das DC „Adresse“ mit der rechten Maustaste, hält es fest und zieht es dann in die geöffnete JSP-Seite und lässt wieder los. Es erscheint ein Pop-up-Menü, aus dem wir auswählen können, dass wir eine „ADF Table“ erstellen wollen, die Änderungen zulässt.

Zur Laufzeit können wir eine Zeile der Tabelle selektieren und einzelne Felder ändern. Wie aber werden diese Änderungen gespeichert? Wir erinnern uns: Jede Instanz eines Application Module entspricht einer Datenbanktransaktion, und tatsächlich finden wir in den Data Controls die beiden Operationen *commit* und *rollback* wieder. Auch diese können wir durch Drag and Drop auf die Seite ziehen und als Buttons definieren.

Zurück zu den Bounded Task Flows, die per definitionem aus mehreren Activities bestehen können. Einzelnen oder auch allen Activities liegt ein DC zugrunde, wodurch während des Ablaufs des BTFs Updates auf Datenbanktabellen durchgeführt werden können. Es sind also die Data Controls, die aus der Sicht des Bounded Task Flows unter eine Transaktionskontrolle gestellt werden müssen, wenn ein konsistenter Datenbankzustand erhalten bleiben soll und wenn die Änderungen, die während des Ablaufs nur für den agierenden Benutzer sichtbar sind, nach einem Commit für alle Benutzer sichtbar werden sollen (**Abb. 4**).

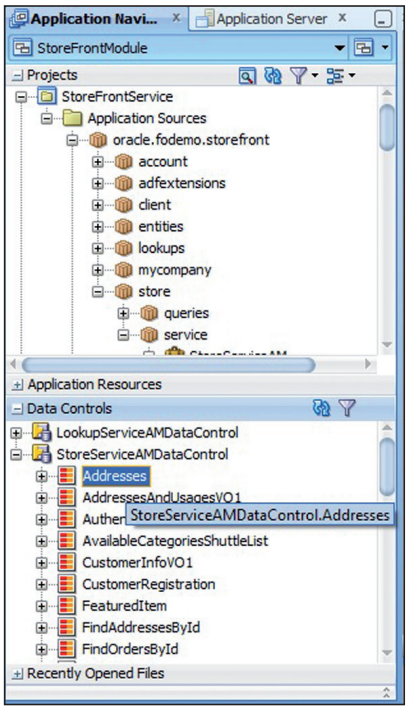


Abb. 3: Data Controls im JDeveloper

Die Data Controls und die darunter liegenden Business Components übernehmen das Transaktions-Handling, allerdings unter der Kontrolle des ADF Controllers, der zwei Stellschrauben hat, an denen gedreht werden kann. Die Einstellungen werden direkt am BTF auf einer Übersichtsseite im JDeveloper vorgenommen.

Bei den Einstellungen für die Data Controls geht es um das Verhalten, wenn ein BTF aufgerufen wird. Soll er eigene Instanzen der von ihm eingesetzten DCs verwenden oder die vom Aufrufer? Im ersten Fall darf im Feld *Share data controls with calling task flow* das Häkchen nicht gesetzt sein, und man spricht von Isolated Scope. Im zweiten Fall muss es gesetzt werden, und dann spricht man von einem Shared Scope.

Im Hintergrund wird ein Data Control Frame verwaltet, im Grunde eine Liste der verwendeten Data Controls, die zur Laufzeit für einen Unbounded Task Flow oder einen BTF mit Isolated Scope angelegt wird. Wenn allerdings ein BTF Shared Scope auswählt, dann verwendet er die Liste des Aufrufers. Mit *DataControlFrame* gibt es eine Klasse, die man programmtechnisch bearbeiten kann – vor allem im Sinne, dass alle DCs entweder „committet" oder zurückgerollt werden können. Für die grafische Darstellung gibt es eine *task-flow-return*-Komponente, deren *end-transaction*-Attribut entweder als Commit oder Rollback eingestellt werden kann. Wichtig ist, diese End Transactions nicht mit den oben angeführten *commit*- bzw. *rollback*-Operationen zu verwechseln. Diese wirken immer nur auf ein DC. Bei den Einstellungen für die Task Flow Transaction kann man zwischen folgenden Optionen wählen (**Abb. 5**):

- *<No Controller Transaction>*
- Always Begin New Transaction
- Always Use Existing Transaction
- Use Existing Transaction if Possible

Es würde den Rahmen sprengen, alle Kombinationen zu beschreiben. Theoretisch sind es bei zwei Bounded Task Flows 64 Möglichkeiten. Es gibt in [6] eine Beschreibung der wichtigsten Kombinationen und auch Codebeispiele, die man sich herunterladen kann, um mit den wichtigsten Kombinationen vertraut zu werden.

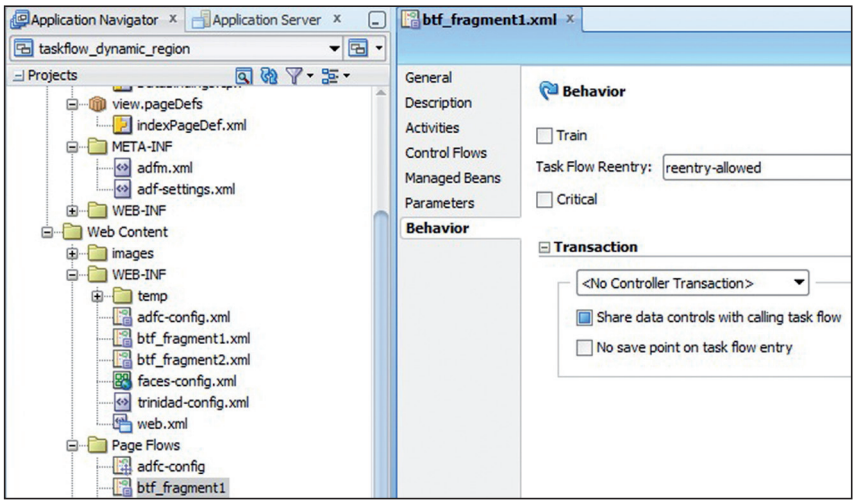


Abb. 4: Data Control Scope Behavior eines Bounded Task Flows

Exception Handling

Fehler können auftreten, das ist auch bei ADF Task Flows nicht anders. Schließlich können Programmteile, die in Java implementiert sind, aufgerufen werden, z. B. bei der Verwendung einer Method Call Activity. Eine Exception kann aber auch auftreten, wenn ein Benutzer eine Activity aufrufen möchte, für die er keine Rechte hat.

Sowohl bei Unbounded Task Flows als auch bei Bounded Task Flows kann eine Activity als Ex-

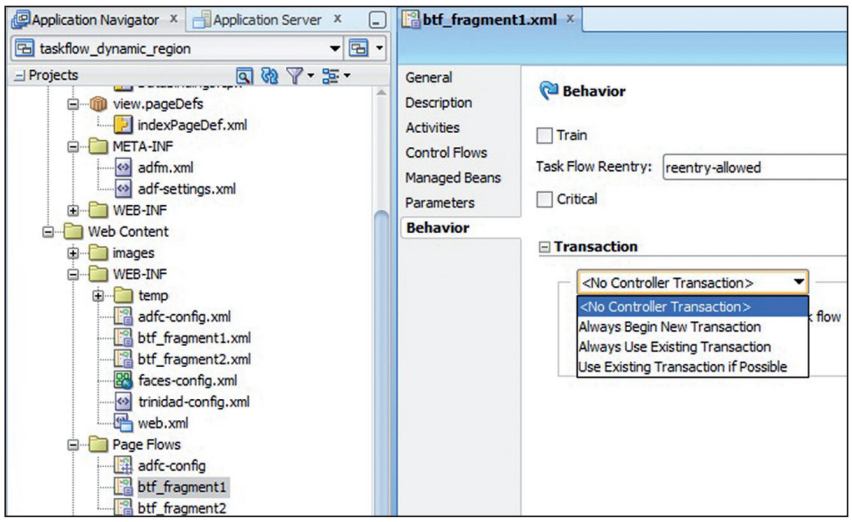


Abb. 5: Task Flow Transaction Behavior eines Bounded Task Flows

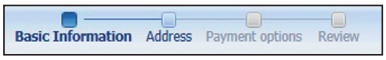


Abb. 6: TrainButtonBar mit Train Stops

ception Handler bestimmt werden. Wenn die Exception auftritt, wird mit dieser Aktivität fortgefahren.

Einem Task Flow kann nur ein Exception Handler zugeordnet werden. Allerdings kann er sich von dem des Aufrufers unterscheiden. Wir haben bereits die Regions kennengelernt und ihre Unabhängigkeit von den Gegebenheiten der Seite, in die sie eingebettet sind. So ist es auch im Hinblick auf das Exception Handling. Eine Region kann ihren eigenen Exception Handler zugeordnet bekommen und damit ein anderes Fehler-Handling als das der „beherbergenden" Seite.

Die Vorgehensweise zur Definition eines Exception Handlers sind in Kapitel 22.5 in [2] beschrieben. Es lohnt ein Blick in diese Beschreibung, am besten, bevor man mit der Umsetzung eines ADF-Projekts beginnt. Denn man findet dort wertvolle Hinweise, wie man z. B. verfährt, wenn eine Exception während einer Transaktion auftritt. Es werden auch Tipps gegeben, wie man durch Validierung vermeidet, dass die Exception Handling Activity aufgerufen werden muss.

Templates

Für ein einheitliches Exception Handling eignet sich ein Task Flow Template. Anstatt jedem BTF eine Exception Handling Activity zu spendieren, kann man diese in ein Template legen, das dann bei der Erzeugung des BTFs angezogen wird. Ausschließlich für Bounded Task Flows verwendbar, bieten die Templates eine Vereinfachung für immer wiederkehrende Abläufe und/oder die Gestaltung eines einheitlichen Verhaltens. Die Gestaltungs- und Einsatzmöglichkeiten sind vielfältig.

Es kann eingestellt werden, ob sich Änderungen am Template sofort auf die BTFs auswirken. Diese Einstellung kann später wieder zurückgenommen werden. Es können sich allerdings Konflikte zwischen den Definitionen des Parent Task Flow Templates und denen des Child Task Flows bzw. Child Task Flow Templates ergeben (ein Template kann auf ein anderes Template aufbauen), die vom ADF Controller aufgelöst werden müssen. Was passiert z. B., wenn das Template eine Default Activity (Activity, die als Erstes ausgeführt wird) definiert, die sich von der des BTFs, der auf das Template aufbaut, unterscheidet? Laut Tabelle in [1] Kapitel 22.9 gewinnt dann die Definition des BTFs.

Train

Bei Prozessabläufen findet man immer häufiger Trains, wie sie in **Abbildung 6** dargestellt sind. Die Einstellung, dass er die *train*-Komponente verwenden soll, muss bei der Erzeugung des BTFs oder eines Templates festgelegt werden. Erst danach können die Activities, die im Train auftauchen sollen, per Drag and Drop in den BTF gezogen werden.

Es kann für einen Bounded Task Flow immer nur eine *train*-Komponente definiert werden. Die Definition ist im Designer des JDeveloper – also grafisch – möglich. Es können nur Views – also interaktive Komponenten – und Aufrufe von Child Bounded Task Flow als Train Stops definiert werden. Obwohl das Kontextmenü für eine Methode die Auswahl anbietet, wird das Tag *<train-stop/>* nicht in die Definition für die Methode in der XML-Datei des BTFs eingefügt.

Im einfachen Fall korrespondieren die Train Stops mit View, also visuellen Komponenten. Es besteht aber auch die Möglichkeit, mehrere Activities zu gruppieren und einem Train Stop zuzuordnen. Das Weiterschalten in der Gruppe geschieht dann mit den eingangs erläuterten Mechanismen, also z. B. durch Drücken eines Buttons, der eine Action auslöst und damit im Bounded Task Flow zur nächsten Activity weiterschaltet.

Die Train Stops müssen nicht unbedingt sequenziell durchlaufen werden. Diese Standardeinstellung kann für jeden Train Stop abgeschaltet werden, sodass er nicht mehr dann erst aufgerufen werden kann, nachdem man seinen Vorgänger oder seinen Nachfolger besucht hat. Schließlich besteht noch die Möglichkeit, Train Stops zu überspringen. Alle diese Einstellungen sind auch mittels EL Expressions möglich.

Sicherheit

ADF bietet mit ADF Authentication and Authorization ein umfassendes Sicherheitsmodell, dessen Benutzung empfohlen wird. JDeveloper bietet einen Wizard an, mit dessen Hilfe die Sicherheitseinstellungen vorgenommen werden können. Es lässt sich sehr feingranular einstellen, wer auf einen Bounded Task Flow zugreifen darf. **Abbildung 7** zeigt, dass z. B. auf den BTF *checkout-taskflow* nur Mitglieder der Gruppe *FOD Users* zugreifen dürfen. Weitere Informationen findet man in [2], Kapitel 35.

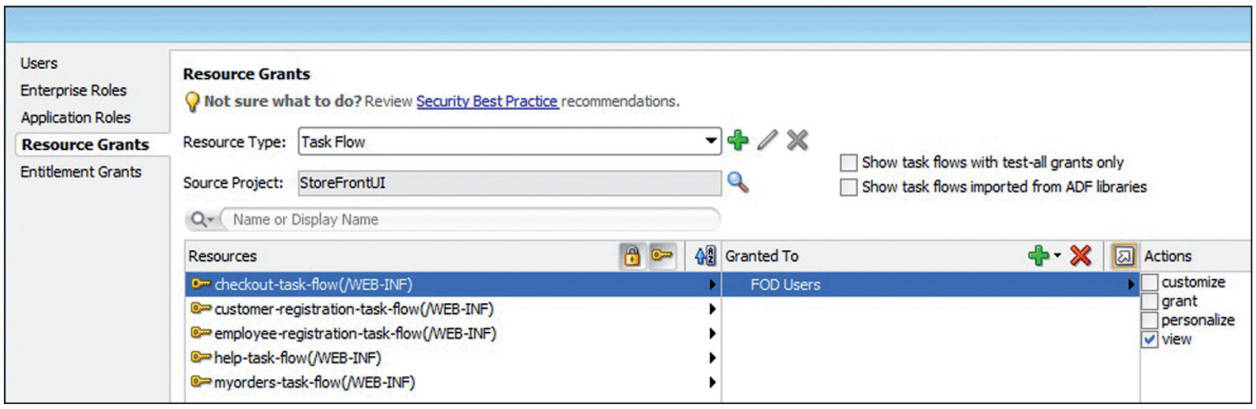


Abb. 7: Rechte auf Ressourcen

Aktuelles und Nützliches

Im Oracle Magazine Ausgabe März/April 2013 [7] werden Hinweise für das Error Handling in Unbounded bzw. Bounded Task Flows gegeben. Eine weitere Quelle, wie man es nicht machen soll bzw. besser machen kann, ist [8].

Zusammenfassung

Eingangs wurde ein Bezug zu JSF Faces Flows bzw. Spring Web Flow hergestellt. Spring Web Flow gibt es bereits seit 2006, und es hat einen Umfang, der in [9] bzw. [10] dokumentiert ist.

Faces Flow ist laut [11] eines der drei Big Ticket Features des JSR-344, der Ende März 2013 final werden soll. In [12] wird etwas detaillierter auf JSF Faces Flow eingegangen, und es sind Merkmale erkennbar, die es bei ADF Task Flows bereits gibt: Es gibt einen Scope, der durch eine Annotation *@FlowScoped* in einer Backing Bean definiert wird, die dann statt einer View einen ganzen Flow unterstützt. Diese Annotation hat einen Parameter *id*, der eine Bezeichnung des Flows speichert. Diese Bezeichnung kann dann verwendet werden, in dem JSF *meta-data facility* einer Flow Defining View um den Flow zu initialisieren.

Um einen Flow zu starten, kann man einen *commandLink* verwenden, dessen Action ebenfalls diese ID verwendet. In der bekannten Datei *faces-config.xml* können Definitionen eingetragen werden, die in ihrer Wirkungsweise den Annotationen entsprechen. Damit gibt es einen zentralen Platz, wo die Definitionen für den Faces Flow gespeichert werden. Das dient der Übersichtlichkeit. Es können fast die gleichen Aktivitäten aufgerufen werden wie bei ADF Task Flows: Views, Switches, Returns, Method Calls und andere Faces Flows.

Die Zeit wird zeigen, ob JSF Faces Flow ähnlich mächtig wird wie es ADF Task Flows bereits ist, z. B. durch eine grafische Unterstützung oder ein Transaktions-Handling.

Die programmtechnische Behandlung von ADF Task Flows musste hier ausgespart werden. Für die Version 11.1.2 wurde angekündigt, dass mittels *oracle.adf.controller.metadata.MetaDataService* auf die Metadaten des Task Flows zugegriffen werden kann. Wir werden auf dieses Thema zurückkommen.

Martin Künkele ist Inhaber der Firma SMK Software Management Kommunikation GmbH. Mit seiner Firma entwickelt er Webapplikationen auf der Basis von JDeveloper/ADF. Er ist zertifizierter Projektmanager nach P.M.I. und Certified Scrum Master der Scrum Alliance. Er beschäftigt sich mit der Frage, wie man ADF-Projekte agil durchführen kann und welche Aspekte aus Sicht des Projektmanagements zusätzlich zu berücksichtigen sind, um ein Projekt erfolgreich abzuschließen. Sein Blog befindet sich auf http://martinkuenkele.blogspot.de.

Links & Literatur

[1] JavaServer Faces Specification Version 2.2 Ed Burns Editor: http://download.oracle.com/otndocs/jcp/jsf-2_2-pfd-spec/

[2] Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g Release 2 (11.1.2.3.0)

[3] http://martinkuenkele.blogspot.de

[4] Implement Contextual Events By Frank Nimphius, in Oracle Magazine May/June 2011: http://www.oracle.com/technetwork/issue-archive/2011/11-may/o31adf-352561.html

[5] http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html

[6] Oracle ADF Task Flow Transaction Fundamentals by Chris Muir, 07/AUG/2012: http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adf-Task-Flow-trans-fund-v1-0-1723395.pdf

[7] Nimphius, Frank: „Catch Me If You Can", in Oracle Magazine March/April 2013: http://www.oracle.com/technetwork/issue-archive/2013/13-mar/o23adf-1897193.html. Abschnitt „UnBounded Task Flow: Oracle ADF Controller Error Handling" bzw. „Bounded Task Flow: Oracle ADF Controller Error Handling"

[8] http://community.oraclepressbooks.com/downloads/S316856%20-adf-api-mistakes.pdf

[9] http://www.springsource.org/spring-web-Flow

[10] http://static.springsource.org/spring-webFlow/docs/2.3.x/reference/htmlsingle/spring-webFlow-reference.html

[11] http://java.net/projects/javaserverfaces-spec-public/lists/jsr344-experts/archive/2012-05/message/59

[12] http://jdevelopment.nl/jsf-22/

Mit Puppet und RPM

Schneller in die Produktion!

„It works on my machine“ hat sicher jeder schon einmal gehört. In der Entwicklung verhält sich die Software wie erwartet. Bis diese in der Produktion ist, dauert es lange. Einmal live, treten dann unerwartete Fehler auf, während die Softwareentwicklung längst an einer ganz anderen Stelle ist. Wie kann dieser Spagat umgangen werden? Dieser Artikel reflektiert, wie wir mit diesem Problem auf Basis einer paketorientierten Deployment Pipeline umgehen.

von André von Deetzen und Oliver Wehrens

Im Juli 2012 standen wir vor der Herausforderung, ein neues Produkt zu entwickeln. Da die Entwicklung nach einem agilen iterativen Vorgehensmodell stattfindet, wollten wir das Produkt auch nach Abschluss einer oder weniger User-Storys regelmäßig unseren Kunden zur Verfügung stellen können. Unsere Plattform ist zwar bereits modularisiert, wird jedoch nur im Verbund getestet und ausgerollt. Die Auslieferungszeit von vier bis sechs Wochen war uns jedoch nicht schnell genug. Daher mussten wir das Verfahren für unser Produkt anpassen. Eine weitere Herausforderung war der gleichzeitige Umbau auf ein geändertes Paradigma in unserer Softwarearchitektur. Wir wollen unsere Architektur auf einen Micro-Service-basierten Ansatz umstellen. Dazu werden Funktionalitäten in fachliche Säulen aufgeteilt. Jeder dieser Säulen wird in der Produktion ein Servertyp zugewiesen, sodass auf jedem Server genau ein Service für die Plattform bereitgestellt wird. Um bei Bedarf diese Micro Services zu aktualisieren und einen gefundenen Fehler zu korrigieren, benötigen wir auch für jeden Dienst eine eigene unabhängige Deployment Pipeline. Da einer unserer Architekturgrundsätze die Pflege von abwärtskompatiblen Schnittstellen ist, können wir unsere Dienste jederzeit aktualisieren.

Softwarestack

Unser Produkt basiert auf einer in Java geschriebenen Webanwendung. Als Frameworks nutzen wir Spring [1] und Jersey [2]. Da wir jedoch auch Produkte haben, die auf Grails-Basis entwickelt werden, sollen die Komponenten unseres Ansatzes jedoch untereinander kompatibel sein. Betrieben werden die Applikationen unter Linux in einem Apache Tomcat 7. Da wir in der Produktion und in der Entwicklung jeweils die gleiche Linux-Distribution einsetzen und diese auch nicht so schnell ändern, haben wir uns dazu entschlossen, die spezifischen Aspekte unserer Linux-Distribution zu nutzen. Wir nutzen das gleiche Paketformat und halten uns an die Konventionen, wie z. B. *Init*-Skripte für den Start unserer Anwendung. Aus Sicht der Linux-Distribution sollen sich unsere Anwendungen genauso einfügen wie z. B. ein SSH- oder HTTPD-Dienst. Durch diese Entscheidung können wir auf alle bereits implementierten Funktionen unseres Konfigurationsmanagementwerkzeugs zurückgreifen. Es besteht keine Notwendigkeit, eine angepasste Implementierung für die Installation oder die Verwaltung von Diensten bereitzustellen.

Deployment Pipeline

Im Vorfeld haben wir uns überlegt, welche Stages wir für die Auslieferung unserer Software benötigen, sowie welche Softwareversion in welche Stage installiert wird. Wir entschieden uns für ein Konzept mit drei Stages: eine Dev Stage, eine Testing Stage sowie eine Stable Stage. In der Dev Stage wird der Softwarestand der aktuellsten Entwicklerversion, in der Testing Stage der Release Candidate und in der Stable Stage das Certified Release ausgerollt. Die Software darf je eine Stage weiter gelangen, wenn alle Qualitätsansprüche, die durch automatisierte Tests abgesichert sind, erfolgreich erfüllt wurden. Dazu haben wir eine Testpyramide, die aufzeigt, welche Tests in welcher Stage ausgeführt werden sollen (**Abb. 1**).

Durch unseren vorher definierten Prozess können wir unsere Deployment Pipeline an diesen Vorgaben ausrichten. Als Paketformat und Transporttechnologie

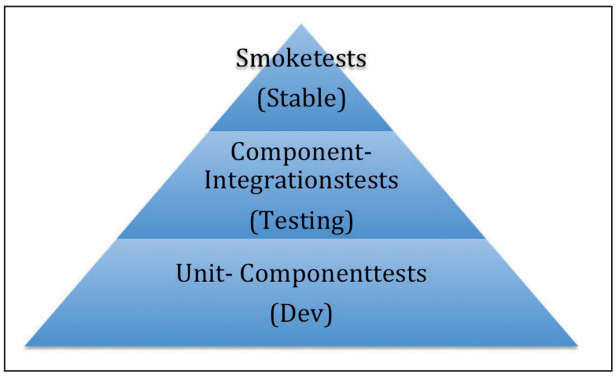


Abb. 1: Testpyramide

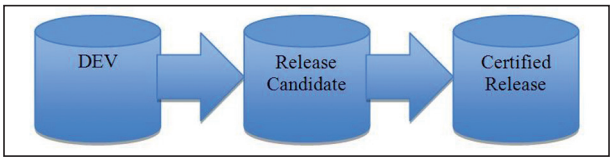


Abb. 2: Deployment Pipeline

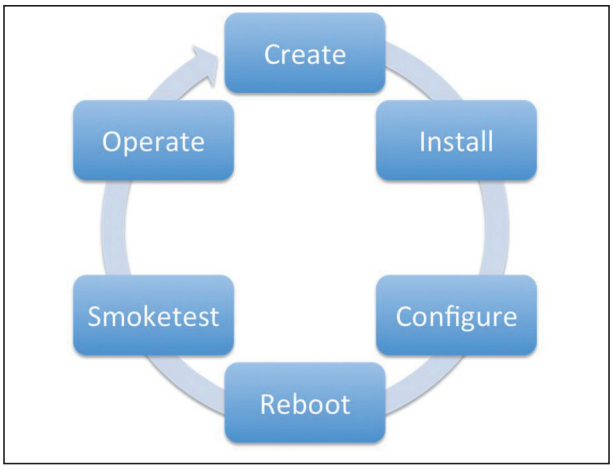


Abb. 3: Lifecycle

für unsere Software haben wir uns für RPM-Pakete und YUM Repositories [3] entschieden, da unsere Systeme in der Produktion sowie Entwicklungsabteilung RPM-basierte Linux-Systeme sind. Für jede unserer Stages haben wir ein eigenes YUM Repository, aus dem die Systeme installiert werden. Damit die Software auf dem Weg in die Produktion nicht mehr verändert wird, ist es ein Grundsatz unserer Deployment Pipeline, dass nur Binärartefakte durch das Staging laufen. Hat die Software den Versionsstand Certified Release erreicht, kann sie in der Produktion installiert werden. Dazu wird das Certified Release Repository in die Produktion gespiegelt. Damit dieses Modell erfolgreich ist, benötigen wir zwischen der Entwicklungsabteilung und der Produktion drei definierte Schnittstellen:

1. Ein YUM Repository pro Komponente mit einem definierten Endpunkt
2. Ein einheitliches Puppet für die Konfiguration
3. Die gleiche Betriebssystemversion in der Produktion sowie in den Entwicklungsumgebungen

Wie sieht der Arbeitsablauf für einen Softwareentwickler aus, der diese Deployment Pipeline nutzt? Nach dem Einchecken in das Versionskontrollverwaltungssystem werden auf dem Continuous-Integration-Server alle Tests ausgeführt und die Software paketiert. Von diesem Zeitpunkt an wird das Paket nicht mehr verändert.

Als nächster Schritt wird die Development Stage mit der passenden Konfiguration automatisiert neu aufgesetzt und das erzeugte Paket installiert. Gegen dieses System laufen die Komponenten- und Integrationstests. Sind diese erfolgreich, so wird das gesamte YUM Repository in die Testing Stage kopiert und die Software hat den Status eines Release Candidate.

Nach dem Aufsetzen der Software in der Testing Stage werden Smoke Tests ausgeführt. Unser Ziel ist es, dass diese Smoke Tests ausführbare Artefakte sind, die auf der Testing Stage und in der Produktion genutzt werden können. Sind diese ebenfalls erfolgreich, so wird das YUM Repository in die Stable Stage überführt und erhält den Status Certified Release. Jetzt kann ein Deployment-Ticket erstellt und die Software in der Produktion ausgerollt werden (**Abb. 2**).

Wie gehen wir mit lokalen Laufzeitabhängigkeiten um, wie z. B. dem JDK oder dem Tomcat? Hier haben wir uns für den Ansatz eines Self Containing Repositorys entschieden. Außer den Betriebssystempaketen enthält jedes unserer Komponenten-Repositorys sämtliche Pakete, die unsere Server zum Betreiben des Diensts benötigen. Somit hat jedes Team die Kontrolle über die Laufzeitabhängigkeiten wie z. B. die Version des eingesetzten JDKs oder des Tomcats. Die Konfiguration unserer Systeme erfolgt mithilfe von Puppet [4]. Puppet ist ein Konfigurationsmanagementtool und hilft uns, in den verschiedenen Stages unterschiedliche betriebliche Konfigurationen zuzusteuern, wie z. B. Datenbankpasswörter oder Endpunkte benötigter Dienste.

Nachdem der Transportweg für die Softwarekomponenten geklärt war und die Konfiguration der Systeme in der Hand der Teams lag, haben wir uns ebenfalls mit der Bereitstellung der Testsysteme beschäftigt. Wir möchten zu jedem beliebigen Zeitpunkt jeden Server wiederherstellen können. Deshalb haben wir die Installation unserer Systeme vollständig automatisiert. Wir benutzen dafür die von unserem Serverbetriebssystem bereitgestellten Technologien Kickstart [5] und RPM. Kickstart ist eine Software, die das komplett automatisierte Bootstrapping eines Systems durchführt. Nachdem die Installation erfolgreich war, übernimmt Puppet und bringt den Server in den Zielzustand, indem es die Repositories für eine Komponente konfiguriert, die Software installiert und konfiguriert. Nach der Puppet-Ausführung erfolgen ein Reboot und die Ausführung der Smoke Tests [6]. Smoke Tests überprüfen das neu erzeugte System und prüfen die wesentlichsten Geschäftsfälle der Komponente ab. Danach wird das neu erzeugte System dem Gesamtsystem zur Verfügung gestellt, in dem es in den *Load Balancer* eingehängt wird (**Abb. 3**).

Alles, was wir bis zu diesem Zeitpunkt betrachtet haben, findet nach dem Commit eines Entwicklers statt. Für unsere Java-Projekte setzen wir Maven als Build-Managementtool ein, d. h. der Import in die lokale Entwicklungsumgebung, Kompilieren und Testen von Projekten sind jedem Java-Entwickler bekannt. Doch die Paketierung von RPM-Paketen oder das Testen von Puppet-Änderungen werden nicht durch alle von uns genutzten Desktopbetriebssysteme wie Windows, OS X und auch nicht durch jede Linux-Distribution unterstützt. Bei der lokalen Verifizierung von Konfigurationen soll vermieden werden, dass der eigene Arbeitsplatzrechner versehentlich in den Zustand eines Webservers oder einer Datenbank überführt wird. Auch hier setzen wir auf Virtualisierung. In einer lokalen virtuellen Maschine kann sich jeder Entwickler die gleiche Linux-Version wie in der Produktion installieren, die Paketierung sowie unsere Puppet-Skripte testen oder weiterentwickeln. Damit nicht alle Entwickler ihre eigene Maschine installieren und sich um Updates und Konfiguration kümmern müssen, nutzen wir Vagrant [7]. Damit können beliebige, und damit auch produktionsnahe Umgebungen jederzeit reproduzierbar auf dem eigenen Arbeitsrechner aufgesetzt werden. Dabei wird VirtualBox von Oracle [8] als Virtualisierungsprodukt verwendet und durch eine Abstraktion, die Vagrant-Konfiguration, in einer DSL beschrieben und die Erzeugung automatisiert. Dabei setzt es auf eine Basebox, die das Basisimage für eine solche produktionsnahe Umgebung enthält. Die Erstellung dieses Basisimage kann mit dem Veewee-Projekt [9] mithilfe einer Kickstart-Installation automatisiert werden.

Ausblick

Wie am Anfang des Artikels erwähnt, reflektiert dieser Artikel die Entscheidungen und Erfahrungen, die uns im letzten dreiviertel Jahr während der Produktentwicklung beschäftigt haben. Themen, die wir aktuell lösen, sind das Datenbank-Deployment und separate Auslieferungsstränge für unsere Puppet-Module sowie die Nutzung von Hiera [10], um die Konfiguration noch besser steuern zu können. Das Thema Paket- und Repository-Signierung ist auch ein Sicherheitsaspekt, den wir bei Bedarf noch näher betrachten werden.

Fazit

Wir haben es mit unserem Vorgehen geschafft, den Releaseprozess pro Komponente bis zu einem Certified Release auf etwa dreißig Minuten zu reduzieren. Unserer Implementierung werden wir die fehlenden Funktionen zügig hinzufügen können und stetig in einem iterativen Verfahren für Verbesserung sorgen. Durch die möglichst offene und gemeinsame Gestaltung des Prozesses, mit der Entwicklung und dem Betrieb, kann ein Entwicklungsteam bei Erweiterungen oder Problemen unterstützen. Dass sich Softwareentwicklungsteams mit der lokalen Erstellung von Paketen beschäftigen, ist eine Hürde, da diese Technologie nicht unter jedem Betriebssystem von Hause aus zur Verfügung steht. Dennoch haben wir es geschafft, den kompletten Zyklus lokal testbar zu gestalten. Die Wahl von Self Containing Repositorys bringt zwar eine große Flexibilität für die Teams, bedeutet aber auch, dass redundante Pakete vorgehalten werden. Die automatisierte Wiederherstellbarkeit von Systemen ermöglicht für diese Komponenten ein Disaster-Recovery-Szenario [11] und ist damit essenziell für die Sicherstellung der Business Continuity [12]. Die Entscheidung, seinen Auslieferungsprozess in dieser Form zu gestalten, ist nicht nur eine technische, sondern vor allem eine organisatorische. Ohne einen Vertrauensvorschuss vom Management ist die Etablierung solcher Arbeitsweisen kaum möglich. Eine Self-Service-Infrastruktur und Konventionen sind für die Automatisierung eines solchen Vorgehensmodells zwingend erforderlich. Sie dürfen jedoch auch nicht zu sehr einschränken. Den schmalen Grat zwischen zu vielen Infrastrukturaufgaben und einer zu hohen Abstraktion zu begehen, ist schwer, unserer Erfahrung nach jedoch den Aufwand wert.

Oliver Wehrens ist Principal Engineer bei der Deutschen Post E-Post Development GmbH in Berlin. Er beschäftigt sich mit Enterprise-Integration, Entwicklungsprozessen und auch gerne (und leider zu wenig) mit Usability von Webseiten.

@owehrens

André von Deetzen ist Principal Engineer bei der Deutschen Post E-Post Development GmbH in Berlin.

@vdandre

Links & Literatur

[1] http://www.springsource.org

[2] http://jersey.java.net

[3] http://yum.baseurl.org

[4] https://puppetlabs.com

[5] https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/ch-kickstart2.html

[6] http://blog.m.artins.net/deployment-smoke-tests-is-anyone-being-slack/

[7] http://www.vagrantup.com

[8] https://www.virtualbox.org

[9] https://github.com/jedi4ever/veewee

[10] http://projects.puppetlabs.com/projects/hiera

[11] http://de.wikipedia.org/wiki/Disaster_Recovery

[12] http://de.wikipedia.org/wiki/Business_Continuity

Anzeige

Java Embedded auf dem BeagleBone

Ran an den Knochen!

Mitte 2008 wurde unter großer Medienresonanz die kostengünstige und lüfterlose Entwicklerplatine namens BeagleBoard, Vorgänger des BeagleBone, vorgestellt. Ein kompaktes Format, niedriger Stromverbrauch und geringe Wärmeentwicklung zeichnen dieses Entwicklungsboard für den Embedded-Bereich aus. Ein Erfahrungsbericht über Java-Entwicklung auf dem BeagleBone.

von Matthias Wenzl und Sigrid Schefer-Wenzl

Das BeagleBoard ist ein Open-Hardware-Projekt, das von einigen Texas-Instruments-Mitarbeitern ins Leben gerufen wurde, um einer wachsenden Embedded-Entwicklungsfangemeinde kostengünstige Entwicklungsboards zur Verfügung zu stellen [1]. Das neueste und in diesem Artikel näher beleuchtete Mitglied der BeagleBoard-Familie ist der BeagleBone [2], der derzeit für ca. 89 US-Dollar vertrieben wird. Die erste Version wurde im November 2011 vorgestellt. Die neueste Fassung trägt die Revisionsnummer A6 und wurde im Mai 2012 veröffentlicht.

Der BeagleBone ist eine vielfältig einsetzbare und erweiterbare Variante des BeagleBoards von der Größe einer Kreditkarte (**Abb. 1**). Die Anwendungsmöglichkeiten des BeagleBone erstrecken sich von rechenleistungsintensiven Szenarien wie der Visualisierung von 3-D-Grafiken und der Anzeige von HD-Videos über Echtzeitanwendungen wie der Ansteuerung von 3-D-Druckern bis hin zu energieeffizient arbeitenden Robotersteuerungen und Webservern. Die Softwareentwicklung kann unter Ångström Embedded, Ubuntu, Android, Open Embedded, Windows Embedded, Symbian und anderen Betriebssystemen erfolgen.

Im Gegensatz zu seinen Geschwistermodellen verfolgt der BeagleBone bei der Realisierung von eigenen Projekten den Ansatz eines Baukastens. Der BeagleBone dient hierbei als Basis, die um eigens entwickelte Hardware (sog. *Capes* [3]) oder um Drittkomponenten erweitert werden kann (z. B. LCD Cape [4], Audio and DVI-D Cape [5] etc.).

An dieser Stelle bietet sich ein kurzer Vergleich mit dem wesentlich bekannteren Raspberry Pi [6] an. Aufgrund seiner Ausstattung eignet sich der Raspberry Pi primär für Desktop-Computing und Videoprojekte, während der BeagleBone für den Embedded-Bereich optimiert ist. Dies macht BeagleBone nicht zuletzt dank seiner Vielzahl an frei zugänglichen Peripheriegeräten deutlich (Tabelle 1).

Die Hardware und Software

Herzstück des BeagleBone ist ein System on Chip (SoC) von Texas Instruments (AM3359) [2]. Bei einem SoC handelt es sich allgemein um die Implementierung eines vollständigen Computersystems (z. B. inklusive Prozessor, internem Speicher, Grafikchip, USB-Controller, Ethernet-Controller ...) in einem einzigen Chip. Das BeagleBone SoC verfügt über einen superskalaren ARM-Cortex-A8-Prozessor, der unter anderem auch in Apples iPhone 4 verwendet wird [7]. Zusätzlich ausgestattet ist das SoC mit einem stromsparenden ARM-Cortex-M3-Prozessor sowie mit zwei kleineren RISC-(Reduced Instruction Set Computer-)CPU-Kernen für die Verarbeitung von Echtzeitaufgaben. Der Cortex-A8-Prozessor kann mit 720 MHz (Netzteilversorgung) oder mit 500 MHz (USB-Versorgung) betrieben werden. Diese beiden Prozessoren können über in Hardware implementierte Mailboxen und Spinlocks miteinander kommunizieren bzw. sich synchronisieren. Dies eröffnet eine Spielwiese, die einem gewöhnlich nur in High-End-Embedded-Entwicklunsgplattformen geboten wird. An Arbeitsspeicher stehen 256 MB DDR2 RAM (off-chip) zur Verfügung. Für Grafikberechnungen nutzt der BeagleBone eine PowerVR-SGX530-3-D-Grafikeinheit. Als Massenspeicher steht dem BeagleBone eine 4-GB-microSD-Karte

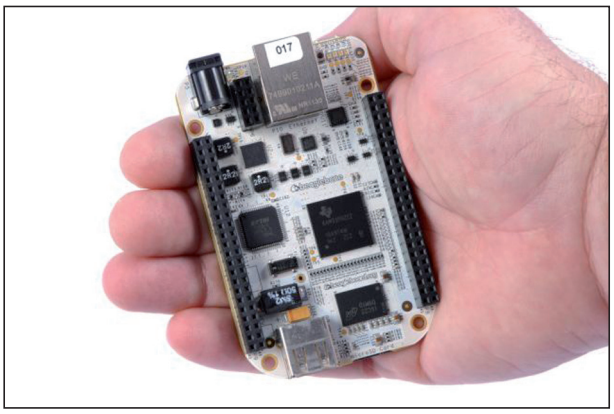


Abb. 1: BeagleBone: Kleiner Bone mit großer Leistung [2]

mit der vorinstallierten Ångström-Linux-Distribution zur Verfügung.

Die älteren Geschwister des BeagleBone werden zwar mit deutlich mehr vorgefertigten Anschlüssen ausgeliefert, der BeagleBone erlaubt jedoch aufgrund der Vielzahl an frei zugänglichen Ports eine höhere Einsatzbandbreite (siehe *Capes* [3]). Konkret besitzt das BeagleBoard einen OMAP3530 System on Chip (SoC) von Texas Instruments, der neben einem ARM-Cortex-A8-Prozessor auch einen Grafikprozessor von PowerVR Technologies enthält [8]. Es verfügt über 256 MB Arbeitsspeicher und 256-NAND-Flash-Speicher. Die Grafikausgabe des BeagleBoards erfolgt über HDMI oder S-Video. Des Weiteren besitzt das BeagleBoard Anschlüsse für USB, RS-232, SD-Karte, JTAG sowie Stereoein- und -ausgabe. Die Stromversorgung kann über USB oder über ein 5-V-Netzteil erfolgen. Durch den Anschluss weiterer Peripheriegeräte via USB kann das BeagleBoard an individuelle Bedürfnisse angepasst

	Raspberry Pi (Model B)	BeagleBone (Rev A6)
Preis	35 USD	89 USD
Prozessor	ARM11	ARM Cortex-A8, ARM Cortex-M3
Grafikprozessor	Broadcom VideoCore IV	PowerVR SGX530
Taktfrequenz	700 MHz	720 MHz
RAM	512 MB	256 MB
Massenspeicher	SD-Karte	4 GB microSD
Digital GPIO	8	66
Analoge Eingänge	0	7 zu je 12 bit
PWM	0	8
TWI/I²C	1	2
SPI	1	1
UART	1	5
Ethernet	10/100	10/100
USB Master	2 USB 2.0	1 USB 2.0
Videoausgang	HDMI, Composite	0
Audioausgang	HDMI, Analog	Analog

Tabelle 1: Vergleich zwischen Raspberry Pi und BeagleBone

werden. Als Weiterentwicklung wurde Ende 2010 das PandaBoard vorgestellt. Neben einem Zweikernprozessor mit 1 GHz (OMAP4430 SoC) und 1 GB RAM besitzt das PandaBoard drei Videoausgänge, LAN, WLAN und Bluetooth [9].

Viele Linux-Distributionen stellen eigene Portierungen für ARM-Prozessoren bereit, die mittlerweile auch den BeagleBone unterstützen. Eine ausführliche Liste samt Informationen und Links findet sich unter [10]. In diesem Artikel stellen wir unsere Erfahrung mit der mitgelieferten Linux-Distribution Ångström vor [11].

Von der Schachtel bis zur Kommandozeile

Gleich nach dem Auspacken kann man den BeagleBone über den Micro-USB-Stecker und das mitgelieferte Kabel an einen freien USB-Port am eigenen Computer anstecken. Der BeagleBone meldet sich als USB-Massenspeicher sowie mit zwei USB Serial Ports an. Der zweite der neu im System registrierten USB Serial Ports stellt anfangs eine Konsole für die Nutzerinteraktion zur Verfügung. Um eine konsolenbasierte Verbindung zu BeagleBone aufzubauen, konfiguriert man ein Terminalprogramm (z. B: PuTTY [12]) mit folgenden Parametern: Baud-Rate 115 200, 8 Datenbits, 1 Stopbit. Nach einer erfolgreichen Verbindung zum Board erscheint der Log-in Prompt. Der Benutzername ist „root“; es gibt kein Passwort.

Um den BeagleBone komfortabler bedienen zu können (und um Java zu installieren), erfolgt im nächsten Schritt die Konfiguration des Netzwerks. Wenn ein DHCP-Server im Netz zur Verfügung steht, muss man lediglich *dhclient eth0* am Shellprompt eingeben. Bei einer manuellen Konfiguration sind die folgenden Schritte notwendig:

```
root@beaglebone:~# ifconfig eth0 <freie IP Adresse> netmask
                                                  <Netzmaske> up
root@beaglebone:~# route add default gw <IP Adresse des Routers>
```

Abschließend müssen noch die DNS-Server in die Datei */etc/resolv.conf* eingetragen werden. Unter Ångström Linux steht hierbei der Editor „vi“ zur Verfügung. Nun kann man sich auch über SSH auf dem BeagleBone einloggen und über das Netzwerk weitere Komponenten, wie z. B. eine Java Virtual Machine, nachinstallieren.

Und Java?

Die Hardwareausstattung des BeagleBone erlaubt es, die ganze Bandbreite an JVMs für ARM-Prozessoren auszunutzen. So ist es einerseits möglich, die ressourcensparende und für den Einsatz in Embedded-Systemen optimierte, jedoch im Funktionsumfang limitierte Java Virtual Machine Micro Edition (ME) [13] zu installieren. Andererseits kann aber auch das JDK 7 Update 6 verwendet und somit der volle Funktionsumfang von Java genutzt werden. Dieser Artikel bezieht sich in weiterer Folge auf die Installation des Java Runtime Environments der OpenJDK 6 (siehe Kasten: „Cross Development“) [14], einer Open-Source-Implementierung der Oracle JVM mit nahezu gleicher Codebasis.

Um die JRE des OpenJDK auf dem BeagleBone zu installieren, muss der Paketmanager der Linux-Distribution bemüht werden. Im Fall von Ångström Linux heißt der Paketmanager *opkg*. Der Paketmanager verfügt über eine interne Datenbank, die den aktuellen Zustand (z. B. installiert, nicht installiert) sämtlicher für das System verfügbarer Softwarepakete widerspiegelt. Um die JRE des OpenJDK 6 zu installieren, werden die folgenden Schritte durchgeführt.

```
(1) root@beaglebone:~# opkg update
(2) root@beaglebone:~# opkg install openjdk-6-java
(3) ln –s /usr/lib/jvm/java-6-openjdk/jre/bin/java /usr/bin/java
```

Anweisung (1) bringt hierbei die interne Paketdatenbank auf den neuesten Stand, während Anweisung (2) das gewünschte Paket installiert. Abschließend muss das Java Binary noch in den Suchpfad des Systems eingetragen werden. Dies kann über das Setzen eines symbolischen Links, wie in Anweisung (3) beschrieben, passieren. Die

Cross Development

Da eine Vielzahl von Embedded-Entwicklungsboards, unter anderem auch der BeagleBone, über keine oder nur sehr eingeschränkte direkte Interaktionsmöglichkeiten (in Form eines Bildschirms, oder einer Tastatur) bzw. geringe Hardwareressourcen (z. B. schwacher Prozessor, wenig RAM) verfügen, hat sich „Cross Development“ als die primäre Art, Software unter Embedded-Systemen zu entwickeln, durchgesetzt. Cross Development beschreibt hierbei den Prozess, Software auf einem Entwicklungssystem (Host) zu entwickeln und auf einem Zielsystem (Target) zur Ausführung zu bringen. Der Sourcecode wird hierbei auf dem Entwicklungsrechner oder einem eigenen Build-Rechner kompiliert.
Im demonstrierten Beispiel wurde die Software auf einem PC (Host) unter Eclipse entwickelt und anschließend als kompilierte Class Files über eine SSH-Verbindung (z. B. PuTTY) auf das BeagleBoard kopiert. Eventuelle Debug Sessions wurden über die Remotefähigkeiten des Java-Debuggers durchgeführt. Eine Debug Session kann wie folgt gestartet werden:

```
ssh –t root@<Beagle_Bone's IP Adresse> ' java –Xdebug
    -Xrunjdwp:transport=dt_socket,server=y,address=<Portnummer>,
                                                  suspend=y Bone '
```

Die ssh-Option *–t* veranlasst hierbei die Ausführung des unter einfachen Hochkommata angegebenen Kommandos auf dem angegebenen Zielsystem. Die Option *suspend* bei der Parametrisierung des Java-Debuggers gibt an, dass der Debugger auf eine eingehende Verbindung warten soll. Somit ist am Target nur eine JRE anstatt einer kompletten JDK vonnöten.

BeagleBone-Netzwerkkonfiguration ist abgeschlossen, Java ist installiert. Dann mal ran an den Speck!

Blinking LEDs: Der Klassiker unter den ersten Gehversuchen mit neuer Hardware

Auch, wenn das Blinken-lassen von LEDs trivial erscheinen mag, so hat es eine lange Tradition in der Embedded-Softwareentwicklung. Der Grund dafür liegt auf der Hand: Wenn die Lampe leuchtet, funktioniert meine Software offensichtlich richtig. Ich habe also verstanden, wie mein System grundlegend funktioniert – so lautet die Grundannahme unzähliger Embedded-Entwickler, die sich mit neuer Hardware konfrontiert sehen. Um diese schöne Tradition aufrecht zu erhalten, präsentieren wir hier ebenfalls ein Beispielprogramm zum Blinken von LEDs auf dem BeagleBone. Genauer gesagt soll die Beispielanwendung ein Lauflicht, auch bekannt als Knight-Rider-Licht (wie die TV-Serie aus den 80ern), realisieren (Listing 1). Das Programm instanziiert hierbei drei unterschiedliche GPIO (General Purpose IO) Pins, die im Endeffekt als Ausgänge definiert werden und drei LEDs in der oben definierten Funktion ansteuern. Die importierte Klasse *BoneGPIO* im Package *BeagleBone* ist hierbei selbst implementiert und nicht Bestandteil des BeagleBone. Ihre Funktion wird jedoch im Folgenden erläutert.

Blinking LEDs: Die Details

Der BeagleBone verfügt – so wie viele andere Embedded-Entwicklungsboards – über GPIO Pins. Im Gegensatz zu Pins mit einer dezidierten Aufgabe (z. B. vertikale Synchronisation, horizontale Synchronisation eines VGA-Signals), die von einem Peripheriegerät bereitgestellt werden (z. B. VGA-Grafikmodul), ist es bei GPIO Pins dem Entwickler überlassen, entsprechende Funktionen zu programmieren. Die Idee hinter GPIO Pins ist die, eine komplett generische Schnittstelle für Prototyping oder Testzwecke (z. B. Anschluss von LEDs) zur Verfügung zu haben.

Das Lauflicht wird somit über GPIO Pins realisiert. In unserem Fall werden die Pins GPIO_0_26, GPIO_1_14 und GPIO_1_31 [2] verwendet. Diese Pins sind willkürlich gewählt, jedoch unter der Einschränkung, dass kein Pin bereits in anderer Funktion in Verwendung ist (siehe Kasten: „Ein Pin, viele Funktionen“). Die Bezeichner sind hierbei im Fall des BeagleBone wie folgt zu interpretieren: GPIO_X_YY steht für: GPIO Pin YY am Port X. Diese hersteller- und modellspezifische Gruppierung fasst GPIO Pins zu einer administrativen Einheit zusammen.

Um nun auf einzelne GPIO Pins zugreifen zu können, müssen die von Ångström Linux bereits mitgebrachten GPIO-Kerneltreiber verwendet werden. Dies

erfolgt über eine Reihe von Linux SysFs-Dateien (diese stellen eine textbasierte Schnittstelle zwischen Kernelobjekten und Userspace zur Verfügung, https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt) und kann unter Java z. B. über die Verwendung von *FileWriter()*- bzw. *FileReader()*-Objekten erfolgen. Die entsprechenden Dateien und Verzeichnisse sind in Tabelle 2 aufgelistet.

Um einen GPIO Pin anzusteuern, muss im ersten Schritt ein Link auf diesen erzeugt werden. Da jeder GPIO Pin im GPIO-Kerneltreiber einen eindeutigen Bezeichner besitzt, muss dieser in die Datei *export* (Tabelle 2) geschrieben werden. Der Bezeichner errechnet sich hierbei wie mit der Formel *Port x 32 + Pin = Bezeichner*. Für Pin GPIO_1_14 lautet der Bezeichner demnach *1 x 32 + 14 = 46*. Anschließend muss die Pin-Multiplexer-, sowie die Pin-Konfiguration durchgeführt werden (Listing 2). Hierbei ist noch zu beachten, dass der errechnete Konfigurationswert als Hexadezimalstring ohne führendes *0x* in die entsprechende Datei geschrieben werden muss. Die Werte der Klassenvariablen *myMuxMode*, *myMuxPinDir*, *myMuxPudn* und *myMuxPudnEn* liegen dem Schema in Tabelle 3 zugrunde und werden in der Implementierung über die entsprechenden *Set*-Methoden geschrieben (Listing 1). Abschließend muss noch der GPIO Pin selbst als Ausgang konfiguriert werden

Listing 1

```
import BeagleBone.BoneGPIO;
public class Bone {

  public static void main(String[] args) {

    BoneGPIO[] pins = new BoneGPIO[3];
    pins[0] = new BoneGPIO(0,26);
    pins[1] = new BoneGPIO(1,14);
    pins[2] = new BoneGPIO(1,31);

    for(int i = 0;i < 3;i++) {
      pins[i].setMuxMode(BoneGPIO.MODE_MSK_GPIO);
      pins[i].setMuxDir(BoneGPIO.PINDIR_MASK_OUT);
      pins[i].setMuxPudnEn(BoneGPIO.PUDN_MSK_DISABLE);
      pins[i].applyMuxSettings();
      pins[i].setPinDirection(BoneGPIO.pinOutput);/*als ausgang setzen*/

    }/*for*/

    /*5 lauflicht durchleaufe*/

    for(int i = 0; i< 5;i++) {

      for(int j = 0;j < 3;j++) {

        if(j == 0) {

          pins[2].setPinValue(BoneGPIO.pinLow);
          pins[j].setPinValue(BoneGPIO.pinHigh);

        } else {
          pins[(j-1)].setPinValue(BoneGPIO.pinLow);
          pins[j].setPinValue(BoneGPIO.pinHigh);
        }

        Thread.sleep(1000);
      }/*end inner for*/
    }/*end for;*/
    for(int i = 0; i < 3;i++) {
      pins[i].releasePin();
    }
  }/*end main*/
}/*end class*/
```

Verzeichnis	Dateiname	Zweck
/sys/class/gpio	export	Schnittstelle zum Anlegen eines neuen Links auf einen GPIO Pin
	unexport	Schnittstelle zum Löschen eines Links auf einen GPIO Pin
/sys/class/gpio<id>	direction	Bestimmt die Richtung eines GPIO Pins (Eingang oder Ausgang)
	value	Definiert den am Pin anliegenden binären Wert (0 oder 1)
/sys/kernel/debug/omap_mux	<Pin Mode 0 Bezeichner>[BoneSRM]	Pin Multiplexer und Konfigurationseinstellungen

Tabelle 2: Wichtige SysFs-Verzeichnisse und -Dateien

Bit(s)	Bedeutung
2–0	Mode0 (0b0) – Mode 7 (0b111); beschreibt, welches Peripheriegerät an den Pin angeschlossen wird
3	Aktiviert (0b1)/deaktiviert (0b0) den Pull-up- bzw. Pull-down-Widerstand
4	Konfiguriert den Pin mit Pull-up-Widerstand (0b1)/Pull-down-Widerstand (0b0)
6–5	Konfiguriert den Pin als Eingang (0b01) oder Ausgang (0b10)
15–7	Nicht von Bedeutung

Tabelle 3: Pin-Multiplexer-Konfiguration

(Listing 3), wobei dies in der Beispielimplementierung wieder über eine entsprechende *Set*-Methode realisiert wird (Listing 1). Nun kann der GPIO Pin als Ausgang verwendet und über die Methode *setPinValue()* in der Beispielimplementierung auf den entsprechenden Pegel gesetzt werden (Listing 1).

Fazit

Dank dateibasierter Schnittstellen zu den Pin-Konfigurationsregistern ist es ein Leichtes, mittels Java direkten Zugriff auf die Pins des BeagleBone zu erhalten. Durch die große Leistungsfähigkeit sowie die flexiblen Erweiterungsmöglichkeiten (siehe Capes [3]) eröffnet der BeagleBone eine regelrechte Spielwiese, um mit einer komfortablen Programmiersprache und erprobten Entwicklungstools den Einstieg in die Embedded-Welt zu wagen. Da im Embedded-Bereich heterogene Mehrkernprozessorsysteme gerade *en vogue* sind (z. B. Samsung Galaxy S3 und weitere aktuelle Smartphones), eignet sich der BeagleBone auch optimal für Experimente à la wie lasse ich ein Java-Programm auf einem Cortex-A8-Prozessor mit einem C-Programm auf einem Cortex-M3-Prozessor miteinander kommunizieren? So ist der kleine Hundeknochen auf jeden Fall einen Blick wert.

Danksagung: Diese Arbeit wurde zum Teil von der Magistratsabteilung 23 der Stadt Wien unter der Projektnummer MA 23-Projekt 10-09 unterstützt.

Listing 2

```
private static final String muxPath ="/sys/kernel/debug/omap_mux";
(...)
int mux_cfg = myMuxMode | myMuxPinDir | myMuxPudn | myMuxPudnEn;

FileWriter mux = new FileWriter(muxPath+"/"+myMuxName);
String t = Integer.toHexString(mux_cfg);
mux.write(t);
mux.flush();
mux.close();
(...)
```

Listing 3

```
public static final String pinInput = "in";
public static final String pinOutput = "out";
(...)
myPinPath = new String("/sys/class/gpio/gpio"+myPinId);
(...)
myPinDir = dir;
FileWriter pindir = new FileWriter(myPinPath+"/"+pinDirection);
pindir.write(myPinDir);
pindir.close();
(...)
```

Ein Pin, viele Funktionen

Da Mikro-Controller respektive SoCs über weniger Pins als Funktionen verfügen, werden diverse Pins mit einer Reihe von konfigurierbaren Doppelbelegungen versehen. Dies funktioniert wie folgt:
Zwei Peripherieeinheiten (z. B. GPIO-Modul und Hardware-Timer-Modul) sollen sich einen physischen Pin teilen. Es ist somit möglich, dass entweder das GPIO-Modul oder das Timer-Modul Zugriff auf den Pin bekommt. Der Zugriff wird über einen Multiplexer, der über ein Register konfiguriert werden kann, gesteuert. Demnach kann in Abhängigkeit von der Multiplexer-Konfiguration entweder das GPIO-Modul oder das Timer-Modul auf den SoC Pin zugreifen. Im Falle des BeagleBone können bis zu sieben Hardwaremodule über diese Art und Weise auf einen einzelnen physischen Pin zugreifen.

Matthias Wenzl ist an der FH Technikum Wien am Institut für Embedded Systems in den Bereichen Forschung und Lehre tätig. Des Weiteren ist er aktuell Lektor an der TU Wien für paralleles Programmieren und hält Kurse für Kinder zum Thema Java und LEGO MINDSTORMS.

wenzl@technikum-wien.at

Sigrid Schefer-Wenzl arbeitet in Forschung und Lehre an der WU Wien sowie an der Fachhochschule Campus Wien. Neben ihrer Forschungstätigkeit im Bereich IT-Security hält sie unter anderem Lehrveranstaltungen zum Thema Java-Programmierung.

sschefer@wu.ac.at

Links & Literatur

[1] BeagleBoard.org: http://beagleboard.org/

[2] BeagleBone: http://beagleboard.org/bone

[3] BeagleBone Capes: http://circuitco.com/support/index.php?title=BeagleBone_Capes

[4] BeagleBone LCD3: http://circuitco.com/support/index.php?title=BeagleBone_LCD3

[5] BeagleBone DVI-D with Audio: http://circuitco.com/support/index.php?title=BeagleBone_DVI-D_with_Audio

[6] Raspberry Pi: http://www.raspberrypi.org/

[7] Apple – iPhone – Technische Daten des iPhone 4: http://www.apple.com/de/iphone/iphone-4/specs.html

[8] BeagleBoard-Hardware: http://beagleboard.org/hardware

[9] PandaBoard: http://pandaboard.org/

[10] BeagleBone eLinux.org: http://elinux.org/BeagleBone

[11] The Ångström Distribution: http://www.angstrom-distribution.org/

[12] PuTTY: http://www.putty.org/

[13] Oracle Java ME: http://www.oracle.com/technetwork/java/embedded/resources/me-embeddocs/index.html

[14] OpenJDK: http://openjdk.java.net/

Visual Java für Einsteiger

Einführung in Processing

Java hat sich in vielen Bereichen der Informatik unverzichtbar gemacht – was auch gut so ist. Im Fokus dieser neuen Serie steht allerdings nicht Java, sondern eine andere, auf Java basierende Open-Source-Programmiersprache samt Entwicklungsumgebung: Processing. In deren Umfeld ist es ein Kinderspiel, MS Kinect oder Playstation-Controller anzuschließen, Computerbilderkennung zu betreiben, ästhetische – auch ungewöhnliche – Visualisierungen zu erstellen oder einfach nur unterhaltsame und interaktive Anwendungen zu bauen. Im ersten Artikel der Serie möchten wir die Basics abdecken, bevor wir im zweiten auf die einzigartigen Möglichkeiten eingehen.

von Stefan Siprell

Processing ist eine Programmiersprache samt Entwicklungsumgebung, die seit ihrer Erfindung am MIT 2001 kontinuierlich weiterentwickelt wird. Ursprünglich wurde die Sprache erschaffen, um Programmieranfängern eine visuelle Lernumgebung zu bieten. Mittlerweile kann man aber auch professionelle und abgeschlossene Anwendungen erstellen. Aufgrund der knappen Syntax und der visualisierungslastigen APIs bietet Processing sich bei der Erstellung von Grafiken, Animationen und natürlich interaktiven Anwendungen an.

Da Processing auf Java aufbaut, ist die Sprache selbst sowohl objektorientiert als auch stark typisiert. Durch eine vereinfachte Syntax und reichhaltige Kontextobjekte ist der Quelltext sehr lesbar und knapp gehalten im Vergleich zu regulären Java-Anwendungen. Schauen wir uns das erste Programm – in Processing auch *Sketch* genannt – an (Listing 1).

Als Java-Entwickler erkennt man die Syntax wieder, und man kann sofort zwei Methodenimplementierungen erkennen, vermisst allerdings die Klassendefinition. Dies geschieht implizit und leitet sich von der Basisklasse *PApplet* ab. Diese Basisklasse sieht nicht nur die Hauptmethoden *setup()* und *draw()* vor, sondern stellt eine ganze Reihe von Methoden zur grafischen Ausgabe, für verschiedene Eingabemechanismen, Farbberechnung, mathematische Funktionen und vieles mehr zur Verfügung.

In der *setup()*-Methode wird die Bildrate auf 24 Wiederholungen pro Sekunde gesetzt, das Anti-Aliasing für die 2-D-Ausgabe aktiviert und eine Ausgabefenstergröße von 600 mal 600 Pixel erstellt. *Setup()* wird einmalig vor der ersten Ausgabe durch Processing aufgerufen und dient daher der Initialisierung des Sketch.

Die *draw()*-Methode wird hingegen für jede Aktualisierung des Bilds aufgerufen – in unserem Beispiel 24-mal pro Sekunde. Beim Neuzeichnen des Bildschirms wird die vorherige Ausgabe nicht gelöscht. Das nutzen wir, um einen Fade-Effekt zu generieren, bei dem sich

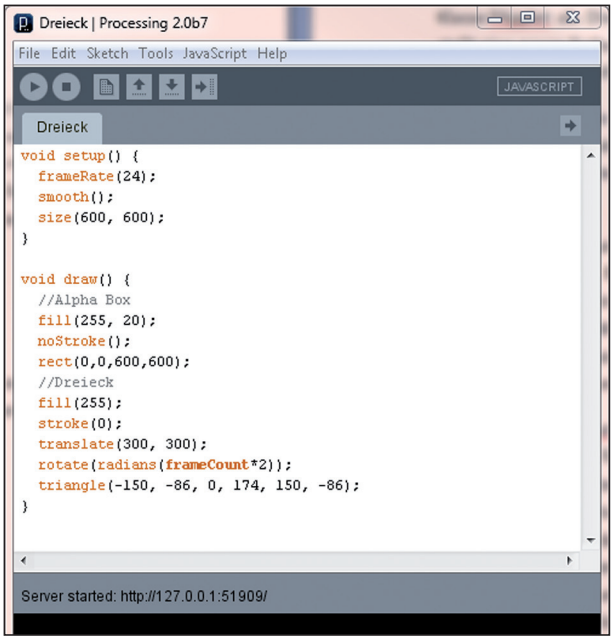


Abb. 1: Processing IDE im JavaScript-Modus

Artikelserie

Teil 1: Einführung in Processing, Nutzen der 2D Renderingengine
Teil 2: Nutzen der 3D Renderingengine mit Kamerafahrten
Teil 3: Computer Vision und Augmented Reality mit Processing
Teil 4: Professionelle Datenvisualisierung mit Java

vorherige Zeichenoperationen graduell ausblenden lassen. Wir legen hierzu ein semitransparentes weißes Rechteck über das gesamte Fenster. Die *fill()*-Anweisung definiert die Farbe und den Alpha-Wert, *noStroke()* verzichtet auf einen gezeichneten Rand, und *rect()* zeichnet letztendlich das Quadrat.

Jetzt wollen wir das Dreieck rotieren lassen. Zunächst müssen wir die Zeichenoperationen für die Füllung und Umrandungen wieder auf volles Weiß bzw. Schwarz mit *fill()* und *stroke()* zurücksetzen. Das Dreieck soll um den Mittelpunkt des Umkreises rotiert werden. Da Processing alle Rotationsoperationen um den Ursprung (0|0) ausführt, muss das Dreieck so gezeichnet werden, dass der Mittelpunkt im Koordinatenursprung liegt. Das geschieht mit der Methode *triangle()*. Damit das Dreieck über die obere linke Ecke rotiert wird, verschieben wir das Koordinatensystem um 300 Pixel nach rechts und unten mit der *translate()*-Anweisung. Zu guter Letzt wird aus der Anzahl der bisherigen Wiederholungen der Drehwinkel bestimmt, und per *rotate()* wird die Rotation schlussendlich auch ausgeführt.

Die Anwendung kann direkt aus der IDE gestartet werden, die im Hintergrund aus dem Processing Code vollwertigen Java Sourcecode für ein Applet erstellt, kompiliert und das Applet anschließend startet. In der neuesten Version ist es möglich, den Processing-Code per HTML5 und Canvas direkt im Browser auszuführen oder als Android-Anwendung im Emulator zu starten. Diese so genannten *Modes* (Java, Android und JavaScript) können direkt in der IDE ausgewählt werden. Wenn man mit der Entwicklung der Anwendung fertig ist, kann man den Sketch für das entsprechende Modul exportieren – damit lässt sie sich ohne IDE direkt auf dem Desktop, Browser bzw. Android-Gerät starten.

Listing 1

```
void setup() {
  frameRate(24);
  smooth();
  size(600, 600);
}
void draw() {
  //Alpha Box
  fill(255, 20);
  noStroke();
  rect(0,0,600,600);
  //Dreieck
  fill(255);
  stroke(0);
  translate(300, 300);
  rotate(radians(frameCount*2));
  triangle(-150, -86, 0, 174, 150, -86);
}
```

Ansprechende Visualisierungen mal anders

Ein weiteres Beispiel, wie man mit sehr wenig Code spannende Ausgaben erzeugen kann, ist die Erzeugung grafischer Auswertungen von regionalen Daten. Die folgende Anwendung (Listing 2) visualisiert die Fitch Ratings der EU-Nationen. Die grünen Töne stellen gute, die roten Töne weniger gute Ratings dar.

Die Hashmaps werden in der *setup()*-Methode aus den String-Arrays befüllt. Die *HashMap fitchRatingsEurope* enthält das ISO-Kürzel der Länder als Schlüssel und die Ratings als Werte. Die *ratingColors* Map hat wieder die Länderkürzel als Schlüssel, enthält aber berechnete Farbcodes als Werte. Man hätte dies alternativ direkt aus z. B. einer XML-Datei (*loadXML()*) laden können, aber aus Anschauungszwecken wurden die Daten direkt inline im Quellcode abgelegt. In der *setup()*-Methode wird außerdem eine SVG-Grafik geladen. Die Grafik enthält die Umrisse aller europäischen Staaten. Mit Inkscape wurde sichergestellt, dass die Umrisse das ISO-Kürzel des jeweiligen Landes als ID tragen.

In der *draw()*-Methode wird die Ausgabe verkleinert; die CSS-Eigenschaften der Karte wurden deaktiviert und ein Grauton wurde als Standardfüllfarbe definiert. Mit diesen An-

Abb. 2: Grafische Ausgabe der Dreiecksrotation

Listing 2

```
import java.util.Map;
import java.util.Iterator;

PShape europe;
String[] euCountries = {"be", "bg", "dk", "de", "ee", "fi", "fr", "gr", "ie", "it", "lv",
   "lt", "lu", "mt", "nl", "at", "pl", "pt", "ro", "se", "sk", "si", "es", "cz", "hu", "uk",
                                                                    "cy", "hr" };
String[] fitchRatings = {"AA", "BBB-", "AAA", "AAA", "A+", "AAA", "AAA", "B-",
     "BBB+", "A-", "BBB-", "BBB", "AAA", "A+", "AAA", "AAA", "A-", "BB+", "BBB-",
                          "AAA", "A+", "A", "BBB", "A+", "BBB-", "AAA", "B", "BBB-" };
String[] ratingList = {"AAA", "AAA-", "AA+", "AA", "AA-","A+", "A", "A-","BBB+",
    "BBB", "BBB-", "BB+", "BB", "BB-","B+", "B", "B-","CCC+", "CCC", "CCC-", "CC+",
                                                    "CC", "CC-","C+", "C", "C-"};

HashMap<String, String> fitchRatingsEurope = new HashMap();
HashMap<String, Integer> ratingColors = new HashMap();

void setup(){
  for(int i = 0; i < euCountries.length; ++i){
    fitchRatingsEurope.put(euCountries[i], fitchRatings[i]);
  }
  for(int i = 0; i < ratingList.length; ++i){
    ratingColors.put(ratingList[i], color(i*10, 255-(i*10), 0));
  }
  size(800, 600);
  background(255);
  europe = loadShape("Map.svg");
}

void draw(){
  scale(0.075);
  europe.disableStyle();
  fill(color(128,128,128));
  shape(europe, 0, 0);
  drawCountries(fitchRatingsEurope, ratingColors);
}
```

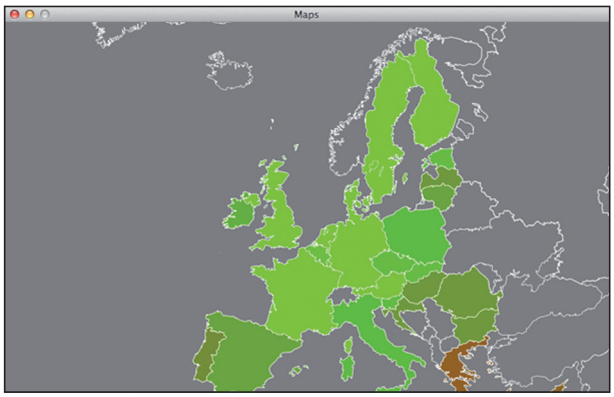


Abb. 3: Eingefärbte Europakarte

gaben werden die Umrisse gezeichnet. In der Hilfsmethode *drawCountries()* wird für jedes Land der EU der entsprechende Farbcode für das Ranking ermittelt und das Land mit der entsprechenden Füllung gezeichnet. Da die Umrisse in der SVG-Datei zuvor mit den korrekten IDs versehen worden sind, kann man *europe.getChild()* sehr intuitiv nutzen.

Koordinatensystem, Zeichnen und Transformationen

Das zweidimensionale Koordinatensystem hat seinen Ursprung in der oberen linken Ecke und erstreckt sich von dort aus auf das gesamte Fenster, welches mit der *size(width, heigh)* erstellt wurde. Neben einfachen geometrischen Basisformen:

- Ellipsensegmenten: *arc()*
- Ellipsen: *ellipse()*
- Linien: *line()*
- Punkten: *point()*
- Quadraten: *quad()*
- Rechtecken: *rect()*
- Dreiecken: *triangle()*

kann man auch beliebig komplexe Formen auf den Bildschirm ausgeben. Diese Formen lassen sich als:

- Kurven mit den *bezier()*- oder den *curve()*-Funktionen erstellen.
- Polygone mit den *vertex()*-Funktionen erstellen.

Das Zeichnen in Processing kann man sich sehr bildhaft vorstellen. Man kann sich sein Zeicheninstrument festlegen, und alle folgenden Zeichenoperationen erfolgen mit dem Zeicheninstrument, bis man es wieder ändert. Zum Festlegen der Zeichenoperationen kann man die Funktionen

- *strokeCap()*, *strokeJoin()*, *strokeWeight()*, *stroke()* und *noStroke()* für die Definition der Linien
- *fill()*, *noFill()* für die Definition der Füllung

nutzen (Listing 3).

Erwähnenswert sind auch die *Translation*-Funktionen. Damit kann man das Koordinatensystem, auf dem die Zeichenausgaben erfolgen, manipulieren, sodass die Ausgaben gedreht, gestaucht, gedehnt o. ä. werden (siehe Kasten: „Translation-Funktionen“).

Listing 3

```
void setup(){
  size(300,300);
  smooth();
}

void draw(){
  noStroke();
  fill(color(255,0,0));

  beginShape();
  vertex(250, 110);
  vertex(250, 290);
  vertex(50, 290);
  vertex(50, 110);
  endShape();

  fill(color(0,255,0));
  bezier(50, 110, 50, 10,
      250, 110, 250, 110);
}
```

Echte IDEs und los geht's

Es sollte nicht unerwähnt bleiben, dass man auch ohne die sehr einfach gehaltene IDE arbeiten kann. Durch die Einbindung der Processing JARs kann man auch ohne Weiteres die Anwendungen in Eclipse schreiben. Hierzu gibt es eine sehr gute Anleitung auf der Processing-Homepage [1]. Man muss lediglich die Startklasse von der erwähnten *PApplet*-Klasse ableiten, und schon kann es losgehen. Natürlich sind mit Eclipse etc. nur Java-Module nutzbar. Benötigen Sie JavaScript- oder Android-Unterstützung, sollten Sie weiterhin die Processing IDE nutzen.

Falls Sie nun Interesse haben, schauen Sie auf der Downloadseite [2] vorbei und laden Sie das Paket herunter. Packen Sie das Archiv aus, starten Sie die IDE – und in wenigen Minuten läuft Ihre erste Anwendung. Happy Sketching.

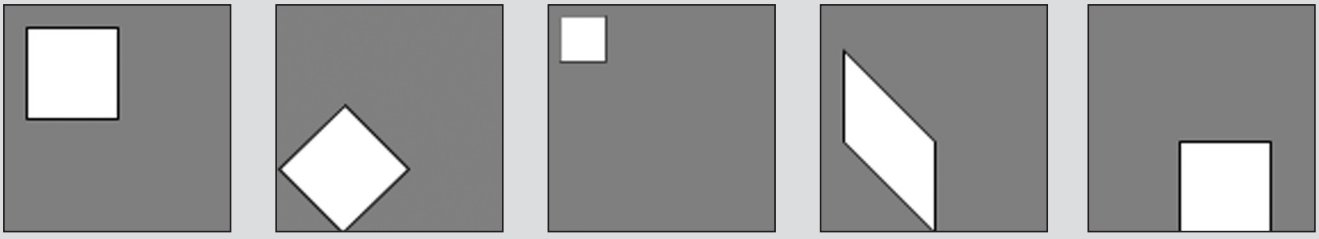
Stefan Siprell (stefan.siprell@exxeta.com) ist Teamleiter bei der Exxeta AG in Karlsruhe. Seine fachlichen und technologischen Schwerpunkte liegen im agilen Software Engineering. In der Freizeit beschäftigt er sich mit Arduino, Processing und Co.

Links & Literatur

[1] http://processing.org/learning/eclipse/

[2] http://processing.org/download/

Translation-Funktionen



Standardausgabe | Drehen mit *rotate()* | Vergrößern und verkleinern mit *scale()* | Scheren mit *shearX()* | Verschieben mit *translate()*

Wie wichtig ist Softwarearchitektur?

Architektur sollte man vermeiden

Ganz auf Softwarearchitektur zu verzichten, kann zum Problem werden. Das liegt daran, dass die Auswirkungen von „schlechter Architektur" beträchtlich sein können. Es ist trotzdem immer wieder notwendig, über den Stellenwert von Architekturarbeit zu diskutieren. Und das ist auch nachvollziehbar, denn im echten Leben verdichten sich beim Architekten alle Probleme der Softwareentwicklung. Immer wenn was richtig schief geht, dann ist es auf die Softwarearchitektur zurückzuführen. Meistens sind erfahrene Architekten auch noch die mit Abstand teuersten Berater am Markt. Also lohnt es sich, das Geld in die Hand zu nehmen? Die kurze Antwort lautet: ja, aber man sollte es vermeiden.

von Niklas Schlimm

Die Frage, ob Architektur wirklich wichtig ist, kann man auf unterschiedliche Art und Weise beantworten. Ich stelle im Folgenden vier unterschiedliche Argumentationslinien vor. Zunächst erfolgt eine kurze Betrachtung, wie sich Softwareentwicklung über die Jahre verändert hat. Das vermittelt ein gutes Gefühl dafür, ob und wann Softwarearchitektur wirklich wichtig ist. Auf Basis einer Fallstudie wird danach beschrieben, welche Ursachen und Folgen Architekturprobleme haben können. Dann drehen wir den Spieß um, denn Softwarearchitektur ist nicht nur eine gute Strategie zur Problemvermeidung. Es wird beschrieben, wie durch den Einsatz von Frameworks konkrete betriebswirtschaftliche Potenziale entstehen. Zuletzt beschreibe ich, warum Architektur für die Wettbewerbsfähigkeit des Unternehmens von zentraler Bedeutung ist.

Architektur eines Softwaresystems

In der Literatur gibt es viele Definitionen zum Begriff Architektur eines Softwaresystems. In [1] wird Architektur wie folgt definiert: „The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." Diese Definition ist nur teilweise zufriedenstellend. Sie weist auf den Strukturaspekt von Architektur hin, sowie darauf, dass die Strukturelemente in Beziehung stehen. Es geht hier um das Ergebnis von Architekturarbeit. Wenig intuitiv ist jedoch der dargestellte Hinweis auf die „extern sichtbaren Eigenschaften". Es ist schwierig, bei einem imaginären Produkt wie Software von extern sichtbaren Eigenschaften zu sprechen. Alles, was einem Laien dazu einfällt, ist allenfalls die Benutzeroberfläche. Des Weiteren fehlt der Bezug zur Umgebung, zu den Stakeholdern, die ja die wichtigen architekturrelevanten Anforderungen an die Architektur definieren. Deswegen wird folgende Definition aus [2] ergänzt: „The architecture of a system constitutes what is essential about that system considered in relation to its environment." Über diese Definition wird der wichtige Aspekt ergänzt, dass es sich bei Architektur immer um die wirklich essenziellen Systemaspekte handelt. Ob was essenziell ist, ergibt sich aus der Beziehung zur Umwelt. Zum Beispiel daraus, welche Erwartungen die wichtigsten Stakeholder haben. Dieser Aspekt wird auch in [3] betont, wenn es heißt: „Software Architecture = { Elements, Form, Rationale }." Architektur entsteht immer aus einem bestimmten Grund („Rationale"). Diese Gründe sind idealerweise die Anforderungen des Kunden.

Wie sich Softwareentwicklung verändert hat

Das Verstehen einer Technik setzt eine gewisse Grundkenntnis über das Werden des Zu-Verstehenden voraus. Im Laufe der Zeit haben sich die Problemstellungen in der Softwareentwicklung seit Beginn der 50er Jahre

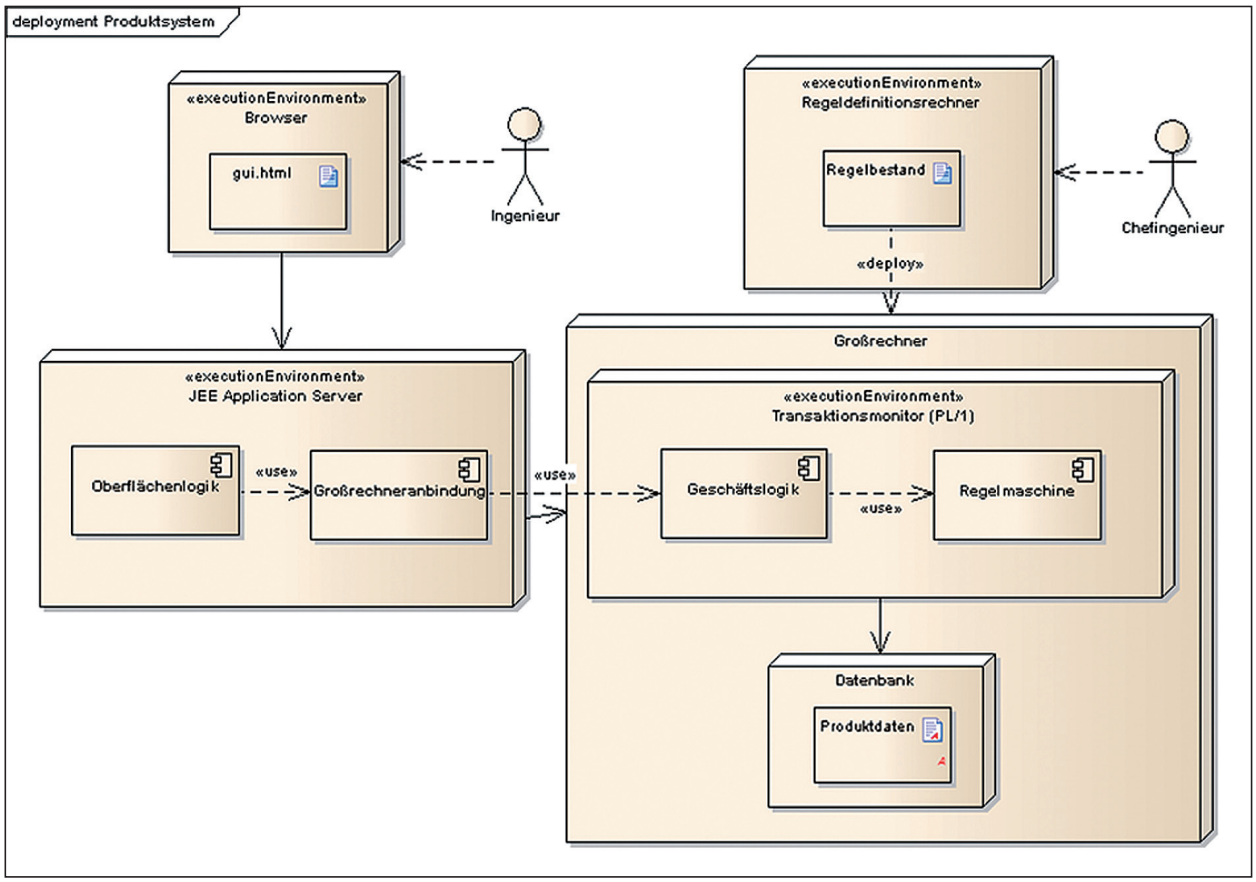


Abb. 1: Architektur eines Produktsystems in der Automobilbranche

erheblich verändert. Ein Grundverständnis über diese Entwicklung hilft dabei, wirklich fundiert die Notwendigkeit von Softwarearchitektur für den eigenen Kontext zu beurteilen.

Zusammenfassend würde ich aus der historischen Betrachtung [5-7] folgende Aspekte auflisten, die die Bedeutung von Softwarearchitektur begründeten bzw. erhöhten:

- Software wurde im Laufe der Zeit viel größer, das erforderte immer mehr Abstraktion, um das gesamte System noch zu verstehen.
- Aufgrund der Größe musste die Arbeit auf mehrere Entwickler aufgeteilt und Software in Files und Module zerlegt werden.
- Die Verteilung von Software auf mehrere Systeme erfordert die Integration eben dieser Systeme zur Durchführung der Aufgabensynthese.
- Die an der Erstellung von Software beteiligten Personen vertreten unterschiedliche Interessen.
- Die Auswahl an Lösungsalternativen ist durch das Internet und Open-Source-Software deutlich größer geworden.
- Die offenen Architekturen aus der Open-Source-Gemeinde sind eine gute Alternative gegenüber den vorgefertigten Architekturen der etablierten Anbieter.
- Software sollte für Produktfamilien entwickelt werden, das heißt auf verschiedenen Endgeräten ausführbar sein. Parallel stieg diese Heterogenität der Endgeräte.

Diese Faktoren haben dazu geführt, dass Softwarebau heutzutage besonders komplex ist. Die Komplexität ergibt sich einerseits durch die schiere Größe der Software bzw. des zu lösenden Problems und andererseits durch die Auswahlmenge an Alternativen und Einflussfaktoren bei Architekturentscheidungen. Softwarearchitektur ist eine junge Disziplin, die diese Komplexität reduzieren soll.

Fallstudie: Ursachen und Auswirkungen von Architekturproblemen

Softwaresysteme sind groß und komplex. Kein Wunder also, dass bei der Entwicklung Probleme auftreten. Welche das auf Architekturebene sind, verdeutlicht folgende Fallstudie. Ein internationaler Automobilkonzern baut ein neues Produktsystem, um Autos an weltweit verteilten Standorten zu entwickeln. Mit dem Kunden werden folgende nicht funktionale Anforderungen besprochen:

1. Das neue System soll bei 90 Prozent der Benutzertransaktionen eine Antwortzeit von unter zwei Sekunden haben.
2. Das System soll es ermöglichen, innerhalb von vier Wochen neue Produktkonfigurationen einzuführen.
3. Das System soll über diverse Endgeräte zugänglich sein, die gegenwärtig und in absehbarer Zukunft am Markt erhältlich sind.

4. Bestehende PL/1-Geschäftslogik auf dem Großrechner soll möglichst wiederverwendet werden.
5. Ein hoher Funktionsumfang des Systems zum festgelegten Einführungstermin ist wichtiger als eine hohe Ressourceneffizienz.

Die konzeptuelle Architektur ist in **Abbildung 1** dargestellt. Beim Entwurf wurde berücksichtigt, dass das System besonders gut änderbar sein muss. Das System wurde dazu „regelbasiert" entworfen. Auf diese Weise können Ingenieure eigenständig umfangreiche Funktionsanpassungen über einen Regeldefinitionsrechner vornehmen. Um eine große Anzahl unterschiedlicher Endgeräte zu unterstützen, hatte der Kunde sich für eine konventionelle Weboberfläche entschieden. Durch die gewählte Technologie wird ein Web Application Server als Middleware zwischen Endbenutzerclient und Großrechner notwendig. Es wird sich für einen Java EE Server entschieden. Java ist eine weit entwickelte Sprache und offene Plattform mit sehr großer Verbreitung, hohem Funktionsumfang und die Programme sind auf unterschiedlichsten Computersystemen ausführbar.

Herausfordernd sind nun die Performanzanforderungen sowie die Wiederverwendung bestehender Geschäftslogik in PL/1. Zur Wiederverwendung der Geschäftslogik bestehen grundsätzlich zwei Alternativen:

1. Die neue Geschäftslogik für das Bestandssystem wird in Java entwickelt und es erfolgt eine Anbindung der bestehenden PL/1-Programme.
2. Es werden auch die neuen Programmteile in PL/1 auf dem Großrechner entwickelt.

Wenn die neuen Programmteile in Java entwickelt werden würden, befürchtet man viele Fernzugriffe pro Benutzertransaktion auf den Großrechner. Um die Anforderungen an die Antwortzeiten zu erfüllen, soll die Wartezeit durch Netzwerkkommunikation jedoch minimiert werden. Es wird sich daher für PL/1 auf dem Großrechner entschieden. Des Weiteren wird der Großrechner auch als Laufzeitumgebung für die Regelmaschine bestimmt. In Summe entsteht so die

Produktfamilien

Dem Thema Produktfamilien [4] kommt im Kontext von Softwarearchitektur ein besonderer Stellenwert zu. Eine Produktfamilie ist – vereinfacht ausgedrückt – eine Menge von Programmen, die gemeinsame Eigenschaften haben. In den 60ern hatte man erst einmal eine Produktversion fertiggestellt. Danach wurde von dieser aus das zweite „Mitglied der Familie" entwickelt, zum Beispiel dasselbe Betriebssystem für eine andere CPU-Architektur. Das Vorgehen erzeugte hohen Änderungsaufwand von einer Version zur nächsten. Man erkannte, dass es sich auszahlt, gemeinsame Eigenschaften der Familie erst zu erkennen, bevor man ein Programm ganz fertigstellt. Denn auf diese Weise können zuerst die gemeinsamen Entwurfsentscheidungen getroffen werden. Später können dann von einem gemeinsamen Programmstand aus unterschiedliche Versionen von ausführbaren Systemen erstellt werden.

Ein typisches Beispiel einer Produktfamilie ist ein Betriebssystem, das für unterschiedliche CPU-Architekturen entwickelt wird. Hier trifft man typischerweise sofort die Entscheidung, das die systemnahen Anweisungen in eine „untere Schicht" gekapselt werden. Auf dieser Schicht bauen dann die höheren benutzernahen Schichten und Dienste auf. Unterschiedliche CPU-Architekturen führen dann Idealerweise nur zu neuen Versionen, die sich höchstens in den maschinennahen Schichten unterscheiden. Die benutzernahen Schichten der Familie sind jedoch die gleichen in allen Versionen. Solche abstrakten Entscheidungen kommen dem, was wir heute unter einer Architekturentscheidung verstehen, sehr nahe. Denn auch hier müssen wir gemeinsame Eigenschaften (von Modulen, Klassen, Anwendungen) erkennen und darauf aufbauend bestimmte Entwurfsentscheidungen früh treffen. Diese Entscheidungen haben dann für das gesamte System bindenden Charakter und deren Rückabwicklung verursacht extrem hohe Kosten.

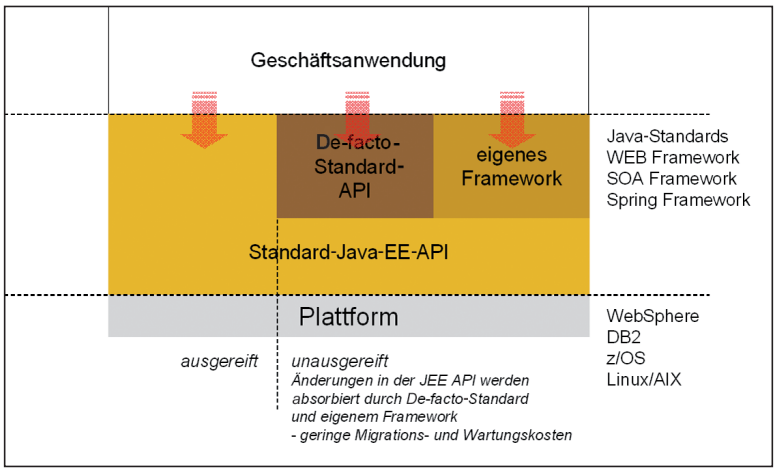


Abb. 2: Produktfamilie auf Basis von Java-Frameworks

geringste Wartezeit zwischen den fachlich eng verzahnten Komponenten.

Das Ergebnis der Überlegungen ist eine klassische Drei-Schichten-Architektur bestehend aus HTML-basierter Oberfläche im Browser, Oberflächenlogik in Java auf dem JEE Application Server und der Geschäftslogik in PL/1 auf dem Großrechner als Backend-System. Das System erfüllt in Produktion alle oben aufgeführten Anforderungen. Es gibt aber trotzdem einige Probleme, deren Ausmaß leider erst nach Produktionseinführung erkannt wird.

Es kommt zu einem sehr hohen CPU-Verbrauch, wodurch der Betrieb der anderen Anwendungen auf dem Großrechner gefährdet wird. Es ist sehr kostspielig, die CPU-Ressourcen auf dem Großrechner zu erhöhen. Der Zukauf von CPU und Speicher soll das Problem also alleine nicht lösen. Bei der Regelmaschine handelt es sich um generierten Programmcode in ANSI C. Dieser kann daher auch für eine günstige dezentrale Windows-Plattform kompiliert und dorthin ausgelagert werden. Die neu entwickelten PL/1-Programme müssen auf dem Großrechner verbleiben und unter hohem Aufwand optimiert werden. Die Auswirkungen dieser Maßnahmen sind:

1. Durch die Softwareverteilung entsteht jetzt mehr Netzwerkkommunikation und dies führt zu dauerhaft höheren Antwortzeiten.
2. Für Hardwarezukauf sowie für die Performanceoptimierung muss der Kunde noch mal einen sechsstelligen Betrag aufwenden.
3. Das System wird auf Dauer zu einer geringeren „Verarbeitungsgeschwindigkeit“ im Fachbereich führen. Die geplante Prozessoptimierung kann nur teilweise erreicht werden.

Das zeigt, dass etwas Essenzielles schief gelaufen war. Aber was? Alle formulierten Anforderungen wurden korrekt erfüllt. Daher müsste man diesbezüglich von einem Erfolg sprechen, das System arbeitet wie spezifiziert. Leider wurden die bestehenden Anforderungen an den Ressourcenverbrauch jedoch nicht formuliert. Dadurch hat das Team beim Entwurf des Systems maßgeblich auf Antwortzeiten, Wiederverwendung und Änderbarkeit des Programms geachtet. Bei der Auswahl der Lösungen sind ebenfalls Fehler unterlaufen. Gut änderbare Programme, wie regelbasierte Systeme, stehen grundsätzlich unter dem Verdacht, nicht besonders ressourceneffizient zu sein. Daher hätte dafür möglicherweise eine günstigere Plattform von vornherein als Lösungsalternative betrachtet werden müssen. Auf dieser hätte man den hohen Ressourcenverbrauch wahrscheinlich durch Zukauf von Hardware verkraften können. Unter diesen Umständen hätten die neuen Programmteile möglicherweise doch komplett in Java gebaut werden können. Denn dann hätte die neue Geschäftslogik und die Regelmaschine zusammen im JEE Application Server lokal deployt werden können. Auf diese Weise hätte man Hardwarekosten sparen können und die Antwortzeiten hätten sich verbessert.

Die Fallstudie hat (stark vereinfacht) in den Entscheidungsprozess auf Architekturebene eingeführt, und hat verdeutlicht, wie sich Architekturentscheidungen auf den Projekterfolg auswirken. Es geht um Entscheidungen über essenzielle Eigenschaften des Systems. Wenn diese Entscheidungen falsch getroffen werden, kommt es zu schwerwiegenden Auswirkungen. Das liegt daran, dass die Korrektur- und Folgekosten falsch getroffener Architekturentscheidungen hoch sind. Gleichzeitig werden die Entscheidungen auf Basis unvollständiger Informationen getroffen. Daher sind sie mit einer hohen Unsicherheit versehen. In der Fallstudie wurde der zu erwartende CPU-Verbrauch zum Beispiel nicht in die Entwurfsentscheidungen mit einbezogen.

Es ist unmöglich, jedes Problem während der Entwicklung oder nach der Produktionseinführung vorherzusehen. Das bedeutet ferner, dass nicht alle Entwurfsentscheidungen auch als architekturrelevant eingestuft und wirklich bewusst getroffen werden. In der Fallstudie ist man aufgrund der gewählten Programmiersprache PL/1 erst mal an den Großrechner gebunden. Als man spät erkannt hat, dass die Anwendung sehr CPU-intensiv ist, wurde das zum Problem. Die Aufrüstung der Hardware ist nämlich mit dauerhaften Folgekosten verbunden, da auf CPU-Verbrauchsbasis abgerechnet wird. Solche „Sackgassenszenarien“ vorherzusehen kostet Zeit und erfordert viel Erfahrung im Bereich Architektur. Alle möglichen Szenarien vorherzusagen ist unmöglich. Das ist das Dilemma, in dem wir alle stecken und der Hauptgrund, warum man Architektur vermeiden sollte. Aber darauf gehe ich später noch mal ein.

Potenziale durch frameworkbasierte Produktfamilien

Zuvor haben wir Softwarearchitektur sozusagen als Zwangsdisziplin vorgestellt, die man machen muss,

weil es sonst erhebliche Probleme gibt. Jetzt drehen wir den Spieß einmal um und sagen: es gibt betriebswirtschaftliche Vorteile durch Softwarearchitektur! Als konkretes Beispiel gehe ich im Folgenden auf den Aufbau von Produktfamilien auf Basis zentral entwickelter Frameworks ein.

Viele Unternehmen betreiben mehr als fünfzig verschiedene Java-EE-Anwendungen. Für solche Szenarien ist es sehr ratsam, die einzelnen Geschäftsanwendungen auf einem zentral entwickelten Framework aufzubauen. Alle Anwendungen sind auf diese Weise Mitglied einer Produktfamilie, sie ähneln sich also in weiten Teilen. **Abbildung 2** zeigt ein praxisnahes Architekturbeispiel für eine Produktfamilie Java-basierter Geschäftsanwendungen. Jede Geschäftsanwendung arbeitet auf Basis unterschiedlicher Frameworks. Die Nutzung von Java EE und Spring in einer Anwendung wird besonders von den jeweiligen Religionsvertretern kontrovers diskutiert. Sie ergibt sich jedoch ganz natürlich aufgrund der Evolution, der nicht funktionalen Anforderungen sowie der organisatorischen Gegebenheiten in großen Unternehmen. Das hauseigene Framework bietet Funktionen, die keiner der Standards unterstützt. Durch den Frameworkeinsatz können viele Basisdienste zentral an einer Stelle implementiert werden.

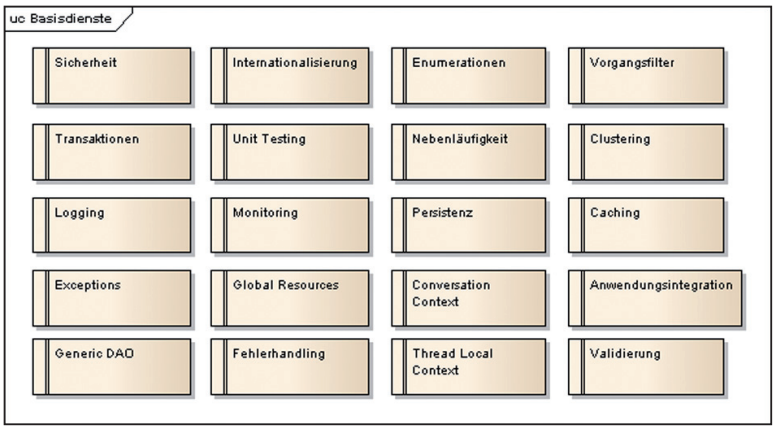


Abb. 3: Technische Basisdienste einer konkreten Produktfamilie

Abbildung 3 zeigt wichtige Basisdienste, die in einer konkreten Kundensituation umgesetzt werden sollten. Diese Themen sollten dann für alle Mitglieder der Familie am besten einmal gelöst werden. Durch den Einsatz von Produktfamilien wurde vermieden, dass die Themen immer wieder neu entwickelt werden. Stattdessen werden sie einmal im Framework bereitgestellt. Ein neues Mitglied einer Produktfamilie ergibt sich nun durch die Portierung einer Geschäftsanwendung auf eine neue Plattform (Standardsystem-Produktfamilie). Oder es wird auf derselben Plattform eine neue Geschäftsanwendung mit dem Framework entwickelt (Standardplattform-Produktfamilie). Durch die Gründung einer solchen Produktfamilie entsteht ein ganz entscheidender wirtschaftlicher Vorteil, der nur erzielt werden kann, wenn die Softwarearchitektur entsprechend entworfen und im Unternehmen durchgesetzt wurde.

Die hohe Governance durch die Gründung von Produktfamilien hat weitere nennenswerte Vorteile, die nicht unbedingt direkt „auf der Hand“ liegen. Betrachten wir zum Beispiel die Automatisierung des Deployment-Verfahrens. Hier stellen sich die Gesamtkosten für die Automatisierung K_G als Funktion über den Kosten der Automatisierungsmaßnahme K_E und der Anzahl unterschiedlicher Anwendungen n dar: $K_G = K_E * n$. Anwendungen sind unterschiedlich, wenn sie verschiedene Deployment-Spezifikationen aufweisen. Durch ein einheitliches Framework ist die Konfiguration für eine Zielumgebung immer gleich. Das bedeutet: n = 1. So sind die Automatisierungskosten sehr gering und es kann ausgehend vom Kundenauftrag an die IT bis zum Release in die Test- oder Produktionsumgebung eine umfangreiche Automatisierung stattfinden.

Zusammenfassend lässt sich festhalten, dass Softwarearchitektur gezielt zur Realisierung betriebswirtschaftlicher Potenziale eingesetzt werden kann. Denn durch Produktfamilien werden Erstellungs- und Wartungskosten von Software deutlich reduziert. Das wird ermöglicht, indem die Implementierung von Basisdiensten an zentraler Stelle erfolgt, anstatt in jeder einzelnen Fachanwendung.

Geschäft und Architektur beeinflussen sich gegenseitig

Jetzt spannen wir den Bogen zum Schluss noch etwas weiter. „Strategische“ Architekturentscheidungen sind solche, die sich nicht nur auf eine Anwendung, sondern auf das gesamte Unternehmen beziehen. Sie überdauern viele Jahre, weil deren Rückabwicklung erhebliche Kosten mit sich bringen würde. PL/1 auf dem Großrechner wurde z. B. vor vierzig Jahren in viele Unternehmen eingeführt und diese Unternehmen haben heute 15 Millionen Zeilen PL/1-Code im Bestand. Eine einfache Migration in moderne Java-Umgebungen ist aus wirtschaftlichen Gründen unmöglich. Andererseits drängt der Markt die Unternehmen dazu, modernere Technologien zu verwenden. Die jüngeren Kollegen wollen nicht mit „verstaubten“ Technologien des letzten Jahrhunderts arbeiten. Im Ergebnis müssen moderne Systeme nun mit dem Bestand an Geschäftslogik in Altsystemen integriert werden. Wir finden heute mehr oder weniger heterogene Systemlandschaften vor.

Die Geschäftsprozesse des Fachbereichs sind mit der Systemlandschaft des Unternehmens eng verzahnt. Aktivitäten werden vollautomatisiert ausgeführt, oder es wird eine Benutzeroberfläche bereitgestellt, um teilautomatisierte Tätigkeiten im Rahmen einer Mensch-Maschine-Kommunikation abzuwickeln. Mensch und Maschine sind beide Aufgabenträger (Hardware) im Geschäftsprozess. Das menschliche Gehirn und die entwickelte Software enthalten die Programme zur kooperativen Aufgabendurchführung. Unternehmen entwickeln neue Produkte und optimieren Geschäfts-

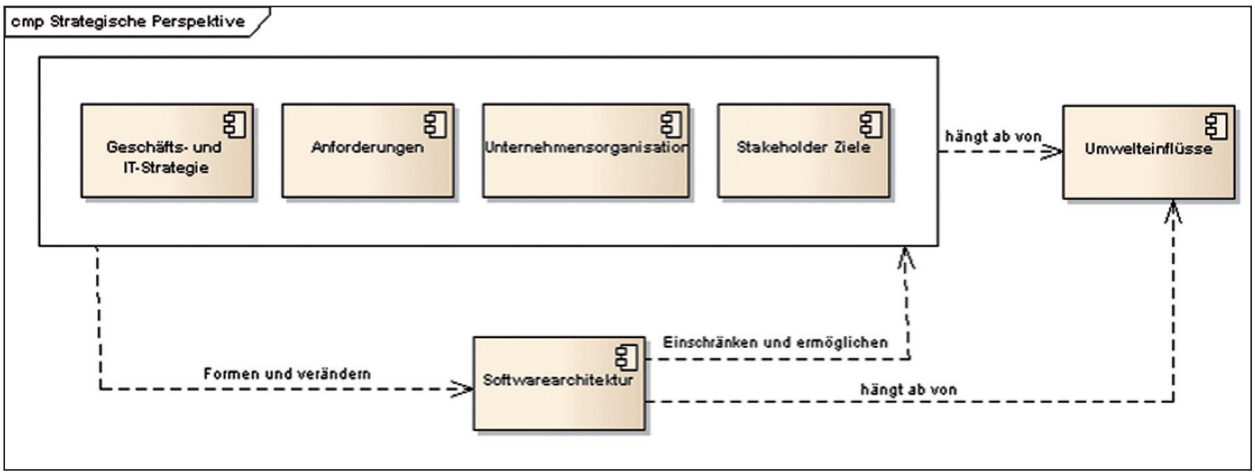


Abb. 4: Wechselwirkungen zwischen Softwarearchitektur und Geschäft

prozesse, um dem Kostendruck mit immer höherer Produktivität zu begegnen. Solche Veränderungen haben direkte Auswirkungen auf alle Aufgabenträger. Während sich das menschliche Gehirn verhältnismäßig schnell autonom umstellen kann, bedeutet die Änderung der Software hohen Aufwand. Dieser muss nun bei der Entscheidungsfindung mit berücksichtigt werden. Wir sind, wie man so schön sagt, normalerweise nicht mehr „auf der grünen Wiese", wenn wir neue IT Projekte planen und umsetzen.

Es bestehen also Abhängigkeiten zwischen Geschäftsprozessen des Fachbereichs und komplexer Systemlandschaft. Deswegen nehmen einmal getroffene Architekturentscheidungen erheblichen Einfluss auf die Fähigkeit eines Unternehmens, auf geänderte Geschäftsstrategien und Wettbewerbsbedingungen zu reagieren. Sie bestimmen maßgeblich das Kosten-Nutzen-Verhältnis von zukünftigen Initiativen und zeigen Grenzen bezüglich der Machbarkeit von Produkt- und Prozessinnovationen auf (**Abb. 4**). Damit wirken sich Architekturentscheidungen nicht nur auf den Projekterfolg aus, sie betreffen unter Umständen den Erfolg des gesamten Unternehmens. Dieser Gedankengang führt das Thema Architektur auf eine Ebene von strategischer Bedeutung für den langfristigen Erfolg des Unternehmens.

Architektur vermeiden ist sinnvoll

Nachdem was ich bisher zusammengetragen habe, machen der Titel des Artikels und diese Aussage vielleicht keinen Sinn. Doch in den letzten Kapiteln sollte auch folgender zentraler Aspekt klar geworden sein: Architekturentscheidungen werden unter Unsicherheit getroffen und sind nur schlecht wieder rückgängig zu machen. Gleichzeitig nehmen sie erheblichen Einfluss auf den Erfolg des Projekts oder des Unternehmens. Wir haben auch gelernt, dass Architekturentscheidungen besonders komplex sind. Es wäre unter all diesen Umständen ungünstig, besonders viele solcher Entscheidungen treffen zu müssen. Eine mögliche Lösung ist, Architekturentscheidungen zu vermeiden.

Aber wie soll das gehen? Martin Fowler hat in seinem Artikel „Who needs an Architect?" [8] angemerkt: „I think that one of an architect's most important tasks is to remove architecture by finding ways to eliminate irreversibility in software designs." Wenn man seine Entscheidungen problemlos ändern kann, dann ist es auf einmal weniger wichtig, sie richtig zu treffen. Daraus folgt, dass Softwareentwürfe modifizierbar sein sollten. Hiermit ist nicht gemeint, dass sie abstrakt sein sollen, sodass niemand mehr den Code versteht. Vielmehr geht es darum, an Stellen, an denen Änderungen schon absehbar sind, auch Änderungen einzuplanen. Der Einsatz von JDBC als Abstraktionsebene zwischen Geschäftslogik und Datenbank macht zum Beispiel das Datenbanksystem weitgehend frei austauschbar. Der Einsatz von Web Services ermöglicht es, ganz verschiedene Technologien an der Benutzeroberfläche zu unterstützen. IOC Container ermöglichen es, ein objektorientiertes System ganz unterschiedlich zu konfigurieren und flexibel auf geänderte Bedürfnisse anzupassen. Das alles sind Beispiele, wie eigentlich essenzielle Eigenschaften eines Systems auf Dauer änderbar bleiben. An der Stelle darf nicht verschwiegen werden, dass solcher Luxus natürlich auch etwas kostet: CPU-Ressourcen.

Ein weiteres probates Mittel, Architekturentscheidungen zu vermeiden, ist, wenn man sich auf geschlossene Baukastenarchitekturen der großen Anbieter reduziert. Von Microsoft, IBM oder Oracle kann man das komplette Programm einkaufen und das wird auch funktionieren. Allerdings hat man so nur Auswahlentscheidungen reduziert. Die liegen nun im Bauchladen des Anbieters vor. Übrig bleibt immer noch der Bau der Anwendungen, die fachliche Architektur des Systems. Dieses muss zerlegt werden, damit mehrere Programmierer daran arbeiten können. Bei dieser Alternative entscheidet sich der Kunde bewusst, einen Teil der Architekturentscheidungen abzugeben und deren Lösung einzukaufen. Daraus ergeben sich manchmal unerwünschte Abhängigkeiten zum Lieferanten und es kann auch im Vergleich zu offenen Architekturen ziemlich teuer werden.

Aus der XP-Gemeinde kommt die kontrovers diskutierte Forderung, kein „Big Design Up Front“ zu machen. Darunter wird eine zu lange Analysephase ohne Kodierung verstanden. Die Betonung liegt hier auf dem „Big“ und nicht auf dem „Up Front“. Das Problem einer langen Analysephase ohne Programmierung liegt darin, dass es unmöglich ist, alle Probleme vorherzusehen, die sich während der Entwicklung ergeben. Außerdem ändern sich die Anforderungen laufend. Also direkt losprogrammieren? Das geht leider auch nicht, denn leider müssen einige essenzielle Eigenschaften des Systems entschieden werden, bevor programmiert wird: die Programmiersprache, die Laufzeitumgebung, wenigstens ein paar fachliche Anwendungsfälle und einiges mehr. Die modernen Vorgehensweisen zur Architekturentwicklung versuchen deswegen die goldene Mitte zu finden und schlagen vor, ein „Skeletal System“ [1] zu entwickeln oder „Experimente“ [9] durchzuführen, um Unsicherheiten zu reduzieren. In Feature-driven Development wird erst ein Gesamtmodell des Systems erstellt [10]. Die Zeitspanne zwischen dem Projektstart und dem Beginn der Programmiertätigkeit wird in allen Fällen kurz gehalten. Denn im Rahmen der Programmiertätigkeit können Unsicherheiten reduziert werden, bevor man eine endgültige Entscheidung trifft.

Zuletzt möchte ich noch auf eine vielleicht triviale Erkenntnis hinweisen, die man nicht oft genug wiederholen kann: ein System sollte unbedingt in Anlehnung an seine konkreten Anforderungen entwickelt werden. Systeme, deren tatsächliche Lebenszeit kurz ist, müssen nicht wartbar sein. Onlineanwendungen brauchen in der Regel nicht dieselbe Performance wie Batch-Anwendungen. Systeme, die nicht portiert werden sollen, müssen natürlich nicht plattformunabhängig sein. Es ist sehr ratsam, sich die Anforderungen an die Architektur des Systems genau zu betrachten, um jegliches „Over-Engineering“ tunlichst zu vermeiden. Auch dadurch wird die Anzahl von Architekturentscheidungen erheblich reduziert, indem die Menge relevanter Architekturtreiber auf ein Minimum herabgesetzt wird.

Fazit

Durch Softwarearchitektur soll Komplexität reduziert werden. Das funktioniert, indem Informationen verständlich aufbereitet und Auswahlentscheidungen methodisch fundiert getroffen werden. Es lassen sich durch Architekturarbeit Folge- und Korrekturkosten im Rahmen der Neuentwicklung und Wartung minimieren. Durch gezielten Einsatz von Architektur, zum Beispiel durch Produktfamilien, lassen sich auch konkrete Einsparpotenziale realisieren. Gut getroffene Architekturentscheidungen erhöhen zudem die Flexibilität eines Unternehmens, auf geänderte Umweltbedingungen zu reagieren. Aus diesen Gründen ist Architektur notwendig und definitiv wichtig.

Problematisch stellt sich dar, dass Architekturentscheidungen in der Regel auf Basis unsicherer Information getroffen werden müssen. Sie betreffen gleichzeitig essenzielle Eigenschaften des Systems. Dadurch können sie später schlecht rückgängig gemacht werden. Eine Lösung dieses Problems liegt darin, den Entwurf des Systems so zu gestalten, dass es an den essenziellen Stellen veränderbar bleibt. Dadurch wird die Irreversibilität so mancher Entscheidung aufgehoben. Die Eigenschaften, die zuvor essenziell und daher architekturrelevant waren, sind es jetzt nicht mehr. Es geht sprichwörtlich darum, sich möglichst viele Türen offen zu halten, falls eine Veränderung ansteht. Architektur vermeiden bedeutet Irreversibilität aufheben. Sehr hilfreich ist es auch, wenn man ein System entlang seiner Anforderungen baut, also nur die Architekturtreiber berücksichtigt, die tatsächlich relevant sind. Unsicherheiten können durch frühe Programmierung reduziert werden. Wem das alles immer noch zu kompliziert ist, der geht einfach einkaufen bei einem der großen Anbieter von geschlossenen Baukastenarchitekturen. Das ist aber nicht so spannend, und entspricht auch so gar nicht unserem „Freigeist“: Java ist nun mal eine offene Architektur, ein schöner bunter Blumenstrauß, Mischkultur und keine Monokultur.

Niklas Schlimm ist Leiter des Architekturteams der Provinzial Rheinland Versicherung AG. Er arbeitet als Architekt seit fünfzehn Jahren in EDV-Projekten im Java-Umfeld, meist mit Großrechneranbindung. In diesen Jahren war er in unterschiedlichen Beratungshäusern und Versicherungsunternehmen tätig. Die Provinzial Rheinland gehört zu den führenden deutschen Versicherungsunternehmen und ist Marktführer in ihrem Geschäftsgebiet.

Links & Literatur

[1] Bass, L.; Clements, P.; Kazman, R.: „Software Architecture in Practice“, 2nd Edition, Addison-Wesley, 2003

[2] The Institute of Electrical and Electronics Engineers Standards Board. Recommended Practice for Architectural Description of Software Intensive Systems, IEEE-Std-1471-2000, September 2000

[3] Perry, D. E.; Wolf, A. L.: „Foundations for the Study of Software Architecture“, ACM Software Eng. Notes, vol. 17, no. 4, 1992, pp. 40–52

[4] Parnas, D. L.: „On the Design and Development of Program Families“, IEEE Trans. Software Eng., vol. 2, no. 1, 1976, pp. 1–9

[5] Soni, D.; Nerd, R.; Hofmeister, C.: „Software Architecture in Industrial Applications“, 1995

[6] Kruchten, P.; Obbink, H.; Stafford, J.: „The Past, Present, and Future of Software Architecture“, 2005

[7] Shaw, M.; Garlan, D.: „An Introduction to Software Architecture“, 1993

[8] http://www.in-ag.eu/uploads/media/whoNeedsArchitect.pdf

[9] Lattanze, A.: „The Architecture Centric Development Method“, 2005

[10] http://www.nebulon.com/articles/fdd/latestfdd.html

Hochskalierbare Transaktionsplattformen mit dem LMAX Disruptor

Wenn RAM zu langsam ist

Warum der herkömmliche Architekturansatz hochskalierbarer Transaktionsplattformen im Widerspruch zur Funktionsweise moderner Hardware steht, wird im folgenden Beitrag erläutert. Am Beispiel der London Multi Asset Exchange (LMAX) wird dargestellt, welches Transaktionsvolumen die Java Virtual Machine abwickeln kann, wenn man die ausgetretenen Pfade verlässt.

„The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry."

– Henry Petroski

von Eugen Seer, Oliver Selinger und Sebastian Bohmann

Die LMAX ist ein von der englischen Financial Services Authority (FSA) zugelassenes multilaterales System für den Direkthandel von privaten und institutionellen Anlegern, kurz gesagt: eine offene Börse [1]. Als die LMAX vor einigen Jahren die Entwicklung ihrer Handelssoftware in Angriff nahm, waren folgende nicht funktionale Anforderungen zu erfüllen:

1. Vorhersagbare durchschnittliche Systemlatenz von unter einer Millisekunde, um das finanzielle Risiko wegen der Spreads (Unterschied zwischen Ankaufs- und Verkaufskurs) zu minimieren
2. Abwicklung von Millionen Transaktionen pro Sekunde
3. Hochverfügbarkeit und Ausfallsicherheit
4. Vertretbare Betriebskosten

Contention

Als *Contention* bezeichnet man einen Konflikt beim Zugriff auf exklusive Ressourcen. Dieser tritt auf, wenn verschiedene Threads versuchen, auf dieselbe Ressource zeitnah lesend und schreibend zuzugreifen.

Die Analysen und Performancemessungen der involvierten Ingenieure ergaben, dass alle herkömmlichen Ansätze wie relationale Datenbanken, Java EE, Concurrent Queues oder Aktoren nicht in der Lage waren, das hohe Transaktionsaufkommen auch nur annähernd mit den angeforderten Werten für Durchsatz und Latenz abzuwickeln. Selbst bei weitgehender Vermeidung von I/O-Zugriffen verbrachten die Systeme bei steigender Last mehr Zeit mit der Administration der Nebenläufigkeit als mit der Ausführung der Geschäftslogik [2].

Der Grund lag darin, dass die vielen Threads kontinuierlich von unterschiedlichen Prozessorkernen aus auf dieselben Daten zugriffen. Weil diese Daten und deren Änderungen für alle Prozessorkerne sichtbar sein mussten, erfolgten diese Zugriffe an den Caches vorbei (man spricht hierbei von *Cache Misses*) direkt in das RAM. Dabei ist anzumerken, dass die Geschwindigkeit von RAM-Zugriffen sich seit vielen Jahren nicht entscheidend verbessert hat. Was sich jedoch deutlich weiterentwickelt hat, sind die Caches der Prozessorkerne. Diese sind erheblich schneller und durch die 64-Bit-Adressierung auch größer geworden.

Zusammengefasst nutzen herkömmliche nebenläufige Architekturansätze zwar vordergründig alle Prozessorkerne. Sie scheitern jedoch daran, dass auf

moderner Hardware die durch sie verursachten Cache Misses und RAM-Zugriffe die Performance stark beeinträchtigen.

Concurrent Queues: trügerischer Komfort

Seit Java 5 sind Concurrent Queues ein beliebtes Mittel zur einfachen und komfortablen Kommunikation zwischen zwei oder mehreren Threads, vor allem im Vergleich zu systemnäheren Mechanismen wie beispielsweise Locks.

Eine Queue wird von einem so genannten *Producer* gefüllt und von einem *Consumer* gelesen. Beim Einstellen von Daten in die Queue führt der Producer einen schreibenden Zugriff auf den Speicherbereich der Queue durch. Der Consumer wiederum bedient sich – der Intuition widersprechend – auch eines schreibenden Zugriffs, da er die Daten nicht nur liest, sondern diese auch aus der Queue entfernt. Somit entsteht eine Konkurrenzsituation zwischen dem Producer und dem Consumer, da beide schreibenden Zugriff auf dieselbe Datenstruktur benötigen. Dieser Zugangskonflikt (eine *Contention*, siehe Kasten „Contention“) wird durch Locks gelöst. Allerdings ist das Anfordern eines Locks aufwändig und birgt die Gefahr eines Kontextwechsels im Betriebssystem, durch den die Caches ihre Daten verlieren können.

Abgesehen von den Locks liegt der zweite gravierende Nachteil der Queues in ihrem Speicherlayout. Anfang und Ende der Queue liegen im Speicher zu nahe beieinander in einer Cache Line [3], [5]. Da der Anfang jedoch vom Producer aus einem Thread und das Ende vom Consumer aus einem anderen Thread beschrieben werden, kommt es zu False Sharing und Cache Misses.

Disruptor und Ring Buffer – hardwarenah und schnell

Einen Ausweg bietet das Disruptor Pattern, das an die Funktionsweise von modernen Prozessoren angepasst ist. Ausschließlich ein Thread darf jeweils in einen bestimmten Speicherbereich schreiben. Man spricht hier vom *Single-Writer*-Prinzip. Lesender Zugriff auf diese Speicherbereiche bereitet keine Probleme, da über Inter-Links aktuelle Werte effizient gelesen werden können. Ständige Ausführung des Codes und laufende Speicherzugriffe auf die Daten bewirken zudem, dass diese in den Caches gehalten werden.

Der Disruptor stellt dieselbe Funktionalität wie Concurrent Queues zur Verfügung. Er beruht jedoch auf einem Ring Buffer und verzichtet komplett auf Locks. Der Ring Buffer ist eine Datenstruktur mit optimiertem Speicherlayout, die aus einem Array besteht, das zyklisch adressiert wird (**Abb. 1**). Alle Elemente des Ring

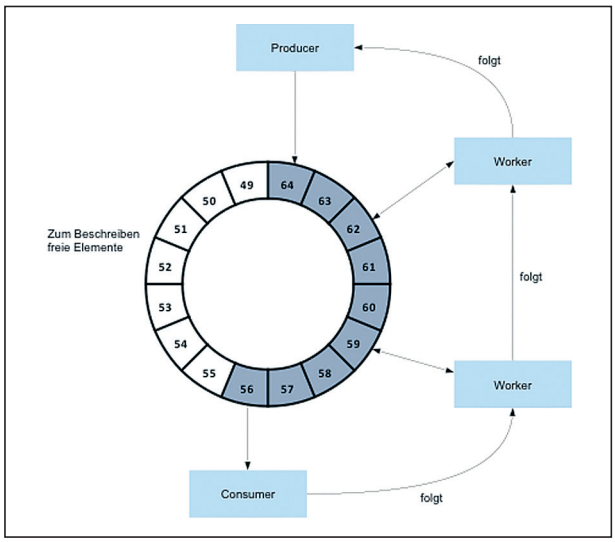


Abb. 1: Ring Buffer mit sechzehn Elementen zur Veranschaulichung

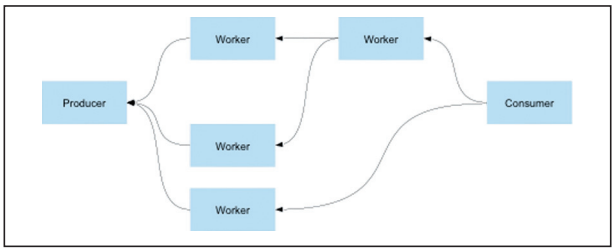


Abb. 2: Abhängigkeiten zwischen Consumer, Workern und Producer als gerichteter Graph

Buffers werden bereits zum Zeitpunkt seiner Erstellung als kontinuierlicher Block im Speicher alloziert. Das ermöglicht die Ausnutzung des Prefetching-Mechanismus des Prozessors. Dieser lädt Daten in den Cache, bevor die CPU diese anfordert, wenn der Speicherzugriff nach einem bestimmten Muster erfolgt. Dies ist bei Blöcken von aneinander gereihten Daten der Fall.

Cache Line Padding

Der Cache in einem modernen Prozessor ist in Cache Lines organisiert, die auf Standardhardware 64 Bytes groß sind. Wenn sich zwei Variablen in derselben Cache Line befinden und von unterschiedlichen Threads beschrieben werden, entsteht ein Zugriffskonflikt, der als *False Sharing* bezeichnet wird. Durch simples Auffüllen (sog. *Padding*) der Cache Lines mit unabhängigen, gleichzeitig beschriebenen Variablen kann False Sharing vermieden werden (siehe auch [3], [4]):

```
private volatile long cursor = INITIAL_CURSOR_VALUE;
public long p1, p2, p3, p4, p5, p6, p7; // cache line padding
```

	Lauf 1	Lauf 2	Lauf 3	Mittelwert
Disruptor	58.111.574	56.909.748	57.706.264	57.575.862
LinkedBlockingQueue	1.964.504	2.077.605	1.990.593	2.010.901
ArrayBlockingQueue	2.282.317	2.242.736	2.273.806	2.266.286

Tabelle 1: Leistungsvergleich Disruptor/Java Concurrent Queues anhand der Transaktionen/s

Vorteil: Die logische Kopplung der Geschäftslogik an die Datenbank wird aufgehoben, der Impedance Mismatch zwischen Objekten und Relationen entfällt.

Die Synchronisation zwischen Producer und Consumer erfolgt im einfachsten Fall ausschließlich über Memory Barriers, die durch volatile Variablen umgesetzt sind [6]. Dieser als *Piggyback Synchronization* bekannte Ansatz erfordert angepasste Algorithmen und Datenstrukturen, da die Garantien nicht so weit gehen wie beim Locking. Im Gegenzug entfällt jedoch bei korrektem Einsatz die gesamte Lock Contention.

Der Algorithmus im Detail

Der Disruptor steuert die gemeinsame Bearbeitung von Daten durch mehrere Threads. Dabei wird immer der Grundsatz eingehalten, dass innerhalb eines Elements im Ring Buffer jede schreibende Instanz nur eine bestimmte Speicherstelle verändert.

Der Producer schreibt Daten, der Consumer liest Daten. Dazwischen können mehrere Worker zur Datenvorverarbeitung geschaltet sein. Es ist möglich, gerichtete Abhängigkeiten zwischen mehreren Workern zu definieren. Beispielsweise muss der Worker für die Deserialisierung vor dem Consumer für die Geschäftslogik an die Reihe kommen.

Producer, Worker und Consumer haben öffentliche Sequenznummern, die das jeweils letzte von ihnen vollständig bearbeitete Element identifizieren und dadurch eine geordnete Abfolge der Bearbeitung ermöglichen. Ein Element wird zuerst vom Producer beschrieben und dann von allen beteiligten Workern vorverarbeitet. Sobald die letzten Worker fertig sind, ist das Element bereit für den Consumer. Dieser und die Worker beobachten die Sequenznummern der anderen Instanzen und setzen ihre Arbeit jeweils bis zum letzten möglichen Element fort. Der Consumer erhöht ebenfalls nach jedem verarbeiteten Element seine Sequenznummer. Der Producer orientiert sich an dieser, um den Consumer im Ring Buffer nicht zu überholen.

Wenn kein Element bearbeitet werden kann, wird der Thread zyklisch in der Größenordnung von zehn bis hundert Mikrosekunden pausiert, bis wieder Elemente zur Verarbeitung verfügbar sind. Dies ähnelt zwar auf den ersten Blick dem klassischen Anti-Pattern „Busy Polling“, ist in diesem Fall aber unkritisch, weil nicht Latenz und Systemlast minimiert werden sollen, sondern ausschließlich der Durchsatz maximiert.

Da der Algorithmus komplett auf klassische Synchronisationsmittel wie Mutual Exclusions und Semaphore verzichtet, entfallen die damit verbundenen Wartezeiten.

Der Consumer hat als einziger Thread Zugriff auf die Geschäftslogik. Hier spielt Thread Safety keine Rolle, und der Code kann ausschließlich auf Geschwindigkeit optimiert werden. Jegliche einfache, aber aufwändige Vorverarbeitung wird also klassisch über Parallelisierung erledigt. Der Consumer kann sich dadurch rein auf die Geschäftslogik konzentrieren.

Listing 1: Anlegen eines Ring Buffers

```
RingBuffer<ValueEntry> ringBuffer = new
                RingBuffer<ValueEntry>(ValueEntry.ENTRY_FACTORY, SIZE,
ClaimStrategy.Option.SINGLE_THREADED, WaitStrategy.Option.YIELDING);
ConsumerBarrier<ValueEntry> consumerBarrier =
                                    ringBuffer.createConsumerBarrier();
BatchConsumer<ValueEntry> batchConsumer = new
            BatchConsumer<ValueEntry>(consumerBarrier, batchHandler);
ProducerBarrier<ValueEntry> producerBarrier =
                        ringBuffer.createProducerBarrier(batchConsumer);
```

Beispielimplementierung und Performancetests

Die Beispielimplementierung setzt je einen stark reduzierten Worker und Consumer ein. Diese ersetzen die Geschäftslogik durch einfache Berechnungen auf der Basis deterministischer Pseudozufallszahlen, die vom Producer in den Disruptor bzw. eine Queue geschrieben werden. Da der Aufwand im Vergleich mit einer komplexen Geschäftslogik mit Zugriff auf große Datenmengen im RAM sehr gering ist, eignet sich dieses Szenario gut, um den maximal erreichbaren Durchsatz abzuschätzen.

Der Code in Listing 1 zeigt das Anlegen eines Ring Buffers, der von einem Producer und einem Consumer zum Austausch von Daten verwendet wird.

Die Performancetests wurden auf einem Rechner mit einem 3,4 GHz Intel Core i7 und 16 GB DDR3-RAM auf Mac OS X 10.7.5 ausgeführt. Sie zeigen, dass bei diesem einfachen Beispiel im Mittel eine Durchsatzverbesserung um den Faktor dreißig herum gemessen wurde.

Fazit

Auf einem Prozessorkern in einem Thread sind bis zu 8,4 Milliarden Instruktionen pro Sekunde auf einer Ivy-Bridge-Mikroarchitektur (bei 2,4 GHz) möglich. Damit können mehrere Millionen Transaktionen pro Sekunde stattfinden, wenn folgende Grundsätze befolgt werden:

1. Verwendung von Cache-freundlichen Datenstrukturen, die wenig Garbage erzeugen.

2. Ausgefeilte Modellierung der Domain durch wohlformulierten objektorientierten Quellcode mit kurzen Methoden und einfachem Kontrollfluss.
3. Kontinuierliche Performance-Unit-Tests zur Verifikation der Annahmen, die den Optimierungen zugrunde liegen.

Daraus ergeben sich folgende zusätzliche Vorteile für die Geschäftslogik [7]:

1. Keine Nebenläufigkeit, denn Parallelisierung wird außerhalb der Geschäftslogik abgehandelt.
2. Alle Daten befinden sich im Speicher; dadurch werden keine synchronen physischen Zugriffe mehr auf die Datenbank benötigt.
3. Die logische Kopplung der Geschäftslogik an die Datenbank wird aufgehoben, der Impedance Mismatch zwischen Objekten und Relationen entfällt.

Ausblick

In einem weiteren Artikel wird das hier gezeigte Beispiel in eine Real-World-Applikation eingebettet. Dabei wird erarbeitet, wie vorteilhaft sich die Kerneigenschaften des Disruptors auf Hochverfügbarkeit und Ausfallsicherheit des Systems auswirken.

Eugen Seer verfügt über mehr als zwölf Jahre Erfahrung bei der Softwareentwicklung in den Bereichen Java Enterprise, Mobile und Agile. Derzeit ist er als freier Softwarearchitekt und Scrum Master für ein Start-up in Wien tätig.

Oliver Selinger ist freiberuflicher Softwarearchitekt und Berater in Wien. Er beschäftigt sich mit skalierbaren verteilten Systemen und Big Data. Sein Fokus liegt derzeit auf der Entwicklung eines hochperformanten Clearing-Systems basierend auf der Java-Plattform im Finanzbereich.

Sebastian Bohmann ist freiberuflicher Softwarearchitekt in Wien mit Schwerpunkt auf Safety, insbesondere Zugsicherungstechnik, und Embedded Systems. Er beschäftigt sich dabei vor allem mit C++, Java und funktionalen Programmiersprachen.

Links & Literatur

[1] http://de.lmax.com/exchange-handel

[2] http://www.infoq.com/presentations/LMAX

[3] http://mechanitis.blogspot.co.at/2011/07/dissecting-disruptor-why-its-so-fast_22.html

[4] http://mechanical-sympathy.blogspot.co.at/2011/07/false-sharing.html

[5] http://lmax-exchange.github.com/disruptor/files/Disruptor-1.0.pdf

[6] http://docs.oracle.com/javase/specs/jls/se5.0/html/memory.html

[7] http://martinfowler.com/articles/lmax.html

Android rockt die MTC und die JAX 2013

Beweglicher Androide

Viele Jahre legte Google bei der Entwicklung von Android Wert auf Performance und Funktionalität. Als Folge davon blieben Design und Usability auf der Strecke. Android galt als cool und geeky, aber hässlich. Das änderte sich mit Ice Cream Sandwich und den Design Guidelines. Inzwischen ist Android UI so stark in den Fokus getreten, dass es ein vorherrschendes Thema in den Android-Tracks der MobileTech Conference 2013 [1] und der JAX 2013 [2] geworden ist. Und das ist auch gut so.

von Claudia Fröhling

Denn es gibt viel zu erzählen und zu diskutieren in Sachen Android UI. Zum einen haben Entwickler seit Ice Cream Sandwich ganz neue Möglichkeiten für ihre Apps erhalten, die Umsetzung ist aber noch nicht überall gelungen. Die Fragmentierung wird oft als Pitfall herangezogen – da müssen Webentwickler dann oft müde lächeln. Schließlich bauen sie seit vielen Jahren Anwendungen, die auf verschiedenen Browsern und Bildschirmgrößen funktionieren müssen. Heutzutage sind Webseiten responsiv, um auf dem Desktop **und** auf dem mobilen Gerät optimal zu laufen. Und genau das muss auch für Android-Apps möglich sein. Wir haben das alles im Web schon einmal durchgemacht, warum also jetzt das Rad neu erfinden?

Responsive Design für Android

Kein Problem, sagt Juhani Lehtimäki von Snapp TV, der den Android Day auf der MTC eröffnete. Android-Apps müssen von den Erfahrungen der Web-Apps und -entwickler profitieren und dieselben Prinzipien studieren. Denn mit den Tools, die für Android verfügbar sind, lassen sich heute Apps bauen, die auf jedem Device laufen, egal welche Screengröße vorliegt. Juhani muss es wissen, schließlich beschäftigt er sich seit über zehn Jahren mit Android und betreut mit dem Blog www.androiduipatterns.com eine der wichtigsten Quellen zum Thema UI Design Patterns. Es gibt keine Android-App, die Juhani noch nicht in den Fingern hatte.

Mit Responsive Design müssen sich Android-Entwickler beschäftigen, so Juhani, und die Plattform hilft ihnen dabei. Man müsse weggehen von der Prämisse, es gäbe auf der einen Seite Smartphone-Apps und auf der anderen Seite Tablet-Apps. Eine App für alles, das ist das Ziel. „There shouldn't be such thing as an Android phone app", so Juhani in seinem Talk. Interessant: Im Raum saßen zu neunzig Prozent Android-Entwickler, mit Responsive hatten sich aber laut Handzeichen maximal zehn Prozent beschäftigt.

Google macht uns vor, wie responsive Apps aussehen müssen: Google Play Store, Google Mail, YouTube-App – das sind alles Beispiele für Apps, die überall und immer gut aussehen. Wie erwähnt ist das alles kein Hexenwerk, sondern mit den Mitteln umsetzbar, die Android heute bietet. So gibt es beispielsweise die Funktion, zwei Items nebeneinander darzustellen, wenn der User sich in der Landscape View eines Tablets befindet. Wechselt er auf das Smartphone, wird nur noch ein Item angezeigt, nach Tap auf einen Punkt gelangt man dann in Item 2. Genau so funktioniert auch die Google-Mail-App auf dem Smartphone.

Eine weitere Möglichkeit von Android ist die Grid View, wie z. B. in der YoutTube-App zu sehen. Der Entwickler/Designer entscheidet zu Beginn, wie viele Columns das Grid haben soll. Durch einstellen des *stretchmode* auf *columnWidth* werden diese Grids dann automatisch an die jeweilige Screengröße angepasst. Columns auf dem Smartphone werden vertikal dargestellt.

Am Ende gab Juhani den Teilnehmern der MTC noch einen Tipp mit auf den Weg: Tablets First. Konzipiert eure App als Erstes für das Tablet und wandelt von dort die Version für das Smartphone ab. Dieser Weg ist, so Juhani, um Längen unkomplizierter als umgekehrt. Sicherlich ist die Einstellung recht extrem, nichtsdestotrotz eine valide Vorgehensweise.

Supercharge your Android UI

Wie anfangs erwähnt hatte iOS lange Zeit die Nase vorn im mobilen Business, weil iOS-Apps rein objektiv betrachtet besser aussahen und ergonomisch besser bedienbar waren. Apple hatte von Anfang an darauf geachtet, dass jede App, die in den Store gelangt, zu hundert Prozent die iOS-UX-Paradigmen erfüllt. Wer schon mal eine App für iPhone oder iPad entwickelt hat, kennt die Schmerzen, die mit dem Apple-Reviewprozess verbunden sind.

Der Erfolg einer App hängt maßgeblich davon ab, wie sie sich dem Nutzer präsentiert. Um Android-Apps einen außergewöhnlichen Look zu verleihen, steht unter anderem das Android Graphics Framework zur Verfügung. Dominik Helleberg und Jonas Gehring (inovex) präsentierten auf der JAX die Möglichkeiten des Frameworks in ihrem Talk, und sprachen über Performance und Best Practices. Mit dem Framework lassen sich zum Beispiel Animationen und 3-D-Effekte in der App einbauen. Mehr Informationen gibt es auf der Graphics-Seite von Google [3].

Apropos Animationen: Lars Vogel und Sergej Shafarenka präsentierten auf der JAX eindrucksvoll, wie man das Properties-API und die Hintergrundbearbeitungsfähigkeiten von Android verwenden kann, um reaktive und schöne User Interfaces zu generieren.

Android UIs für alle

Mehr über Responsive Design lernten MTC-Teilnehmer von Andreas Hölzl (canoo): Seit Honeycomb gibt es das neue Feature „Fragments“, das die Smartphone- und Tabletentwicklung vereinen soll. Die jüngste Qualitätsoffensive von Google, gezielt App-Entwickler für die Tabletentwicklung zu gewinnen, zeigt, dass es im Bereich der Cross-Device-UI-Entwicklung noch Informationsbedarf gibt. Andreas stellte das Fragmentkonzept vor und konzentrierte sich dabei auf Tablet-UIs. Wichtig war ihm, festzuhalten, dass man auf der Android-Plattform nicht (mehr) von Fragmentierung, sondern eher von Differenzierung sprechen sollte. Derzeit gibt es 2 244 unterschiedliche Geräte für eine einzige Codebasis, und das sollte man als Chance und nicht als Problem wahrnehmen. Wir sprachen vor seinem Talk mit Andreas, das Video gibt es unter [4]. Und wer sich in dieses Thema einlesen will: Andreas hat einen spannenden Artikel für das Mobile Technology Magazin 1.2013 geschrieben [5].

Fazit

Die Android Tracks der MobileTech Conference Spring und der Android Day der JAX waren der Auftakt in ein spannendes Jahr für Googles mobile Plattform. Jetzt liegt der Ball bei Google und die Entwicklergemeinde hofft inständig, dass die anstehende Google I/O viel Schwung in die Plattform bringen wird. Über die Ergebnisse von Googles Hauskonferenz sprechen wir dann auf der nächsten MobileTech Conference im September!

Links & Literatur

[1] www.mobiletechcon.de

[2] www.jax.de

[3] http://developer.android.com/guide/topics/graphics/overview.html

[4] http://bit.ly/15hoKZ6

[5] http://it-republik.de/it/sonderhefte/mobile360/

Miracast: Mehrere Bildschirme mit Jelly Bean

Android für Couch Potatoes

Obwohl auf der Google I/O 2013 wohl wieder ein Nachfolger zu Android 4.2 vorgestellt wurde, stecken selbst aktuelle Features teilweise noch in den Kinderschuhen. Jelly Bean (4.2) erlaubt etwa den Betrieb von Android-Geräten mit mehreren Bildschirmen und insbesondere mit Miracast-tauglichen Empfangsgeräten, aber welche Funktionalität ist mit den neuen Schnittstellen möglich? Was ist Miracast und warum ist es für Android-Entwickler interessant? Welche Geräte sind überhaupt verfügbar und wie sind die ersten Erfahrungen mit Miracast? Dieser Artikel möchte einen ersten Überblick über die durchaus spannenden Möglichkeiten von Android 4.2 in diesem Umfeld geben.

von Daniel Bälz und Christian Meder

Man darf das Verhältnis von Hardware und Software im Android-Umfeld durchaus als emotionales Wechselbad der Gefühle bezeichnen. Eine riesige Auswahl an hochgradig unterschiedlichen Geräten in allen erdenklichen Farben, Formfaktoren und Ausstattungen zusammengeführt in einer einheitlichen App-Plattform, für jeden Geschmack etwas (*one ring to rule them all,* wie böse Zungen behaupten). Auf der anderen Seite eine Zersplitterung und Verästelung, die es immer schwieriger macht, neue Funktionalitäten zur Android-Plattform hinzuzufügen und annähernd zeitnah auch mit einer breiten Hardwareunterstützung aufwarten zu können. Bestes Beispiel dieses Konflikts ist das *Presentation*-API von Android 4.2 und der damit eng verbundene Miracast-Standard.

Was ist Miracast?

Die englischsprachige Wikipedia-Seite beschreibt Miracast in einem schönen Vergleich als ein drahtloses HDMI-Kabel (*High Definition Multimedia Interface*) zur Übertragung des Bildschirminhalts eines Geräts auf einen anderen Bildschirm inklusive Implementierung der Kopierschutzmaßnahmen durch die Emulation von HDCP (*High-Bandwidth Digital Content Protection*): ein einfach verwendbarer kabelloser Ersatz für den Anschluss eines Geräts an einen Bildschirm also, vereinfacht dargestellt [1], [2]. Ein Gerät muss als Bildquelle dienen (*source*), deren Bildschirminhalt mitgeschnitten wird und ein anderes Gerät, der Empfänger (*sink*), zeigt die empfangene Aufzeichnung dann an.

In **Abbildung 1** ist das Verfahren schematisch dargestellt und man sieht, dass bekannte Codecs zur Kodierung der Inhalte verwendet werden. Grafische Inhalte werden in H.264 verpackt, Audiodaten als LPCM oder auch AAC bzw. AC3 (*Dolby Digital*) kodiert. Alle Daten werden in einem MPEG-2 TS (*transport stream*) gemischt, ein Format, das auch in der digitalen Ausstrahlung von Inhalten per Satellit und Kabel (DVB-S/C/T) oder auch als Bestandteil des Blu-ray-Formats verwendet wird.

Für den Transport der Daten baut Miracast auf bereits durch die Wi-Fi Alliance zertifizierte Technologien auf,

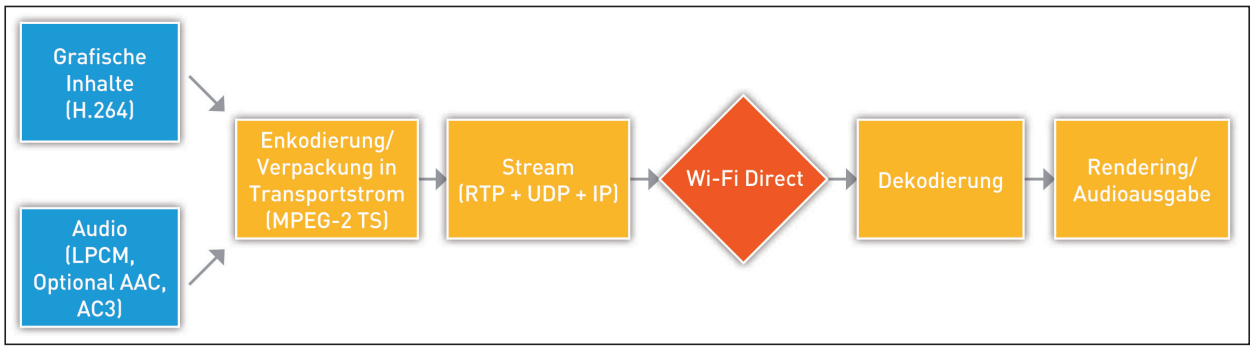


Abb. 1: Prinzipieller Aufbau des Miracast-Ablaufplans

um die Implementierung zu erleichtern und somit eine möglichst schnelle Verbreitung sicherzustellen. Durch die Zertifizierung soll die Interoperabilität zwischen den Geräten sichergestellt und damit ein fehlerloser Betrieb gewährleistet werden:

- Als Standard für drahtlose Netzwerke zwischen Sender und Empfänger wurde IEEE 802.11n gewählt. Für den Einsatz mit Miracast muss die Implementierung durch die Wi-Fi Alliance als „Wi-Fi CERTIFIED n" zertifiziert sein.
- Der Datenverkehr wird mit „Wi-Fi Protected Access 2" (WPA2) gesichert. Dies stellt Teile der Funktionalität des IEEE-802.11i-Standard bereit.
- Mit „Wi-Fi Multimedia" (WMM) wird Quality of Service bereitgestellt, sodass eine Priorisierung des Datenverkehrs über den Steuerverkehr möglich ist.
- Um drahtlose Verbindungen einfach aufzubauen, wird „Wi-Fi Protected Setup" (WPS) verwendet, welches für die Erkennung der Geräte untereinander und den Verbindungsaufbau über eine mit WPA2 gesicherte Verbindung zuständig ist.
- Nach dem Verbindungsaufbau nutzt Miracast „Wi-Fi Direct", um zwischen den Geräten zu kommunizieren. Es wird eine direkte Verbindung zwischen den zu kommunizierenden Geräten, ohne Nutzung eines Access Points, durchgeführt.
- Optional unterstützt Miracast mit „WMM Power Save" Stromsparmodi, welche den Akkuverbrauch der beteiligten Geräte senken. Falls Sender und Empfänger auf demselben Access Point verbunden sind, können sie mittels „Tunneled Direct Link Setup" (TDLS) eine direkte Verbindung, ohne die Nutzung von „Wi-Fi Direct", aufbauen.

Betrachtet man sich die verwendeten Technologien, so wird klar, dass einer Unterstützung für Miracast auf breiter Front aus technischer Sicht wenig entgegensteht. Die verwendeten Wi-Fi-Technologien (802.11n, WPA2, WPS, Wi-Fi Direct) sind gang und gäbe in einer Vielzahl von Geräten vom Fernseher und Blu-ray-Abspieler bis zu mobilen Geräten aller Art (Tablets, Smartphones). Die Dekodierung von Inhalten in der Kombination aus MPEG-2 TS mit H.264 und LPCM ist Bestandteil typischer hardwarebeschleunigter Medienabspieler aller Plattformen. Anspruchsvoller wird es auf der sendenden Seite: Die Hardware des Senders muss in der Lage sein, neben der „normalen" Anzeige der Inhalte auf seinem lokalen Bildschirm, den Bildschirminhalt annähernd zeitgleich zu kodieren, zu verpacken und mit schnellem Durchsatz drahtlos an den Empfänger zu versenden. Das klingt doch nach zusätzlicher Prozessorunterstützung für das Kodieren und Verpacken, sowie moderner Wi-Fi-Hardware, um die Übertragungsrate zu sichern.

Hardwareunterstützung für Miracast

Die aktuelle Referenzkombination, die wir ebenfalls in unseren Tests verwendet haben, ist sicherlich die Kombination aus einem Google *Nexus 4* auf der Senderseite und dem Netgear *Push2TV PTV3000* auf der Empfängerseite. Während das Nexus 4 auch in Deutschland verfügbar ist, mussten wir den *PTV3000* noch über den Umweg via amazon.com bestellen. Das Nexus 4 sollte aus einem früheren Artikel dieser Reihe noch hinreichend bekannt sein, interessant ist hier nur nochmals, an die 4-Kern-Prozessorarchitektur zu erinnern, die zusätzliche Kapazitäten für die Last der Kodierung der Audio- und Videodaten im Miracast-Betrieb bietet.

Beim PTV3000 handelt es sich um einen Empfänger für Miracast, der bereits von der Wi-Fi Alliance für diesen Einsatzzweck zertifiziert wurde. Die durch den PTV3000 per drahtloser Verbindung empfangenen Inhalte können über einen HDMI-Ausgang an ein Anzeigegerät wie einen Monitor, TV-Bildschirm oder einen Beamer weitergegeben werden.

Der PTV3000 war schon vor der Veröffentlichung von Miracast erhältlich und konnte damals bereits als Empfänger für Intel Wireless Display verwendet werden. Mit der Veröffentlichung von Miracast wurde der Empfänger zusätzlich für diesen Standard zertifiziert und konnte ab der Firmware 2.2.7 auch für diesen Einsatz verwendet werden. Das vorliegende Gerät wurde zuerst auf die zu diesem Zeitpunkt erhältliche Firmware 2.2.12 aktualisiert. Inzwischen wurde durch Netgear die Firmware 2.2.15 veröffentlicht, welche nun auf dem PTV3000 zum Einsatz kommt.

In unseren ersten Tests lief die Kombination aus Nexus 4 und PTV3000 auch bei längeren Videoübertragungen stabil. Auch Tests mit (3-D-)Spielen verliefen tendenziell positiv, allerdings sind hier gelegentlich Hänger oder größere Latenzen zu beobachten. Es ist aktuell schwierig zu beurteilen, inwiefern sich diese

Probleme durch weitere Softwareupdates lösen lassen oder ob es sich dabei tatsächlich um Beschränkungen der verwendeten Hardware handelt.

Grundsätzlich unterstützt Android 4.2 auf Plattformebene Miracast [3] und bietet daher auch mit API-Level 17 das *Presentation*-API zur Ansteuerung zusätzlicher Bildschirme an [4]. Miracast kann daher von Geräten mit geeigneter Hardware und Jelly Bean 4.2 verwendet werden. In der Praxis gibt es jedoch zusätzlich Geräte, die auch mit 4.1 bereits Miracast als reine Bildschirmspiegelung unterstützen.

Daneben unterstützt Intel Wireless Display (WiDi) als Source ab Firmware 3.5 das Senden an Miracast-Empfänger und einige WiDi-Empfänger (wie das PTV3000) sind entsprechend aktualisiert worden bzw. werden es, damit sie Miracast empfangen können. Alle zertifizierten Sender (Source) inklusive Referenzplattformen der Hardwarehersteller sind unter [5] zu finden. Bekannte Android-Geräte, die Miracast als Sender unterstützen, sind:

- Google Nexus 4
- HTC One
- LG Optimus G
- Samsung Galaxy S III (Android 4.1, Miracast über „AllShare Cast“)
- Samsung Galaxy S4
- Samsung Galaxy Note II
- Sony Xperia Z (Qualcomm APQ8064 wie Nexus 4, hat laut Homepage Android 4.1 und Miracast [6])
- Galaxy Note 10.1 (Tablet, Miracast über „AllShare Cast“)
- Asus PadFone Infinity (Hybrid Tablet + Smartphone)

Chipsätze, die Miracast als Sender unterstützen, sind:

- Qualcomm Snapdragon S4 (8960 MTP3.1 ICS ist zertifiziert, damit auch der APQ8064 des Nexus 4)
- Tegra laut Whitepaper von Nvidia [7], allerdings unterstützt das Nexus 7 kein Miracast
- Broadcom BCM943236USB 802.11 a/b/g/n Wireless Adapter (USB-Adapter)
- Intel Centrino Advanced-N 6235 AGN (Wi-Fi-/Bluetooth-Modul, bisher benutzt für WiDi)

Samsung unterstützt mit neueren Geräten ebenfalls Miracast [8], hatte aber bereits das proprietäre Samsung AllShare Cast mit älteren Android-Versionen für das einfache „Bildschirm spiegeln“ unterstützt [9]. Daher ist die klare Differenzierung im Samsung-Umfeld zwischen den unterschiedlichen Android-Versionen und Smartphone-Versionen aktuell äußerst schwierig. Alle zertifizierten Empfänger (Sink/Display) sind unter [10] zu finden.

Die Liste ist gegenwärtig schon beachtlich lang, insbesondere, da viele Typenvarianten einzeln aufgeführt

Listing 1: Ausschnitt der Verwendung des Presentation-API

```
private ExternalPresentation mPresentation;
private SelectionMethod mSelectionMethod =
                                    SelectionMethod.DISPLAY_MANAGER;

private void initDisplay(SelectionMethod selection) {
  Display display;
  if (selection == SelectionMethod.DISPLAY_MANAGER) {
    DisplayManager displayManager = (DisplayManager)
                            getSystemService(Context.DISPLAY_SERVICE);
    Display[] displays = displayManager.getDisplays(DisplayManager.
                                    DISPLAY_CATEGORY_PRESENTATION);
    if (displays.length == 0) {
      return;
    }
    display = displays[0];
  } else if (selection == SelectionMethod.MEDIA_ROUTER) {
    MediaRouter mediaRouter = (MediaRouter)getSystemService(Context.
                                            MEDIA_ROUTER_SERVICE);
    MediaRouter.RouteInfo route = mediaRouter.
                getSelectedRoute(MediaRouter.ROUTE_TYPE_LIVE_VIDEO);
    display = route.getPresentationDisplay();
    if (display == null) {
      return;
    }
  } else {
    return;
  }

  if (mPresentation != null) {
    mPresentation.dismiss();
  }
  mPresentation = new ExternalPresentation(this, display);
  mPresentation.show();
}

private final class ExternalPresentation extends Presentation {
  public ExternalPresentation(Context context, Display display) {
    super(context, display);
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.presentation_display);
    TextView tv = (TextView)findViewById(R.id.external_tv);

    Point resolution = new Point();
    getDisplay().getSize(resolution);
    String metric =  "Auflösung: " + resolution.x + "x" + resolution.y;
    String displayID = "Display ID: " + getDisplay().getDisplayId();
    tv.setText(displayID + "\nName: " + getDisplay().getName()
                                                + "\n" + metric);
  }
}
```

sind. Vor allem Fernseher, Blu-ray-Spieler und Set-top-Boxen einiger großer Hersteller sind hier zu finden. Die meisten Geräte sind allerdings nicht oder noch nicht in Europa lieferbar. Hervorzuheben sind an dieser Stelle vor allem LG, die bereits viele Geräte aus dem Bereich Blu-ray und TV zertifiziert haben [11].

Weitere sehr spannende technische Erfahrungswerte mit verschiedenen Empfängern und Sendern aus der *Cyanogenmod*-Community sind unter [12] beschrieben.

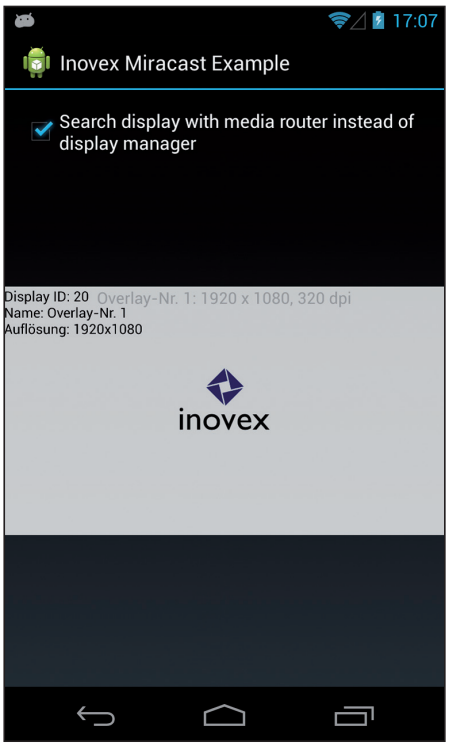


Abb. 2: Zweiten Bildschirm per Overlay simulieren

Presentation-API

Aus reiner Entwicklersicht gestaltet sich die Verwendung zusätzlicher Bildschirme erfreulicherweise erstaunlich einfach. Unabhängig davon, ob per Miracast oder traditionellem Kabel angeschlossen, bietet das mit API 17 eingeführte *Presentation*-API prinzipiell zwei Wege, zusätzliche Bildschirme anzusteuern [4]. Zum einen kann man den *MediaRouter* befragen, welches der aktuell präferierte Bildschirm für einen bestimmten Inhaltstyp (Audio/Video) ist. Hierbei überlässt man der Plattform die Entscheidung für den am besten geeigneten Bildschirm.

Der zweite Weg nutzt den *DisplayManager*, um vom System eine Aufzählung aller aktuell verbundenen Bildschirme und ihrer Eigenschaften zu erhalten.

Beide Wege sind beispielhaft in Listing 1 skizziert. Ein einfaches Beispiel ist auch unter [13] zu finden. Beide Bildschirme können hierbei über das einfache „Bildschirm spiegeln“ auch unterschiedliche Darstellungen wählen. Damit sind beispielsweise angereicherte Informationen auf dem Tablet und eine reine Präsentationsansicht auf dem großen Bildschirm denkbar.

Sehr hilfreich ist die Entwickleroption unter Einstellungen | Entwickler Optionen | Zeichnung | Sekundäre Displays simulieren. Wie in **Abbildung 2** zu sehen ist, ist es damit möglich, einen Overlay mit der Anzeige des zusätzlichen Bildschirms auf dem gleichen Gerät zu simulieren, was die initiale Entwicklung ohne zusätzliche Hardware ermöglicht.

Fazit

Miracast hat zahlreiche Vorteile auf seiner Seite: basierend auf offenen, bestehenden, verbreiteten Standards für Transport und Codecs der Daten, Zertifizierung durch eine bekannte und anerkannte Organisation, breite Unterstützung vieler großer Hersteller aus dem Bereich mobiler Endgeräte und Audio/Video Home Entertainment. Auch das *Presentation*-API macht einen guten Eindruck, bietet es doch zahlreiche Möglichkeiten mit seiner Trennung in zwei unterschiedlich ansteuerbare Displays. Aber das alles sind aktuell noch Versprechen auf die Zukunft. De facto sind eigentlich keine Empfänger offiziell in Deutschland verfügbar. Updates auf Jelly Bean 4.2 haben noch unübersehbare Schwächen in der Umsetzung des *Presentation*-API. Die Lage bei den sendenden Android-Geräten ist unübersichtlich. Gerätehersteller halten an ihren eigenen Lösungen fest.

Wie schrieb Andrew Dodd in [12]: „reminds me ... of Bluetooth and even 802.11 ... in their early days“. Dann hoffen wir, dass Miracast im Laufe des Jahres 2013 genauso selbstverständlich werden wird, wie Bluetooth und Wi-Fi es heute sind. Das Versprechen steht im Raum.

Daniel Bälz studiert Informatik an der Hochschule Karlsruhe. In seiner Bachelorarbeit bei der inovex GmbH beschäftigt er sich mit dem Einsatz von Miracast unter Android.

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] http://en.wikipedia.org/wiki/Miracast

[2] http://www.wi-fi.org/wi-fi-certified-miracast%E2%84%A2

[3] http://www.android.com/whatsnew/

[4] http://developer.android.com/reference/android/app/Presentation.html

[5] http://bit.ly/UJjMfQ

[6] http://www.sonymobile.com/de/products/phones/xperia-z/specifications/#tabs

[7] http://blogs.nvidia.com/2012/07/tegra-enhances-miracast-wireless-display-on-hdtvs

[8] http://bit.ly/OZWhfa

[9] http://developer.samsung.com/allshare-framework

[10] http://bit.ly/10fkS9p

[11] http://de.slideshare.net/droidcon/droidcon2013-miracast-final2

[12] https://plus.google.com/101093310520661581786/posts/bWVtgq78aRv

[13] http://blog.stylingandroid.com/archives/1394

[14] http://www.goodreads.com/author_blog_posts/4057300-android-4-2-hdmi-presentation-support

API zum Laden von Daten: Loader

Daten richtig laden

GOOD TO KNOW

Daten, die in einer Android-App angezeigt werden, müssen irgendwoher kommen. Dieses „irgendwo“ ist selten initial im Speicher verfügbar und muss daher von einer externen Datenquelle wie Datei, Datenbank oder Netzwerk geladen werden. Damit dies nicht zu einer langsam reagierenden App führt oder noch schlimmer, zu dem Dialog „Application not responding“, sollte das Laden in einem Hintergrundthread passieren.

von Lars Röwekamp und Arne Limburg

Jeder, der den Code einer App von Android Version 2.x auf Version 3 oder 4 aktualisiert, wird verschiedene Stellen im Code finden, die auf einmal von der IDE (jedenfalls von Eclipse) durchgestrichen werden. Ein Major-Versionssprung geht in Android eigentlich immer damit einher, dass einige Methoden auf *deprecated* gesetzt werden, was zu dem beschriebenen Phänomen führt. Ein häufig anzutreffendes Stück Code dieser Art ist die Activity-Methode *startManagingCursor*. Diese ist seit Android 3 *deprecated* und das aus gutem Grund.

Probleme von „managed“ Cursors

Dabei könnte es so einfach sein: Man möchte Daten aus der Datenbank anzeigen und diese sollen sich automatisch aktualisieren, wenn sich der Datenbankinhalt ändert. Die Methoden *startManagingCursor* und *managedQuery* (welche eine Query absetzt und dann *startManagingCursor* aufruft) bieten hierfür tatsächlich einen ebenso einfachen, wie funktionierenden Mechanismus an: Der Cursor der Query wird an den Lebenszyklus der Activity gebunden. Das heißt, wenn die Activity gestoppt wird, wird der Cursor deaktiviert und beim Zerstören der Activity wird auch der Cursor automatisch geschlossen. Damit nicht genug, zusätzlich werden jedes Mal, wenn die Activity wieder in den Vordergrund geholt wird, die Daten automatisch aktualisiert. Im Zusammenspiel mit dem *SimpleCursorAdapter* ist es sogar möglich, dass die Daten direkt bei der Änderung aktualisiert werden. Wo ist also das Problem mit diesem Mechanismus?

- Problem 1: Das Laden der Daten geschieht nicht, wie in der Einleitung empfohlen, im Hintergrund sondern direkt im UI-Thread. Bei kleinen Datenmengen ist das zwar in der Regel nicht so dramatisch, führt aber in jedem Fall zu einem langsamer reagierenden UI.
- Problem 2: Jedes Mal, wenn die Activity in den Vordergrund geholt wird, werden die Daten automatisch neu geladen. Auch das ist bei näherer Betrachtung recht ungeschickt, weil dadurch die betroffene Activity grundsätzlich langsamer angezeigt wird, auch wenn sich die angezeigten Daten gar nicht geändert haben.
- Problem 3: Auch das Zusammenspiel mit dem *SimpleCursorAdapter* ist problematisch, weil es dazu führt, dass das UI unvorhersehbar stockt, nämlich jedes Mal, wenn im Hintergrund die Daten geändert werden und der *SimpleCursorAdapter* daher ein *requery()* vornimmt.

All diese Probleme haben Google dazu bewogen, seit Android 3 einen anderen Weg zu gehen, wenn es darum geht, Daten für das UI zu laden.

Neues API zum Laden von Daten

Seit Android 3 gibt es mit dem *Loader*-Interface [1] und *LoaderManager* ein neues API, um in Android Daten zu laden. Es kann über die Compatibility Library [2] auch in älteren Android-Versionen eingesetzt werden.

Die Idee von Loadern ist eine Entkopplung des Ladens der Daten vom Anzeigen. Dadurch kann transparent eine Asynchronität zwischen Laden und Anzeigen implementiert werden. Damit das nicht jedes Mal neu passieren muss, wird in Android eine fertige Implementierung mitgeliefert: Der *AsyncTaskLoader*. Er verwendet intern einen *AsyncTask*, um das Laden der Daten im Hintergrund durchzuführen. Eigene Implementierungen des *Loader*-Interface sollten immer von *AsyncTaskLoader* ableiten.

Ein Loader kann drei Status annehmen: gestartet, gestoppt und zurückgesetzt (reset). Gestartet bedeutet, dass der Loader läuft und Daten lädt. Wenn ein Loader gestoppt ist, lädt er keine Daten, kann aber auf Datenänderungen hören und sich ggf. selbst wieder in den Status „gestartet“ versetzen. Ein Loader, der zurückgesetzt ist, sollte jegliche Daten, die er enthält, aufräumen, weil es sein kann, dass er nie wieder aufgerufen wird. Auch aus diesem Status heraus kann er wieder gestartet werden.

Die Bedienung eines Loaders kann asynchron aus dem UI-Thread heraus über den *LoaderManager* erfolgen. Dieser hat z. B. die Methode *initLoader*, um einen Loader zu erzeugen und (potenziell asynchron) zu starten. Weitere Methoden gibt es für das Restarten und das Zerstören

eines Loaders. Der *LoaderManager* kümmert sich auch um die Anbindung an den Lebenszyklus der Activity.

Damit die fertig geladenen Daten im UI angezeigt werden können, muss eine Kommunikation zwischen dem asynchron laufenden Loader und dem UI-Thread erfolgen. Auch hier greift die oben beschriebene Entkopplung und zwar über das *LoaderCallbacks*-Interface des *LoaderManagers*, das drei Methoden enthält, um der Activity erstens zu ermöglichen, einen Loader zu erzeugen (*createLoader*) und um sie zweitens (im UI-Thread) über Zustandsänderungen zu informieren (*onLoadFinished*, *onLoaderReset*).

Vom „Managed Cursor" zum Loader

So weit so gut. Aber wie muss dieses Pattern in der Praxis angewendet werden? Was muss ich tun, wenn ich bisher die Methode *startManagingCursor* oder *managedQuery* verwendet habe. Listing 1 zeigt exemplarisch für eine *ListActivity*, wie ein solcher Code in der Regel aussieht. Der Code lädt bereits in der *onCreate*-Methode die Daten aus der Datenbank, befüllt den *SimpleCursorAdapter* mit den Daten und setzt diesen als Adapter für die *ListView*.

Der Trick besteht nun darin, in der *onCreate*-Methode die Liste zunächst mit einem leeren Adapter zu füttern und das Laden im Hintergrund zu starten. Das tatsächliche Befüllen der Daten erfolgt dann in den bereits erwähnten Callbacks über *swapCursor* (Listing 2).

Die in Listing 2 verwendete *LOADER_ID* ist übrigens dazu da, den korrekten Loader zu identifizieren, falls man innerhalb einer Activity mehr als einen Loader verwenden möchte. Ansonsten kann sie auf einen fixen Wert, z. B. 1, gesetzt werden. Da sich nun der *CursorLoader* um das Aktualisieren der Daten kümmert, sollte man gleichzeitig dafür sorgen, dass der *SimpleCursorAdapter* dies nicht mehr tut. Das geschieht durch die Verwendung des korrekten Konstruktors. Der richtige ist der, der als letztes Argument einen int namens *flags* erwartet. Der andere Konstruktor wurde auch mit Einführung des *Loader*-Interface auf *deprecated* gesetzt. Als *flags* sollte hier dann 0 übergeben werden.

Listing 1

```
public class MyListActivity extends ListActivity {
  ...
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Cursor cursor = managedQuery(CONTENT_URI, PROJECTION, ...);
    SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
                                          R.layout.list_item, cursor, ...);
    setListAdapter(adapter);
  }
}
```

Listing 2

```
public class MyListActivity extends ListActivity
      implements LoaderManager.LoaderCallbacks {
  ...
  private SimpleCursorAdapter adapter;

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    adapter = new SimpleCursorAdapter(this, R.layout.list_item, null, ...);
    setListAdapter(adapter);
    getLoaderManager().initLoader(LOADER_ID, null, this);
  }

  public Loader<Cursor> onCreateLoader(int id, Bundle bundle) {
    return new CursorLoader(this, CONTENT_URI, PROJECTION, ...);
  }

  public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
  }

  public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
  }
}
```

Fazit

Über den in Android 3 eingeführten Mechanismus der Loader ist das asynchrone Laden von Daten möglich, ohne dass sich der Entwickler über die Behandlung der Asynchronität Gedanken machen muss. Daten werden automatisch im Hintergrund geladen und auch die Kommunikation mit dem UI-Thread erfolgt automatisch. Der mitgelieferte *CursorLoader* stellt eine direkte Anbindung an einen *ContentProvider* zur Verfügung, inklusive Datenänderungserkennung. Zusammen mit dem *LoaderManager*, der das Zusammenspiel mit dem Lebenszyklus der Activity behandelt, bietet der *CursorLoader* denselben Komfort, den vorher managed Cursors geboten haben. Der entscheidende Unterschied ist allerdings, dass Daten nun nicht mehr im UI-Thread sondern asynchron geladen werden. Wer seine Daten nicht über einen *ContentProvider* laden möchte, hat jederzeit die Möglichkeit, einen eigenen Loader zu implementieren [3], da das Loader-Interface generisch gehalten und somit nicht auf Cursor beschränkt ist.

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

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/components/loaders.html

[2] http://developer.android.com/tools/extras/support-library.html

[3] http://bit.ly/SNWMrX

Vorschau auf die Ausgabe 8.2013

Big Data konkret

Big Data ist eines der gefürchtetsten Buzzwords in unserer Welt, nicht immer passen Erwartung und Realität hier zusammen. Wer muss heutzutage tatsächlich schon mit „Big Data“ hantieren? Sogar Giganten wie Facebook und Yahoo! kommen mit Gigabytes aus. Die Kunst liegt darin, die *richtige* Art von Daten zu sammeln, nicht die schiere Masse. Um Big Data aus der Metaebene herauszuholen, in der es oft heute steckt, wollen wir uns nächsten Monat etwas konkreter mit Beispielen beschäftigen: Ramon Wartala wird uns zeigen, wie man seinen eigenen Hadoop-Cluster auf Basis eines Raspberry-Pi-Klons aufsetzt und Peter Siemen erklärt, wie die Open-Source-Bibliothek Cascading die Produktivität im Schreiben von Hadoop-Jobs enorm steigern kann.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 3. Juli 2013

Querschau

eclipse MAGAZIN — Ausgabe 4.2013 | www.eclipse-magazin.de

- **Jetty 9: Zukunft der Webprotokolle heute**
- **Eclipse Mobile: BB10 IDE und SDK**
- **Team Foundation Server loves Git**

PHP magazin — Ausgabe 4.2013 | www.phpmagazin.de

- **Git und die Sicherheit: Was Sie bei GitHub beachten sollten**
- **Couchbase: Einfach, schnell, elastisch**
- **Neo4J und Gremlin: Von Kobolden und Elefanten**

windows .developer — Ausgabe 7.2013 | www.windowsdeveloper.de

- **Groß, größer, Big Data: Es lebe die Big Data Cloud**
- **Git Backstage: Was steckt hinter der „Magie“?**
- **Gap of War: Stuxnet, Duqu, Flame und Co. im Überblick**

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, Corinna Kern, Diana Kupfer
Chefin vom Dienst/Leitung Schlussredaktion: Nicole Bechtel
Schlussredaktion: Jennifer Diener, Frauke Pesch, Lisa Pychlau
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Flora Feher, Dominique Kalbassi, Laura Keßler, Nadja Kesser, Maria Rudi, Petra Rüth, Franziska Sponer

Autoren dieser Ausgabe:
Sebastian Bohmann, Andy Bosch, Christian Brandenstein, Robert Bruckbauer, Thilo Focke, Klaus Kreft, Martin Künkele, Angelika Langer, Arne Limburg, Christian Meder, Michael Müller, Marc Petersen, Christian Robert, Peter Roßbach, Lars Röwekamp, Zoran Savić, Sigrid Schefer-Wenzl, Niklas Schlimm, Eugen Seer, Oliver Selinger, Thomas Singer, Stefan Siprell, Sandro Sonntag, Kai Spichale, Marc Strapetz, Oliver Wehrens, Matthias Wenzl, Christian Zillmann

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

Einem Teil dieser Ausgabe liegt eine Beilage der Firma oose Innovative Informatik GmbH bei.