Reichen Java und Linux aus, um Android-Apps zu entwickeln? ▸107

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 2.2013

JAVA Mag

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

OAuth 2.0
Der gemeinsame Nenner von Google, Facebook & Co. ▸63

Google App Engine
Neues aus der PaaS-Reihe: In Google we trust? ▸70

JAX 2013
Alle Infos hier im Heft!
▸61

Java EE 7

Evolution statt Revolution

Was ist neu? ▸16

JAX-RS und JMS 2.0 ▸21, 26

JSF 2.2: Interview mit Andy Bosch ▸33

Kommentar: Keine Wolke (Java EE) 7 ▸36

Migration abseits der grünen Wiese ▸38

4 194586 709801 02

Productivity, Productivity, Productivity

Die Fertigstellung der Java EE 7 steht kurz bevor, ein guter Zeitpunkt also, das neue „Java fürs Große“ unter die Lupe zu nehmen. Auch im Vorfeld dieses Release gab es enttäuschte Gesichter, als es hieß, die hippen Cloud-Technologien werden nicht in dieser Version, sondern erst mit der nachfolgenden Version 8 erscheinen.

Wer also das mächtige Cloud-Release, als das EE 7 ursprünglich mit Pauken und Trompeten angekündigt wurde, erwartete, für den fällt die de facto EE 7 gewiss etwas blass aus. Zumal es nicht sonderlich originell wirkt, zum dritten Mal in Folge ein Major-Release unter das Motto „Productivity“ zu stellen. Aber dieser Umstand spricht wohl eher Bände darüber, wie es um dieses Thema zuvor bei der J2EE bestellt war ... So lassen wir uns also ein Productivity-Release nach dem anderen gerne gefallen.

Die Frage dagegen, ob wir tatsächlich eine „Cloud-ready Java EE“ benötigen, lässt sich am besten mit einer Gegenfrage beantworten: „Wie sollte eine Cloud-ready Java EE überhaupt aussehen?“ Ich vermute ja, es waren eher die konzeptionellen Unklarheiten als handfeste technische Probleme, die eine Verschiebung des Wolkenthemas auf die Version 8 zur Folge gehabt haben. In anderen Worten: das Risiko, mit einer Spec für Java EE Cloud Provisioning und Multi-Tenancy voran zu marschieren, ohne dass die maßgeblichen Hersteller nachfolgen, wäre wohl zum jetzigen Zeitpunkt zu hoch gewesen.

Da überlassen wir doch lieber die vielfältigen PaaS-Technologien wie CloudFoundry, Heroku, OpenShift oder Google App Engine dem freien Spiel der Marktkräfte und analysieren dann, welche Konzepte für die Anwender tatsächlich von Nutzen sind. Eine Spec schreiben können wir im Nachhinein dann immer noch.

Überhaupt ist das Zuhören, was auf dem Marktplatz so geredet wird, immer eine gute Idee. So hat Oracle auch eine Umfrage zum Thema Java EE 7 gestartet und kurz vor Jahresfrist die Ergebnisse veröffentlicht. Die Umfrage spiegelt die Erwartungen der Entwicklercommunity an die Java EE 7 wider:

- 73 Prozent meinen, CDI sollte per Default in Java-EE-Umgebungen aktiviert werden.
- 53 Prozent unterstützen die konsistente Nutzung von @Inject in allen JSRs, nur 28 Prozent halten alternative Injection-Annotationen für sinnvoll.
- Sollte das CDI @Stereotype dahingehend erweitert werden, dass Annotationen über CDI hinaus abgedeckt werden? 62 Prozent meinen: Ja! 13 Prozent sind dagegen.
- 96 Prozent sind dafür, Interceptors für alle Java-EE-Komponenten zu ermöglichen. 35 Prozent sehen das auch für Java-EE-Managed-Klassen.

Einen kompakten Einstieg in die Neuerungen der Java EE 7 bieten unsere Beiträge auf den Seiten 16–43.

Android – endlich geschmeidig

In der Kolumne „Good to know“ beschreiben Lars Röwekamp und Arne Limburg eine Android-Innovation, die möglicherweise von zentraler Bedeutung für den weiteren Markterfolg des Mobile-Betriebssystems sein kann (Seite 112). Es handelt sich um das „Project Butter“, das für einen ruckelfreien Fluss aller Animationen und – vielleicht am wichtigsten – für Touch-Reaktionen in gefühlter Echtzeit sorgen soll.

Damit haben die Android-Entwickler endlich eine zentrale Schwachstelle gefixt, die Android-Devices im Allgemeinen und die Tablets im Speziellen bis vor Kurzem noch deutlich unreifer hat erscheinen lassen als die Konkurrenz von Apple.

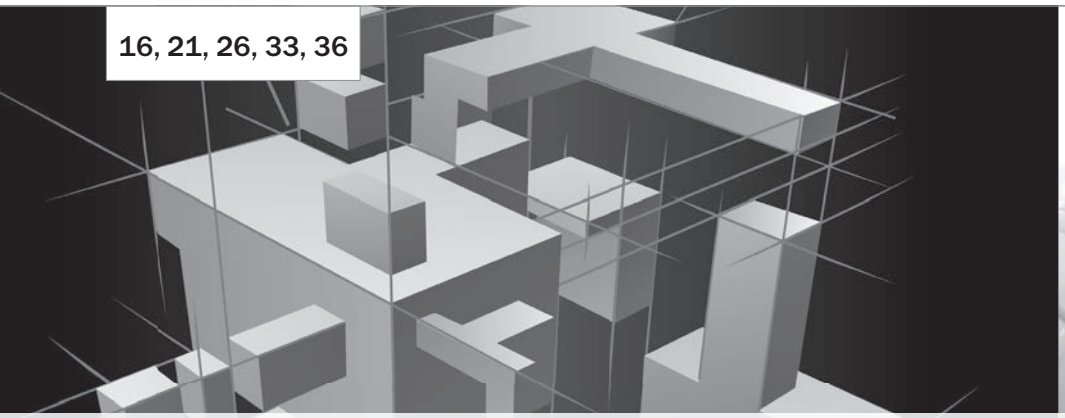
Man muss sich das mal vorstellen: Da haben die Android-Ingenieure keine Mühen gescheut, ihrem Betriebssystem so tolle Sachen wie Multitasking oder viele Schnittstellen zu spendieren, und damit erreicht, dass ihr Betriebssystem Dinge kann, die die Apple-Welt (teilweise) noch nicht beherrscht, aber die direkte und intuitive Interaktion mit dem Nutzer – das Wichtigste was ein Multi-Touch-System beherrschen sollte! – haben sie ihrem Android erst jetzt gegönnt.

Besser spät als nie, jedenfalls stelle ich selbst (als notorischer iOS-Nutzer) fest, dass mir Android mit Jelly Beans erstmals richtig Spaß macht. Überhaupt könnte es sein, dass der August 2012 eines Tages als der Wendepunkt im Kampf „iOS vs. Android“ bezeichnet werden wird – mit Jelly Beans und den gut gemachten, aber kostengünstigen 7-Zoll-Devices von Google und Amazon haben die Apple-Konkurrenten erstmals eine ernsthafte Alternative zum dominierenden iPad vorzuweisen.

Ich wünsche Ihnen jedenfalls einen tollen Start ins neue Jahr und viel Glück und Erfolg bei allen beruflichen und privaten Vorhaben!

Ihr Sebastian Meyen, Chefredakteur

 @smeyen



Java EE 7

Hätte man noch vor wenigen Monaten einen Ausblick auf Java EE 7 gegeben, wären Buzz Words wie Multi-Tenancy und Cloud-Support die Aufhänger des Schwerpunkts gewesen. Mittlerweile hat die JSR 342 Expert Group allerdings ein wenig zurückgerudert und sich deutlich realistischere Ziele gesteckt. Welche, zeigen unsere Autoren Lars Röwekamp und Arne Limburg. Einen ersten Blick auf die wichtigsten Neuerungen in JAX-RS 2.0 sowie in JMS 2.0 wirft Thilo Frotscher. Im Interview erzählt Andy Bosch von der Arbeit an JSF 2.2. Auch, was Java EE 7 auslässt, lassen wir nicht aus: Bernhard Löwenstein kommentiert die Verschiebung der Cloud-Features.

Migration zu Java EE 6

Auf Konferenzen und in Fachartikeln zeigen neue Technologien und Frameworks meist nur ihre Schokoladenseite. Doch Unternehmen, die schon etwas länger auf Enterprise Java setzen, befinden sich oft in einer wenig vorteilhaften Ausgangsposition. Lernen Sie, wie der Wechsel zum aktuellen Java EE 6 Stack dennoch gelingen kann und wann und warum sich eine Migration lohnt.

Magazin

Titelthema

Enterprise

Web

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

OAuth, die Zweite
Facebook, Google, Foursquare oder Pinterest haben eines gemeinsam: Die APIs dieser Dienste setzen allesamt auf OAuth 2.0. Auch in Zukunft werden wohl mehr APIs dieses Protokoll verwenden, zumal es im Vergleich zum Vorgänger viel einfacher zu benutzen ist. Wir betrachten in dieser Artikelserie zunächst den wichtigsten OAuth 2.0 Grant.

Mit J2ME ins Internet
Nachdem wir uns im ersten Teil des Tutorials mit dem Thema GUI für J2ME beschäftigt haben, folgt nun eine detaillierte Analyse der im Hintergrund befindlichen Logik. Dabei gehen wir besonders auf den Internetzugriff und das lokale Speichern von Daten ein – Themen, die für Anfänger oft problematisch sind.

Zwei Seiten einer Medaille?
Erschwingliche Datenflatrates, eine echte Vielfalt an Smartphones und Tablets und eine Vielzahl an Inhalten für jeden Geschmack haben den Siegeszug des mobilen Internets ermöglicht. Waren es früher noch die sperrigen Ökosysteme von BlackBerry oder Nokia, die das mobile Internet abdeckten, ist es heute ein Allgemeingut geworden – und damit umso interessanter für Unternehmen.

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

JVM-Sprache Kotlin M4 wird „JDK-7-friendly“

Die mit dem Preis „Innovativstes Java-Unternehmen 2012“ ausgezeichnete tschechische Softwareschmiede JetBrains gehört zu den aktivsten Playern im Java-Ökosystem. Nachdem letzte Woche die Version 12 der Entwicklungsumgebung IntelliJ IDEA erschienen ist, präsentiert uns Andrey Breslav heute den vierten Meilenstein der JVM-Sprache Kotlin, die maßgeblich von JetBrains entwickelt wird.

Kotlin ist eine JVM-kompatible Programmiersprache, die ein statisches Typensystem, Unterstützung für Variable Type Inference und Closures mitbringt. Die seit 2010 entwickelte Sprache zielt auf Java- und JavaScript-Plattformen ab (Kotlin wird in JVM-Bytecode oder JavaScript kompiliert) und wurde im Februar 2012 Open Source zur Verfügung gestellt.

Als generelle Einschätzung zur neuen Version gibt Kotlin-Entwickler Andrey Breslav aus: Kotlin M4 sei „JDK-7-friendly“ – was heißen will, dass zwar immer noch Java-6-kompatibler Bytecode generiert wird, einige Probleme im Zusammenhang mit der Kompilierung gegen JDK 7 aber gefixt sind. Schnellere Type-Argument-Inferenz, eine neue *@deprecated*-Annotation und eine reichere Quellcodevervollständigung im Kotlin-Tooling gehören zu den weiteren Nettigkeiten.

Neuerungen gibt es auch im KAnnotator, einem Tool zur automatischen Annotation von Libraries, und bei den Data Classes, mit denen sich Daten originell repräsentieren lassen, wie in folgendem Beispiel gezeigt:

```
data class Person(val firstName: String, val lastName: String)
fun Person.asMarried(newLastName: String)
  = this.copy(lastName = newLastName)
```

Wie es sich für einen Tooling-Spezialisten wie JetBrains gehört, ist Kotlin M4 auch gleich in der Entwicklungsumgebung IDEA präsent. Der Kotlin-Support wurde dort noch weiter ausgebaut. So können beispielsweise alle Tests in einem Verzeichnis gestartet werden – IntelliJ IDEA erkennt in Kotlin und in Java geschriebene Tests automatisch.

Wer Kotlin M4 ausprobieren möchte, kann dies am besten in der Community Edition von IntelliJ IDEA 12 tun. Kotlin M4 steht dann im Plug-in-Repository bereit, das direkt in der IDE zugänglich ist.

▶ http://blog.jetbrains.com/kotlin/2012/12/kotlin-m4-is-out/

Browser-IDE Orion 2.0 M1: Android-Swype, Node.js usw.

Die Arbeiten am zweiten Major-Release des Orion-IDE-Projekts haben erste Früchte gezeitigt. Übergeordnetes Ziel der Eclipse-Browser-IDE Orion 2.0 ist die Verbesserung der Import- und Deployment-Prozesse sowie die Unterstützung der JavaScript-Bibliothek Node.js. Noch ist man in diesen Punkten allerdings noch nicht soweit, handfeste Ergebnisse vorlegen zu können, aber immerhin weist der aktuelle Milestone-1-Build einige schöne Featureerweiterungen auf.

So lässt sich die globale Suche jetzt auf bestimmte Dateitypen beschränken. Für Suche- und Ersetze-Aktionen lassen sich jetzt JavaScript-ähnliche reguläre Ausdrücke nutzen. Nett ist die neue Funktion, Dateien, Ordner und Verzeichnisse vom Desktop in den Orion-Editor ziehen zu können, allerdings nur in Chrome. In Firefox funktioniert bisher nur das Drag and Drop von einzelnen Dateien. IE-User müssen die Action-Menü-Option IMPORT LOCAL FILE bemühen.

Signifikante Verbesserungen weist auch der Android-Editor auf: virtuelle Keyboards, Sprachinput und Gesteneingaben à la Android 4.2 Swype. Wie anfangs erwähnt, ist das große Thema Node.js noch nicht in einem präsentablen Zustand. Allerdings können Bleeding-Edge-Interessierte Orion-Freunde auf GitHub einen Prototyp des Node-basierten Orion-Servers begutachten.

Orion 2.0 M1 hat noch mehr zu bieten, wie auf dem Orion-Blog beschrieben wird. Wer es genauer wissen möchte, findet im JAXenter-Artikel „Eclipse Orion: Coding on the Web“ von Orion-Entwickler Simon Kaegi den vertiefenden Lesestoff, inklusive Projekthistorie und Blick unter die Motorhaube.

▶ http://it-republik.de/jaxenter/artikel/Eclipse-Orion-Coding-on-the-Web-4650.html

Agil und flexibel mit adVANTAGE

Die adesso AG hat kürzlich ihr eigenes agiles Vorgehensmodell für die Softwareentwicklung vorgestellt: adVANTAGE ist bei dem IT-Dienstleister als Modell für die Entwicklung von Individualsoftware im Einsatz.

Im Gespräch mit dem Java Magazin erklärt adesso-Kovorstandsvorsitzender Dr. Rüdiger Striemer den Grund für adVANTAGE: „Auf beiden Seiten, also sowohl beim Kunden als auch beim IT-Dienstleister, besteht häufig eine Vollkaskomentalität". Das bedeutet in der Regel viel zu lange und teure Spezifikationsphasen, das Projektergebnis lässt häufig zu viele Interpretationsspielräume zu und geht am eigentlichen Ziel vorbei.

Der Einsatz eines agilen Verfahrens ist natürlich nichts Neues, adesso setzt meist auf den bewährten Scrum-Ansatz. Das Neue am adVANTAGE-Modell: Bereits in einer sehr frühen Phase, in der es noch keine Spezifikation gibt, definiert adesso die Aufwände. Das ist für das Unternehmen natürlich mit einem gewissen Risiko verbunden, wenn die Prognosen zu optimistisch sind. Warum: Nach der Aufwandsschätzung priorisiert der Kunde die einzelnen fachlichen Anforderungen. Schließlich wird die erste Gruppe User Stories umgesetzt. Hat diese Umsetzung dann etwa statt der vorher festgesetzten 30 Tage Aufwand beispielsweise 40 Tage gedauert, rechnet adesso die 30 Tage normal ab und die zusätzlichen zehn nach einem „schmerzhaften" Tagessatz.

„Das ist ein Tagessatz, der für uns noch kostendeckend ist, für den wir aber eigentlich nicht arbeiten wollen", erklärt Dr. Striemer. „Der Kunde weiß, dass dieser Tagessatz für uns unattraktiv ist. So haben wir eine eingebaute Risikoteilung."

Warum aber kommt adVANTAGE jetzt und nicht schon vor fünf Jahren? „Agile Verfahren sind jetzt immer besser durchsetzbar, das ist ein schleichender Prozess. Vor fünf Jahren gab es diese Erfahrung zwar bei uns schon, aber beim Kunden noch nicht."

Diese Aussage deckt sich mit den Eindrücken des Agile Days, der kürzlich auf der W-JAX in München stattfand und der auch für die kommende JAX in Mainz wieder auf dem Programm steht. Der Tag hat eine lange Tradition in der Konferenz, dabei hat sich das Vorwissen der Teilnehmer über die Jahre deutlich gewandelt. Mittlerweile sind jedem Besucher die Grundregeln agiler Methoden bekannt, viele haben selbst schon agil entwickelt. Das hilft IT-Dienstleistern wie der adesso AG bei der erfolgreichen Umsetzung ihres eigenen agilen wertorientierten Vorgehensmodells.

▶ http://adesso.de/de/leistungen/softwaredevelopment/advantage/advantage.jsp

Leserbrief

Der Artikel „MDSD ohne Einfluss auf die fachliche Architektur“ von André Pflüger (Ausgabe 12.2012) wirft einige Fragen zu unserem vorhergehenden Artikel „EJB 3 modellgetrieben entwickeln“ (Java Magazin 3.2012) auf. Als Autoren nehmen wir diese gerne auf und wollen genauer darauf eingehen.

Zunächst wird die Frage gestellt, ob die Arbeit des Anwendungsentwicklers durch die gleichzeitige Verwendung von Annotations und Deployment-Deskriptoren (DD) zur Angabe von JEE-Metadaten erschwert wird und warum diese Aufteilung erfolgt. Unsere Praxiserfahrung zeigt, dass die Arbeit des Entwicklers dadurch in keinster Weise negativ beeinflusst wird. Im Gegenteil, für ihn sind die sich aus dem Modell ergebenden generierten Metadaten – ob als Annotation oder per DD – vollständig transparent. Gerade das stellt einen zentralen Vorteil des Vererbungs-Patterns dar: Da sämtliche Modellinformationen in der Basisklasse oder dem DD abgebildet werden, kann der Entwickler mit einem echten POJO (der ableitenden Implementierungsklasse) arbeiten, das somit frei von jeglichen modellspezifischen Metadaten ist. Dies hat sich in unserem MDD-Projektalltag als Best Practice erwiesen. Warum auch sollte der Entwickler sich im Quellcode mit Modellinformationen auseinandersetzen, ist doch laut MDD-Definition an der Stelle das Modell und nicht der Code führend!

Zugegebenermaßen ist die Nutzung des DD ein „kleiner Notnagel“. Idealerweise würden sich alle Metadaten als Annotation in die Basisklasse generieren lassen. Wie jedoch in unserem Artikel beschrieben, sind nicht alle benötigten EJB-3-Annotationen wie gewünscht ableitbar. Daher ist es erforderlich, einige wenige Metadaten in den Deployment-Deskriptor auszulagern. Allerdings ist das – insbesondere bei Entities – nur ein sehr geringer Anteil.

Auch, dass sich durch das Vererbungs-Pattern die „fachliche Architektur“ ändere, können wir nicht nachvollziehen. Um beim in beiden Artikeln verwendeten Beispiel der Entity Customer zu bleiben: Nach wie vor wird eine Geschäftsentität Customer modelliert. Gleichfalls steht dem Anwendungsentwickler genau eine Java-Klasse für Codeergänzungen zur Verfügung. Es wird lediglich eine technische Basisklasse eingefügt, von einem Einfluss auf die Fachlichkeit kann nicht gesprochen werden. Der verwendete Ausdruck „Mikroarchitektur“ ist an die Begrifflichkeit in der Fachliteratur angelehnt (vgl. „EJB 3 professionell“ von Ihns et al., S. 93 ff.). Er beschreibt die technischen Artefakte, aus denen sich eine logische Entity oder Session Bean zusammensetzt, d. h. beispielsweise Bean-Klasse, Interface, DD etc. Somit kann die Verwendung des Ausdrucks nicht als Indiz für eine Auswirkung auf die Fachlichkeit herangezogen werden.

Richtig liegt der Artikel mit dem Argument, dass eine Trennung von generierten und manuellen Quelltexten nicht dadurch begründet sein sollte, dass die zu verwaltende Datenmenge aufgrund der nicht benötigten Versionierung der generierten Anteile deutlich abnimmt. In der Tat ist das bei modernen Versionsverwaltungssystemen kaum noch relevant. Das ist jedoch auch nicht die Motivation für die strikte Aufteilung auf Dateiebene.

Vielmehr sollte sich – wenn das MDD-Paradigma konsequent umgesetzt wird – die logische Trennung zwischen Modell und ergänzendem Code auch auf physikalischer Ebene niederschlagen. Folglich kommt der Entwickler idealerweise nicht mit den generierten Artefakten in Berührung. Durch die Trennung im Dateisystem ist es für ihn besser ersichtlich, welche Dateien er problemlos erweitern kann und welche Verzeichnisse für Änderungen „tabu“ sind.

Als wir vor der Entscheidung standen, welchen Ansatz wir für ein großes MDD-Projekt wählen – Vererbung oder geschützte Bereiche („ProRegs“) –, haben wir uns bewusst für den Vererbungsansatz entschieden. Selbst wenn sich das Pattern nicht perfekt auf EJB 3 übertragen lässt, da einige wenige JEE-Metadaten in Deployment-Deskriptoren ausgelagert werden müssen, haben wir uns aufgrund der Nachteile, die sich zumindest für das konkrete Projekt ergeben hätten, gegen die ProRegs entschieden.

Insbesondere der Umstand, dass alle sich aus dem Modell ergebenden Metadaten vor dem Entwickler gekapselt werden, sodass er sich im Normalfall nicht mit ihnen auseinandersetzen muss, war ein zentrales Ziel.

Durch die unsaubere Trennung von Generat und manuellen Ergänzungen bei ProRegs kann dies nicht erreicht werden. Zusätzlich werden die Klassen mit technischen, Generator-spezifischen Kommentaren zur Verwaltung der ProRegs verschmutzt (siehe Listing 3 bei Pflüger). Des Weiteren wird durch ProRegs der Codegenerator komplexer, da er geschützte Bereiche verwalten, erkennen und erhalten muss. Deshalb werden ProRegs auch nicht von allen Generator-Frameworks unterstützt.

All diese Punkte führten dazu, dass wir im konkreten Fall das etablierte Vererbungs-Pattern umgesetzt haben – mit großem Projekterfolg.

Christian Schwörer und Konrad Pfeilsticker, NovaTec – Ingenieure für neue Informationstechnologien GmbH (www.novatec-gmbh.de)

Anzeige

Autor des Monats

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.

Wie bist du zur Softwareentwicklung gekommen?

Das war Ende der 90er, als überall die Rede vom eklatanten Fachkräftemangel und den guten Einkommen in der IT war. Ich war mitten im BWL-Studium und kaufte mir mein erstes Programmierbuch „Teach Yourself C++ in 21 Days“. 21 Tage später war klar, dass das mein Ding ist.

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Ich mag den großen Hebel bei der Softwareentwicklung. Man kann mit überschaubaren Investitionen großartige Produkte bauen. Außerdem ist die Tätigkeit abwechslungsreich und verlangt eine herausfordernde Kombination unterschiedlichster Fähigkeiten wie Lösungskompetenz, dauerhaftes Lernen, gute Abstraktionsfähigkeiten für Struktur und Namensgebung sowie effizienten Umgang mit anderen Menschen.

Was ist für dich ein weniger schöner Aspekt?

Der Hebel wirkt natürlich auch in die andere Richtung. Ein paar falsche Personen an den richtigen bzw. falschen Stellen können ein ganzes Projekt zum Kippen bringen.

Wie und wann bist du auf Java gestoßen?

Ich denke, das war um die Jahrtausendwende herum. Für einen C++-Entwickler hat sich die Syntax angenehm angefühlt. Die Sprache und die mächtige Standardbibliothek haben den Spaß- und Produktivitätsfaktor beim Programmieren enorm gesteigert. Auf das Gewürge mit C++ hatte ich danach keine wirkliche Lust mehr.

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

Die Sprache selbst ist in den meisten Bereichen gut gelungen. Es stören mich nur Kleinigkeiten, beispielweise dass der Umgang mit Synchronisierung bei StringBuffer und StringBuilder (oder auch Vector und ArrayList) über (nicht sehr sprechende) Klassennamen gelöst worden ist. In Hinblick auf Java als Plattform würde ich anregen, den Fokus weniger auf den JCP und mehr auf agile, alternative Innovationen zu richten. Die wirklich tollen Innovationen wie Maven, Spring, Android, Netty, Wicket etc. wurden nicht jahrelang spezifiziert und standardisiert, sondern passierten einfach, wenn sie passieren mussten.

Was ist zurzeit dein Lieblingsbuch?

Ich habe „Agile Estimating and Planning“ von Mike Cohn gerade zwei Mal gelesen und kann es sehr empfehlen. Neben der Fachliteratur lese ich gerne Biografien.

Was machst du in deinem anderen Leben?

Ich habe eine Familie mit zwei Kindern. Außerdem spiele ich gerne Tennis und nehme an regionalen Laufwettbewerben teil.

Produktive Softwareentwicklung

von Hans-Jürgen Plewan und Benjamin Poensgen

Bei produktiver Softwareentwicklung geht es um weit mehr als nur um das Coden der Programme. Das Buch richtet sich vor allem an Projektleiter. Ein guter Projektleiter wiederum muss dem Entwicklerumfeld entstammen, so die Autoren. Auch Entwickler mit Zielrichtung Projektleiter oder einfach nur mit Blick über den Tellerrand sind potenzielle Leser.

Was bedeutet produktive Softwareentwicklung überhaupt? Diese Frage beantworten die Autoren im ersten Teil. Insbesondere geht es hier um das Verhältnis von Ertrag zu Aufwand. Soll der Output, aber nicht der Zeitaufwand erhöht werden, so kann die Qualität leiden, d. h. Korrekturen werden erforderlich und die Produktivität sinkt. Die Verbesserung der Qualität ist ein wesentlicher Baustein für die produktive Entwicklung.

Aufwand in Form von Kosten und Zeit ist recht gut messbar. Wie aber steht es mit dem Ertrag? Hier geht es nicht (nur) um den späteren Erlös oder Nutzen der Software, sondern darum, den Funktionsumfang einer Software frühzeitig zu messen. So gelingt es, Projekte miteinander zu vergleichen und zu bewerten. Der zweite Teil widmet sich diversen Messmethoden, wobei die Autoren die Function-Point-Methode präferieren.

Bevor dann acht elementare Produktivitätsfaktoren vorgestellt werden, erfährt der Leser erst einmal etwas über schlechte Praktiken – vage Ziele, unrealistische Projekte, unklare Anforderungen und Methoden, Politik im Projekt und mehr. Hier wird leicht ersichtlich: so nicht. Im letzten Teil werden dann die genannten acht Faktoren ausführlich vorgestellt. Teilweise handelt es sich um scheinbare Selbstverständlichkeiten – wie das klare Formulieren von Zielen –, die in der Praxis leider allzu oft missachtet werden. An diesen Stellen ist das Buch der Mahner, an diese Dinge zu denken und sie auch umzusetzen. Teilweise sind es Gedanken, die deutlich über diese elementaren Anforderungen hinausgehen. Wie finde ich die optimale Teamgröße? Was bringt methodisches Vorgehen? Wie helfen Code-Reviews? Insgesamt ergibt sich eine Aufstellung der Leitlinien, deren Beachtung nicht nur eine produktive Softwareentwicklung ermöglicht, sondern auch die Erfolgschancen eines Projekts erhöht. Insbesondere der letzte Teil eignet sich in diesem Sinne auch als eine Art Nachschlagewerk.

Locker verpackt, übersichtlich gestaltet und mit Grafiken und Tabellen untermauert, gefällt dieses Werk. Für Entwickler, die ausschließlich ihren Code leben, ist das Buch gänzlich ungeeignet. Für Projektleiter und alle, die sich über das Kodieren hinaus mit ihrem Projekt beschäftigen, ist es hingegen eine lohnenswerte Lektüre.

Michael Müller

Hans-Jürgen Plewan, Benjamin Poensgen

Produktive Softwareentwicklung

Bewertung und Verbesserung von Produktivität und Qualität in der Praxis

262 Seiten, 39,90 Euro
dpunkt.verlag, 2011
ISBN 978-3-89864-686-4

Kanban in der IT

von Dr. Klaus Leopold und Dr. Siegfried Kaltenecker

Die Autoren beschreiben in diesem Buch, wie die Kanban-Methodik dabei unterstützt, kontinuierliche Veränderungsprozesse in der Softwareentwicklung erfolgreich einzuführen und zu etablieren. Dabei richtet sich der Band nicht nur an Leser mit einem grundsätzlichen Interesse an Kanban in der IT und Change Manager in IT-Organisationen, sondern auch an alle, die sich bereits näher mit der Einführung von Kanban beschäftigt haben bzw. bereits damit zu Gange sind. Die wichtigsten Erkenntnisse jedes Kapitels sind jeweils am Ende zusammengefasst und optisch hervorgehoben.

Die heterogene Leserschaft im Kopf, haben die Autoren das Buch in drei Teile gegliedert. Der erste Teil vermittelt die grundlegenden Ansätze der Methodik: die Visualisierung von Abläufen und Engpässen mittels Kanban Board, die Notwendigkeit von Work-in-Progress-Limits zur Beschleunigung der Durchlaufzeiten sowie die Bedeutung von Serviceklassen zur Risikobewertung jeder einzelnen Aufgabe. Anschließend konkretisieren die Autoren, wie Kanban in Softwareentwicklungsteams betrieben und koordiniert werden kann. Sie erläutern wesentliche Plattformen, wie Daily Standup, Queue Replenishing Meeting und Teamretrospektiven, und schließen den ersten Teil damit, welche Metriken und Verbesserungsmethoden ihrer Erfahrung nach helfen, den Erfolg dieser Vorgehensweise zu sichern.

Der zweite Teil befasst sich mit Change Management im Allgemeinen. Ausgehend von der Annahme, dass Veränderung eine Konstante des Tagesgeschäfts ist, wird beleuchtet, wie Change-Prozesse professionell gestaltet werden können. Im Mittelpunkt stehen die Menschen und ihre Emotionen sowie die Unternehmenskultur und -politik. Als Kernkompetenzen für erfolgreiches, zielorientiertes Change Management werden aufmerksame Wahrnehmung, professionelle Kommunikation und die agile Gestaltung der Veränderungsprozesse beschrieben.

Für die zuletzt aufgeführte Kompetenz geht es im dritten Teil dann in medias res. Unterlegt mit praktischen Kanban-Erfahrungen von Führungskräften unterschiedlicher Branchen, wird ausführlich und konkret erklärt, wie die Methodik gut vorbereitet eingeführt, gelebt und weiter ausgebaut werden kann. Dabei werden nicht nur blühende Landschaften beschrieben, sondern auch mögliche Widerstände und Konflikte.

Das Buch liest sich unterhaltsam und adressiert den Leser persönlich. Nach der Lektüre hat man eine Vorstellung davon gewonnen, was diese Vorgehensweise für das eigene Entwicklungsteam bringen könnte. Wer seine theoretischen Kenntnisse über Change Management nicht weiter zu vertiefen braucht, kann den Mittelteil überblättern. Sonst: Lesen! Steckt man schon mitten im Kanban-Prozess, dient vor allem der dritte Teil als Nachschlagwerk und Ideengeber für das Kanban-Instrumentarium.

Caroline Buck

Dr. Klaus Leopold, Dr. Siegfried Kaltenecker

Kanban in der IT

Eine Kultur der kontinuierlichen Verbesserung schaffen

331 Seiten, 34,90 Euro
Carl Hanser, 2012
ISBN 978-3-446-43059-4

tinylog: schlanke log4j-Alternative für Java

Logging light

Loggen ist mittlerweile in jeder komplexen Anwendung üblich. Als Logging-Framework war log4j der Vorreiter in der Java-Welt und hat sich zum De-facto-Standard entwickelt. Aber genügt log4j immer allen Anforderungen? „tinylog“ wagt als schlanke Alternative den Vorstoß.

von Martin Winandy

Das Protokollieren des Programmablaufs vereinfacht die Fehlersuche und das Reproduzieren von Bugs. Dabei macht die Anwendung mithilfe eines Loggers regelmäßig Ausschriebe über den Programmablauf und aufgetretene Fehler. Später kann der Entwickler für gemeldete Bugs anhand der erzeugten Log-Dateien problemlos nachvollziehen, was im Programm genau während des Fehlers passiert ist. Dies ist besonders bei Bugs hilfreich, die in Produktion oder beim Kunden auftauchen. Damit Log-Dateien wirklich hilfreich sind, muss eine Anwendung genügend und vor allem aussagekräftige Log-Einträge ausgeben, und das liegt allein in der Hand der Entwickler. Je einfacher ein Logging-Framework zu verwenden ist, desto häufiger wird es auch tatsächlich genutzt – und der Programmablauf damit umso lückenloser protokolliert.

„tinylog“ [1] ist ein schlankes Logging-Framework für Java, das eine Alternative zum De-facto-Standard log4j darstellt. Die Idee zu tinylog entstand aus mehreren Verbesserungsvorschlägen für log4j [2] und mündete schließlich in ein eigenes Open-Source-Projekt:

- Das Erzeugen einer Logger-Instanz für jede Klasse, in der geloggt werden soll, ist umständlich und fehleranfällig.
- Texte in den Logging-Methoden können weder formatiert noch mit Parametern versehen werden.
- In Anwendungen, bei denen es auf schnelle Reaktionszeiten ankommt, kann das Erzeugen und Schreiben von Log-Einträgen zu Performanceproblemen führen.

Statischer Logger

In den drei großen bekannten Logging-Frameworks log4j, Logback [3] und dem Java-Logging-API [4] wird üblicherweise für jede Klasse eine eigene Logger-Instanz erzeugt. In log4j sieht dies beispielsweise wie in Listing 1 aus.

Die Instanziierungsanweisung in Zeile 5 wird in jeder Klasse benötigt und daher oft per Copy and Paste aus einer anderen Klasse herauskopiert. Dabei kann leicht vergessen werden, den Klassenparameter zu ändern. Dies führt dazu, dass falsche Log-Einträge erzeugt werden. In tinylog gibt es nur einen einzigen Logger, der statisch ist. Damit entfällt das Instanziieren eines Loggers (Listing 2).

Zu loggende Texte können in tinylog formatiert und mit Parametern versehen werden. Dies funktioniert genau wie beim bekannten *MessageFormat.format()* [5]. Der Unterschied zwischen *Logger.info("Pi ist {0,number,0.00} und die Antwort auf alle Fragen lautet {1}", Math.PI, 42)* und *Logger.info(MessageFormat.format("Pi ist {0,number,0.00} und die Antwort auf alle Fragen lautet {1}", Math.PI, 42))* ist, abgesehen vom kompakteren Code, dass der Text nur dann erzeugt wird, wenn der Log-Eintrag wirklich ausgegeben werden soll. Ist der Logger deaktiviert oder sollen nur Fehler und Warnungen ausgegeben werden, wird der

Listing 1

```
import org.apache.log4j.Logger;

public class Application {

  private static final Logger logger = Logger.getLogger(Application.class);

  public static void main(String[] args) {
    logger.info("Mein Log-Eintrag ...");
  }
}
```

Listing 2

```
import org.pmw.tinylog.Logger;

public class Application {

  public static void main(final String[] args) {
    Logger.info("Mein Log-Eintrag ...");
  }

}
```

Log-Eintrag und damit der Text erst gar nicht erzeugt und dadurch Rechenzeit eingespart. „tinylog“ verfügt über fünf Logging-Level: *ERROR*, *WARNING*, *INFO*, *DEBUG* und *TRACE*. Standardmäßig werden nur die Logging-Level *ERROR*, *WARNING* und *INFO* ausgegeben. Dies kann über die Property *tinylog.level=<level>* geändert werden. Properties werden typischerweise als Parameter (z. B. *-Dtinylog.level=DEBUG*) übergeben oder befinden sich in der Properties-Datei *tinylog.properties*. Liegt diese im Root-Package, so wird die Datei beim Start automatisch angezogen.

Logging Writers

Standardgemäß werden in tinylog alle Log-Einträge in der Konsole ausgegeben. Dies sieht für das Beispiel in Listing 2 wie folgt aus:

```
2012-08-14 08:52:27 [main] Application.main()
INFO: Mein Log-Eintrag ...
```

Neben dem eigentlichen Text enthält der Log-Eintrag das Logging-Level, einen Timestamp mit Datum und Uhrzeit, den Namen des Threads sowie die Klasse und Methode, in der der Log-Eintrag erzeugt wurde. Die Ausgabe lässt sich über die Property *tinylog.format* beliebig mit einem Pattern anpassen. So können z. B. zusätzlich der Name der Java-Datei und die Codezeile mit ausgegeben oder der ganze Log-Eintrag in einer einzigen Zeile geschrieben werden.

In produktiven Umgebungen ist normalerweise die Ausgabe der Log-Einträge in Log-Dateien gewünscht. Der *FileWriter* schreibt alle Log-Einträge in eine konfigurierbare Log-Datei, die bei jedem Start geleert wird. Weitaus mächtiger ist der *RollingFileWriter*. Dieser kann eine beliebige Anzahl von alten Log-Dateien als Backup aufheben und nach konfigurierbaren Events neue Log-Dateien beginnen. Dies ist vor allem für langlaufende Anwendungen – wie z. B. Server – sinnvoll, damit die Log-Datei nicht irgendwann GBs belegt und die Festplatte vollläuft. So lassen sich z. B. alte Log-Dateien für den Zeitraum von sieben Tagen erhalten, und täglich um Mitternacht kann eine neue Log-Datei begonnen werden:

```
tinylog.writer=rollingfile
tinylog.writer.filename=log.txt
tinylog.writer.policies=daily: 00:00
tinylog.writer.backups=7
```

Sollten die vorhandenen Logging Writers nicht ausreichen, können auch eigene geschrieben werden. Hierzu muss nur das Interface *LoggingWriter* implementiert werden. Der neue Logging Writer kann anschließend als Service registriert und somit wie die bereits vorhandenen über Properties konfiguriert werden.

Reaktionen der Leser

Dieser Artikel erschien ursprünglich auf **http://jaxenter.de**.
Folgende Leserkommentare wollen wir Ihnen nicht vorenthalten:

Leser Nr. 1: „Bei tinylog wird für jede Log-Meldung der Stacktrace ausgewertet, um die loggende Klasse zu ermitteln. Also ich kann mir gut vorstellen, dass sich das gravierend auf die Performance auswirkt ...“

Martin Winnandy: „Es wird nicht der ganze Stacktrace ausgewertet, sondern nur das eine benötigte Stacktrace-Element. Daher sind die Auswirkungen auf die Performance gering. Wenn z. B. auch der Name der Methode ausgegeben werden soll, ist tinylog deutlich schneller als andere Logging-Frameworks: http://www.tinylog.org/benchmark.“

Leser Nr. 2: „Überflüssig, es gibt SLF4J mit LogBack. Die hätten das lieber als Backend anlegen müssen.“

Leser Nr. 3: „‚In den drei großen bekannten Logging-Frameworks log4j, Logback [3] und dem Java Logging API [4] wird üblicherweise für jede Klasse eine eigene Logger-Instanz erzeugt.’ Das klingt so, als täten das die Logging-Frameworks. Vielmehr sind es ja die Anwender, die glauben, es wäre sinnvoll, die Logger so zu organisieren. Die Logger-Hierarchie muss nicht der Klassenhierarchie folgen. Viel sinnvoller ist es, die Logger sachlich in Kategorien zu organisieren, um z. B. kontrolliertes Logging zum Verarbeitungsdurchlauf von Daten zu erhalten. Ich zumindest will oft nicht wissen, welche Daten bei allen Methoden einer bestimmten Klasse durchgeschleift werden, sondern vielmehr, was bestimmte, an Verarbeitung X beteiligte Methoden diverser Klassen eigentlich mit den verarbeitungsrelevanten Daten tun.“

Leser Nr. 4: „Grundsätzlich haben die Entwickler von tinylog das Problem der bestehenden Logging-Frameworks erkannt: Diese sind derzeit noch für die Arbeit mit einem Prozessor ausgelegt. Ja, es gibt im log4j auch die Möglichkeit, die Messages in eine Queue zu legen. Aber wenn dann eine bestimmte Anzahl von Messages (zum Beispiel 100) erreicht ist, werden eben diese 100 Nachrichten synchron ins *LogFile* geschrieben (und damit wird der Programmablauf für kurze Zeit unterbrochen). Vielleicht sollte man eher einen entsprechenden Appender für LogBack/log4j erstellen, der asynchron arbeitet und mit mehreren Cores zurechtkommt.“

Performance

Für die Entwicklung von tinylog ist die Performance ein ganz entscheidender Punkt. Aus diesem Grund enthält das Projekt einen automatischen Benchmark, um zu verhindern, dass neue Features die Performance beeinträchtigen. Bei zeitkritischen Anwendungen kann Loggen möglicherweise zu Performanceproblemen führen. Bei schnellen Internetzugängen sollte google.de nicht viel mehr als 100 ms benötigen, um die Startseite an den Browser zu senden. Um solche Reaktionszeiten zu erreichen, dürfen nicht viele Millisekunden für das Loggen verloren gehen. Ähnlich sieht es bei grafischen Anwendungen aus: 60 fps bedeuten 16,67 ms Zeit pro Frame!

Um zu verhindern, dass die eigentliche Anwendung durch Schreib-Operationen des Loggers ausgebremst wird, kann ein separater Writing Thread aktiviert werden. Dieser Thread läuft mit niedriger Priorität und kümmert sich um das Schreiben der erzeugten Log-Einträge. Sobald der Main Thread fertig ist, beendet sich der Writing Thread automatisch. Letzterer kann auch so konfiguriert werden, dass er auf einen beliebigen anderen Thread wartet.

Ausblick

„tinylog“ befindet sich in der öffentlichen Betaphase. Die Version 1.0 des Open-Source-Loggers ist für Ende 2012 geplant. Vorschläge für neue Funktionalität oder das Melden von Bugs sind ausdrücklich erwünscht. Bis zur Version 1.0 sind noch zwei weitere Testversionen vorgesehen, sodass noch ausreichend Zeit für neue Funktionalität bleibt. Es gibt Planungen zur Unterstützung von mehreren parallelen Logging Writers, und die Entwicklung einer Facade für log4j ist bereits im Gange. Die log4j-Facade ermöglicht es, Log-Einträge, die über das log4j-API erzeugt werden, an tinylog weiterzuleiten. Dies soll es vereinfachen, bestehende Programme auf tinylog umzustellen, und es zu ermöglichen, externe Bibliotheken, die log4j nutzen, mit tinylog zu verwenden. „tinylog“ kann unter http://www.tinylog.org/de/download heruntergeladen werden und steht unter der Apache License 2, die auch die Verwendung von tinylog in kommerziellen Projekten erlaubt.

Martin Winandy ist Diplom-(BA)-Informatiker und arbeitet seit 2008 im Bereich Product Data Management mit dem Schwerpunkt Eclipse-RCP-Entwicklung. Er ist der Initiator von tinylog. Sie können ihn unter martin.winandy@tinylog.org erreichen.

Links & Literatur

[1] http://www.tinylog.org/de

[2] http://logging.apache.org/log4j/1.2/

[3] http://logback.qos.ch/

[4] http://docs.oracle.com/javase/7/docs/technotes/guides/logging/overview.html

[5] http://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html

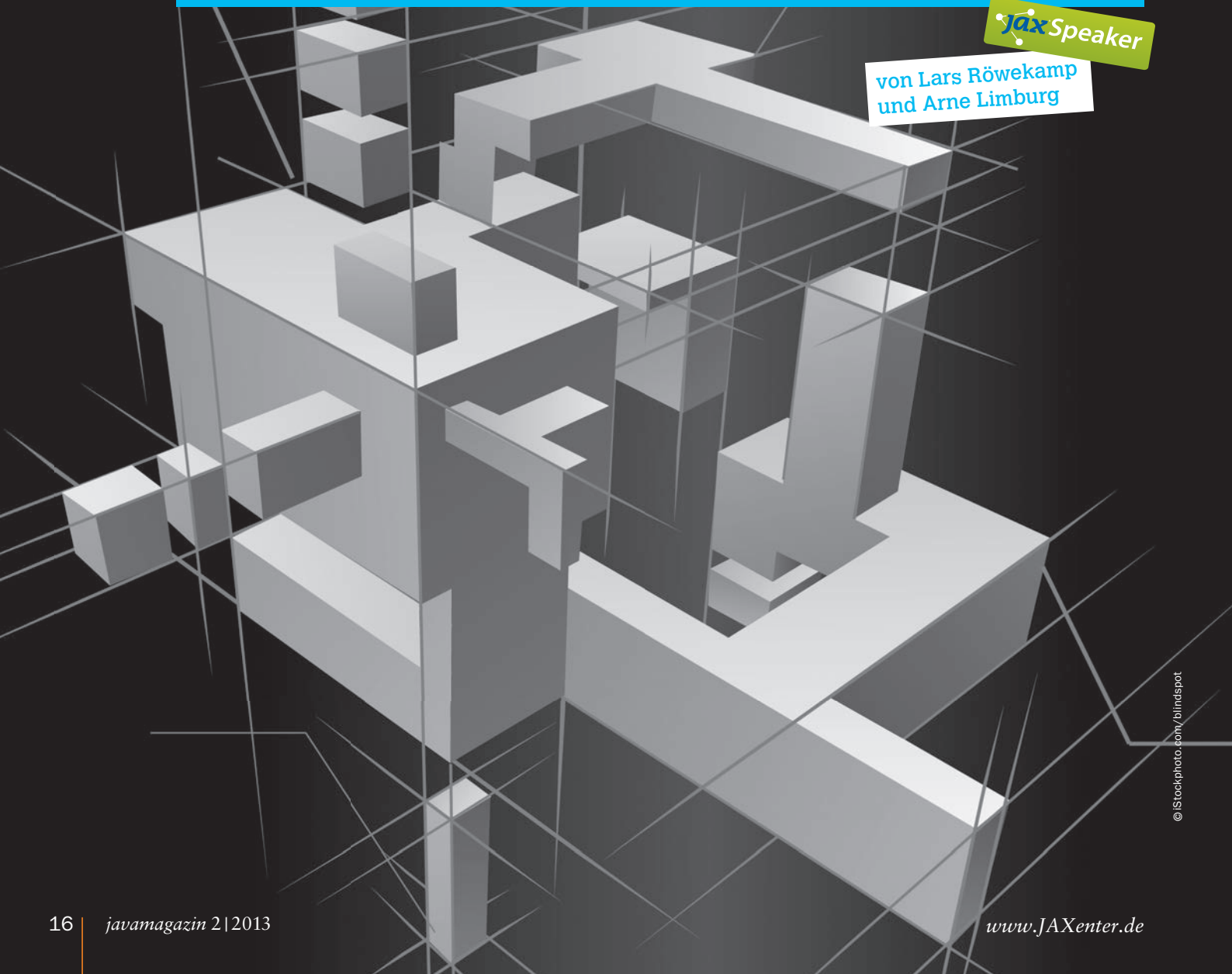
Java EE 7 im Überblick

Heiter bis wolkig

Hätte man noch vor wenigen Monaten einen Ausblick auf Java EE 7 gegeben, wären Schlagworte wie Multi-Tenancy und Cloud-Support die Aufhänger des Artikels gewesen. Mittlerweile hat die JSR 342 Expert Group allerdings ein wenig zurückgerudert und sich deutlich realistischere Ziele gesteckt – und das ist auch gut so.

JAX Speaker

von Lars Röwekamp und Arne Limburg



Seit Ende November steht der Early Draft Review 2 der Java-EE-7-Spezifikation [1] zum Download bereit und seitdem sammelt die Expert Group fleißig das aktive Feedback [2] der Community. Nachdem zunächst relativ hochtrabende Ziele, wie standardisierte Mandantenfähigkeit für PaaS-Provider und allgemeiner Cloud-Support, verfolgt wurden, stellt der aktuelle Stand der Dinge eher eine gesunde Evolution hin zu den „neuen Paradigmen" des Enterprise-Java-Standards dar. *Ease of Development*, *Convention over Code/Configuration* und *Flexibility* stehen auch in dieser Version – wie schon zuvor in Java EE 5 bzw. 6 – im Fokus.

Wer sich einen Überblick darüber verschaffen möchte, in welche Richtung die neue Java-EE-Spezifikation ursprünglich gehen sollte, der wirft am besten einen Blick auf die Sektion 2 des Original Request [1]. Dort finden sich nach wie vor an sehr prominenter Stelle entsprechende Ausführungen zu den Themen Cloud und Multi-Tenancy. Die Lektüre der entsprechenden Abschnitte ist insofern von Interesse, da die Features laut Expert Group lediglich verschoben – Java EE 8 lässt grüßen – nicht aber aufgehoben worden sind. Nun aber zurück zum aktuellen Stand der Dinge.

Weniger ist mehr

Fangen wir erst einmal damit an, was es zukünftig nicht mehr geben wird – und vergessen es dann ganz schnell wieder. Gemeinsam mit der Idee der Java EE Profiles wurde mit Java EE 6 auch das Konzept der *pruned* APIs eingeführt. Darunter versteht man ein API, das in einer zukünftigen Java-EE-Version als *optional* markiert werden wird und ab dann nicht mehr unterstützt werden muss. Das Konzept geht somit einen deutlichen Schritt weiter als das aus Java bereits bekannte *Deprecated*. Ziel dieses Konzeptes ist es, veraltete APIs entsorgen zu können und dabei trotzdem standardkonform zu bleiben. In Java EE 7 stehen gleich mehrere, bereits in Java EE 6 als pruned markierte APIs auf der Abschussliste:

- EJB Entity Beans inkl. EJB QL
- JAX RPC 1.1
- JAXR 1.0
- Java EE Deployment 1.2

So weit, so gut. Diesen APIs wird wahrscheinlich kaum ein Entwickler eine Träne nachweinen. Interessanter ist aber natürlich, was sich bei den restlichen APIs getan hat und welche neuen APIs hinzukommen werden.

Mehr ist mehr

Nachdem die beiden Zugpferde Cloud und Mandantenfähigkeit aus der Spezifikation verbannt wurden, musste schnell ein neues Motto für die kommende Java-EE-Version gefunden werden. Nichts leichter als das: Productivity und HTML5.

Wenn man dieses Motto liest, wundert es nicht, dass mit dem *Java API for Websockets 1.0* (WebSockets/JSR 356) und dem *Java API for JSON Processing 1.0* (JSON-P/JSR 353) gleich zwei neue APIs in Java EE aufgenommen wurden, deren Fokus klar in Richtung Modern & High Scalable Web Applications ausgerichtet ist.

Das WebSocket API hat sich zum Ziel gesetzt, den offiziellen WebSocket-Standard [3] auch innerhalb des Enterprise Java Stacks zugänglich zu machen. Dies ist insbesondere für Anwendungen von Interesse, die von häufigen, serverseitigen Datenupdates leben. Stock Ticker, Chat-Anwendungen und Live Maps sind nur einige von unzähligen Aspiranten. Möglich wird dies durch den bidirektionalen Kommunikationsansatz der WebSocket-Verbindung, welcher die typischen Problemfelder rein HTTP-basierter Lösungen, wie Latency, Skalierung und Performanz, umgeht. Das neue WebSocket API spezifiziert, wie sich mittels Java WebSocket-Verbindungen aufbauen lassen, wie WebSocket Events initiiert und intercepted werden, wie man WebSocket-Nachrichten verarbeitet und vieles mehr. Natürlich werden im Kontext der Java-EE-Spezifikation auch übergreifen-

de Themen angegangen und zum Beispiel definiert, wie sich WebSocket-Anwendungen innerhalb des Java-EE-Security-Modells verhalten.

Auch das neue JSON Processing API versucht eine bestehende Lücke in der Java-EE-(Web-)Welt zu schließen. Während JSON durchaus als etabliert bezeichnet werden kann und der Java-EE-Standard bereits in der Lage ist, entsprechende Streams via (RESTful) Web Services zu erzeugen, mangelt es nach wie vor an einer standardisierten Art und Weise, wie sich JSON-Objekte schreiben bzw. einlesen lassen. Die neue Spezifikation bietet daher entsprechende APIs, die sowohl einen Streaming-basierten Ansatz (ähnlich StAX für XML) erlauben, als auch einen API-basierten (ähnlich DOM API für XML). Nicht dagegen im Fokus des JSON Processing API ist das direkte Binding von JSON-Objekten zu Java-Objekten – vice versa.

Passend zu den eben beschriebenen APIs und dem oben aufgezeigten Motto der aktuellen Java-EE-Spezifikation wurde auch dem *Java API for RESTful Web Services* (JAX-RS/JSR 339) ein Face-Lifting spendiert, welches neben vielen anderen sinnvollen Änderungen und Erweiterungen, wie zum Beispiel Filter und Interceptoren, u.a. endlich auch einen genormten REST-Client spezifiziert. Details zu den wesentlichen Änderungen und Erweiterungen dieses API finden sich in Thilo Frotschers Artikel auf Seite 21 ff. in diesem Heft.

Und natürlich kommt auch das Web-UI-API – Java-Server Faces – nicht drum herum, sich an die wundersame neue Welt des HTLM5 anzupassen. Mit JSF 2.2 (JSR 344) wird es in Java EE 7 eine stark überarbeitet Version des aktuellen Standards geben. Neben einigen Anpassungen in Richtung HTML5 bringt die zukünftige JSF-Version mit hoher Wahrscheinlichkeit auch eine Reihe neuer Konzepte mit sich, die derzeit zum Teil noch heftig diskutiert werden. So soll es zum Beispiel, in Anlehnung an die ADF Task Flows der Oracle Fusion Middleware, ab JSF 2.2 die Möglichkeit geben, so genannte Faces Flows zu deklarieren. Ein Faces Flow fasst eine Gruppe von Views zusammen und bestimmt gleichzeitig, wie innerhalb dieser Gruppe navigiert werden kann. Neben der regelbasierten Navigation – inkl. „GOTOs“ – sind auch Features wie Method Calls zwischen den einzelnen Navigationsschritte geplant, sodass Faces Flows schon beinahe als Mini-Workflow-Engine angesehen werden kann. Mehr hierzu und zu den weiteren Neuerungen in JSF 2.2 findet sich im Interview mit Andy Bosch auf der Seite 33 ff. in diesem Heft.

Die eben beschriebene JSF-Spezifikation macht noch einen weiteren Trend innerhalb der Java-EE-Spezifikation deutlich: *Vereinheitlichung*. Die in Java EE 5 und 6 eingebrachten Konzepte werden in Java EE 7 konsequent fortgeführt. Annotationen statt XML-Konfigurationen werden zur Norm. Die Integration von CDI wird deutlich in die Breite getragen. So wird zukünftig auch innerhalb von JSF Validatoren und Convertern das Injizieren von CDI Managed Beans möglich sein. Darüber hinaus wird JSF 2.2 mit hoher Wahrscheinlichkeit eine CDI Extension anbieten, die es erlaubt, den JSF-spezifischen View Scope auch im CDI-Kontext zu nutzen, sodass dessen Verwendung nicht – wie bisher – automatisch auch einen Verzicht auf CDI Managed Beans mit sich bringen muss. Dies ist insofern nicht unbedeutend, da mit der Spezifikation von JSF 2.2 auch zum ersten Mal laut darüber nachgedacht wird, das Package *javax.faces.beans* als Deprecated zu markieren und CDI Managed Beans offiziell als bessere Wahl zu empfehlen [4].

Zwei weitere, neue APIs sollen an dieser Stelle ebenfalls nicht unerwähnt bleiben: *JCACHE Temporary Caching API* (JSR 107) und *Batch Applications for the Java Platform* (JSR 352). Das Caching API tritt an, um endlich eine standardisierte Lösung für ein Standardproblem im Java-SE/EE-Umfeld zu liefern, denn wohl kaum eine große (Web-)Anwendung kommt ohne eine eigenentwickelte oder kommerzielle Caching-Lösung aus. JSR 107 ist bereits etwas älter. Allerdings war es in den vergangenen Jahren ruhig um ihn geworden. Für Java EE 7 ist er wiederbelebt worden, auch um als Basis für JSR 347, die „Java Data Grid Specification“ zu fungieren, die es allerdings nicht in Java EE 7 schaffen wird, sondern zunächst einmal auf Java EE 8 verschoben ist.

JCache basiert auf dem Prinzip des Object Chaching und erlaubt sowohl By-Value als auch – optional – By-Reference Caching. Vorgesehen ist neben einer In-Process-Variante auch die Option verteilter Lösungen. Lokale und verteilte Transaktionen sind ebenfalls Bestandteil der Spezifikation, wenn auch nur optional.

Wie schon zuvor das Caching API richtet sich auch die neue *Batch Applications for the Java Platform*-Spezifikation an Java SE und EE. Wie es der Name der Spezifikation schon vermuten lässt, können mit ihrer Hilfe Batch Jobs – inklusive Ablaufsteuerung – definiert, angelegt und ausgeführt werden. Die Spezifikation definiert dazu eigene, deploybare Einheiten – Batch Applications – und spezielle Container – Batch Executer. Ein Batch besteht dabei aus Steps, die wiederum aus dem Lesen, Verarbeiten und Schreiben von Items bestehen können. Wer schon einmal mit Spring Batch [5] gearbeitet hat, dem dürften das Vorgehen und die Terminologie bekannt vorkommen. Etwas befremdlich wirkt, dass in der gesamten Spezifikation keine Referenz auf andere Java-EE-Spezifikationen zu finden ist, und somit z. B. keine standardisierte Integration in dem restlichen Java EE Stack existiert. Die Definition eines Batches erfolgt über XML und die Referenzierung von Java-Objekten über ihren Namen. In diesem Punkt fällt die Spezifikation im Rahmen von Java EE etwas aus der Reihe, da alle anderen Spezifikationen bestrebt sind, vermehrt auf Annotations und Typsicherheit zu setzen.

Kosmetische Korrekturen

Eher kosmetischer Natur sind die Änderungen in den übrigen APIs von Java EE. Bean Validation 1.1 (JSR 349) wird als wesentliches Feature um *Method Level Validation* erweitert. Mithilfe dieses Features soll es anderen Technologien – wie zum Beispiel CDI – ermög-

licht werden, Methodenparameter und -rückgabewerte via Bean Validation zu validieren. Zusätzlich soll die bisher lediglich in JSF 2 und JPA 2 stattgefundene Integration auch auf andere APIs, wie JAX-RS oder JAXB, ausgeweitet werden. Innerhalb von JSF sollen darüber hinaus – neben den bereits vorhandenen Property Level Constraints – zusätzlich auch Class Level bzw. Cross-Parameter Constraints unterstütz werden, um so auch Validierungen über mehrere Parameter einer Managed Bean hinweg zu erlauben.

In ähnlich homöopathischen Dosen spielen sich die Änderungen in CDI 1.1 (JSR 346) ab, was durchaus als ein Zeichen für die bereits gute Qualität der CDI-1.0-Spezifikation gedeutet werden kann. Bei den meisten Änderungen handelt es sich um *Clarifications*, also Stellen, an denen die Spezifikation ungenau oder mehrdeutig war. Diese sind als direktes Resultat des Community-Feedback zu CDI 1.0 entstanden.

Als Beispiel für ein echtes neues Feature dagegen kann das *Priority Ordering* aufgeführt werden, welches es zukünftig erlauben wird, eine Aufrufreihenfolge für Interceptors und Decorators auf Basis von Prioritäten anzugeben. Dieses bisher fehlende Feature führt bei CDI 1.0 immer wieder zu Problemen im praktischen Einsatz. Ebenfalls interessant ist die Möglichkeit, in Szenarien, in denen man keine Injection und auch keine Referenz auf den BeanManager zur Verfügung hat, auf CDI Beans zuzugreifen. Dies ist ab CDI 1.1 über die statische Methode *CDI.current()* möglich.

Natürlich wird es auch Änderungen in der EJB-Spezifikation geben, die sich direkt auf Java EE 7 auswirken. Die wohl wesentlichste Änderung in EJB 3.2 ist der bereits oben beschriebene optionale Wegfall der allseits unbeliebten Entity Beans aus EJB 1.x und 2.x. Diese müssen derzeit noch, aus Gründen der Abwärtskompatibilität, von jedem Java EE Full Profile Container unterstützen werden. Zukünftig dagegen bleibt es dem Hersteller freigestellt, ob er dies tun möchte oder nicht. Entsprechend findet sich das zugehörige Feature auch nicht mehr in der eigentlichen EJB-Spezifikation (JSR 345: Enterprise JavaBeans, Version 3.2 EJB Core Contracts and Requirements), sondern wurde in ein eigenes Dokument verschoben (JSR 345: Enterprise JavaBeans, Version 3.2 EJB Optional Features).

Ebenfalls interessant ist die Tatsache, dass es Überlegungen gibt, klassische EJB-Funktionalitäten auch anderen APIs zur Verfügung zu stellen, indem sie in eigene Subspezifikationen ausgelagert werden. Ein erstes praktisches Beispiel hierfür gibt es bereits seit EJB 3.1 in Form des Interceptor API 1.1 [6], das es u. a. erlaubt, allgemeine Interceptors zu implementieren, die dann zum Beispiel im CDI-Umfeld verwendet werden können. Für Java EE 7 ist etwas Ähnliches für den Bereich der Transaktionen geplant. Es soll eine eigene *@Transactional*-Annotation geben, hinter der sich eine EJB-unabhängige, aber trotzdem durch den Container gemanagte Transaktionssteuerung verbirgt [7].

Dass auch die EJB Expert Group empfänglich für das Feedback der Community ist, zeigt sich anhand der Aufwertung des EJB Lite API Subsets. Ab EJB 3.2 wird es sowohl möglich sein, asynchrone Session Beans als auch nicht-persistente Timer innerhalb eines Java-EE-Web-Profile-konformen Servers zu nutzen. Beides sind Features, die bisher von vielen vermisst wurden.

Bleibt noch das Java Persistence API 2.1. Die meisten der angedachten Änderungen für diese derzeit als Early Draft 2 vorliegende Spezifikation zielen darauf hin, dem Entwickler mehr Einfluss auf Mappings und Datenbankzugriffe – insbesondere Abfragen – zu ermöglichen. So wird zum Beispiel für JPQL und Criteria API ein neues Schlüsselwort *ON* eingeführt, mit dessen Hilfe – wie aus SQL bekannt – Outer Joins mit ON-Conditions ausgeführt werden können. Ebenfalls neu ist das Schlüsselwort *TREAT*, das ein Downcasting innerhalb einer Query ermöglicht und so völlig neue Perspektiven für Abfragen erlaubt. Das bereits aus JPA 2.0 bekannte

Aufrufen von DB- und User-Functions via Criteria API wird ab JPA 2.1 – dank *FUNCTION(...)* – auch innerhalb einer JPQL-Abfrage möglich sein. Freunde des Criteria API werden sich außerdem darüber freuen, dass zukünftig sowohl Bulk Updates als auch Bulk Deletes mithilfe eigens dafür eingeführten Interfaces möglich werden.

Als wirkliche Innovation ist die angedachte Unterstützung von *Stored Procedures* zu sehen. Bisher waren Aufrufe von *Stored Procedures* nur über *createNativeQuery* möglich. Out-Parameter konnten nach der Ausführung gar nicht abgefragt werden. Zwar wird auch in JPA 2.1 lediglich der Aufruf einer Stored Procedure und nicht deren Definition möglich sein, Out-Parameter können aber nach dem Aufruf abgefragt werden. Alleine dieses Feature ist wahrscheinlich vielen Entwicklern schon den Umstieg bzw. Aufstieg auf JPA 2.1 wert. Ebenfalls interessant dürfte für den einen oder anderen JPA-Anwender die Möglichkeit sein, das Ergebnis einer nativen Query via Constructor Result – einer speziellen Mapping-Information – direkt in gemanagte Entitäten zu überführen. Diese befinden sich dann allerdings im Status *new* oder *detached*. Ein Überbleibsel der Cloud-Fokussierung innerhalb der JPA-2,1-Spezifikation ist die Möglichkeit, in der *persistence.xml* Referenzen auf Datenbankskripte zu hinterlegen, die eine neue Datenbank anlegen, mit Daten befüllen oder droppen können.

Und auch innerhalb der JPA-2.1-Spezifikation – wie auch in etlichen anderen Java-EE-7-Spezifikationen – finden wir eine verbesserte Integration von CDI. War es bisher zum Beispiel nicht möglich innerhalb eines Entity Listener eine CDI Managed Bean zu injizieren, wird dies zukünftig kein Problem mehr sein.

Last but not least

Einer besonderen Erwähnung ist die nach gefühlten 100 – und realen 10 – Jahren mehr als überfällige Überarbeitung des Java Message Service wert (JSR 343). Das gesamte Programmiermodell von JMS wurde deutlich überarbeitet und stark vereinfacht. Für alle denkbaren JMS-spezifischen Ressourcen wird es entsprechende Annotationen geben. Und auch CDI-Unterstützung wird nicht fehlen. So lässt sich zukünftig zum Beispiel ein *MessageContext* einfach via *@Inject* injizieren. Näheres zu den Änderungen und Neuerungen in JMS 2 findet sich in dem passenden Artikel von Thilo Frotscher auf Seite 26 ff.

Ebenfalls einer besonderen Erwähnung ist die neue Version 3.0 der Expression Language wert. Zum einen, weil sie zukünftig auch außerhalb eines Web Containers genutzt werden kann, und zum anderen, da sie etwas geschafft hat, was der aktuellen Java-SE-Spezifikation verwehrt geblieben ist. Nach aktuellem Stand wird die EL 3.0 nämlich Lambda Expressions unterstützen. Wer also schon einmal einen Eindruck davon bekommen möchte, was uns in Java SE 8 erwartet, der kann dies in der EL-3.0-Spezifikation in Abschnitt 1.20 nachlesen.

Fazit

Es erscheint sinnvoll, dass von den ehemaligen Zielen *Cloud* und *Multi-Tenancy* zunächst wieder abgerückt wurde und mit *Productivity* & *HTML5* deutlich realistischere Ziele in Angriff genommen worden sind. Zu unausgegoren und uneinheitlich waren die bis dato vorgestellten Ansätze für mandantenfähige PaaS-Provider. Schade nur um die verlorene Zeit, denn mit seinen mittlerweile mehr als drei Jahren bricht die Java-EE-7-Spezifikation leider alle Negativrekorde.

Auch wenn der aktuelle Stand der Spezifikation mittlerweile (wieder) einen recht runden Eindruck macht, gibt es nach wie vor noch den einen oder anderen Punkt, der weiterer Klärung bedarf. Insbesondere die Frage nach der genauen Ausgestaltung des Web Pofile in Abgrenzung zum Full Profile, sowie nach einer verbesserten Einbettung von CDI in Java EE sind immer noch „under discussion“. Hier ist nun insbesondere die Community gefragt und jeder – wirklich jeder – sollte diese Chance nutzen. In diesem Sinne: „Help create your favorite JSRs.“

Java EE auf der JAX

Wollen Sie mehr über die hier besprochenen Themen erfahren? Auf der JAX 2013 wird es wieder viele spannende Talks rund um Java EE geben: www.jax.de.

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] Java-EE-7-Spezifikation: http://jcp.org/en/jsr/detail?id=342

[2] Java EE 7 Feedback Survey: https://www.surveymonkey.com/s/javaee7

[3] WebSocket-Standard: http://tools.ietf.org/html/rfc6455

[4] Future of JSF Managed Beans and CDI: http://java.net/jira/browse/JAVASERVERFACES_SPEC_PUBLIC-976

[5] Spring Batch: http://static.springsource.org/spring-batch/

[6] EJB 3.1 und Interceptor API 1.1: http://jcp.org/aboutJava/communityprocess/final/jsr318

[7] Transactional Interceptors: https://blogs.oracle.com/ldemichiel/entry/transactional_interceptors

Ein erster Blick auf die wichtigsten Neuerungen in JAX-RS 2.0

Rund gemacht

In Java EE 6 war mit JAX-RS 1.1 erstmals standardisierte Unterstützung für REST-basierte Anwendungen enthalten. Während diese Version im Allgemeinen sehr gut angenommen wurde, zeigte sich schon bald, dass das Eine oder Andere noch fehlte, um JAX-RS wirklich rund zu machen. Insbesondere existierte bislang kein standardisiertes Client-API. Das und vieles andere soll in JAX-RS 2.0 nun nachgereicht werden.

von Thilo Frotscher

Die Arbeiten an JAX-RS 2.0 sind bereits seit 2011 im Gange. Kurz vor Jahresende 2012 liegt die Spezifikation nun als Public Review vor. Die Referenzimplementierung, die wie gehabt im Rahmen des Jersey-Projekts [1] erstellt wird, hat den Meilenstein 9 erreicht. Somit sind die Arbeiten zwar noch nicht vollständig abgeschlossen, und es kann nicht ausgeschlossen werden, dass sich das Eine oder Andere kurzfristig noch ändert. Das wesentliche Erscheinungsbild von JAX-RS 2.0 dürfte aber inzwischen feststehen. Grund genug also, sich genauer anzusehen, was Entwickler von dem neuen API erwarten dürfen. Im Folgenden werden die wichtigsten Neuigkeiten vorgestellt.

Client-API

Die sicherlich wichtigste Neuerung in JAX-RS 2.0 besteht in der Standardisierung des Client-API. Zwar konnten auch bislang schon Clients für REST-Anwendungen mit den diversen Frameworks implementiert werden, nur verwendeten diese eben alle proprietäre Client-APIs. Bei der Entwicklung des Standard-API für die Clientseite wurden nun Ideen und bewährte Konzepte sowohl aus diversen proprietären Lösungen, als auch von dem bereits standardisierten serverseitigen API zusammengeführt.

Als Einstiegspunkt in das neue Client-API dient die Klasse *ClientFactory*. Wie der Name vermuten lässt, dient sie als Fabrik für Objekte, die das Interface *Client* implementieren, und bietet hierfür entsprechende Methoden. Die Implementierungsklasse des zurückgelieferten *Client* ist spezifisch für das eingesetzte Framework. Im Falle von Jersey handelt es sich beispielsweise um die Klasse *JerseyClient*.

Hat man einen *Client* erhalten, können dort optional Provider oder Features registriert werden. Provider dienen dazu, Erweiterungen und Anpassungen für das Framework zu realisieren. Die bereits von dem serverseitigen API bekannten *MessageBodyReader* und *MessageBodyWriter* sind Beispiele für solche Provider. In ihrem Fall besteht die Erweiterung daraus, spezifische Umwandlungen zwischen Ressourcen und bestimmten Repräsentationen zu implementieren, die das Framework von Haus aus nicht unterstützt.

Während Provider bereits aus JAX-RS 1.1 bekannt waren, sind Features ein neues Konzept in JAX-RS 2.0. Sie sind letztlich nichts anderes als ein spezieller Typ von Provider, sie sollen also ebenfalls der Erweiterung des Frameworks dienen. Der Unterschied zwischen beiden liegt darin, dass Features eine umfangreichere Erweiterung kapseln sollen, die beispielsweise aus mehreren Providern und gegebenenfalls zusätzlichen Laufzeitkonfigurationen bestehen könnte. Das zugehörige Interface *Feature* definiert dazu eine Methode namens *configure*, die vom Framework in der Bootstrapping-Phase aufgerufen wird. So haben Features die Möglichkeit, die von ihnen implementierte Erweiterung zur Laufzeit zu konfigurieren, also etwa zusätzliche Provider zu registrieren oder bestimmte Properties zu setzen. Die folgenden Codezeilen demonstrieren, wie ein *Client* mit JAX-RS 2.0 erstellt wird, und wie Provider und Features registriert werden können:

```
Client client = ClientFactory.newClient();
client.register(OrderReader.class);
client.register(OrderWriter.class);
client.register(MyFeature.class);
client.setProperty("myProperty", aValue);
```

Darüber hinaus dient ein *Client* dazu, ein so genanntes *WebTarget* zu erzeugen, zum Beispiel unter Angabe eines *URI* oder eines *Link* (mit der Klasse *Link* werden in JAX-RS 2.0 nun auch Hypermedia-Links unterstützt). Ein *WebTarget* repräsentiert eine bestimmte Ressource, kapselt deren URI und bietet diverse Methoden, mit denen z. B. der Pfad erweitert, Query-Parameter gesetzt oder spezifische Repräsentationen angefordert werden können. Die Methode *request()* liefert schließlich einen *Invocation.Builder*, der beim Aufbau bzw. der Vorbereitung eines Requests an das *WebTarget* behilflich ist.

Listing 1

```
List<Order> orders = ...
WebTarget target = client.target("http://server/app/orders/{orderId}");

for (Order order : orders) {
  Response r = target.resolveTemplate("orderId", order.getId())
                     .request()
                     .put(Entity.xml(order));
  // ... process Response
}
```

Listing 2

```
WebTarget ordersWT = client.target("http://.../orders/latest");
Invocation latestOrderReq = ordersWT.request("text/xml")
                                    .header("myHeader1", "value1")
                                    .header("myHeader2", "value2")
                                    .buildGet();
while (keepPolling) {
  Response res = latestOrderReq.invoke();
  // ...
}
```

Listing 3

```
@Provider
public class AuthenticationFilter implements ContainerRequestFilter {

  @Override
  public void filter(ContainerRequestContext req) throws IOException {
    URI loginUri = null;
    String authToken = req.getCookies().get("myAuthToken").getValue();

    if (!isValid(authToken)) {
      req.abortWith(Response.temporaryRedirect(loginUri).build());
    }
  }

  private boolean isValid(String authToken) {
    // ...
  }
}
```

Die Ausführung des Requests erfolgt dann beispielsweise durch Aufruf der Methoden *get*, *put* oder *post*. Dabei kann angegeben werden, in welchen Java-Typ die Response umgewandelt werden soll. Dies alles wurde als so genanntes Fluent-API entworfen:

```
Money balance = client.target("http://.../account/12345/balance")
                      .queryParam("pin", "1234")
                      .request(MediaType.TEXT_XML)
                      .get(Money.class);
```

Ebenfalls unterstützt wird die Verwendung von URI Templates. Hierzu werden in bekannter Weise Platzhalter mithilfe von geschweiften Klammern definiert, die anschließend mit der Methode *resolveTemplate* aufgelöst werden können (Listing 1). Eine *Response* kapselt sämtliche Informationen einer HTTP-Response, wie Headers, oder einer enthaltenen Entity. Hier wurde die von der Serverseite aus JAX-RS 1.1 bekannte Klasse wiederverwendet. Neu ist jedoch auch hier die Unterstützung für Hypermedia-Links.

Eine weitere interessante Möglichkeit besteht darin, Requests mithilfe des API zunächst nur zusammenzubauen, jedoch noch nicht unmittelbar abzuschicken. Dazu bedient man sich einer der verschiedenen *build…()*-Methoden, die ein Objekt von Typ *Invocation* zurückliefern. Auf diese Art und Weise könnte man etwa mehrere Requests zunächst vorbereiten und zu einem späteren Zeitpunkt in einer Art Batch-Lauf abschicken. Ein anderer Einsatzzweck besteht darin, einen spezifischen Request, der wiederholt benötigt wird, mehrmals wiederzuverwenden anstelle jedes Mal einen neuen, immer gleichen Request zusammenzubauen (Listing 2).

Filter

Auch Filter sind eine spezielle Art von Provider und dienen der Erweiterbarkeit des Frameworks. Ihr hauptsächlicher Verwendungszweck ist die Realisierung von Querschnittsfunktionen wie Logging oder Security. Hierzu können sie die Header von Requests oder Responses verarbeiten und gegebenenfalls modifizieren. Filter können sowohl auf der Serverseite als auch auf dem Client zum Einsatz kommen und werden dort jeweils in Ketten verwaltet (Request Chain und Response Chain), die sequenziell abgearbeitet werden. Auf der Serverseite ist der Container dafür zuständig, eingehende Requests zunächst durch alle Filter der Request Chain zu leiten, bevor sie an die jeweilige Zielressource übergeben werden. Analog wird eine von der Ressource erzeugte Response vom Container zunächst durch die einzelnen Filter der Response Chain geleitet, bevor sie letztendlich versandt wird. Auf der Clientseite kümmert sich dagegen das Client-API darum, einen Request vor seinem Versand einerseits und die eingehende Response vor ihrer Auslieferung an die Clientanwendung andererseits durch die clientseitig konfigurierten Filterketten zu leiten.

Filter zeichnen sich dadurch aus, dass sie mindestens eines der beiden Interfaces *ContainerRequestFilter* oder *ContainerResponseFilter* implementieren. Diese definieren jeweils nur eine Methode namens *filter*, die ein Kontextobjekt als Parameter erwartet. Mit dessen Hilfe erhalten die Filter Zugriff auf den gerade zu verarbeitenden Request oder die Response. Dabei ist es auch möglich, dass Filter ihre Kette unterbrechen, indem sie einen Fehler erzeugen. So könnte etwa ein Sicherheitsfilter auf der Serverseite im Falle einer fehlgeschlagenen Authentifizierung durch den Client eine Response mit dem Status 401 (Unauthorized) erzeugen und an diesen zurück senden. In einem solchen Fall würden weder die restlichen Filter der Request Chain, noch die eigentlich vom Request adressierte Ressource erreicht (Listing 3). Es ist zu beachten, dass Filter ebenso wie alle anderen Arten von Providern mit der Annotation *@Provider* zu markieren sind.

Standardmäßig haben alle bei der JAX-RS Runtime registrierten Filter globale Gültigkeit, das heißt sie werden für sämtliche Ressourcen ausgeführt. Alternativ können mithilfe der Metaannotation *@NameBinding* so genannte Name-Binding-Annotationen erstellt werden. Filterklassen, die mit einer Name-Binding-Annotation markiert sind, kommen nur für solche Ressourcen oder Ressourcenmethoden zum Einsatz, die die gleiche Name-Binding-Annotation aufweisen (Listing 4).

Listing 4

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(value = RetentionPolicy.RUNTIME)
@NameBinding
public @interface Logged { }


@Logged
public class LoggingFilter
      implements ContainerRequestFilter, ContainerResponseFilter {
   ...
}


@Path("/")
public class OrderResource {
   @GET
   @Produces("text/xml")
   @Path("{orderId}")
   @Logged
   public String getOrder(@PathParam("orderId") long id) {
    Order order = getOrderById(id);
       return order;
   }
}
```

Interceptors

Wie wir gesehen haben, kommen Filter auf der Serverseite entweder zum Einsatz bevor oder nachdem die adressierte Ressource einen eingehenden Request verarbeitet. Dies ist soweit auch sinnvoll, da ein Filter entweder die Header des Requests oder die Header der darauf folgenden Response verarbeitet. Analog kommen Filter auf der Clientseite entweder zum Einsatz bevor oder nachdem ein Request versendet wird. Insbesondere werden die Filter somit durchlaufen, bevor empfangene Entities vom zuständigen *MessageBodyReader* aus dem *InputStream* gelesen, bzw. nachdem zu versendende Entities vom zuständigen *MessageBodyWriter* in den *OutputStream* geschrieben werden. Folglich fließen diese Entity-Streams nicht durch die Filter. Was aber tun, wenn eine Erweiterung benötigt wird, die Entities manipulieren kann?

Hierfür sind die so genannten (Entity-)Interceptors vorgesehen. Sie kapseln den Aufruf von *MessageBodyReader* und -*Writer*, haben somit direkten Zugriff auf die Entity-Streams und können ebenfalls verkettet werden. Durch den Zugriff auf den Entity-Stream ist es beispielsweise möglich eine Erweiterung zu schreiben, welche die GZIP-Kompression von Entities realisiert (Listing 5).

Analog zum API für Filter definiert JAX-RS 2.0 mit *ReaderInterceptor* und *WriterInterceptor* auch für Interceptoren zwei Interfaces, die zu implementieren sind. Auch diese definieren jeweils nur eine einzige Methode, die ein Kontextobjekt als Parameter erwartet. Neben dem Zugriff auf den Entity-Stream stellt das Kontextobjekt insbesondere die Methode *proceed* bereit. Jeder Interceptor muss (nach der eventuellen Manipulation des Entity-Streams) diese *proceed*-Methode aufrufen, um die Kontrolle an den nächsten Interceptor in der Kette weiterzugeben. Befindet sich kein weiterer Interceptor in der Kette, wird der zuständige *MessageBodyReader* oder -*Writer* aufgerufen. Das Binden von Interceptors an Ressourcen oder einzelne Ressourcenmethoden erfolgt wie bei Filtern mithilfe von Name Bindings.

Listing 5

```
@Provider
class GzipInterceptor implements ReaderInterceptor, WriterInterceptor {

  @Override
  public Object aroundReadFrom(ReaderInterceptorContext ctx) throws IOException {

    if (!"gzip".equals(ctx.getHeaders().getFirst("Content-Encoding"))) {
      // skip...
    }

    InputStream in = ctx.getInputStream();
    ctx.setInputStream(new GZIPInputStream(in));
    return ctx.proceed();
  }

  @Override
  public void aroundWriteTo(WriterInterceptorContext ctx) throws IOException {

    // Check if client sent "Accept-Encoding" => gzip
    // ...

    OutputStream old = ctx.getOutputStream();
    ctx.setOutputStream(new GZIPOutputStream(old));

    ctx.getHeaders().add("Content-Encoding", "gzip");
    ctx.proceed();
  }
}
```

Listing 6

```
@Path("/longRunningCalculation")
public class MyResource {

  @GET
  public void longRunningCalulation(@Suspended final AsyncResponse ar) {
    Executors.newSingleThreadExecutor().submit(
      new Runnable() {
        public void run() {
          ComplexCalculationResult calcResult = executeLongRunningCalculation();
          ar.resume(calcResult);
        }
      }
    );
    // Return immediately and handle other requests
  }
}
```

Wenn's mal wieder etwas länger dauert ...

Eine weitere Neuigkeit von JAX-RS 2.0 besteht in der Unterstützung für asynchrone Abläufe, und zwar sowohl auf Seiten des Servers, als auch auf der Clientseite. Auf der Serverseite kann es zum Beispiel Fälle geben, in denen die Beantwortung eines Requests sehr viel Zeit beansprucht. Dies kann etwa daran liegen, dass mit der Response eine sehr große Entity verschickt werden soll oder eine Entity, deren Erstellung schlicht lange dauert. Ein HTTP Worker Thread, der einen solchen Request entgegennimmt und in den Aufruf einer lange laufenden Ressourcenmethode umwandelt, muss all diese Zeit warten. Er ist somit blockiert und kann derweil keine anderen Requests entgegennehmen. Mit JAX-RS 2.0 wird es nun möglich sein, solche langwierigen Abläufe in einen separaten Thread auszulagern, der vom HTTP Worker Thread entkoppelt ist. Der Request wird sozusagen vorübergehend geparkt, sodass der HTTP Worker Thread anstelle zu warten in der Zwischenzeit andere Requests bedienen kann. Sobald der von der Ressourcenmethode gestartete Thread seine Arbeit beendet hat, wird der geparkte Request wiederaufgenommen und die Response geschrieben.

Um diesen neuen Mechanismus nutzen zu können, muss die Ressourcenmethode einen Parameter vom Typ *AsyncResponse* haben, der zudem mit der Annotation *@Suspended* versehen ist. Hieran erkennt der Container, dass Requests an diese Ressourcenmethode geparkt werden sollen. *AsyncResponse* bietet dann eine Methode namens *resume*, die von der Ressourcenmethode aufgerufen werden muss, wenn die lange dauernde Arbeit getan ist, um den geparkten Request zu reaktivieren. Beim Aufruf von *resume* kann ein Objekt übergeben werden, das dann als Entity in der Response versandt wird (Listing 6). Alternativ kann auch ein *Throwable* übergeben werden.

Standardmäßig ist in *AsyncResponse* kein Timeout gesetzt, die zugehörigen Requests werden also unendlich lange geparkt. Alternativ kann mit den Methoden *setTimeout* und *setTimeoutHandler* sowohl ein Zeitlimit gesetzt als auch ein Handler übergeben werden, der dessen Überschreitung behandelt.

Falls eine Ressource als Stateless Session Bean implementiert wurde, lässt sich der gleiche Effekt noch einfacher als in Listing 6 erreichen, indem die lange laufende Ressourcenmethode mit *@Asynchronous* markiert wird. In diesem Fall kümmert sich der Container um die Ausführung der Ressourcenmethode in einem separaten Thread. Sie muss daher nicht selbst für das Starten eines Threads sorgen und die entsprechenden Codezeilen können somit entfallen (Listing 7).

Das Parken von Requests und der verzögerte Versand von Responses bieten vielfältige Möglichkeiten. So lassen sich etwa Subscribe/Notify-Szenarien und Long-Polling-Clients realisieren.

Auf der Clientseite bietet das neue Client-API von JAX-RS 2.0 die Möglichkeit, Requests asynchron zu versenden. Das bedeutet, dass auch hier ein separater Thread gestartet wird, der den Versand des Requests und das Warten auf die Antwort übernimmt, während die Clientanwendung unmittelbar weiterarbeiten kann und nicht blockiert ist. Hierzu dient die Methode *async()* des bereits erwähnten *Invocation.Builder*, die einen *AsyncInvoker* zurückliefert. Dieser bietet Methoden zum asynchronen Versand von Requests (GET, PUT, POST, DELETE etc.). Es besteht dann die Möglichkeit, entweder mithilfe des zurückgelieferten *Future*-Objektes zu pollen, oder ein *InvocationCallback*-Objekt zu übergeben, das die Response nach ihrem Eintreffen verarbeitet (Listing 8).

Listing 7

```
@Stateless
@Path("/longRunningCalculation")
public class MyResource {

  @GET
  @Asynchronous
  public void longRunningCalulation(@Suspended final AsyncResponse ar) {
    ComplexCalculationResult calcResult = executeLongRunningCalculation();
    ar.resume(calcResult);
  }
}
```

Listing 8

```
InvocationCallback<List<Order>> callback =
  new InvocationCallback<List<Order>>() {
    @Override
    public void completed(List<Order> orders) {
      // process orders
    }

    @Override
    public void failed(ClientException arg0) {
      // handle error
    }
  };

client.target("http://.../orders/")
      .request("text/xml")
      .async()
      .get(callback);
```

Los geht's!

Das standardisierte Client-API, Filter und Interceptors sowie die Unterstützung für asynchrone Abläufe stellen sicherlich die Highlights von JAX-RS 2.0 dar. Darüber hinaus gibt es jedoch noch viele weitere Neuigkeiten zu entdecken. Laut offiziellem Zeitplan für Java EE 7 ist Ende April 2013 als Erscheinungsdatum vorgesehen. Wer bereits heute die neuen Features von JAX-RS 2.0 ausprobieren möchte, muss nicht so lange warten, sondern kann entweder den aktuellen Milestone des Jersey-Projekts oder einen der so genannten Promoted Builds [2] von GlassFish 4 herunterladen. Insbesondere Entwicklerteams, die gerade mit einem neuen Projekt beginnen, oder die eine Fertigstellung in den nächsten Monaten planen, sollten sich die Neuigkeiten ansehen. Unter anderen ist zu überlegen, ob es sinnvoll ist, jetzt noch mit der Entwicklung eines REST-Clients zu beginnen, der auf einem veralteten, proprietären API basiert. In vielen Fällen wird es sich lohnen, gleich auf den neuen Standard zu setzen.

Thilo Frotscher arbeitet als freiberuflicher Softwarearchitekt und Trainer mit den Schwerpunkten Enterprise Java, Services und Integration. Er unterstützt seine Kunden durch die Mitarbeit in Projekten sowie mit der Durchführung von Reviews und Trainings. Daneben ist er regelmäßig auf internationalen Fachkonferenzen als Sprecher anzutreffen. Zu seinen Publikationen zählen zahlreiche Fachartikel und mehrere Bücher.

www.frotscher.com

Links & Literatur

[1] Jersey: http://jersey.java.net

[2] GlassFish 4, Promoted Builds: http://download.java.net/glassfish/4.0/

JMS 2.0 bietet Entwicklern die lang erwartete Modernisierung

Runderneuert

Vor über zehn Jahren, anno 2002, erschien J2EE 1.4. Zu den Neuerungen zählten damals unter anderem JSP 2.0 inklusive Expression Language, EJB 2.1 oder Support für XML Schema. Im selben Jahr wurde der Euro als Bargeld in Umlauf gebracht, Rudi Völler war Bundestrainer und… Das alles scheint eine halbe Ewigkeit her zu sein. Ist es auch. Aus diesem Jahr stammt das noch immer aktuelle API für JMS 1.1.

von Thilo Frotscher

Java hat seitdem einen weiten Weg zurückgelegt und viele mächtige Erweiterungen erhalten, sowohl als Sprache als auch als Plattform. So wurde die Sprache etwa um Generics, Enumerations und Annotations erweitert. Zudem gibt es nun *try*-Statements mit automatischer Verwaltung von Ressourcen. Was die Java-EE-Plattform angeht, so brachte Java EE 5 zunächst einige deutliche Vereinfachungen für Entwickler. Zu den wichtigsten Stichpunkten zählten hier „Convention over Configuration", der überwiegende Verzicht auf Checked Exceptions sowie der breite Einsatz von Annotations. Mit Java EE 6 hielt dann auch Dependency Injection Einzug in die Plattform.

Von alledem kam JMS lediglich Dependency Injection zugute. So können Ressourcen wie Connection Factory, Queues oder Topics inzwischen injiziert werden. Davon abgesehen fühlen sich Entwickler, die JMS-Anwendungen entwickeln, beim Schreiben des Codes jedoch in lange vergangene Zeiten zurückversetzt. Der Code fühlt sich umständlich an. Und es ist fast schon unglaublich, wie viele Codezeilen notwendig sind, nur um eine einzige Nachricht zu senden oder zu empfangen. Dies fällt ganz besonders dann auf, wenn man die restliche Anwendung mit aktuellen APIs entwickelt.

Doch Abhilfe ist endlich in Sicht. Denn im Zuge der Arbeiten für Java EE 7 wurde nun auch eine Modernisierung des JMS-API beschlossen (JSR-343). Kurz vor Ende des Jahres 2012 liegt die Spezifikation zwar noch immer im Early Draft vor. Da die Fertigstellung von Java EE 7 jedoch für Ende April 2013 geplant ist und zu diesem Zeitpunkt folglich auch JMS 2.0 fertiggestellt sein muss, lohnt sich bereits jetzt ein Blick auf die zu erwartenden Änderungen. Dabei sollte jedoch bedacht werden, dass sich das eine oder andere bis dahin durchaus noch ändern kann.

Listing 1: Versand einer Nachricht mit JMS 1.1

```
@Resource(lookup = "java:global/jms/myConnectionFactory")
ConnectionFactory myConnectionFactory;

@Resource(lookup = "java:global/jms/myQueue")
Queue myQueue;

public void sendMessage(String msg) {
  try {
    Connection conn = myConnectionFactory.createConnection();
    try {
      Session session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageProducer messageProducer = session.createProducer(myQueue);
      TextMessage textMessage = session.createTextMessage(msg);
      messageProducer.send(textMessage);
    } finally {
      conn.close();
    }
  } catch (JMSException ex) {
    // handle exception
  }
}
```

Warum umständlich, wenn's auch einfacher geht?

Wo die Probleme des aktuellen API liegen, wird durch einen Blick auf Listing 1 deutlich. Für den einfachen Versand einer Nachricht sind unglaubliche 13 Zeilen Code notwendig. Das ist unter anderem dadurch begründet, dass Ressourcen explizit geschlossen werden müssen und viele API-Methoden Checked Exceptions werfen, die vom Entwickler zu behandeln sind. Zudem sind vor dem eigentlichen Nachrichtenversand mit *Connection*, *Session* und *TextMessage* zunächst eine ganze Reihe von „Zwischenobjekten" zu erzeugen.

Eine umfassende Vereinfachung des API stand daher ganz oben auf der Wunschliste für JMS 2.0. Da andererseits die Rückwärtskompatibilität gewahrt werden soll, ist eine Vereinfachung nur durch Erweiterung möglich. Dabei wird eine zweistufige Strategie verfolgt. Zum einen wird das bestehende API erweitert. Zum anderen wird ein gänzlich neues API eingeführt.

Zur Erweiterung des bestehenden API werden den aus JMS 1.1 bekannten Schnittstellen neue Methoden hinzugefügt. So erhält *Connection* beispielsweise zusätzliche Methoden, mit denen das Erzeugen einer *Session* etwas leichter gelingt. Außerdem implementieren *Connection*, *Session*, *MessageProducer*, *MessageConsumer* und *QueueBrowser* nun *AutoCloseable*, sodass ein *try-with-resources*-Block verwendet werden kann, wodurch das explizite Schließen der Ressourcen entfällt. Listing 2

zeigt, dass der für den Nachrichtenversand notwendige Code auf diese Weise schon ein wenig reduziert wird.

Mal was Neues!

Das mit JMS 2.0 neu eingeführte API umfasst im Wesentlichen die drei neuen Interfaces *JMSContext*, *JMSProducer* und *JMSConsumer*. Eine zentrale Rolle nimmt dabei *JMSContext* ein. Es kombiniert die Funktionalität von *Session* und *Connection* aus dem alten API in einem einzigen Objekt und bietet Methoden zum Erzeugen von *JMSProducer* und *JMSConsumer*. Während die Konzepte einer Connection (Verbindung zum JMS Server) und einer Session (Single-Threaded Context zum Senden und Empfangen von Nachrichten) zwar weiterhin bestehen bleiben, gibt es mit dem neuen API für Entwickler jedoch keine Notwendigkeit mehr, diese Objekte explizit zu erzeugen oder zu benutzen. Der *JMSContext* erledigt all diese Schritte transparent im Hintergrund.

Der für den Nachrichtenversand vorgesehene *JMSProducer* vereinfacht die Entwicklung noch weiter. So muss im Unterschied zum alten API auch keine *Message* mehr erzeugt werden, um diese der *send*-Methode zu übergeben. Stattdessen übergibt man einen *String*, eine *Map*, ein Byte-Array oder ein *Serializable*-Objekt. Die Erzeugung einer *Message* des jeweils richtigen Typs übernimmt der *JMSProducer*. Darüber hinaus unterstützt der *JMSProducer* ein Message Chaining für das Setzen von Versandoptionen sowie von Headers und Properties von Nachrichten.

Listing 3 veranschaulicht, wie viel einfacher sich mit dem neuen API arbeiten lässt. Da natürlich auch *JMSContext* die Schnittstelle *AutoCloseable* erweitert, kann wieder ein *try-with-resources*-Block verwendet werden. Der Versand einer Nachricht gelingt so in fünf Zeilen Code, also nochmals deutlich weniger als noch in Listing 2.

Der *JMSConsumer* ist für den blockierenden oder nicht blockierenden Empfang von Nachrichten vorgesehen und hat ein ähnliches API wie der alte *MessageConsumer*. Allerdings bietet er einige neue *receive*-Methoden, die kein *Message*-Objekt zurückliefern, sondern direkt die in der Nachricht enthaltenen Daten. *JMSContext*, *JMSProducer* und *JMSConsumer* werfen ausschließlich Unchecked Exceptions.

Im Fall von Anwendungen, die im Web- oder EJB-Container laufen, kann ein *JMSContext* auch vom Container injiziert werden. Hierzu genügt die Markierung des *JMSContext* mit einer *@Inject*-Annotation. Mithilfe der optionalen Annotation *@JMSConnectionFactory* kann zusätzlich der JNDI-Name der *ConnectionFactory* angegeben werden, die für den *JMSContext* verwendet werden soll. Fehlt diese Annotation, gilt ein Default-Name. Ebenfalls optional sind die Annotationen *@JMSSessionMode* und *@JMSPasswordCredential*, mit denen weitere Hinweise an den Container gegeben werden können, welcher Gestalt der injizierte *JMSContext* sein soll.

Neben der Ersparnis einiger Codezeilen hat ein injizierter *JMSContext* den weiteren Vorteil, dass er am Ende der Transaktion auch automatisch vom Container wieder geschlossen wird. Er wird daher auch *Container-Managed Context* genannt. Der vom Entwickler zu schreibende Code zum Senden einer Nachricht wird so auf ein absolutes Minimum reduziert (Listing 4).

Sonstige Neuigkeiten

Abgesehen von der Vereinfachung des API sollen mit JMS 2.0 einige weitere Neuerungen eingeführt werden. So wird es künftig die Möglichkeit geben, Nachrichten

Listing 2: Versand einer Nachricht mit JMS 2.0

```
@Resource(lookup = "jms/myConnectionFactory")
ConnectionFactory myConnectionFactory;

@Resource(lookup = "jms/myQueue")
Queue myQueue;

public void sendMessage(String msg) throws JMSException {
  try (Connection conn = myConnectionFactory.createConnection();
       Session session = conn.createSession();
       MessageProducer producer = session.createProducer(myQueue);) {

    Message jmsMsg = session.createTextMessage(msg);
    producer.send(jmsMsg);

  } catch (JMSException e) {
    // exception handling
  }
}
```

Listing 3: Das neue API von JMS 2.0

```
@Resource(lookup = "java:global/jms/myConnectionFactory")
ConnectionFactory myConnectionFactory;

@Resource(lookup = "java:global/jms/myQueue")
Queue myQueue;

public void sendMessage(String msg) {
  try (JMSContext context = myConnectionFactory.createContext();) {
    context.createProducer().send(myQueue, msg);
  } catch (JMSRuntimeException ex) {
    // handle exception
  }
}
```

Listing 4: Container-managed JMSContext

```
@Inject
@JMSConnectionFactory("jms/myConnectionFactory")
private JMSContext context;

@Resource(mappedName = "jms/myQueue")
private Queue myQueue;

public void sendMessage (String msg) {
  context.createProducer().send(myQueue, msg);
}
```

verzögert ausliefern zu lassen. Zu diesem Zweck kann für jeden Producer ein Delivery Delay gesetzt werden. Er definiert die Minimalzeit, für die das Messaging-System eine Nachricht zurückhalten soll, bevor sie an einen Consumer ausgeliefert wird. Ein möglicher Einsatzzweck könnte beispielsweise darin bestehen, dass bestimmte Nachrichten erst am Ende eines Geschäftstags verarbeitet werden sollen.

Ebenfalls wird es einen nicht blockierenden Nachrichtenversand geben. Die Schnittstelle *MessageProducer* erhält zu diesem Zweck eine neue *send*-Methode, die die Kontrolle sofort an den Aufrufer zurückgibt, ohne zuvor auf ein Acknowledge des Servers zu warten. Der Methode ist neben der zu versendenden Nachricht zusätzlich ein *CompletionListener* zu übergeben. Dieser wird dann per Callback aufgerufen, sobald das Acknowledge eintrifft. Das gleiche Feature kann natürlich auch mit dem neuen *JMSProducer* genutzt werden (Listing 5).

Bereits in JMS 1.1 existierte die optionale Message Property *JMSXDeliveryCount*. Wird sie verwendet, erhöht der JMS Provider ihren Wert jedes Mal dann, wenn die Nachricht an einen Consumer ausgeliefert wird. Die Property kann sehr nützlich sein, um problematische Nachrichten zu finden, deren Auslieferung wiederholt scheitert. In JMS 2.0 soll sie daher obligatorisch werden.

Listing 5: Nicht blockierender Versand von Nachrichten

```
@Resource(lookup = "jms/myConnectionFactory")
ConnectionFactory myConnectionFactory;

@Resource(lookup = "jms/myQueue")
Queue myQueue;

public void sendMessage(String msg) {

  try (Connection connection = myConnectionFactory.createConnection();
       Session session = connection.createSession(false,
                                                  Session.AUTO_ACKNOWLEDGE);
    MessageProducer producer = session.createProducer(myQueue)) {

    TextMessage textMsg = session.createTextMessage(msg);
    producer.send(textMsg, new CompletionListener() {

  @Override
  public void onCompletion(Message msg) {
    // handle acknowledge
  }

  @Override
  public void onException(Message msg, Exception ex) {
    // handle exception
  }
    });

  } catch (JMSException ex) {
    // handle exception
  }
}
```

Darüber hinaus soll der Umgang mit Durable Subscriptions vereinfacht werden. Unter anderem sollen Container einen Default Subscription Name für Message-driven Beans generieren.

Eine weitere voraussichtliche Neuerung betrifft die Definition benötigter Ressourcen wie Connection Factory oder Queues und Topics in Java-EE-Anwendungen. Zwar können Entwickler mithilfe der *@Resource*-Annotation und der Angabe eines JNDI-Namens diese Ressourcen automatisch in den Anwendungscode injizieren lassen, jedoch ist es die Aufgabe desjenigen, der die Anwendung deployt, dafür zu sorgen, dass diese Ressourcen auch entsprechend angelegt oder konfiguriert werden. In JMS 2.0 soll es nun möglich sein, benötigte Connection Factories und Destinations zu definieren, und zwar entweder per Annotation (*@JMSConnectionFactoryDefinition*, *@JMSDestinationDefinition*) oder mithilfe eines Deployment Descriptors. Der Application Server kann diese Informationen dann verwenden, um diese Ressourcen automatisch anzulegen. Ein ganz ähnlicher Mechanismus existiert bereits für das automatische Erzeugen von Data Sources (*@DataSourceDefinition*).

Fazit

Mit dem neuen JMS 2.0 wird endlich die längst überfällige Modernisierung eines API geliefert, das in seiner aktuellen Version schlicht nicht mehr zeitgemäß ist. Der Hauptschwerpunkt liegt dabei ganz eindeutig darauf, seinen Einsatz für Entwickler deutlich zu vereinfachen. Das geschieht in zwei Schritten: Zum einen wird das existierende API vereinfacht, wo dies ohne Bruch der Rückwärtskompatibilität möglich ist. Zum anderen wird ein neues API eingeführt, das weniger (Zwischen-)Objekte erfordert. Der vom Anwendungsentwickler zu schreibende Code wird auf diesem Weg deutlich reduziert. Darüber hinaus bietet JMS 2.0 eine Reihe neuer Features, wie Nachrichtenversand mit zeitlicher Verzögerung oder nicht blockierender Versand. Auch wenn vielleicht noch nicht alles endgültig fertiggestellt ist und sich hier und da noch kleinere Änderungen bis zur finalen Verabschiedung des API ergeben können, lohnt bereits heute ein genauerer Blick auf den aktuellen Stand. Eines steht in jedem Fall bereits fest: Das Arbeiten mit JMS 2.0 wird deutlich einfacher von der Hand gehen und deshalb viel mehr Spaß machen.

Thilo Frotscher arbeitet als freiberuflicher Softwarearchitekt und Trainer mit den Schwerpunkten Enterprise Java, Services und Integration. Er unterstützt seine Kunden durch die Mitarbeit in Projekten sowie mit der Durchführung von Reviews und Trainings. Daneben ist er regelmäßig auf internationalen Fachkonferenzen als Sprecher anzutreffen. Zu seinen Publikationen zählen zahlreiche Fachartikel und mehrere Bücher.

www.frotscher.com

Links & Literatur

[1] JMS Specification Project Home: http://jms-spec.java.net

[2] JSR 343: http://jcp.org/en/jsr/detail?id=343

Anzeige

Anzeige

Anzeige

Anzeige

Interview mit Andy Bosch

„JSF hat sich extrem gut in der Industrie etabliert.“

Java Magazin: Hallo Andy! Wir stehen im Moment (Stand November) kurz vor dem Public Review Draft für JSF 2.2. Wie weit seid ihr gerade mit der Arbeit an JSF 2.2?

Andy Bosch: Wenn ich eine typische Entwicklerantwort geben darf: Fast fertig, bestimmt schon bei über 90 Prozent. Als guter Projektleiter weiß man aus dieser Antwort zu schließen, dass man ca. bei der Hälfte ist. Nein, im Ernst: Wir befinden uns tatsächlich auf der Zielgeraden. Es werden noch Feinabstimmungen in der Spezifikation besprochen, und es wird über Details diskutiert. Ein größeres Feature ist noch etwas stärker in der Diskussion, aber auch hier wird sich bald entscheiden, ob man es eventuell erst für JSF 2.3 dazu nimmt. Wenn dann Anfang Dezember der erste Public Review Draft veröffentlicht wird, wird hoffentlich auch einiges an Feedback auf die Expert Group zukommen. Das ist auch gut so und soll helfen, den Standard möglichst nah an den Wünschen der Community zu bauen. Bis die Community Zeit für den Review hatte und das Feedback ausgewertet und eingearbeitet wurde, vergehen dann sicherlich nochmals ein paar Wochen. Es ist aber auch so, dass JSF 2.2 Bestandteil von Java EE 7 sein soll. Und hier sieht der Zeitplan einen Termin im April 2013 vor. Man sieht somit, die Zügel sind straff angezogen und eine finale Version ist in greifbare Nähe gerückt.

JM: JSF besteht jetzt seit gut elf Jahren – in dieser Zeit hat sich in der Webwelt viel verändert. HTML5 und JavaScript haben das Web revolutioniert. Ist JSF auf die neuen Herausforderungen vorbereitet?

Bosch: In der Tat gibt es JSF schon seit einigen Jahren. Die Anfänge der Spezifikation wurden bereits 2001 gemacht, eine öffentliche Version wurde 2004 schließlich freigegeben. JSF hat sich extrem gut in der Industrie etabliert, sehr viele große und mittelständische Firmen setzen auf dieses Framework. Kritik an JSF gab und gibt es immer wieder. Und das ist auch gut so. Die Expert Group muss sich damit auseinandersetzen und passende Antworten liefern, die JSF fit für die Zukunft machen. Genau das passiert momentan, auch in JSF 2.2 sind wieder viele Anregungen aus der Community aufgenommen worden. Klar ist aber auch, dass man an der urtypischen Ausgestaltung dieses Standards, nämlich dass es sich um ein serverseitiges Komponentenframework handelt, kaum etwas rütteln wird. Es bleibt daher für die Zukunft spannend, wie sich JSF auf den Trend einstellt, dass immer mehr im Client, also größtenteils mit JavaScript, erfolgt und nicht mehr auf der Serverseite. An der großen Expert Group und der sehr regen Beteiligung der Community an JSF sieht man deutlich,

Porträt

jaxSpeaker

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 seit Neuestem Trainingsvideos zu JSF, CDI und Portlets bereit. Er ist Autor mehrerer Bücher zu JSF und hält regelmäßig Vorträge auf nationalen und internationalen Konferenzen.

dass JSF aktiv weiterentwickelt und für die Zukunft vorbereitet wird. Für mich persönlich ist es immer wichtig, dass Anregungen aus der Praxis an die Expert Group nicht auf einem Schreibtisch verstauben, sondern zeitnah in passende Konzepte eingearbeitet werden. Und genau das funktioniert bei JSF zum Glück sehr gut.

JM: Wie genau passt sich JSF dem aktuellen HTML5-Trend an?

Bosch: Hier sollte man zunächst etwas genauer unterscheiden. Die JSF Views werden in der Regel als XHTML-Seiten aufgebaut. Das hat somit erst einmal nichts mit HTML5 zu tun. In den XHTML Views beschreibe ich meine Seite, also welche Eingaben oder Buttons ich benötige. Innerhalb des JSF-Lebenszyklus wird diese View gerendert und das resultierende Markup zum Browser geschickt. Dieses Markup kann HTML, aber auch XSL oder RSS sein. JSF ist nicht auf HTML als Ausgabeformat angewiesen. Somit ist HTML5 nur ein mögliches von vielen Formaten, die JSF schon immer unterstützt hat. Mit JSF 2.2 wird es künftig jedoch so sein, dass standardmäßig der HTML5-Doctype aus den XHTML-Seiten generiert wird.

Ein weiteres wichtiges Feature, das JSF künftig bringt, sind so genannte *PassThrough*-Attribute. Dies sind Attribute, die für JSF keinerlei Bedeutung haben, aber im resultierenden HTML wichtig sind. Man denke hier z. B. an die *data-**-Attribute, die frei definiert werden können und vorwiegend für JavaScript-Funktionen verwendet werden. Ein anderes Beispiel sind die *type*-Attribute bei input-Tags, z. B. bei *<input type="email">*. Durch die Angabe dieser *PassThrough*-Attribute können beliebige Attribute ins Markup eingebaut werden und werden mehr wie bislang ausgefiltert.

„Kritik an JSF gab es immer wieder. Und das ist auch gut so.“

JM: Der HTML5-friendly Markup-Support ist auch eines der so genannten sechs „Big Ticket Features“ von JSF 2.2. Kannst du uns ein bisschen was über die anderen fünf Features erzählen?

Bosch: Ja gerne. Ein großes Highlight ist sicherlich das Konzept von *FacesFlows*. Die Ideen dahinter sind zunächst nicht spektakulär neu, sondern entstammen aus Open-Source-Initiativen und Frameworks wie Spring Web Flow oder ADF Task Flows. Man hat sich im Vorfeld sehr genau die existierenden Lösungen angeschaut und versucht, die besten Ansätze in einen Standard zu kippen. Letzten Endes geht es darum, wiederverwendbare und in sich geschlossene Seiten-Flows zu bauen. Ein typisches Anwendungsgebiet sind Wizards, die eine in sich geschlossene Abfolge von Seiten haben und eventuell innerhalb dieses Ablaufs eigene Daten (also Beans) benötigen. Im Zusammenhang mit diesen Flows wird auch ein neuer Scope – der *FlowScope* – eingeführt, der sich nahtlos in das gesamte Konzept integriert.

Ein weiteres wichtiges Feature ist die „Cross-site Request Forgery Protection“. Das hört sich komplizierter an, als es eigentlich ist. Kurz gesagt geht es darum, die Sicherheit von Webanwendungen zu verbessern. Es darf nicht möglich sein, dass z. B. durch eingeschleusten Schadcode ein Request abgesetzt wird, der nicht von der eigentlichen JSF-Anwendung initiiert wurde. Hier war JSF in der Vergangenheit schon sehr gut positioniert, existiert in jeder gerenderten View ein *javax.faces.ViewState*-Hidden-Feld. Mithilfe von *ProtectedViews* wird dieser State künftig verschlüsselt, um ein höheres Maß an Sicherheit zu erreichen.

In der Liste der „Big Ticket Features“ taucht ebenfalls das „Laden von Facelets über den *ResourceHandler*“ auf. Es wird hierfür das API im *ResourceHandler* ein wenig erweitert. Auch wichtig in diesem Zusammenhang ist zu wissen, dass mit 2.2 eine eigene *FaceletFactory* definiert werden kann. Das ist jetzt nicht unbedingt eine umwälzende Neuerung. Vielmehr wird hier ein Fehler aus der JSF-2.0-Version beseitigt, bei der es nicht möglich war, konfigurativ oder programmatisch eine eigene FaceletFactory für die Erzeugung von Komponenten zu verwenden. Wichtig ist hier mehr die Botschaft zwischen den Zeilen. Bei JSF wird extrem viel Wert darauf gelegt, dass es an allen Ecken und Enden erweiterbar und anpassbar ist. Sei es, wenn man einen eigenen Navigation-Handler erzeugen oder einen spezifischen *ExceptionHandler* im Framework registrieren möchte. Durch die Möglichkeit, eine eigene FaceletFactory implementieren zu können, ist man hier dann insgesamt wieder „sauber“.

Eine neue Standardkomponente in JSF wird endlich ein FileUpload sein. In der Vergangenheit gab es solche Komponenten schon in verschiedensten Ausprägungen, je nachdem, welche zusätzliche Komponentenbibliothek man mit im Projekt inkludiert hatte. Jetzt hat man endlich im JSF-Standard bereits die Möglichkeit, einen Fileupload zu realisieren. Der JSF-Fileupload nutzt hier die Basis, die mit Servlet Version 3 gelegt wurde. Es wird beim Upload in einem Managed Bean ein *javax.servlet.http.Part*-Objekt bereitgestellt, an dem ein Stream abgeholt werden kann. Eine sehr schön in die übrige EE-Welt integrierte Lösung.

Last but not least kommt noch ein Konzept von Multi-Templating dazu. Wobei ich hier genauer sagen sollte: kommt *vielleicht* dazu. Speziell über dieses Feature wird aktuell noch sehr stark diskutiert. Es wird vermutlich eher so sein, dass eine vereinfachte Version davon in JSF 2.2 reinrutschen wird, die zunächst etwas mehr Flexibilität in der Verwendung von View-Templates mit sich bringt. Hier werden aber die nächsten Wochen mehr Klarheit bringen.

JM: Du hast vor Kurzem den JSF Day auf der W-JAX moderiert. Mit welchen Fragen kommen die Teilnehmer auf euch zu?
Bosch: Ein Großteil der Fragen betraf natürlich die neuen Features in JSF 2.2. Doch man merkt deutlich, dass JSF in vielen Firmen bereits etabliert und das Niveau der Fragen sehr hoch ist. Es geht somit mehr um konkrete Probleme, die in der täglichen Arbeit mit JSF auftreten. Interessant ist auch, dass doch schon sehr viele auf dem neuesten JavaEE-Standard sind und über einen Software-Stack bestehend aus JSF, CDI und JPA verfügen. Daher war auch die Session über JSF und CDI innerhalb des JSF Day noch ein wichtiger Impulsgeber.

Für viele Teilnehmer war auch die Vision wichtig, was JSF für den mobilen Bereich anzubieten hat. In meiner Session am Vortag des JSF-Days habe ich verschiedene Ansätze aufgezeigt, mobile Webanwendungen mit JSF zu bauen. Im Umfeld des mobilen Webs kommen verständlicherweise verstärkt JavaScript und JavaScript-Frameworks zum Einsatz. Daher waren auch viele Fragen darunter, wie sich JSF mit diesen Technologien kombinieren lässt. In diesem Zusammenhang wird auch gerne nachgefragt, ob JSF sich mit JavaScript gut versteht oder das eher Konkurrenten sind. Hier konnte ich die Teilnehmer natürlich beruhigen, dass sich JavaScript auf der Clientseite sehr harmonisch mit JSF verheiraten lässt.

JM: Du hattest die Kombination JSF und CDI erwähnt. Sollte künftig jedes JSF-Projekt automatisch CDI mitverwenden?
Bosch: Diese Kombination ist sicherlich der Weg, der von beiden Expert Groups favorisiert ist. Natürlich kann JSF auch weiterhin „standalone", also ohne CDI, verwendet werden. Eine Verwaltung von Managed Beans über Angaben in der *faces-config* wird natürlich weiterhin unterstützt. Es wird jedoch stark davon abgeraten, langfristig auf die *@ManagedBean*-Annotation zu setzen, da diese künftig wohl deprecated sein wird. CDI ist nun mal *die* Komponententechnologie innerhalb des JavaEE-Stacks. Speziell auch mit JSF 2.2 wird eine stark verbesserte CDI-Integration kommen, sodass man in jedem Projekt einen gemeinsamen Einsatz von JSF und CDI favorisieren sollte. Ein Hinweis, wie eng JSF mit CDI künftig verheiratet sein wird, zeigt auch der neue *FlowScope*, der als CDI Custom Scope realisiert ist. Möchte man daher dieses Feature nutzen, kommt man an CDI nicht mehr vorbei.

JM: Selbst hast du auch einen Talk über UI-Architekturen in JSF gehalten mit der Aussage, dass der Einsatz von JSF alleine nicht genügt, um ein User Interface zu realisieren. Welche Weichen werden beim UI denn gestellt?
Bosch: Wenn man sich als Frontend-Entwickler outet, landet man oftmals zu Unrecht in einer falschen Schublade: „Frontend-Entwickler, das sind doch die, die nur die bunten Pixel programmieren!" hört man nicht selten. Diese Klischee-Vorstellung ist jedoch falsch, da ich selbst vorwiegend in der Frontend-Entwicklung aktiv bin, von Design, Farben und schicken Bildern aber sehr wenig Ahnung habe. Dass das kein Widerspruch ist, lässt sich damit erklären, dass es im Frontend zwei Bereiche gibt: die Gestaltung der Frontend-Seiten und die Programmierung der Frontend-Technik. Gerade im Bereich der Frontend-Programmierung werden oftmals grobe Architekturfehler eingebaut, die Auswirkungen auf das gesamte Projekt haben können. Wenn man hier die falschen Konzepte zugrunde legt, endet das nicht selten in sehr aufwändigen und damit teuren Refactorings. Bezogen auf JSF heißt das, dass das Beherrschen der reinen Syntax zwar schon eine gute Basis für ein Projekt ist. Aber viel wichtiger ist die Architektur im UI, die mit der JSF-Syntax letzten Endes realisiert ist. Viele der Aussagen, die ich in meinem Talk über UI-Architekturen getroffen habe, gelten übergreifend auch für andere Frontend-Technologien. JSF hat darüber hinaus noch einige Stolpersteine, die man beim Bau von Frontends beachten sollte.

JM: Vielen Dank für das Gespräch!

Der Himmel kann warten

Keine Wolke (JEE) 7

Die Spezifikation für Java EE 7 soll im Frühjahr 2013 offiziell verabschiedet werden. Nach einigem Hin und Her steht nun fest, dass sie das Thema Cloud gänzlich außen vor lassen wird. Aller Kritik zum Trotz: Diese Entscheidung war absolut richtig.

von Bernhard Löwenstein

„Sie liebt mich – sie liebt mich nicht – sie liebt mich – …" Millionen von Margeriten mussten für dieses Kinderspiel sicherlich schon ihre Blüten lassen. Was hat das nun mit Java zu tun? Lesern, die eifrig alle Nachrichten verfolgen, die über die Java-Editionen veröffentlicht werden, kommt wohl langsam der Verdacht auf, dass nach einem ähnlichen Verfahren entschieden wird, wenn es um die Aufnahme von neuen Features und Technologien in eine der Plattformeditionen geht. So hieß es anfänglich, dass die Modularisierung der Plattform und die Java-Module mit Java SE 8 kommen sollen. Daraus wird nun aber doch nichts, denn das Thema wurde auf Version 9 verschoben. Ähnlich gestaltete es sich mit Java EE 7 und den Cloud-Features. Den anfänglichen Plänen nach sollte diese Version ganz im Zeichen der Wolke stehen. Seit einigen Wochen steht allerdings fest, dass dies nicht der Fall sein wird, da die Java-Community die Zeit als noch nicht für reif dafür erachtet. Diese beiden Verschiebungen degradieren jedenfalls beide Editionen zu besseren Zwischenversionen, die ein paar nette Verbesserungen enthalten. In meinem Kopf hat sich durch die häufigen Wechsel jedenfalls ein Bild eingeprägt, das ich nicht mehr loswerde: Ich sehe Larry Ellison, umgeben von seinen technologischen Chefberatern, in seinem Büro sitzen – mit einer Margerite in der Hand – und jedes Mal, wenn er an der Blume zupft, geht eine aktualisierte Pressemitteilung raus.

Trotz meiner kritischen Worte halte ich die Entscheidung, die nächste Generation der Java-Enterprise-Plattform noch frei von sämtlichen wolkigen Funktionalitäten zu halten, für absolut richtig [1]. Dass dies auch viele andere Java-Freunde so sehen, bestätigt eine entsprechende Umfrage von JAXenter [2], bei der fast zwei Drittel die Entscheidung gutheißen. Wie sich bereits bei EJB3 gezeigt hat, kann der Code-First-Ansatz – im Gegensatz zur Spec-First-Devise bei EJB2 – wahre Wunder bewirken und uns letztendlich mit einer Technologie beglücken, die sich kaum eleganter spezifizieren lässt. Dem Spring Framework sei Dank, denn es übernahm in Verbindung mit EJB3 quasi die Code-First-Rolle. Da selbst Oracle kein Orakel im Keller stehen hat, das die Zukunft vorhersagt, und die Zahl der von irgendwelchen Elfenbeinturmpredigern erstellten Technologiespezifikationen mittlerweile groß genug ist, kann man den in der Java-Welt eingekehrten Pragmatismus durchaus sehr positiv sehen.

Dass heute noch keiner so recht weiß, wohin die Reise überhaupt genau gehen soll, zeigt sich am breiten Spektrum der aktuell verfügbaren Cloud-Lösungen. Das Java Magazin stellte passend dazu erst kürzlich in einer vierteiligen PaaS-Serie ausgewählte Vertreter vor: Heroku [3], Windows Azure [4], Amazon Beanstalk [5] und Red Hat OpenShift [6]. Auch auf der W-JAX 2012 in München konnten sich die Besucher bei der PaaS-Parade über die unterschiedlichen Cloud-Plattformen mit Java-Unterstützung informieren. Eine Kategorisierung der aktuell verfügbaren Lösungen lässt sich unter anderem nach folgenden Kriterien vornehmen:

- Unterstützt das Cloud-System Java oder Java Enterprise?
- Stellt es lediglich eine Laufzeitumgebung bereit oder unterstützt es den Entwickler während des gesamten Zyklus der Applikationsentwicklung?
- Ist es nur öffentlich nutzbar oder kann es auch in Form einer Private Cloud betrieben werden?
- Welche zusätzlichen Dienste und Systeme stehen innerhalb der Cloud-Plattform zur Nutzung bereit?

Klar ist jedenfalls, dass Java (Enterprise) in der Wolke auf PaaS-Level verfügbar gemacht und die Java-Enterprise-Plattform in Richtung Mehrmandantenfähigkeit adaptiert werden muss. Das Rollenmodell sieht hierbei folgende Stakeholder vor: Cloud Provider, Cloud Account Manager, Cloud Customer, Application Submitter, Application Administrator und End-User. Rege Diskussionen gibt es hingegen noch in Bezug auf die Spezifikation der SLAs, QoS, Elastizität, automatische Systemanpassung und bedarfsabhängige Kapazitätsbereitstellung.

Ein weiterer Grund, weshalb ich die Verschiebung für nicht besonders tragisch halte, ist die Tatsache, dass das Thema in der Wirtschaft und in der Industrie noch gar nicht so recht Fuß gefasst hat. Ich habe erst vor wenigen Wochen bei den IBM developerWorks Days 2012 in Zürich das Auditorium gefragt, wie relevant Cloud Computing für die Kunden ist. Gerade mal für einen von rund

achtzig Entwicklern war das Thema von Bedeutung, und auch hier sollte lediglich die vorsichtige Variante, nämlich eine in einem herkömmlichen Rechenzentrum betriebene Private Cloud, zum Einsatz kommen. Das halte ich persönlich zwar für eine Fehlentwicklung, da das elastische Verhalten der Wolkenmaschine – meiner Meinung nach das wichtigste Unterscheidungsmerkmal im Vergleich zu den früheren Ansätzen – nur beschränkt erreichbar ist. Es ist aber auf alle Fälle ein Einstieg in die Materie. Erst bei Nutzung einer Public PaaS oder dem Aufbau einer privaten Cloud-Plattform auf Basis einer Public-IaaS-Lösung spielt diese ihre ganze Stärke aus. Bei allen anderen Lösungen muss man nämlich selbst wiederum die Ressourcen für die Lastspitzen vorhalten. Grundsätzlich lässt sich aber festhalten: Angekündigte Revolutionen finden nun einmal selten statt – wie so oft wird auch hier der technologische Wandel langsam über viele Jahre vonstatten gehen. Klar ist für mich mittlerweile aber, dass es sich bei Cloud Computing nicht nur um einen temporären Hype handelt, sondern dieser Ansatz langfristig unsere Art, wie wir Software entwickeln und vor allem betreiben, maßgeblich beeinflussen und verändern wird.

Wie sieht nun meine Vision für eine Java-Plattform in der Cloud aus? Lassen Sie uns einen Blick in die Vergangenheit werfen. Vor etwas mehr als einem Jahrzehnt kamen die ersten Applikationsserver zur Ausführung von Java-Enterprise-Applikationen auf. Jene nehmen dem Programmierer seitdem viel Arbeit ab, da sie durch die Bereitstellung diverser Dienste die Entwicklung auf ein höheres Level heben. So ist es heute möglich, eine vollständig transaktional ablaufende Geschäftsanwendung zu implementieren, ohne dass man dafür eine einzige Zeile Code schreiben muss. Dank *Convention over Configuration* muss der Programmierer dieses Verhalten nicht einmal mehr deklarativ festlegen. Denn der Container sorgt dafür, dass der Aufruf jeder EJB-Methode im transaktionalen Kontext erfolgt. Der Entwickler kann sich voll und ganz auf das Schreiben der eigentlichen Applikationslogik konzentrieren. Er muss seinen Quellcode lediglich gegen die entsprechenden APIs entwickeln. Interessanterweise hat sich die Nutzung des Applikationsservers über die Jahre grundlegend verändert. War es früher gang und gäbe, einen einzigen schwergewichtigen Applikationsserver für all seine Applikationen zu verwenden, so ist mittlerweile eindeutig ein Trend zu vielen leichtgewichtigen Applikationsserverinstanzen mit jeweils einer Applikation an Bord erkennbar. Darauf zielen beispielsweise auch die Architekturänderungen am JBoss AS7 ab. Zwecks besserer Skalierbarkeit und Ausfallsicherheit begnügt man sich meist nicht mehr damit, seine Anwendung auf nur einem Knoten zu betreiben, sondern in den meisten Fällen kommt ein ganzer Cluster zum Einsatz. Genau hier müsste meiner Meinung nach nun die Java-PaaS ansetzen. Was mit dem Applikationsserver begann, sollte die Wolkenmaschine weiterführen – und damit die Java-Entwicklung auf ein noch höheres Level als bisher heben. Was gäbe es Schöneres, als wenn man seine Anwendung gemäß dem bisherigen Programmiermodell entwickeln könnte, sich aber auch um Dinge wie Skalierbarkeit und Hochverfügbarkeit nicht mehr selbst kümmern müsste – sondern lediglich das gewünschte Skalierungsverhalten innerhalb eines Deployment Descriptors über ein paar Konfigurationsattribute anzugeben hätte? Durch die Wahl von geschickten Voreinstellungen und *Convention over Configuration* ließen sich selbst diese Einstellungen auf ein Minimum beschränken, wenn nicht sogar gänzlich einsparen. Das wäre ein wahrer Mehrwert der wolkigen Java-Plattform gegenüber allen bisherigen Lösungen. Man müsste sich dabei auch im Vorfeld keine technischen Gedanken mehr darüber machen, ob das System mit der Last zurechtkommt. Darum kümmert sich dann nämlich einzig und allein die Cloud-Plattform.

Die grundlegend erforderlichen Änderungen ließen sich meiner Meinung nach mit adäquatem Aufwand hinbekommen. In der Java-EE-Spezifikation muss in Sachen Systemarchitektur der Applikationsserver durch eine verteilte Laufzeitumgebung ersetzt werden. Das Thema Skalierbarkeit lässt sich so gut in den Griff bekommen. Lediglich bei der Hochverfügbarkeit wird man passen müssen. Der Ausfall einzelner Knoten lässt sich zwar durch das redundante Ablegen aller Statusdaten bewältigen. Doch sobald ein ganzes Rechenzentrum ausfällt – wie wir erst kürzlich gesehen haben, kann das durchaus der Fall sein [7] –, ist einem Cloud-Nutzer damit nicht mehr geholfen. Da müsste man die Daten schon über die Rechenzentrumsgrenzen hinweg replizieren. Das führt aber schnell dazu, dass man mit unterschiedlichen Rechtssystemen konfrontiert wird. Am besten wäre es, die Ausfallssicherheit seiner Softwaredienste durch Replikation über die Herstellergrenzen hinweg sicherzustellen. Die wissenschaftliche Community spricht dann von *Sky Computing*. Angesichts der aktuell noch zu klärenden Fragen in Zusammenhang mit der Cloud scheint mir dieses Thema derzeit aber wenig praxisrelevant – anders formuliert: Der Himmel kann sicherlich noch ein bisschen warten!

Bernhard Löwenstein (bernhard.loewenstein@java.at) ist als selbstständiger IT-Trainer und Consultant für javatraining.at und weitere Organisationen tätig. Als Gründer und ehrenamtlicher Obmann des Instituts zur Förderung des IT-Nachwuchses führt er außerdem altersgerechte Roboter-Workshops für Kinder und Jugendliche durch, um diese für die IT und Technik zu begeistern.

Links & Literatur

[1] http://www.jaxenter.de/artikel/5306

[2] http://it-republik.de/jaxenter/quickvote/results/1/poll/167

[3] Eberhard Wolff: „Die Heroku PaaS Cloud – PaaS-Pionier – auch für Java“, Java Magazin 9.2012 (S. 24–26)

[4] Holger Sirtl: „PaaS auch für Java – Welche Möglichkeiten bietet Azure für Java-Entwickler?“, Java Magazin 10.2012 (S. 72–77)

[5] Eberhard Wolff: „Beanstalk – PaaS von Amazon – Die PaaS-Lösung des Cloud-Markführers“, Java Magazin 11.2012 (S. 78–81)

[6] Bernhard Löwenstein: „Red Hat OpenShift – Open-Source-PaaS mit Java-EE-6-Unterstützung“, Java Magazin 12.2012 (S. 100–104)

[7] http://bit.ly/NNRnzB

Migration zu Java EE 6 abseits der grünen Wiese

Generalüberholung – Java EE 6 Style

Neue Technologien und Frameworks zeigen sich auf Konferenzen und in Fachartikeln meist von ihrer guten Seite, insbesondere wenn diese für Neuentwicklungen herangezogen werden. Doch Unternehmen, die schon etwas länger auf Enterprise Java setzen, befinden sich selten in einer derartig luxuriösen Ausgangsposition. Wie dennoch der Wechsel zu dem aktuellen Java EE 6 Stack erfolgen und wann und warum eine Migration sich lohnen kann, zeigt dieser Artikel.

von Jens Schumann

jax Speaker

Hat sich eine technologische Neuerung erst einmal den Weg in die Konferenzlandschaft oder in eine Fachzeitschrift gebahnt, so lassen Aussagen zu deren Effizienz, Eleganz, Performance und Wartbarkeit nicht lange auf sich warten. Auch der Spaßfaktor beim Einsatz spielt heute eine entscheidende Rolle, sodass man als Unternehmen mit einem gewissen Anteil an individuell entwickelter Java-Bestandssoftware recht schnell den Stempel „Old School IT“ abbekommt.

Natürlich wünschen wir uns alle eine IT-Landschaft, in der technologische Neuerungen leicht integriert werden können, und in der Entwicklung bzw. Wartung Spaß machen, ohne dabei täglich mit unnötiger Fleißarbeit konfrontiert zu werden. Leider besitzt aber die durchschnittliche Enterprise-Java-Bestandssoftware, speziell mit einem Ursprung vor 2005, aus heutiger Sicht zahlreiche Altlasten und architektonische Fehlentscheidungen, sodass fachliche und technologische Weiterentwicklung bisweilen stark behindert wird.

Seit dem Erscheinen von Java EE 6 hat sich trotz anfänglicher Skepsis nun gezeigt, dass die neue Enterprise-Java-Plattform für unglaublich viele Bereiche Lösungen und Antworten anzubieten hat, die durchaus den eingangs genannten Schlagworten Effizienz, Eleganz, Performance und Spaßfaktor gerecht werden. Doch auch wenn damit die Zieltechnologien klar umrissen sind, stellt sich oft die Frage, wie Hundertausende bis Millionen Zeilen Java-Code und Hunderte bis Tausende JSP/View-Fragmente migriert werden können, ohne dass die damit verbundenen Aufwände und Risiken den Mehrwehrt des Wechsels in Frage stellen.

Warum migrieren?

Bei allem Sex-Appeal einer neuen Technologie muss man sich in jedem Fall die Frage gefallen lassen, warum eine Migration einer bestehenden Enterprise-Java-An-

wendung zu einem neuen Technologie-Stack notwendig und sinnvoll sein soll.

Für Enterprise-Java-Anwendungen mit einem Alter von fünf, sieben oder gar zehn Jahren gelten natürlich die klassischen Gründe, wie nicht (mehr) vorhandener bzw. auslaufender Support sowie oft eine geringe Entwicklungsgeschwindigkeit (meist in Verbindung mit einer oft technologiebedingten geringen Testabdeckung). Aber auch eine hohe Komplexität in der Entwicklung und fehlender Funktionsumfang können eine Migration notwendig machen, insbesondere wenn für die verwendeten Technologien heute nur noch wenig bis gar kein Know-how mehr am Markt verfügbar ist.

Interessanterweise zeigt sich auch, dass immer mehr Unternehmen wegen des eingesetzten Technologie-Stacks Schwierigkeiten haben, neue Mitarbeiter zu rekrutieren, da gerade Berufseinsteiger ungern ihre Kariere in einem Umfeld starten wollen, das im Jahr 2005+ schon als veraltet galt.

Glücklicherweise existiert nun mit Java EE 6, und allen voran mit *JPA*, *CDI*, *JAX-RS* und *JSF*, nachweislich eine ausgereifte, interessante und zukunftssichere Plattform, die eine hohe Akzeptanz in der Enterprise-Java-Community besitzt und somit den Weg für eine Migration aufzeigt.

Wie migrieren?

In einer idealen Welt ist eine Migration zu einer neuen technologischen Plattform recht einfach: Die Fachabteilungen müssen nur ein paar Monate bis Jahre auf die Umsetzung neuer Anforderungen verzichten, und schon kann eine Anwendung vollständig und nachhaltig migriert werden. Dabei können die Erkenntnisse und Erfahrungen im Umgang mit der bestehenden Anwendung direkt in die Migration mit einfließen, sodass zusätzlich zum Austausch der verwendeten Technologien bekannte architektonische Fehler korrigiert werden können. Doch so verlockend dieser Ansatz auch klingen mag, in der Praxis wird er kaum vorkommen.

Eine Alternative zur vollständigen Migration ist eine an den architektonisch vorhandenen Schichten orientierte Migration, die schrittweise pro Schicht die verwendete Technologie austauscht. Dies funktioniert in der Regel besonders gut im Business- und je nach Ausgangstechnologie auch im Data-Access-Tier und lässt sich oft automatisieren, sodass der Austausch der verwendeten Technologien parallel zur laufenden fachlichen Weiterentwicklung vorbereitet und innerhalb einer sehr kurzen Code-Freeze-Phase durchgeführt werden kann. Bei der Automatisierung hat sich die Verwendung von eigenentwickelten Refactorings als besonders positiv herausgestellt, wie sie zum Beispiel dank des Eclipse JDTs [1], [2], [3] selbst für Java-Entwickler ohne große Compiler-Bauerfahrung leicht realisierbar sind.

Bei umfangreichen Webanwendungen zeigen sich meist im Presentation-Tier die Grenzen einer schichtenorientierten Migration, da ein Austausch der Visualisierungstechnologie oft ein sehr komplexer und aufwändiger Prozess ist, den man selten automatisieren kann. Die schichtenorientierte Migration entspricht in dieser Schicht somit meist einer vollständigen manuellen Migration mit entsprechend langen Code-Freeze-Phasen, was bei businessrelevanten Anwendungen selten durchsetzbar ist.

Stattdessen empfiehlt sich gerade im Bereich der Visualisierung eine an fachlichen Modulen orientierte Migration, die sich vor allem auf Module mit hohem (Weiter-)Entwicklungsbedarf konzentriert und stabile, funktionsfähige Module deutlich hinten anstellt. Mit diesem Vorgehen lassen sich im Entwicklerteam auch die notwendigen Erfahrungen im Umgang mit der neuen Technologie langsam aufbauen, sodass Fehler mit nachhaltiger Auswirkung deutlich leichter erkannt und mit deutlich geringerem Aufwand korrigiert werden können. Grundsätzlich lässt sich deshalb sagen, dass eine an fachlichen Modulen orientierte Migration immer genau dann sehr erfolgreich ist, wenn die Migrationen im Wesentlichen manuell erfolgen.

Wohin migrieren?

Setzt man technologisch nun auf den Java-EE-6-Standard, so kommen beim Einsatz von relationalen Datenbanken für die Datenzugriffsschicht nur *JPA 2* und *CDI*, für die Businesslogikschicht *CDI* und *EJB 3.x* und in der Präsentationsschicht *CDI* und *JSF 2* als Zieltechnologien in Frage.

Aufgrund der schieren Menge an möglichen Enterprise-Java-Ausgangstechnologien sind die mit einer Migration verbundenen Aufgaben und Herausforderungen sehr vielschichtig und auch sehr unterschiedlich, womit eine ganzheitliche Betrachtung aller Migrationsvarianten den Rahmen dieses Artikels deutlich sprengen würde. Die folgende Betrachtung von Migrationsansätzen konzentriert sich deshalb auf die am stärksten verbreiteten technologischen Ausgangsszenarien, wie sie bei fünf, sieben oder zehn Jahre alten Java-Anwendungen zu finden sind.

OR Mapping oder nicht?

In der Datenzugriffsschicht sind neben einfachen *SQL Template Engines* entweder klassische *JDBC-Datenbankabfragen*, manchmal aber auch eigenentwickelte *OR-Mapping-Frameworks* bzw. *EJB 2.x Persistence (CMP, BMP)* oder *Hibernate* unter Verwendung von *org.hibernate*-Klassen stark verbreitet.

Die Zieltechnologie *JPA 2* stellt in Verbindung mit *CDI* nunmehr ein ausgereiftes und leicht integrierbares OR-Mapping-Framework zur Verfügung, mit dessen Hilfe eine stark fachlich orientierte Anwendung und damit die fachlichen Entitäten und ihre ausmodellierten Beziehungen persistiert und von einer Datenbank geladen werden können.

Kommt man nun von einer klassischen JDBC-Datenzugriffsschicht, so ist der Mehrwert eines OR-Mapping-Frameworks (trotz aller typischen Performancebedenken) im täglichen Entwicklungsprozess sofort sichtbar. Eine Migration hin zu JPA ist daher auf jeden Fall zu empfehlen, auch wenn der Prozess der Migration sich unter Umständen schwieriger gestaltet als man auf den ersten Blick erwarten mag. Die Ursache hierfür ist das Fehlen eines ausgeprägten fachlichen Domainmodells, da eine klassische JDBC-Datenzugriffsschicht häufig nur simplen Value Objects nutzt, die weder Domainlogik, Vererbungshierarchien noch ausformulierte Beziehungen zwischen Entitäten besitzen.

Diese blutleeren Domainmodelle [4] können so zwar auch von JPA für die Interaktion mit relationalen Datenbanken genutzt werden, jedoch kommt der eigentliche Mehrwert von JPA – und zwar das *objektrelationale Mapping* – so kaum zum Tragen. Es ist daher zwingend zu empfehlen, dass mit der Migration zu *JPA* auch das fachliche Domainmodell einer Anwendung eine Überarbeitung erfährt, sodass das Entwicklungsmodell in der Datenzugriffsschicht auch einem echten OR Mapping entspricht. Leider handelt es sich dabei vor allem um eine fachliche Herausforderung, die aufgrund der dafür notwendigen Abstimmung mit den Fachabteilungen schnell der technischen Migration im Wege stehen kann.

Etwas einfacher kann sich die Migration eines eigenen OR-Mapping-Frameworks gestalten, sofern der grundsätzliche OR-Mapping-Ansatz und der Lebenszyklus von Entitäten sich nicht zu stark von dem JPA-Ansatz und dessen Entitäten-Lebenszyklus unterscheiden. Die Migration zu JPA kann deshalb auch erst nach eingehender technischer Analyse des vorhandenen OR-Mapping-Frameworks erfolgen und sollte unbedingt ergebnisoffen durchgeführt werden. In der Vergangenheit hat sich gezeigt, dass viele OR-Mapping-Frameworks, die in den Jahren 2001 bis 2003 entstanden sind, eine erstaunliche Ausgereiftheit besitzen, sodass eine Migration zu JPA oft nur geringen Mehrwert oder sogar einen Rückschritt bedeuten kann.

Sollte der Mehrwert von JPA oder aber die mit dem eigenentwickelten OR-Mapping-Framework verbundenen Risiken eine Migration erforderlich machen, so ist diese meist relativ gefahrlos durchführbar. Ursache hierfür ist, dass die meisten OR-Mapping-Frameworks aus dieser Zeit auf einen generativen Ansatz setzen und auf der Basis einer zentralen OR-Mapping-Beschreibung den notwendigen Persistenzcode generieren. Mithilfe dieser Beschreibung lassen sich in der Regel mit überschaubarem Aufwand auch JPA-Entitäten generieren und in Verbindung mit automatischen Refactorings in die bestehende Software integrieren. Spannend bleibt hierbei die Migration einer gegebenenfalls vorhandenen eigenen Query Language, die jedoch selten bis gar nicht anzutreffen ist. Kommen stattdessen für dynamische Anfragen reine SQL Queries zum Einsatz, so ist die Migration zu JPA mithilfe von JPA-Bordmitteln, den so genannten native Queries, leicht realisierbar.

Etwas problematischer kann sich die Migration von *EJB 2.x Entity Beans* hin zu JPA gestalten. Dies liegt zum einen an der potenziellen Remoting-Fähigkeit von Entity Beans, aber auch an dem oft anzutreffenden blutleeren Domainmodell einer EJB-2.x-Entity-Beans-Persistenzschicht. Somit bestehen als Problembereiche für die Migration nicht nur die oben erwähnte Einführung eines reichen Domainmodells sondern gegebenenfalls auch der Austausch der Kommunikation mit der Datenzugriffsschicht, da JPA eine rein lokale Persistenztechnologie ist.

Sofern heute in einer Anwendung noch EJB-2.x-Persistenz zum Einsatz kommt, kann eine Migration aus Performance- oder Featuregründen ausgeschlossen werden, da die damit verbundenen Probleme schon deutlich eher zu einer Migration geführt hätten. Nun werden mit Java EE 7 und damit mit EJB 3.2 die EJB 2.x Entity Beans zum optionalen Bestandteil der Java-EE-Plattform „degradiert“, sodass ein Application-Server-Hersteller in Zukunft selbst entscheiden kann, ob er diese EJB-Technologie noch unterstützt. Auch wenn es sehr unwahrscheinlich ist, dass die Hersteller eine vorhandene Technologie in ihren Produkten deaktivieren, so ist jedoch das Signal eindeutig, dass von einer Verwendung von EJB 2.x Entity Bean in Zukunft abzuraten ist.

Sofern nur wenige Entity Beans zum Einsatz kommen, kann man von einer manuellen Migration ausgehen. Bei

umfassenden Datenmodellen sollte versucht werden, die Migration mithilfe des JDT oder ähnlichen Ansätzen zu automatisieren.

Aus historischen Gründen findet man nach wie vor noch einige Anwendungen, die statt JPA das klassische Hibernate API und damit Klassen aus dem Package *org.hibernate* nutzen. In den meisten Fällen kommen in diesen Anwendungen neben den durch JPA standardisierten Features noch einige wenige proprietäre Hibernate-Features wie *Query By Example* [5] zum Einsatz. Da mittlerweile die Anzahl der Hibernate-Features, die nicht vom JPA-Standard abgedeckt sind, eine leicht überschaubare Dimension erreicht hat, empfiehlt sich in jedem Fall der Wechsel zum Standard. Für die wenigen verbleibenden Fälle, in denen man unbedingt auf proprietäre Hibernate-Features zurückgreifen muss, bietet Hibernate in der Regel proprietäre Annotation an bzw. erlaubt dem JPA API den Zugriff auf die hinter einem EntityManager liegende Hibernate-Session.

Da JPA faktisch ein Subset des Hibernate Featureset darstellt, ist die Migration hin zu JPA ein rein technischer Migrationsprozess, der in der Regel vollkommen automatisiert werden kann.

Komponentenmodell – aufgeräumt und leichtgewichtig

Im Bereich der Businesslogikschicht und damit im Bereich der Enterprise-Java-Komponentenmodelle findet man in Bestandsanwendungen drei klassische Vertreter: EJB 2.x *Stateless*, *Stateful* und *Message Driven Beans*, das *Spring Framework* sowie jegliche Formen und Varianten an Eigenentwicklungen.

Versucht man nun eine EJB-2.x-Anwendung hin zu den Java-EE-6-Komponentenmodellen *EJB 3.x* und/oder *CDI* zu migrieren, stellen sich im Wesentlichen zwei Herausforderungen: Zum einen müssen zahlreiche Interfaces entfernt und XML-Informationen zu Annotationen überführt werden. Darüber hinaus beinhalten klassische EJB-2.x-Anwendungen eine unglaublich große Menge an Infrastruktur und Pattern-Code, der durch Dependency Injection und den Wechsel hin zu einfachen Java-SE-Klassen, Interfaces und Annotationen obsolet geworden ist.

Da sich mit EJB 3.x zwar das Entwicklungsmodell, aber nicht das Laufzeitverhalten geändert hat, besteht die Migration zu EJB 3.x und/oder CDI aus dem großflächigen Löschen von Code und XML und der Einführung einiger weniger Annotationen, was bei entsprechend hoher Anzahl an EJBs weitestgehend automatisch erfolgen kann und sollte. Die Migration ist damit erneut ein rein technischer Prozess, der aufgrund aktueller Bestrebungen des Java-EE-7-Standards, auch EJB 2.x *Stateless*, *Stateful* und *Message Driven Beans* als optional zu deklarieren, mittelfristig erfolgen sollte.

Inwieweit die Migration einer Spring-Anwendung zum Java-EE-Standard erfolgen sollte, lässt sich an dieser Stelle nicht abschließend beantworten. Grundsätzlich kann man sagen, dass eine Migration nur in wenigen Fällen sinnvoll ist, und zwar auch nur dann wenn hauptsächlich das Spring Dependency Injection Framework zum Einsatz gekommen ist und weiterführende Spring-Framework-Features kaum oder gar nicht genutzt werden. Im direkten Vergleich bietet der Java-EE-Standard auch nur wenig, was im Spring Framework so nicht zu finden ist.

Soll jedoch das CDI-Programmiermodell mit qualifizierter, fachlicher Dependency Injection und CDI Events zum Einsatz kommen, so empfiehlt sich eine sanfte Migration zu CDI, was mit der einen oder anderen Spring <-> CDI Bridge schrittweise erfolgen kann.

Und was ist mit dem eigenentwickelten Komponentenframework, das in der Regel mit Java-SE-Klassen und mit EJBs umgehen kann? Genau diese Lücke wurde in Java EE 6 mit CDI und der zugrunde liegenden *@Inject*-Spezifikation geschlossen, sodass einer Migration zum Java-EE-Standard nichts mehr im Wege steht. Die proprietären Konfigurationsmöglichkeiten dieser Komponentenframeworks lassen sich normalerweise leicht als CDI Extension realisieren, sodass mit überschaubarem Aufwand die Konfiguration, das Abhängigkeitsmanagement und die Laufzeitumgebung von CDI bereitgestellt werden können. Übrig bleiben dann oft – analog zu EJB-2.x-Anwendungen – der durch die Eigenentwicklung eingeführte Infrastrukturcode und damit verbundene Patterns und Fehlerbehandlungsmechanismen, die sich bei entsprechend großer Codebasis automatisch via JDT entfernen lassen.

UI-Politur – mit Schmerzen

In der nunmehr verbliebenen Präsentationsschicht lassen sich die klassischen Vertreter nicht so ohne Weiteres bestimmen, da neben Eigenentwicklungen eine scheinbar unendliche Anzahl an Webframeworks in den letzten 10+ Jahren die Enterprise-Java-Bühne betreten und in der Regel auch schon wieder verlassen hat. Dennoch lässt sich eine große Gruppe an Webframeworks in die Kategorie von Action-basierten Webframeworks einordnen, die zwar im Fall von *Jakarta Struts 1.x* extrem weit verbreitet sind, aber kaum noch heutigen modernen Ansätzen der Webentwicklung und Web Usability genügen.

Wie kommt man nun von einem Action-basierten zu einem komponentenorientierten und Event-basierten Webframework wie JSF 2? Leider fällt die Antwort an dieser Stelle sehr ernüchternd aus: nur mit viel Aufwand und Energie, und vor allem unter großen Schmerzen. Die Ursachen dafür sind vielschichtig. Zum einen erfolgt die Aufbereitung von Daten in einem Action-basierten Webframework sehr formularbezogen, wohingegen in einem komponentenorientierten Webframework diese Aufbereitung sich an den verwendeten Komponenten orientiert. Statt also eine Methode auszuführen, die pro Web-View sicherstellen muss, dass nach der Ausführung alle Daten im Modell vorhanden sind, gibt es in Frameworks wie JSF zahlreiche Beans, die Daten für eine View bereitstellen und gegebenenfalls selbständig aufbereiten können.

Darüber hinaus unterscheidet sich die Repräsentation einer View in JSF in den meisten Fällen deutlich von der Repräsentation älterer Webframeworks, sodass im Rahmen einer Migration zu einem neuen Webframework ein wesentlicher Bestandteil, und zwar die Definition von Formularen und damit eventuell verbundene clientseitige Details (CSS, JavaScript), vollständig neu geschrieben werden muss. Dies umfasst dann auch das Data Binding zwischen dem Modell und der View und damit verbundener Validierung und Navigation, die heute gänzlich anders formuliert wird als vor 5 bis 10 Jahren.

Der Migrationsaufwand im Bereich der Präsentationsschicht erreicht damit schnell Dimensionen einer kompletten Neuentwicklung oberhalb einer hoffentlich vorhandenen guten Business-(Service-)Schicht und lässt die Migration in vielen Fällen fraglich erscheinen. Im Sinne des Investitionsschutzes und Risikomanagements sollte die Entscheidung, eine Migration des Webframeworks nicht durchzuführen, sehr bewusst getroffen werden, da ältere Webframeworks sich schon heute auf der Liste der aussterbenden Arten befinden und in nicht allzu ferner Zukunft am Markt nur noch wenig verfügbares Know-how für die Pflege und Wartung zu finden sein wird.

Deshalb empfiehlt sich in der Präsentationsschicht eine modulweise Migration hin zu einem neuen Framework, sodass die Webanwendung eine kontinuierliche Überarbeitung erfährt und im Rahmen von fachlicher Weiterentwicklung Schritt für Schritt migriert wird. Als Übergabepunkt eigenen sich hier Attribute in den jeweiligen Scopes (Session, Requests), sodass unterschiedliche Webframeworks auch für eine längere Zeit gefahrlos parallel zum Einsatz kommen können.

Und nun?

Der Weg zu Java EE 6 – und damit zum aktuellen Standard – kann sich in vielen Fällen lohnen oder sogar unbedingt notwendig sein. Je nach Ausgangssituation gestaltet sich der Weg dahin einfach oder aber kompliziert. Die Ablösung von EJB 2.x *Stateless*, *Stateful* und *Message Driven Beans* hin zu EJB 3.x und/oder CDI inklusive der Entfernung obsoleter J2EE Patterns ist sicherlich das Minimum, das es anzugehen gilt. Hierbei kann die Migration weitestgehend automatisiert werden.

Inwieweit der Wechsel zu JPA machbar und sinnvoll ist, sollte in jedem Fall genauer analysiert werden. Gerade klassische J2EE-Anwendungen besitzen selten ein ausgeprägtes bzw. stark ausformuliertes Domainmodell, sodass ein Wechsel hin zu JPA nicht nur eine technische, sondern vor allem auch fachliche Herausforderung bedeuten kann. Mittelfristig wird sich dieser Schritt auf jeden Fall positiv auf eine Java-EE-Anwendung und deren Wartbarkeit auswirken, die damit verbundenen Risiken und Kosten sollten im Vorfeld neutral bewertet werden. Die Nutzung von EJB 2.x Entity Beans birgt sicherlich die größten Risiken, sodass hier über eine Migration unbedingt nachgedacht werden sollte.

Allein im Bereich der Webframeworks ist die Frage nach einer Migration zum Standard (aber auch zu jedem anderen aktuellen Webframework) nicht leicht zu beantworten, da dieser Schritt in der Regel gleichbedeutend mit der Neuentwicklung wesentlicher Aspekte der Webanwendung, insbesondere des gesamten Presentation-Tiers, ist. Gerade umfangreiche Struts-1.x-Anwendungen zeigen sich dabei von ihrer unangenehmen Seite, sodass sich eine vollständige Migration selten kurzfristig oder gar mittelfristig rechtfertigen lässt. Als Alternative bietet sich hier eine modulweise Migration an, die sich im Wesentlichen an fachlichen Neuentwicklungen bzw. Bereichen mit einer hohen Anzahl an neuen Features und Anpassungen orientiert.

Jens Schumann, CTO bei dem IT-Beratungs- und Entwicklungsunternehmen open knowledge GmbH sowie freier Autor, beschäftigt sich seit vielen Jahren mit dem Design und der Umsetzung komplexer Lösungen im Enterprise-Computing-Umfeld.

Links & Literatur

[1] http://www.vogella.com/articles/EclipseJDT/article.html

[2] http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html

[3] http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html

[4] http://martinfowler.com/bliki/AnemicDomainModel.html

[5] http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/querycriteria.html#querycriteria-examples

Dynamische Datenabfrage und leichtgewichtige Visualisierung

Die Macht der Bilder

In Teil 1 dieser Serie haben wir gesehen, welche Kernfunktionen des Statistikframeworks benötigt werden, um Businessdaten zu analysieren und zu speichern. In dieser Ausgabe widmen wir uns weiteren Aufgaben des Frameworks, nämlich der Datenauswertung und -visualisierung. Wieso Datenvisualisierung keine Kernaufgabe des Frameworks ist und mit welchen externen Tools, Bibliotheken man stattdessen arbeiten kann – das alles erfahren Sie im folgenden Beitrag.

von János Vona

Datenauswertung ist ein wichtiges Thema. Wenn man nach Business Intelligence (BI) „yahoot“, „bingt“ oder googelt, findet man viele professionelle Lösungen. Natürlich braucht es kein SAP Business Objects, um unsere Daten auszuwerten. Aber eine gewisse Komplexität haben unsere Daten trotzdem. Am Anfang genügte es, eine HTML-Seite zu haben, die die Zahlen in einer einfachen Tabelle darstellte. Es gab aber keine Möglichkeit, komplexere Daten zu visualisieren. Einige Zeit später kam die Anforderung, die Businessdaten „Managementlike“ zu präsentieren. Das heißt: Einerseits gab es eine gewisse Datenkomplexität (mehrdimensionale Daten), anderseits erhöhte Anforderungen zur Visualisierung. Es führte also kein Weg vorbei an einer Restrukturierung des Codes und der Entkopplung der zwei Funktionen. Damals gab es noch kein HTML5 oder Google Charts [1] und Co., deshalb wurde zuerst nach einer anderen Lösung gesucht. Wenn man zu dieser Zeit im JEE-Bereich grafische Auswertungen von Kennzahlen darstellen wollte, ist man zwangsläufig auf JFreeChart [2] gestoßen. JFreeChart ist eine Open-Source-Java-Bibliothek, mit der man mit wenig Programmieraufwand Charts, 3-D-Bars, Pies usw. erstellen kann. JFreeChart ist unter LGPL lizensiert, welche die kommerzielle Benutzung ohne weitere, große Beschränkungen zulässt.

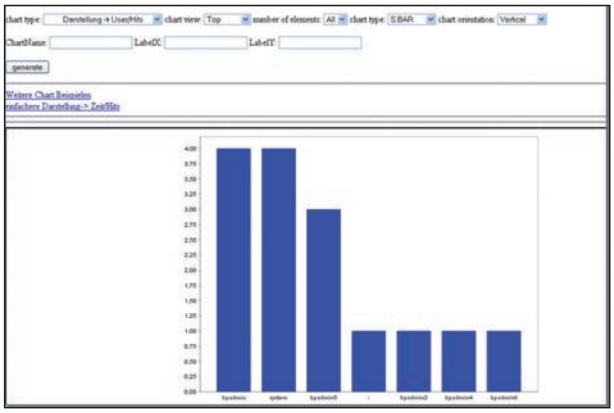


Abb. 1: JFreeChart-Demo

JFreeChart erstellt von den Daten zuerst ein Bild (*.png*), das man in eine HTML-Seite einbetten kann. Diese Funktionsweise ist leider sehr statisch, und ohne das Bild ständig neu zu erzeugen, lässt es keine Interaktion zu. Um etwas mehr Interaktion zu ermöglichen, habe ich eine bestehende Erweiterung von JFreeCharts an meine Anforderungen angepasst. Die Erweiterung heißt *Eastwood* [3] und ist von der Funktion her ähnlich wie Google Charts. Wenn man bei Wikipedia nach Google Charts API sucht, findet man auch Eastwood als Open-Source-Variante. Ich habe also ein *HttpServlet* geschrieben, das nach Benutzereingabe das entsprechende Bild generiert und dieses mit dem *HttpResponse* zurückliefert. Meine Seite sieht man in **Abbildung 1**. Sie sieht zwar immer noch nicht sehr schick aus, aber man konnte seine Kennzahlen auswählen und zusätzliche Darstellungsmerkmale definieren (Anzahl der dargestellten Elemente, Diagrammtyp, Beschriftung usw.). Die Arbeit mit JFreeChart ist kinderleicht. Man kann auf die vorimplementierten Klassen zugreifen, die nur mit Daten gefüttert werden müssen. In Listing 1 sieht man, wie einfach *JFreeChart*-Objekte mithilfe von einem *ChartFactory* erstellt werden können.

„Nach dem Refactoring ist vor dem Refactoring“ – so bleibt die Softwareentwicklung agil. Als ich JFreeCharts komplett integriert hatte, kam mir die Idee, die Auswertung noch interaktiver zu machen. Das geht nur dann, wenn die Abfragen dynamisch und vor allem recht abstrakt sind. Die Servlet-Technologie ist geeignet für solche Abfragen, da das Ergebnis in Form von *HTTPResponse*

Artikelserie

Teil 1: Hauptaufgaben des Frameworks: Business- und Zugriffsdaten analysieren und sichern
Teil 2: Datenauswertung (dynamische Abfrage) und -visualisierung
Teil 3: Modernes Caching

dynamisch zusammengestellt werden kann. So fing die Trennung der Visualisierung von der Auswertung an. Im Grunde genommen haben wir eine Schnittstelle, die dynamische Abfragen (*HTTPRequest*-Parameter) empfängt und sie an das Auswertungsmodul weiterleitet. Das Auswertungsmodul stellt die Abfrage dynamisch von den Parametern zusammen und bereitet das Ergebnis vor. Das Ergebnis wird in der jetzigen Version des Frameworks in Form eines einfachen *CSV* (Comma-separated Value) zurückliefert. Das CSV-Format ist sehr einfach und hat keinen zusätzlichen Overhead wie z. B. XML. Man könnte auch JSON verwenden, wenn man z. B. auf Google Charts umsteigen möchte.

Bevor wir in die Java-Klassen eintauchen, möchte ich zuerst die Bedienung des Servlets erläutern. Meine Idee war, das Servlet mit den gleichen Parametern zu „bedienen", die in der Konfiguration des Frameworks definiert waren. Mit der Weiterentwicklung des Frameworks kamen immer wieder neue Parameter hinzu, die man aber, dank des Aufbaus des Frameworks, sehr schnell integrieren konnte (Feature-driven Development). Eine Liste der aktuellen Parameter findet man in Tabelle 1. Das Servlet macht nicht viel. Es sucht eine geeignete Provider-Klasse aus, die für die Generierung des CSV-Formats anhand der Request-Parameter zuständig ist. Dann setzt es das zurückgelieferte CSV-Datenformat in den Response-Stream – und fertig. Listing 2 zeigt den Aufbau der Servlet-Klasse. Es wird zuerst geprüft, ob die Abfrageparameter anhand eines URL oder eines *aliasName* definiert sind. Dann wird anhand der definierten und der in *Request* vorhandenen Parameter die richtige Provider-Klasse automatisch ausgesucht. Hier könnte man ein weiteres Refactoring vornehmen und das Aussuchen der Provider-Klasse nach dem Paradigma „Konvention vor Konfiguration" (*Convention over Configuration*) implementieren. Somit würde man einige Spring-Konfigurationen sparen.

Die Konfiguration der Provider-Klasse findet man in Listing 3. Wir haben nun also unser Servlet, das von der Provider-Klasse die CSV-Daten in Form von *Strings* bekommt. Wie funktioniert aber eine Provider-Klasse? Man kann in Listing 3 entdecken, dass jede Provider-Klasse eine *StatisticReader*-Klasse injiziert bekommt (via Spring IoC). Man kann auch erkennen, dass im Beispiel eine auskommentierte *xmlStatisticReader* und eine nicht auskommentierte *jpaStatisticReader* definiert sind. Das zeigt, dass man mit einer einfacheren Konfiguration die Möglichkeit hat, zu definieren, welche Datenquelle verwendet werden soll. Das kann eine XML-Datei, eine über JPA definierte Datenbank, ein Cloud-Service oder ein RESTful Web Service sein. Mit dem Austausch der *statisticReader* Bean der Provider-Klasse kann man es je nach Bedarf definieren. Die Provider-Klasse ruft die Reader-Klasse auf und bereitet die gelieferten Daten je nach Rückgabeformat (in unserem Beispiel CSV) vor.

Das Interface der Reader-Klasse sieht man in Listing 4. Man kann erkennen, dass die Methoden, die *deprecated* sind, mit gewöhnlichen *queries* operieren. Diese hatten den Nachteil, dass datenquellenabhängige Abfragen (*queries*) in einer oberen Schicht (Provider-Klasse) benutzt wurden – zum Beispiel XPath-Ausdrücke bei XML. So etwas sollte aber in einem Framework nicht verwendet werden, deswegen sind sie nun *deprecated*. Die Reader-Klassen bekommen also die Parameter, mit denen das Servlet aufgerufen wurde, und sind selbst dafür zuständig, diese in den gewählten „Dialekt" umzuwandeln. Bei XML waren es XPath-Ausdrücke, bei JPA ist es die Zusammensetzung der Criteria-API-Funktionen.

Wir haben also gesehen, dass die statistischen Daten verschiedenartig dargestellt werden können. Wenn

Listing 1

ChartBar.java

```
public JFreeChart getChartViewer(String mainLabel, String labelX, String labelY, List labels,
            List values, int chartOrientation, int chartType, Color bgColor, Color barColor) {
 JFreeChart chart =  ChartFactory.createStackedBarChart(mainLabel, lX,lY, dataset, orient,
                                                                    false, false,false);
 BarRenderer renderer = new StackedBarRenderer();
// set the background color for the chart...
 chart.setBackgroundPaint(bgColor);
 CategoryPlot plot = (CategoryPlot) chart.getPlot();
 //
 ChartRenderingInfo info = null;
 try {
   //Create RenderingInfo object
   renderer.setBasePaint(barColor);
   renderer.setAutoPopulateSeriesPaint(false);
   plot.setRenderer(renderer);
   plot.setRangeAxisLocation(AxisLocation.BOTTOM_OR_LEFT);
   plot.setDomainGridlinesVisible(false);
   plot.setRangeGridlinesVisible(false);
  return chart;
 } catch (Exception ex) {
  //Exception handling
 }
   }
```

ChartCreatorServlet.java

```
  //Process the HTTP Get request
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws
                                                        ServletException, IOException {
    //parse the request information of the user setting dialog
    //get the JFreeChart
    ChartBar chart= new ChartBar();
 JFreeChart chartImage=chart.getChartViewer(mainLabel,labelX,labelY,labels,values,
                                          chartOrientation,chartType,bgColor,barColor);
 HttpSession session = request.getSession();
 // set the content type so the browser can see this as a picture
 response.setContentType("image/png");
 //create the png
  ChartUtilities.writeChartAsPNG(response.getOutputStream(), chartImage, 640, 480);
  response.getOutputStream().close();
  // send the picture ...
}
```

Listing 2

CsvDataProviderServlet.java

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
                                                    ServletException, IOException {
  ICsvDataProvider provider = null;
  try {
    //used url or alias
    String url=getUrlFromRequest(request,response);
    //
    if (url==null) {
      response.getWriter().println("URL is not valid !!!");
      return;
    }
    //get the right provider depend on the configuration
    provider = getProvider(request,url);
    List<String> csv = provider.getCSV();
    //
    response.setContentType("text/csv");
    response.setHeader("Content-Disposition","attachment;filename=StatisticMashApp.csv");
    //
    for (String rowCsv : csv) {
      response.getOutputStream().println(rowCsv);
     }
    response.getOutputStream().flush();
    response.getOutputStream().close();
    //
  } catch (Exception ex) {
    //Exception handling
  } finally {
   //finally block
  }
}
```

man aber über ein Framework spricht, ist die Visualisierung der Daten nicht die Kernaufgabe desselben. Deswegen wurde die Visualisierung der Daten aus dem Framework entfernt, und es wurde stattdessen um eine abstrakte und dynamische Abfragefunktion erweitert. Die Abfragelogik ist in der jeweiligen Provider-Klasse implementiert. Eine Provider-Klasse kann die Daten z. B. nach Top/Flop-Reihenfolge vorbereiten oder ein zweites Aggregationselement hinzufügen. Um diese Logik unabhängig von der Datenquelle zu haben, wurden die Reader-Klassen eingeführt. Sie setzten die Abfrage anhand des implementierten „Datendialekts" um und liefern die Daten zurück. Mit dem Austausch der Reader-Klasse kann man die Datenbeschaffung per Konfiguration ändern.

Daten visualisieren

Wir können nun die Informationen sammeln, sichern und für die Visualisierung bereitstellen. Im vorherigen Abschnitt wurden zwei einfache Möglichkeiten gezeigt, wie man Daten visualisieren kann. Die Visualisierung dieser Art ist aber nicht professionell genug. Außerdem wollte ich meinen Kunden die Möglichkeit geben, die Visualisierung ihrer Daten selbst zu bestimmen und selbst zu gestalten. Deswegen habe ich mir zuerst JasperReports [4] angeschaut. JasperReports ist Java-basiert, hat einen WYSIWYG-(What-You-See-Is-What-You-Get-) Designer und kann mit mehreren Ausgabeformaten umgehen. Bei der Evaluierung habe ich aber gemerkt, dass es für meine „einfachen" Zwecke zu kompliziert und zu teuer ist. Ich hätte alternativ nur die JasperEngine ins

Parameter	Description	Example
url	Return with all results of this url. The url must be configured in the statistic.xml. You can use a part of the url as well (without .do).	http://servlet?url=login.do
name	Same as the url, but more comfortable. Configured in the statistic.xml as alias name.	http://servlet?name=login
tf	Filter the results as top or flop, depends on the value. If the value is bigger than 0, this means top, otherwise flop. This value sets the number of the results.	http://servlet?name=login&tf=3&tf_param=userid
tf_param	This param is required for top or flop. It tells the system which statistic parameter must be used for top or flop.	See above
from	Time filter. You can use it with all parameters. Use it like a simple date (YYYY-MM-DD HH:mm). You can use it without parameter to as well. If the HH:mm is not set, it means 00:00 (greater than 00 hour 00 minutes of the given day).	http://servlet?url=login&from=2008-01-01
to	Time filter. You can use it with all parameters. Use it like a simple date (YYYY-MM-DD HH:mm). You can use it without parameter from as well. If HH:mm is not set, it means 23:59 (less than 23:59 of the given day).	http://servlet?url=login&to=2012-08-01
paramName	Filter by a statistic attribute, defined as paramName.	http://servlet?url=login&paramName=userid&paramValue=norris
paramValue	Filter by a statistic attribute, the value is defined as paramValue	http://servlet?url=login&paramName=userid&paramValue=norris
distinct	Use to distinct aggregated values. For example, if you want to have the logins, aggregated per day but depends on the user id	http://servlet?url=login&distinct=userid

Tabelle 1: Servlet-Parameter

Framework integrieren können, um Kosten zu sparen (ohne den Designer). Das sprach aber gegen das Konzept, dass die Auswertung nicht die Kernaufgabe des Frameworks ist. Also musste eine andere Lösung her.

Dann stieß ich auf ARIS MashZone [5]. Dieses Tool ist eine leichtgewichtige Anwendung, mit der man professionell Kennzahlen, Daten oder Informationen visualisieren kann. Es ist kein BI-Tool und konzentriert sich hauptsächlich auf die Visualisierung von Daten. Die Anwendung ist zurzeit noch Flash-basiert und hat im Wesentlichen zwei Teile: Mit dem *Composer* kann man die Visualisierungskomponenten wie Charts, Pies, geografische Karten (auch Google Maps) usw. zu einem *Mashup* zusammenklicken. „Zusammenklicken" ist dabei wörtlich zu verstehen! Eine kurze Anleitung zur Erstellung der Mashups findet man auf der Webseite [5] von ARIS MashZone. Mit dem *DataFeed*-Editor stehen je nach Lizenzierung verschiedene Datenquellen (in der kostenlosen Variante hauptsächlich Excel, XML und CSV) zur Verfügung. Nach der Auswahl und der Konfiguration der Datenquelle kann man die Daten für die Visualisierung vorbereiten, weiter analysieren, miteinander oder mit anderen Daten aggregieren, verändern usw. Es besteht sogar die Möglichkeit, die Datenquelle über *http* anzubinden, was ich zur Visualisierung der statistischen Daten auch verwendet habe (**Abb. 2**). Ich verbinde also einen MashZone *CSV-DataFeed* mit meinem Servlet (siehe vorige Abschnitte) via URL und gebe dazu die Auswertungsparameter an (Tabelle 1).

Abb. 2: ARIS MashZone

ARIS MashZone ist in der Lage, die Daten interaktiv darzustellen. Mit so genannten *Filtern* kann man Abhängigkeiten zwischen Komponenten erstellen. Wenn ich z. B. aus einer Weltkartenkomponente nur Deutschland auswähle, sehe ich lediglich die Kennzahlen von Deutschland – vorausgesetzt, die zwei Komponenten sind mit einem Filter verknüpft und die Werte beider Komponenten lassen sich über einen Schlüsselwert (hier

z. B.: „Deutschland“) filtern. Diese Art von Interaktion sieht zwar sehr gut aus – wenn man aber die Datenseite der Anwendung anschaut, erkennt man die Grenzen von ARIS MashZone. Zuerst werden alle Daten aus der Quelle geladen (z. B.: Kennzahlen aller Länder) und dann die angezeigte Datenmenge durch die Filterung interaktiv (Klick auf Deutschland aktiviert die Filterung) reduziert. In der kostenlosen MashZone-Version kann man eine 1000 x 32 große Matrix von Daten aus einer Datenquelle lesen. Auch bei einer Professional-Lizenz ist es „nur“ etwa das Zehnfache (10 000 x 48 Matrix von Daten). Die Empfehlung von ARIS MashZone ist, nur Kennzahlen und keine Massendaten zu visualisieren. Bei einem Statistikframework hat man aber normalerweise Massendaten. Die Lösung war zum Glück sehr einfach.

Einerseits bietet ARIS MashZone „benutzerdefinierte Eingabefelder“ an, die man für das Filtern anderer Komponenten auch verwenden kann. Diese Felder kann man bei den *DataFeeds* auslesen. Somit werden die Parameter unserer Abfrage-URLs dynamisch. Das Statistikframework hat z. B. einen *von-* und einen *bis-*Parameter, damit die Businessdaten nur innerhalb eines im Zeitfilter vom Benutzern angeklickten Zeitraums abgefragt werden. So werden nur die Daten geladen, die der Benutzer tatsächlich sieht (was einer Ajax-Abfrage nicht ganz unähnlich ist). Anderseits habe ich in meiner Analyser-Klasse eine Default-Aggregation implementiert, weil z. B. die Zeitfilterkomponente nur Tage/Monate/Quartale und Jahre darstellen kann (also keine Minuten usw.). Die Default-Aggregation rechnet die Daten auf den jeweiligen Tag hoch, damit wir bei der Datenmenge sparen, die nicht visualisiert werden kann. Man kann auch sekundäre Aggregationen per Parameter angeben, wie zum Beispiel: Suche die Anmeldedaten innerhalb Dezember, Tag genau je unterschiedlicher Benutzername

```
http://servlet?url=login&from=2012-12-01&to=2012-12-31&distinct=usrname
```

Das Ergebnis wird also die Anzahl der Anmeldungen pro Tag pro Benutzer innerhalb des Monats Dezember sein. Mit ARIS MashZone kann man sehr viel machen. Am besten startet man mit den Tutorials auf der Webseite [5]. Wenn man aber nicht weiterkommt, lohnt es sich, eine Anfrage an die Community [6] zu senden. Dort wird bei Problemen immer schnell geholfen.

Listing 3

```
<bean id="urlDataProvider" class="com.stat.mashapp.StatisticURLProvider">
  <property name="optionalRequestParamKeyList">
    <list>
     <value>from</value>
     <value>to</value>
     <value>paramName</value>
     <value>paramValue</value>
     <value>distinct</value>
    </list>
  </property>
  <property name="statisticReader" ref="jpaStatisticReader"/>
  <!--property name="statisticReader" ref="xmlStatisticReader"/-->
</bean>
```

Listing 4

```
@StatisticVersion(version = StatisticVersions.V1_0)
public interface IStatisticReader {
  @Deprecated
  @StatisticVersion(version = StatisticVersions.V1_3)
  public List<IStatisticElement> getStatisticElements(String keyQuery);
  @Deprecated
  @StatisticVersion(version = StatisticVersions.V1_3)
  public IStatisticElement getStatisticElement(String keyQuery);
  @Deprecated
  public boolean containsStatisticElement(IStatisticElement statisticElement);
  public StatisticConfig getStatisticConfig();
  @StatisticVersion(version = StatisticVersions.V1_3)
  List<IStatisticElement> getStatElements(String url, HashMap<String, String>reqParams);
}
```

Fazit und Ausblick

Wir haben also gesehen, dass die Trennung der Datenanalyse von der Datenvisualisierung sinnvoll und notwendig ist, damit das Framework unabhängig vom Visualisierungstool bleibt. Es gibt z. B. Kunden, die die Ergebnisse (CSV-Datei) in SAP-BO importieren und auswerten. Andere nutzen lieber Excel oder ARIS MashZone dafür. Das Framework hat also sein Ziel in Bezug auf Unabhängigkeit auf jeden Fall erreicht. Mit den dynamischen Abfragen kann man die Daten beliebig auswerten und für die Visualisierung vorbereiten. Die Weiterentwicklung ist leicht, was den Ansatz von Continuous Delivery möglich macht. Eine Frage bleibt aber noch offen: Ist es schon ein Framework, und wenn ja, wann ist es fertig, was fehlt noch? Das werden wir im letzten Teil der Serie sehen.

János Vona ist Senior Manager ARIS Customized Solutions der IDS Scheer Consulting GmbH, einer Tochtergesellschaft der Software AG. Er leitet Individualprojekte im Kundenauftrag und beschäftigt sich seit mehr als zehn Jahren mit Java.

Janos.Vona@softwareag.com

Links & Literatur

[1] https://developers.google.com/chart/?hl=de-DE

[2] http://www.jfree.org/jfreechart

[3] http://www.jfree.org/eastwood

[4] http://jasperforge.org/projects/jasperreports

[5] http://www.mashzone.com

[6] http://www.ariscommunity.com/group/aris-mashzone

Anzeige

Modernizing Legacy, die sanfte Modernisierung einer relationalen DB durch aspektorientierte Programmierung

AOP – wir übernehmen

Der Datenbestand einer über Jahrzehnte gewachsenen Anwendung – traditionellerweise abgelegt in einer relationalen Datenbank – ist aus Sicht des Betreibers der Anwendung nicht selten existenzkritisch. So wie die Luft zum Atmen werden die Daten fürs Geschäft benötigt. Wehe dem, der den Bestand unsachgemäß verändert, gar korrumpiert! Doch was macht man mit solch einem Bestand, wenn die zugrunde liegende Anwendung modernisiert wird, ersetzt durch aktuelle Technologien? Bereinigen? Womöglich Millionen Informationseinheiten neu strukturieren? Ein Albtraum beginnt …

von Werner Gross

Modernizing Legacy – die Modernisierung alter, teilweise schon seit Jahrzehnten im Einsatz befindlicher Anwendungen – ist ein im wahrsten Sinne heißes Thema. Die ungebremsten Innovationszyklen der IT sorgen hier scheinbar für einen permanenten Bedarf der Erneuerung. Doch sowohl Algorithmen als auch die Daten, auf die sie einwirken, widersetzen sich nicht selten einer reibungslosen Erneuerung. Wer weiß noch nach all den Jahren, was die Software im Detail macht? Selbst eine überschwängliche Dokumentation im Quellcode der Legacy-Software kann mehr verwirren als wirklich helfen.

Der (relationale) Datenbestand einer Altanwendung – und um selbigen und seine Erneuerung geht es im Wesentlichen in diesem Artikel – ist in diesem Zusammenhang ein ganz spezielles Thema. Denn fast immer scheint er unüberbrückbar mit der Software, von der er verwendet wird, verwachsen zu sein, haben sich in all den Jahren der Benutzung Daten angesammelt, von denen sich keiner mehr trennen will – man weiß ja nie.

Modernizing Legacy der Datenbank

Spricht man heutzutage von der Modernisierung von Softwaresystemen, meint man fast immer die Einhaltung des objektorientierten Paradigmas in der neuen Softwarelösung. Die Objektorientierung beeinflusst aber nicht nur die Art und Weise der Programmerstellung, sondern natürlich auch die Art und Weise, wie Daten in Informationssystemen – vornehmlich also solchen, die zur Speicherung massenhaft vorzuhaltender Daten geeignet sind – zu speichern und wiederzubeschaffen sind. *ORM* – objektrelationales Mapping – ist hier das Schlagwort. ORM bedeutet, dass Objekte der objektorientierten Welt auf Entitäten der verbundenen relationalen Datenbank abgebildet werden. Um die technischen Details dieser Abbildung muss man sich als Softwareentwickler in aller Regel nicht kümmern. Besitzt man eine solche Abbildung zu einer gegebenen Datenbank, so muss man als Softwareentwickler die objektorientierte Welt nicht mehr verlassen, wenn man auf den Datenbestand des Systems zugreifen möchte. Statt sich also in Details der SQL-Abfragen zu verlieren und um sich nicht selbst um die mühselige und fehlerträchtige Umwandlung der Ergebniszeilen einer SQL-Abfrage in Objektinstanzen kümmern zu müssen, verwendet man die der SQL nachempfundene *OQL* (*Object Query Language*). Diese erledigt all diese Belange automatisch.

ORM ist im Rahmen der Softwareerneuerung also ein erstrebenswertes Ziel. Wo ist also das Problem? Machen wir es doch einfach …

Das Problem mit dem Erbe

Leider ist es nicht so einfach mit dem „Machen wir mal ORM“. Schauen wir uns dazu einmal die typischen Probleme und Altlasten einer Legacy-Datenbank an. Immer wieder stößt man auf die gleichen Probleme: Aus Sicht des Betreibers ist die Datenbank „normal“. Aber normalisiert im Sinne der Informatik ist sie deswegen noch lange nicht. Hier einige typische Beispiele:

- Oftmals sind die Relationen zwischen Entitäten nur hypothetischer Natur, tatsächlich (physikalisch) durchsetzen lässt sich der Constraint aber nachträglich nicht. Hierzu zählen z. B. Parent-Child-Beziehungen mit verwaisten Child-Entitäten, wenn in nicht sauber modellierten Transaktionen gespeichert wurde und dabei Fehlersituationen aufgetreten sind.
- Spezialbehandlungen – der Klassiker in Legacy-Datenbanken. Die Datenbank ist mit der Anwendung „historisch gewachsen“. Abkürzungen und Kurzschlüsse zwischen Entitäten sollten nachträgliche Anforderungen an die Anwendung schnell und unbürokratisch im Bestand abbilden. Wegen der fehlenden physikalischen Constraints sind eventuell daraus resultierende Constraint-Verletzungen letztlich aber nie aufgefallen.
- Ausgelassene Bestandskorrekturen – auch das findet man: Irgendwann wurde die Software überarbeitet,

Fehler wurden korrigiert, und das eine oder andere Attribut in einer Tabelle wird nun plötzlich anders – nämlich richtig – belegt. Die Legacy-Software kommt mit diesem Umstand klar – fast überwiegend wird in alter Software Zeile für Zeile auf die Ergebnismengen von Abfragen zugegriffen (man vergleiche etwa die Ansätze von *Embedded-Sql* in prozeduralen Programmiersprachen). Nicht relevante Ergebniszeilen werden einfach ausgelassen. In OO-Sprachen – wie z. B. Java – hingegen, wo es Objektsorten gibt, die Mengen repräsentieren, können fehlerhafte Zeilen einer Ergebnismenge plötzlich ein erhebliches Problem darstellen.

Diese Probleme – und noch viele andere, die ich hier im Detail nicht aufführen kann – verhindern oder erschweren erheblich die Verwendung von ORM. Letztlich bedeutet ja der Einsatz von ORM in diesem Zusammenhang die Abbildung eines sauberen Objektmodells auf eine – wie auch immer – unsaubere Datenbank. Es ist klar, dass dies nicht ohne Zusatzaufwendungen funktionieren kann.

Die Migrationshölle

Eigentlich liegt es doch auf der Hand: Mit der Modernisierung der Software ist auch eine Modernisierung der Datenbank erforderlich. Nicht nur, dass die Relationen der Entitäten neu und entsprechend der fachlichen Anforderungen zu definieren sind; auch eine Migration des Bestands von alt nach neu ist fällig. Stellen Sie sich eine mittelschwere Anwendung mit mehr als 500 Tabellen vor, in der alleine im Partnerbestand in den letzten 20 Jahren mehr als 600 000 Partnerinformationen hinterlegt wurden. Stellen Sie sich weiter vor, dass die Software selber in COBOL geschrieben wurde und sich über mehr als 8 000 000 Zeilen Quellcode erstreckt. Um es perfekt zu machen: Zudem war die Datenbasis ursprünglich nicht relational, sondern in VSAM-Dateien abgelegt, bevor sie dann irgendwann mehr schlecht als recht in eine relationale Datenbank überführt wurde. Welche Strategien sind hier möglich, diesen Bestand sicher zu modernisieren?

- **Ansatz 1** – der Big Bang: Alle fachlichen Relationen müssen bekannt sein. Die Abbildung alt auf neu sowie die sichere Migration des Altbestands einschließlich Korrektur, womöglich durch Löschen oder Bilden einer Rückstandsbearbeitung, müssen möglich sein. Zum Zeitpunkt T0 wird das alte System abgeschaltet, das modernisierte eingeschaltet, und nun beten wir, dass alles wieder da ist.
- **Ansatz 2** – Koexistenz in getrennten Beständen: Wir haben erkannt, dass der Big Bang nach hinten losgehen kann und dass das Risiko eines irreparablen Datenverlusts unkalkulierbar ist. Also versuchen wir zwei Bestände – nämlich den alten und einen neuen Bestand – parallel zu betreiben. Die Aufwände und Risiken einer solchen doppelt-koexistenten Datenbasis liegen auf der Hand.
- **Ansatz 3** – Ein evolutionärer Ansatz: Evolution statt Revolution ...

Evolution statt Revolution!

Evolutionäre Strategien in der OO, ob sie nun die inkrementelle Softwareentwicklung als solche oder Vorgehensmodelle betreffen, sind en vogue. Warum sollten sich solche Strategien nicht auch auf die Modernisierung von Datenbanken einer Legacy-Anwendung anwenden lassen? Die Theorie dahinter: Wir nehmen auf den Zustand der Datenbank keine Rücksicht und implementieren unsere Zugriffsschicht auf die Daten nach allen Regeln der OO-Kunst, heißt: Wir verwenden ein sauberes, erneuertes und modernes objektrelationales Modell und erlauben das – entsprechend der fachlich definierten Relationen – Traversieren im Objektgraphen, was ja eigentlich eine absolut normalisierte Datenbank voraussetzt. Wir belassen aber zunächst die Struktur der Datenbank, wie sie ist, einschließlich aller Unzulänglichkeiten und Fehler. Wie kann der Widerspruch „erneuertes Objektmodell per objektrelationalem Mapping vs. krude Datenbank“ nun aufgelöst werden?

AOP – wir übernehmen

Der Hebel für diese sanfte Evolution liegt in unserer Betrachtung an zwei wesentlichen Stellen der erneuerten Software:

- in den Entity-Klassen, die per ORM auf die Entitäten der Datenbank abgebildet werden sowie

- in der Zugriffsschicht, die den Zugriff auf das Datenbanksystem regelt.

Wann immer in Entity-Klassen Bezug auf Eigenschaften des objektrelationalen Modells genommen wird, die sich eigentlich aus bekannten Gründen in der Datenbank (noch) nicht wiederfinden – etwa Relationen zwischen Entitäten – wird gemäß Konfiguration Metacode ausgeführt, der die fehlenden Eigenschaften der Datenbank an genau dieser Stelle kompensiert. Hier zwei Beispiele:

- In der Datenbank fehlt der Constraint zwischen zwei Entitäten A, B und lässt sich wegen verwaister Child-Entitäten nicht nachträglich ergänzen. Im Objektgraphen traversieren wir trotzdem per *A->B*. Die Relation wird durch den „eingestreuten" Code in der *getList*-Methode in A aufgelöst und der Constraint beim Speichern durch in der Zugriffsschicht eingestreuten Code vor dem Lesen bzw. Speichern geprüft.
- In einem Attribut x in einer Tabelle T hat sich die Darstellung für „nicht belegt" im Laufe der Zeit von Leerzeichen auf NULL geändert (typisch COBOL). Wir ergänzen Metacode, der verlässlich im Objekt immer per *getX()* das NULL antwortet, unabhängig davon, was in der Datenbank gespeichert ist.

Listing 1: Beispiel in Java

```
@Entity
@Table("KUNDE")
public class Kunde {
  ...
  private KundenId kundenid;

  @Transient
  private List<Auftrag> auftragList;
  ...
  @VirtualRelation(reltype = RelType.ONE_TO_N,
    service = "KundeDAO", attrib = {kundenid})
  public List<Auftrag> getAuftragList() {
    return auftragList;
  }
);
```

Listing 2: Beispiel in Java

```
@Entity
@Table("AUFTRAG")
public class Auftrag {
  ...
  private AuftragId auftragid;

  @Transient
  private Kunde kunde;
  ...
  @VirtualRelation(reltype = RelType.N_TO_ONE,
    service = "AuftragDAO", attrib = {kunde})
  public Kunde getKunde() {
    return kunde;
  }
);
```

Was heißt nun „Metacode"? Um es zu verdeutlichen: Wir haben hier eine Zwickmühle, aus der wir mit traditionellen Techniken der Softwareentwicklung nicht so einfach herauskommen. Einerseits möchten wir, dass sich im Code die Kompensation der Fehler der Legacy-Datenbank wiederfindet, um selbige nicht per Big Bang modernisieren zu müssen. Aber andererseits wollen wir ja die Mängel der Datenbank Schritt für Schritt evolutionär korrigieren, ohne dazu jedes Mal Änderungen am Zugriffscode durchführen zu müssen. Aufgelöst werden kann dieses Dilemma durch die Verwendung von aspektorientierter Programmierung – AOP. AOP erlaubt es, den Code, den wir für das objektrelationale Mapping verwenden, nachträglich, d. h. entweder zur Compile-Zeit oder sogar erst zur Ausführungszeit der Anwendung, zu instrumentieren. Instrumentation heißt dabei

- Einfluss nehmen, bevor ein Zugriff in einem Objekt erfolgt, etwa, um bestimmte Zustände vor dem Zugriff herstellen zu können (etwa per Dependency Injection)
- Übernahme der vollständigen Kontrolle über einen Zugriff in einem Objekt, etwa, um eine Relation aufzulösen, obwohl sie wegen fehlender Constraints in der Datenbank im ORM-Tool der Wahl nicht definiert werden konnte.
- Einfluss nehmen, nachdem ein Zugriff in einem Objekt erfolgt ist, etwa, um nachträgliche Korrekturen im Objekt vornehmen zu können.

Die Besonderheit dabei ist, dass dieser Metacode sich nicht in den Klassen selber, die davon betroffen sind, wiederfindet, sondern in den Code dieser Klassen eingewoben wird. Diesen Vorgang nennt man *Code-Weaving*. In Java etwa lassen sich die Stellen, an denen Code einzuweben ist – die so genannten *Point-Cuts* – per Annotation markieren oder per XML deklarieren, je nach AOP-Werkzeug der Wahl. Entfernt man die Point-Cut-Deklaration, wird auch kein Code mehr eingewoben. Übersetzt heißt das: Entwickelt sich die Legacy-Datenbank im Verlaufe des Projekts weiter, d. h. es werden Fehler korrigiert bzw. Neuerungen eingezogen, so können auch die der Kompensation dienenden Aspekte im Code durch Weglassen der Point-Cut-Definitionen schrittweise entfallen, und zwar ohne dass dies Einfluss auf die eigentliche Implementierung hätte.

Ein Beispiel

Das folgende Beispiel (Java) zeigt exemplarisch, wie die Kompensation durch Metacode funktioniert. Wir gehen davon aus, dass es zwei Entitäten, nämlich Kunde und Auftrag, im Bestand gibt. Eigentlich gilt das Aggregationsprinzip zwischen Kunde und Auftrag, d. h. es darf

keine Aufträge ohne Kunden im System geben, und die Beziehung zwischen Kunde und Auftrag sei 1:n. Leider wurden im Legacy-System Kunden und Aufträge in getrennten Transaktionen gespeichert, sodass es immer wieder vorkam – Programmabbruch –, dass zwar Aufträge angelegt wurden, aber keine Kunden dazu existieren. Damit lässt sich in der relationalen Datenbank der Constraint zwischen Kunden und Auftrag nicht nachträglich physikalisch aktivieren. Eine Bereinigung des Bestands gemäß „Lösche alle Aufträge ohne Kunden" verbietet sich aufgrund der Komplexität des Datenmodells und der Abhängigkeiten zum Zeitpunkt der Neuimplementierung, wo die Beziehung Kunde/Auftrag bereits benötigt wird. Wir lösen dieses Problem daher zunächst durch einen „virtuellen Constraint" (Listing 1).

Und nun der Auftrag als Gegenstück zum Kunden (Listing 2).

Wegen des fehlenden Constraints in der Datenbank benutzen wir auch keine Relationsdefinition des verwendeten ORM-Werkzeugs, hier angedeutet JPA (*Java Persistence API*). Trotzdem beziehen wir uns auf eine fachlich existierende Relation und erlauben auch das Traversieren im Graphen per *kunde.getAuftragList()* bzw. *auftrag.getKunde()*. Nur Auflösen müssen wir die Relation selber. Das Beispiel in Listing 3 verwendet hierzu AspectJ.

Dieser Aspekt implementiert Verhalten, das allen per *@VirtualRelation* markierten Methoden in Entity-Klassen hinzugefügt wird. Die Notation *before* im Aspekt regelt dabei, wann dieser Code auszuführen ist, hier naheliegend: bevor die markierte *getter*-Methode in der Entity-Klasse ausgeführt wird. Wie der virtuelle Constraint letztlich wirkt, regelt dann die Parametrisierung der Annotation, die den Point Cut im Java-Code markiert.

Die Codebeispiele sind naturgemäß stark abstrahiert. Sie sollen lediglich verdeutlichen, wie der Mangel der fehlenden Relation in der Datenbank durch einen eingewobenen Aspekt in der betroffenen Modellklasse ausgeglichen (kompensiert) werden kann. Für den Entwickler im Modernizing-Legacy-Projekt, der die Relation Kunde/Auftrag verwendet, ist dabei letztlich egal, ob das Mapping durch die Deklaration einer Relation mit den Möglichkeiten des verwendeten ORM-Tools erfolgt ist oder – weil es aus technischen Gründen einfach (noch) nicht möglich war – durch die Verwendung eines eingewobenen Kompensationsalgorithmus, der Gleiches leistet.

Erzeugung der Kompensationsalgorithmen

Bleibt noch zu klären, woher wir wissen, welche Stellen in unseren Entity-Klassen wie zu markieren sind. Offensichtlich benötigt man so etwas wie einen Reverse-Engineering-Schritt im Legacy-Projekt, dessen Aufgabe es ist, nachträglich ein sauberes Entity-Relationship-Modell zu finden, und dies auch noch unter dem Gesichtspunkt der Praktikabilität. Man rufe sich ins Gedächtnis zurück:

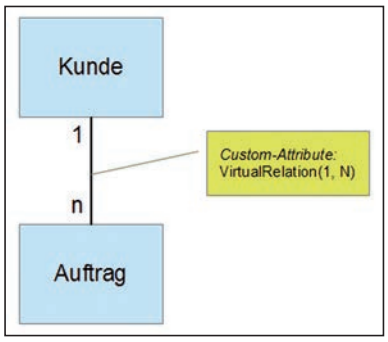


Abb. 1: Auszeichnung der Kante des Graphen mit einem Custom-Attribut

Wir sind von einer Legacy-Datenbank ausgegangen, die alles, nur nicht normalisiert ist. Da in der Datenbank keine – oder wenn, dann nur unvollständige – Constraints zwischen den Tabellen definiert sind, liefert ein Werkzeug zum Auswerten der Beziehungen der Entitäten, basierend auf den Metainformationen der Datenbank, also nur einen ersten Hinweis auf die tatsächlichen Abhängigkeiten. Weitere Analysetätigkeiten sind daher erforderlich, etwa:

- Codeanalyse, um programmatisch abgebildete Beziehungen zu erkennen
- Codeanalyse, um Spezialbehandlungen und Sonderfälle des Datenmodells zu finden
- Unter Umständen kann sogar ein Ähnlichkeitsvergleich von Attributbezeichnern in Kombination mit identischen Type-Vereinbarungen Hinweise auf versteckte Abhängigkeiten liefern.
- Insbesondere entscheiden natürlich fachliche Zwänge, wo Abhängigkeiten existieren sollten bzw. wo nicht.

Listing 3: Beispiel in AspectJ

```
public aspect VirtualRelationHandling {
  ...
  pointcut resolveGraph(...)
    execution(@VirtualRelation * *.get*()) ...
  ...
  before (): resolveGraph() {
    Auswerten der Meta-Informationen ...
    Auflösen des Service DAO ...
    Ausführen des OQL-Statements, um die geforderten Entitäten
      zu ermitteln ...
  }
);
```

Das Konzept der aspektorientierten Programmierung

AOP, aspektorientierte Programmierung, ist ein zentrales Programmierparadigma, das vor allem in Kombination mit einer objektorientierten Programmiersprache – etwa Java – zur vollen Geltung gelangt.
Bei der aspektorientierten Programmierung geht es darum, die technischen Belange einer Softwarelösung von den nichttechnischen so zu trennen, dass sie sich nicht gegenseitig beeinflussen. Wo man früher klassischerweise ein Framework als technische Basis der fachlichen Implementierung verwendet und dieses dann das Paradigma der Realisierung wesentlich beeinflusst hat, werden heute die Querschnittbelange der Software durch entsprechende Techniken in den Code eingewoben, ohne dass dieser in direkte Abhängigkeit zu ihnen gerät. Als möglicher Anwendungsfall sei hier z. B. Dependency Injection genannt – das Injizieren von Abhängigkeiten in Objekten, die über definierte Merkmale (z. B. in Java typischerweise per Annotation) verfügen. AOP gilt mittlerweile als Best Practice und ist aus der modernen Softwareentwicklung nicht mehr wegzudenken.

All dies zusammen ergibt die Beschreibung eines Domänenmodells, das Grundlage für einen generativen Schritt im Entwicklungsprozess der Modernisierung der Anwendung sein kann. Unser objektrelationales Mapping – etwa m. H. von Java – wird also idealerweise generiert. Doch wie kommen nun unsere Aspekte, also etwa virtuelle Constraints (s. o.), hier ins Spiel? Wir müssen dazu unser Modell um zusätzliche „Metaattribute" ergänzen, die Hinweise darüber geben, um welches Verhalten der aus dem Modell generierte (Java-)Code zu ergänzen ist. Nehmen wir wieder unser Kunde-Auftrag-Beispiel. In der rückwärtigen Analyse des Legacy-Code haben wir die Abhängigkeit zwischen den beiden Tabellen durch wiederholte Codesequenzen der Form „Lese einen Auftrag abhängig vom gewählten Kunden mit Schlüssel" aufgedeckt. Diese Abhängigkeit haben wir in unserem Modell als „echte Relation" eingetragen; jedoch wissen wir, dass wir den Constraint wegen der unsauberen Datenbank nicht physikalisch aktivieren dürfen. Diesen Umstand verbuchen wir als Metaattribut im Modell an der Kante des Graphen zwischen Kunde und Auftrag. Damit das funktioniert, muss unser ER-Modellierungs-Tool das Hinzufügen von zusätzlichen Steuerattributen erlauben. Notfalls benutzen wir Kommentarfelder oder ähnliches (**Abb. 1**).

Entscheidend ist, dass wir in der Lage sind, das persistierte Modell auszulesen und eben diese Metainformationen ermitteln zu können, um daraus die Markierungen (Annotationen in Java) für den eigentlichen Code generieren zu können (aus der Datenbank selber können wir die Metainformation aus bekannten Gründen ja nicht gewinnen).

Fazit

Wir haben gesehen, dass moderne Konzepte der Objektorientierung, sofern sie die Datenhaltung und deren Einsatz im Rahmen von Legacy-Projekten betreffen, nicht im Konflikt zu den Altlasten einer verwendeten Legacy-Datenbank stehen müssen, sofern man zwischen objektrelationalem Modell und der Datenbank hinreichend abstrahiert. Um diese Abstraktion zu kontrollieren, sollten Technologien – wie die der Aspektorientierung – eingesetzt werden, möglichst in Kombination mit einem generativen Ansatz. Nur so erfordert die Modernisierung der Datenbank und damit letztlich der gesamten Legacy-Anwendung keinen Big Bang, sondern erlaubt die inkrementelle Fortentwicklung der Datenbank bei gleichzeitigem Erhalt des Bestandes bzw. dessen sanfter Überführung in eine modernisierte Form.

Werner Gross ist Senior IT-Consultant bei der PTA GmbH in Mannheim. Seit über zwanzig Jahren ist er als Softwareentwickler tätig. Den größten Teil dieser Zeit hat er sich der Objektorientierung gewidmet. Sein derzeitiger Schwerpunkt sind Java-JEE-basierte Architekturen.

Continuous Integration für automatisierte Geschäftsprozesse mit Maven und Jenkins

Geschäftsprozesse vom Fließband

Während in Java-Projekten ein automatisierter Build heutzutage Standard ist, wird dies in Projekten, die auf BPEL oder BPMN aufsetzen, oftmals versäumt. Die Modellierungsumgebungen, wie in unserem Fall ActiveVOS, bieten nur rudimentäre Unterstützung für Build-Werkzeuge. Zudem gilt es viele Abhängigkeiten zu WSDL- und Schemadateien zu verwalten, die nicht in Maven, sondern in anderen Repositories versioniert sind. Wir stellen die Projektstruktur vor, die sich im Laufe unseres Projekts entwickelt hat, und die es uns erlaubt, automatisiert alle Projektartefakte inklusive unserer ausführbaren Geschäftsprozesse zu paketieren und zu verteilen.

von Dr. Daniel Lübke und Martin Heinrich

Geschäftsprozesse, die mittels BPMN2 oder BPEL modelliert und ausgeführt werden, finden immer mehr Anklang. Die schnelle Entwicklung von prozessbasierten Anwendungen durch die Nutzung von Geschäftsprozess-Engines sowie die Möglichkeit, Prozesse besser zu verwalten und zu kontrollieren, sind wichtige Vorteile dieser Technologie.

Was viele Marketingprospekte jedoch verschweigen, ist die Tatsache, dass die Modellierung ausführbarer Geschäftsprozesse ebenfalls Programmierung ist. Somit treten dort ähnliche (Qualitäts-)Probleme auf wie in anderen Softwareprojekten. Neben Unit Tests sind also vollautomatisierte Builds und Deployments ein Thema. Aufbauend auf diesen ist es auch möglich, Nightly Builds und Continuous Integration aufzubauen, die nochmals die Qualitäts- durch stabile Software-Builds steigern können.

Die meisten Werkzeuge sind jedoch auf eine Programmiersprache wie Java, C# oder Ruby ausgerichtet. Daher ist es nötig, sich Gedanken zu machen, wie man mit möglichst wenig Aufwand die Geschäftsprozesse wie alle anderen Komponenten in dem Projekt handhaben und in die Build-Umgebung integrieren kann.

Beispielprojekt

In unserem aktuellen Projekt standen wir schnell vor der Herausforderung, ein komplexes Softwareprojekt zur Prozessintegration von mehreren Hundert Partnersystemen effizient zu entwickeln und mit hoher Qualität in die Produktion zu überführen. Der Prozessteil dieses Projekts besteht aktuell aus:

- 12 Prozessen, die mittels ActiveVOS modelliert und ausgeführt werden
- zwei Java-Bibliotheken, um weitere XPath-Funktionen auf dem Server zur Verfügung zu stellen
- 18 BPELUnit-Test-Suiten, die Unit und Integrationstests abdecken

Daneben gibt es noch weitere Java-Webanwendungen und Web Services, die aber klassische Java-Komponenten sind und somit aus Maven- und Jenkins-Sicht nicht interessant.

Im Folgenden wollen wir demonstrieren, wie das komplette Build und Deployment in diesem Projekt mittels Apache Maven und Jenkins automatisiert werden konnte.

Build-Automatisierung mit Apache Maven

Im ersten Schritt musste das komplette Projekt automatisiert erstellt und paketiert werden. Da neben den ausführbaren Prozessen ausschließlich Java-Komponenten entwickelt wurden, fiel die Wahl für das Build-System auf Apache Maven [1].

Damit wollten wir standardisierte Builds und Deployments etablieren, die nicht abhängig vom Entwickler sind, sondern – egal von wo, von wem und zu welchem Zeitpunkt – einheitlich gemacht werden. Neben den Java-Projekten gibt es im Wesentlichen zwei Maven-Modultypen: Module für WSDLs und solche für die Prozesse.

Die WSDLs werden zum einen Teil zentral in einem separaten Subversion Repository verwaltet und von dort ausgecheckt. Diese WSDLs sind extern für Partnersysteme sichtbar und unterliegen daher einem separaten Review- und Freigabeprozess. Der andere Teil beschreibt interne Services, die ohne Freigabe für die ausschließlich

Abb. 1: Modulstruktur

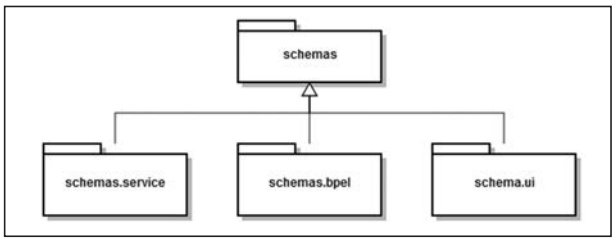
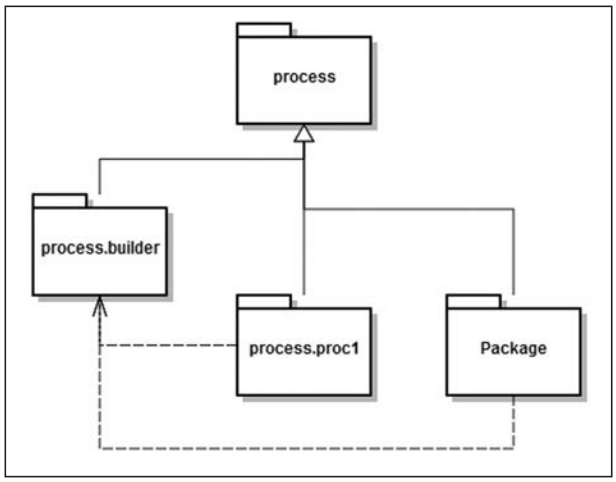


Abb. 2: Maven-Struktur für BPEL-Projekte



interne Benutzung entwickelt werden. Diese WSDLs liegen in der Projekt-Subversion direkt als Maven-Module vor.

Aufbau der zu erstellenden Pakete

Es gibt ja einige Möglichkeiten, wie man Java-Projekte strukturieren, konfigurieren, bauen und zuletzt auch verteilen (deploy) kann. Wir haben uns für Maven entschieden. Im Team war genügend Know-how vorhanden, und man kann mit Maven mit der gleichen Projektkonfiguration arbeiten wie auf dem Build-Server. Uns ist bewusst, dass Maven-Builds nicht den schnellsten Durchlauf haben. Aber dies nehmen wir gerne in Kauf, wenn die Projekte auf allen Umgebungen inklusive der IDE gleich konfigurierbar sind sowie alle Projektbeteiligten das Build-Werkzeug kennen.

WSDL-Pakete

Für die für den Informationsaustausch bestimmten Schemas und WSDLs haben wir zur allgemeinen Verwendung für die BPEL- und Java-Projekte eine eigene Projektstruktur aufgebaut. Dies erlaubt uns:

- das automatische Generieren der notwendigen Klassen für die Schnittstellen
- das explizite Referenzieren der notwendigen Schemas und WSDLs in den BPEL-Projekten

Alle unsere Projektpaketgruppen fassen wir in einem Basisprojekt zusammen, damit wir allgemeine, für alle Pakete gültige Erstellungskonfigurationen definieren können. Dies erlaubt uns in den eigentlichen Subpaketen mit einer minimalen Konfiguration auszukommen, um die JAXB-Klassen und auch das eigentliche Paket zu erstellen. Die daraus entstehende Struktur ist in **Abbildung 1** gezeigt. Die in Tabelle 1 dargestellten Pakete werden zu JARs zusammengebaut und im zentralen Maven Repository zur Verfügung gestellt, damit BPEL- und Java-Projekte diese in ihre Abhängigkeiten aufnehmen und aus dem Artefakt entnehmen können, was sie brauchen, Klassen, Schemas oder WSDLs.

Im übergeordneten Projekt *Schemas* werden alle Plug-ins konfiguriert (JAX-WS für die Generierung von Java-Code für die Web-Service-Schnittstellen, XML-Validierung für alle enthaltenen XML-Dateien) sowie die Einstellungen für die WSDL- und XSD-Paketierung definiert, sodass diese aus den Ressourcen mitpaketiert werden. Das *pom.xml* kann in dem Repository zu diesem Artikel heruntergeladen werden [2].

Durch die globalen Definitionen in dem Parent-Projekt ist das POM des Pakets *schemas.service* sehr kurz. Dies ist insbesondere wichtig, da es viele solcher Projekte gibt und somit unnötige Arbeit vermieden wird (Listing 1).

BPEL-Pakete

BPEL-Projekte in eine Struktur zu bringen, die mit den gleichen oder möglichst ähnlichen Verfahren zu bauen und zu verteilen sind wie all die anderen, „normalen" Java-Pakete, gestaltete sich dagegen aufwändiger.

Es gibt in Maven leider kein „packaging bpel". Somit mussten wir eine Lösung finden, das Projekt in Maven so zu beschreiben, dass es mit dem ActiveVOS-Designer (Eclipse) umgehen konnte und die BPEL-Projekte auch auf dem Build-Server gebaut und automatisch in die ActiveVOS Engine verteilt werden konnten. Die von uns gewählte Struktur ist in **Abbildung 2** gezeigt.

Auch hier verwenden wir ein kleines Maven-Projekt, um mehrere Prozessprojekte in einem Projekt zusammenfassen zu können. Neben den Modulen enthält es die Maven-Profile, die wir weiter unten beschreiben werden. Wir erlauben uns hier, nicht auf alle Maven-Details einzugehen, sondern konzentrieren uns auf die Elemente, die wir für den Maven-basierenden Bau von Prozessen verwenden.

ActiveVOS bietet die Möglichkeit, die Prozesse mit Ant zu bauen und auch zu verteilen. Damit wir das *build.xml* nicht in jedes einzelne Prozesspaket von Hand kopieren mussten, bauten wir ein kleines Maven-Resource-Paket, das zu einem JAR gebaut wird (Listing 2).

Tabelle 1: Beispiele für Projektmodule

Paket	Beschreibung
schemas	Das Basispaket: Enthält die für alle Subpakete gültigen Konfigurationen
schemas.service	Beispiel einer Schnittstellenbeschreibung eines Web Services, enthält die Schemas und die WSDLs
schemas.bpel	Beispiel einer Schnittstellenbeschreibung eines BPEL-Prozesses
schemas.ui	Beispiel einer Schnittstellenbeschreibung eines BPEL-Prozesses, der nur im User Interface verwendet wird

Damit der ActiveVOS ANT Build auch wirklich in Maven funktioniert, muss im Maven-Dependency-Management das *maven-ant-plugin* [3] richtig konfiguriert werden (Listing 3). Darüber hinaus sind neben dem *maven-ant-plugin* noch die Abhängigkeiten auf die ActiveVOS-Bibliotheken zu definieren. Wir wrappen das vom ActiveVOS Designer erstellte Ant-Skript mit einem eigenen, das die gesamte Konfiguration vornimmt. Dieses ist ebenfalls im Repository zu diesem Artikel erhältlich.

Paket process.proc1

Dadurch dass die Schemas und WSDLs als JARs im zentralen Maven-Repository verfügbar sind und für das Erstellen das Ressourcenprojekt *process.builder* auch als JAR im Maven Repository vorhanden ist, können wir mit dem eigentlichen Prozessprojekt beginnen. Prozesse funktionieren ohne die Umsysteme nicht wirklich, daher muss in einem BPEL-Projekt erst sichergestellt werden, dass alle notwendigen Schemas und WSDLs verfügbar sind. In unserem Fall sind diese Dateien in unseren WSDL-Paketen in unserem zentralen Maven-Repository verfügbar und müssen nur noch kopiert, entpackt und die enthaltenen Schemas und WSDLs an einen vorgegeben Platz in der Projektstruktur kopiert werden.

Dies kann mit dem Maven-Plug-in *maven-dependency-plugin* ziemlich einfach realisiert werden. Man muss sich nur Gedanken machen, wohin die Ressourcen kopiert werden sollen. Für temporäre Dateien ist in einem Maven-Projekt das Verzeichnis *target* der richtige Ort. Der ganze Inhalt im *target*-Verzeichnis kann mit dem Maven Goal *clean* gelöscht und danach z. B. mit *compile* und anderen Maven Goals wieder initialisiert werden.

Listing 1

```
<project [...]>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>mygroupid</groupId>
    <artifactId>schemas</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>schemas.services</artifactId>
  <version>0.0.11-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>schema.services</name>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jaxws-maven-plugin</artifactId>
        <executions>
          <execution>
            <phase>process-resources</phase>
            <goals>
              <goal>wsimport</goal>
            </goals>
            <configuration>
              <verbose>true</verbose>
              <extension>true</extension>
              <wsdlDirectory>src/main/wsdl/Internal</wsdlDirectory>
              <wsdlFiles>
                <wsdlFile>Services.wsdl</wsdlFile>
              </wsdlFiles>
              <wsdlLocation>/*</wsdlLocation>
              <sourceDestDir>${basedir}/target/generated-sources/jaxws
                                                        </sourceDestDir>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 2

```
<project [...]>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>mygroupid</groupId>
        <artifactId>process</artifactId>
        <version>1.2.09-SNAPSHOT</version>
    </parent>
    <artifactId>process.builder</artifactId>
    <packaging>jar</packaging>
    <name>process.builder</name>
    <description>Module contains additional scripts to build and deploy a process
                                                        </description>
    <build>
        <resources>
            <resource>
                <directory>src/scripts</directory>
                <includes>
                    <include>**/build.xml</include>
                </includes>
            </resource>
        </resources>
    </build>
</project>
```

Listing 3

```
<properties>
    <maven-antrun-plugin.version>1.6</maven-antrun-plugin.version>
    <activevos.version>9.0.0</activevos.version>
    <ant.version>1.8.2</ant.version>
    <ant-nodeps.version>1.8.1</ant-nodeps.version>
    <commons-net.version>3.0.1</commons-net.version>
    <commons-discovery.version>0.5</commons-discovery.version>
    <commons-logging.version>1.1.1</commons-logging.version>
    <axis.version>1.4</axis.version>
    <wsdl4j.version>1.6.2</wsdl4j.version>
    <axis-wsdl4j.version>1.5.1</axis-wsdl4j.version>
    <spring.javax-activation.version>1.1.1</spring.javax-activation.version>
</properties>
```

Daher kopieren wir nun die im JAR vorhandenen Dateien aus dem Verzeichnis *wsdl* in das lokale Projektverzeichnis *target/wsdl*. Dieses Verzeichnis kann nun im BPEL-Prozess relativ referenziert werden. Außerdem dürfen wir den bereits oben genannten *process.builder* mit dem entsprechenden *build.xml* für den eigentlichen Prozessbau und dem Prozessinstallationsschritt vergessen.

Damit für die Maven Dependency klar ersichtlich ist, dass das Projekt *process.proc1* vom *schemas.service* abhängig ist, sollte man noch die entsprechende Dependency im POM angeben. Da die Abhängigkeit nicht in dem Prozess mitpaketiert werden muss, geben wir für den scope *provided* an (Listing 4).

Wie die Dateien genau kopiert werden, ist in **Abbildung 3** gezeigt. In den Projekten, in denen Schemas (und die genauso aufgebauten Datentransformationsprojekte) gespeichert werden, liegen die Dateien ausschließlich im *src*-Bereich. Auch Abhängigkeiten wie z. B. gemeinsam genutzte XML-Schemas werden in den *src*-Baum kopiert. Dies erlaubt es, in den Editoren korrekte, relative Referenzen zu setzen, verlangt aber, dass man diszipliniert die kopierten Dateien in der Versionsverwaltung seiner Wahl ignoriert. Beim Build werden alle benötigten Dateien in *target/generated-sources* kopiert und von dort paketiert, sodass das Maven-Artefakt auch alle Abhängigkeiten bereits besitzt. Somit müssen andere Projekte nur eine Abhängigkeit auf ein Projekt setzen und nicht selbst alle transitiven Abhängigkeiten verwalten. Unserer Erfahrung nach ist es immens wichtig, dass die relativen Verweise funktionieren, weil sonst viele Tools nicht einsetzbar sind oder erst aufwändig mit XML-Schema-Katalogen konfiguriert werden müssen, sofern das Werkzeug solche überhaupt unterstützt.

In den BPEL-Projekten werden alle benötigten Dateien in das *target*-Verzeichnis kopiert und aus dem BPEL-Prozess relativ referenziert. Beim Paketieren werden alle benötigten Dateien aus dem *src*- und dem *target*-Zweig in die Deployment File kopiert.

Hat man den Prozess im ActiveVOS Designer erstellt, kann dieser nun auch über Maven gebaut und auch auf der ActiveVOS-Instanz installiert (verteilt) werden. Dazu verwenden wir das *maven-ant-plugin*, das im Maven-Plug-in-Management bereits soweit vorkonfiguriert wurde, dass nur noch der eigentliche Aufruf des *build.xml*s konfiguriert werden muss (Listing 5).

Da es in Maven keinen Pakettyp für die ActiveVOS BPRs gibt, haben wir uns mit einem kleinen Plug-in *build-helper-maven-plugin* [4] beholfen, um im Maven-Projekt nicht nur das Pseudo-JAR zu bauen und zu paketieren und ins Maven Repository zu stellen, sondern auch das eigentliche BPR (Listing 6).

Listing 4

```
<dependencies>
  <dependency>
    <groupId>mygroupid</groupId>
    <artifactId>schemas.service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>mygroupid</groupId>
    <artifactId>process.builder</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Listing 5

```
<plugin>
 <artifactId>maven-antrun-plugin</artifactId>
 <executions>
  <execution>
   <id>run-package</id>
   <phase>package</phase>
   <goals>
    <goal>run</goal>
   </goals>
   <configuration>
    <target>
     <ant antfile="target/bin/build.xml" target="package"
                                    inheritAll="true" inheritRefs="true">
      <property name="processname" value="proc1" />
     </ant>
    </target>
   </configuration>
  </execution>
 </executions>
</plugin>
```

Listing 6

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>build-helper-maven-plugin</artifactId>
 <executions>
  <execution>
   <id>attach-artifacts</id>
   <phase>package</phase>
   <goals>
    <goal>attach-artifact</goal>
   </goals>
   <configuration>
    <artifacts>
     <artifact>
      <file>${project.build.directory}/proc1.bpr</file>
      <type>bpr</type>
     </artifact>
    </artifacts>
   </configuration>
  </execution>
 </executions>
</plugin>
```

Prozess-Deployment

Nun fehlt nur noch das Deployment. Das heißt, wir bauen den Prozess automatisch zusammen und wollen ihn auch gleich in einer lokalen oder auf einer entfernten ActiveVOS-Instanz verteilen (deploy). Dazu verwenden wir auch wieder unser oben erwähntes Ant-Skript. Damit wir aber ein lokales und ein entferntes Deployment unterscheiden können, fügen wir in unserem POM zwei weitere Profile ein, wie in Listing 7 gezeigt. Nun kann mit der Angabe des Maven-Profils und dem Maven Goal *install* der gebaute Prozess auf die richtige ActiveVOS-Instanz installiert werden: *mvn clean install -Pdev*.

Zentraler Build mit Jenkins und Staging

Durch die Wahl von Maven kann nun dieselbe Projektkonfiguration für den Nightly Build, der zentral auf einem Development-Server ausgeführt wird, implementiert werden. Es können hier die gleichen Maven Goals verwendet werden wie auf dem lokalen PC des Entwicklers. Je nachdem, wie die eigene Entwicklungsumgebung aussieht, können die unterschiedlichen Environments im Maven-Projekt durch Profile unterschieden werden. Wie im vorherigen Abschnitt beschrieben, verwenden wir zwei Profile:

- *local*: für das lokale Deployment, z. B. für Unit Tests
- *dev*: für das Deployment auf die Development-Umgebung des Entwicklungsteams

Während der Nightly Build mindestens tagesaktuelle Deployments auf der Integrationsumgebung zur Verfügung stellt, wird von den Administratoren auf Anforderung auf das Test- und das Produktionssystem deployt.

Der Release Build wird ebenfalls über Jenkins gesteuert und führt neben den üblichen Artifact Releases noch ein Staging für den Betrieb ein. Die Schnittstelle zum Betrieb sind dabei spezielle Verzeichnisse, in denen fertig paketierte Deployment-Einheiten zur Verfügung gestellt werden. Dies umfasst die WAR-Archive sowie die BPRs für die Geschäftsprozesse. Zusätzlich werden aber auch für alle Server die Einstellungen als Textdateien exportiert. Diese können vom Betrieb mittels Scripts auf die Test- und Produktionsinstanzen angewendet werden. Somit ist sichergestellt, dass alle Einstellungen korrekt übernommen werden. Damit die BPRs richtig paketiert werden können, muss das Maven-POM noch etwas erweitert werden, damit

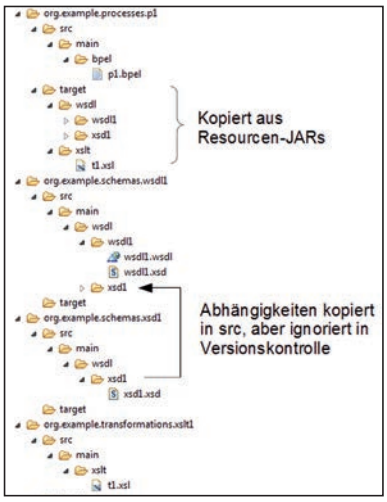


Abb. 3: Ressourcenverwaltung innerhalb der Prozessprojekte

- das richtige BPR mit der korrekten Versionsangabe in das entsprechende Staging-Verzeichnis kopiert werden kann
- das erzeugte Artefakt, welches „gestagt“ wurde, auch in unserem Maven Repository archiviert wird

Das Kopieren in die Staging-Verzeichnisse wurde durch Kopieraktionen gelöst, die mittels eines eingebetteten Apache Ant Copy Tasks arbeiten. Selbstverständlich releasen

wir vor dem Staging alle Projekte unter Verwendung des *maven-scm-plugin*. Um den Rahmen des Artikels nicht zu sprengen, verzichten wir hier auf die genauen Details.

Um die Deployment-Pakete für alle Staging-Umgebungen verwenden zu können, haben wir die URN-Mappings auf dem ActiveVOS-Server genutzt, die es erlauben, die Prozesse mit logischen Endpunkten zu paketieren, die dann durch eine Serverkonfiguration auf physische Endpunkte abgebildet werden. Dieser „Mini-ESB" erlaubt es auf einfache Art und Weise, dass das gleiche Paket für Unit Tests mit gemockten Services bis hin zum produktiven Einsatz genutzt werden kann, die Konfiguration serverseitig geschieht.

Listing 7

```
<profile>
  <id>dev</id>
    <properties>
      <activevos.deploy>http://remotehost:8080/active-bpel/services/
                                              ActiveBpelDeployBPR</activevos.deploy>
      <activevos.username>myuser</activevos.username>
      <activevos.password>mypassword</activevos.password>
    </properties>
    <build>
      <pluginManagement>
        <plugins>
          <plugin>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
              <execution>
                <id>run-deploy</id>
                <phase>install</phase>
                <goals>
                  <goal>run</goal>
                </goals>
                <configuration>
                  <target>
                    <ant antfile="${project.build.directory}/bin/build.xml" target="deploy"
                                                    inheritAll="true" inheritRefs="true">
                      <property name="processname" value="${process.name}" />
                      <property name="archive.deploypath" value="${activevos.deploy}" />
                      <property name="archive.username" value="${activevos.username}" />
                      <property name="archive.password" value="${activevos.password}" />
                    </ant>
                  </target>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </pluginManagement>
    </build>
  </profile>
```

Hinweis für die Modellierungsumgebungen

Beim Initialisieren der Entwicklungsumgebung von neuen Teilprojekten oder einem Clean-Checkout muss eines beachtet werden: Nach dem ersten Checkout und vor dem Öffnen des ActiveVOS Designers (oder anderen Prozessmodellierungsumgebungen) muss zuerst dafür gesorgt werden, dass alle notwendigen WSDLs und Schemas im *target*-Verzeichnis vorhanden sind. Dazu kann mit dem Maven Goal *compile* dafür gesorgt werden, dass die Maven-Phase *generate-sources* ausgeführt wird. Maven führt dann im obigen Beispiel das Goal *unpack* des *maven-dependency-plugin* aus und kopiert die Dateien.

Lessons learned

Auch wenn wir anfangs Probleme hatten, wie wir die Paketierung am besten lösen, so haben wir nun eine pragmatische Lösung gefunden. Dabei haben wir die Integration der WSDL-Dateien, die verschiedene Schemas aus unterschiedlichen Standards und die Maven-Integration der Geschäftsprozesse gelöst, sodass mittels eines einzigen Maven Builds alle Softwarekomponenten kompiliert und paketiert werden. Die durch automatisierte Builds und Nightly Builds gewonnene Sicherheit ist sowohl qualitativ als auch subjektiv für die Entwickler wichtig.

Was in Java-Projekten mittlerweile eine Selbstverständlichkeit ist, lässt sich mit ein wenig Arbeit auch für andere Artefakte erreichen. Die Erleichterung, auch in Stresssituationen auf Knopfdruck das Projekt bauen und deployen zu können, vermeidet Fehler; gerade im Vergleich zu anderen Projekten, wo die Geschäftsprozesse direkt aus der Entwicklungsumgebung auf Produktionsserver deployt werden, was über kurz oder lang zu einem Versionschaos und schwer nachzuvollziehenden Fehlern führt.

Daneben sind die integrierten Tests, auf die wir im Rahmen des Artikels nicht weiter eingehen konnten, ein Teil der Qualitätssicherungsstrategie. Hier sei auf [5] verwiesen, wo BPELUnit beschrieben wird, das ebenfalls über Maven-Unterstützung von Haus aus verfügt.

Dr.-Ing. Daniel Lübke arbeitet bei der innoQ Schweiz GmbH und berät dort Kunden in SOA- und BPM-Projekten. Er ist Mitautor des Buchs „Geschäftsprozesse automatisieren mit BPEL" und Maintainer des BPELUnit-Projekts, das Entwickler beim Testen von BPEL- und BPMN-Prozessen unterstützt.

daniel.luebke@innoq.com

Martin Heinrich, eidg. dipl. Wirtschaftsinformatiker, ist mit seiner eigenen Firma Marisma GmbH als Senior Enterprise Architect tätig. In enger Zusammenarbeit mit der innoQ Schweiz GmbH realisiert er SOA-und MDA-Projekte. Zudem zeichnet er sich als Richtungsverantwortlicher für die Prüfungen des eidg. Fachausweises „ICT-Applikationsentwickler" verantwortlich.

mhe@marisma.com

Links & Literatur

[1] Apache Maven: http://maven.apache.org/, Stand: 25.11.2012

[2] Repository zum Artikel: https://github.com/dluebke/article-prozesse-vom-fliessband, 25.11.2012

[3] Maven AntRun Plug-in: http://maven.apache.org/plugins/maven-antrun-plugin/, Stand: 25.11.2012

[4] Build Helper Maven Plug-in: http://mojo.codehaus.org/build-helper-maven-plugin/, 25.11.2012

[5] Daniel Lübke, Tammo van Lessen: „Testobjekt: Geschäftsprozess. Geschäftsagilität durch Tests sichern", in Java Magazin 5.2012

Anzeige

Anzeige

NEUE SERIE

OAuth 2.0: die Clientseite

OAuth, die Zweite

Facebook, Google, Foursquare oder Pinterest haben eines gemeinsam: Die APIs dieser Dienste setzen allesamt auf OAuth 2.0. OAuth 2.0 ist ein Protokoll zur Autorisierung von API-Zugriffen, beispielsweise durch server- oder clientseitige Webanwendungen oder mobile Apps. Trotz des spektakulären Rückzugs von OAuth-2.0-Editor Eran Hammer werden wohl auch in Zukunft mehr und mehr APIs auf dieses Protokoll setzen: OAuth 2.0 hat sich bereits bei den großen Services (Google, Facebook) etabliert und ist im Vergleich zu OAuth1 viel einfacher zu benutzen. Wir betrachten zunächst den wichtigsten OAuth 2.0 Grant, den Authorization Code Grant.

von Sven Haiges

jaxSpeaker

Zugegeben: Der Blog Post, mit dem sich Eran Hammer von der OAuth-2.0-Spezifikation verabschiedet hat [1], war für viele schockierend zu lesen. Als Hauptgrund für seinen Rückzug gibt er „Indecision Making“ an, also das Fehlen von Entscheidungen. Die OAuth-2.0-Spezifikation sei wegen der Mitwirkung zu vieler Parteien zu offen, zu viele Bereiche seien absichtlich ungenau gehalten und diverse Implementierungsmöglichkeiten offen zu halten. Viele dieser Gründe sind im Kern sicherlich richtig. Dennoch bewegt sich die Welt der APIs mit großen Schritten in Richtung OAuth 2.0. OAuth 2.0 ist ein sicheres und gut durchdachtes Protokoll, auch wenn manche sich vielleicht Details besser spezifiziert gewünscht hätten. Und auch wenn die APIs von Google und Facebook derzeit und vielleicht auch für immer

Artikelserie

1. **Authorization Code Grant**
2. Weitere OAuth 2.0 Grants
3. Implementierung eines OAuth-2.0-Servers anhand des OAuth-2.0-Moduls von Spring Security

nicht hundertprozentig standardkonform sein werden: OAuth 2.0 – in leicht unterschiedlichen Ausprägungen – ist bereits überall im Einsatz. In diesem Artikel möchte ich nicht allzu lange philosophieren. Fakt ist, dass Kenntnisse über OAuth 2.0 relevanter sind denn je. Egal ob Mobile, klassischer Webclient oder moderne In-Browser-JavaScript/HTML5-App – um den Clientcode zu autorisieren und damit mit Zustimmung des Anwenders auf seine Ressourcen zuzugreifen, benutzt man OAuth 2.0. Und auch wenn die verschiedenen OAuth-2.0-kompatiblen APIs noch leichte Unterschiede aufweisen, so muss man OAuth 2.0 nur einmal aus der Vogelperspektive verstanden haben.

Der erste Teil dieser dreiteiligen OAuth-2.0-Serie geht auf die Sicht der OAuth-2.0-Clients ein und stellt Ihnen besonders den allgegenwärtigen OAuth 2.0 Authorization Code Grant vor. Im Laufe dieser Serie werden zahlreiche Codebeispiele die praktische Anwendung aus der Sicht des Clients und Servers beleuchten. Die Codebeispiele für die Benutzung aus Sicht des Clients sind dabei einer Gaelyk-Webapplikation entnommen, und meistens handelt es sich um Code aus Gaelyk-Controllern. Weitere Informationen zu Gaelyk, einem schlanken Webframework für Google App Engine, finden Sie unter [2]. Neben den OAuth-2.0-„Flows“ erhalten Sie auch Informationen dazu, welcher Flow für welches Anwendungsszenario am besten geeignet ist.

Rollen

Um OAuth 2.0 besser verstehen zu können, sollte man die Rollen dieses Autorisierungsprotokolls verstanden haben. Es gibt den Resource Owner, den Resource Server, den Client und den Authorization Server.

- Der **Resource Owner** ist oftmals der Endanwender, beispielsweise ein Benutzer, der auf seine in der Cloud gespeicherten Bilder zugreifen möchte.
- Der **Resource Server** ist ein Server, auf dem die Daten/Dienste des Resource Owners vorliegen. Diese Ressourcen sind dadurch geschützt, dass der Resource Server ein so genanntes *Access Token* verlangt. Nur wenn das Access Token validiert werden konnte, genehmigt der Resource Server den Zugriff.
- Der **Client** ist eine Applikation, die für den Resource Owner auf die Daten/Dienste zugreift. Um auf diese zugreifen zu können, muss er im Besitz eines Access Tokens sein.
- Der **Authorization Server** vergibt die Access Tokens, die den Zugriff auf den Resource Server ermöglichen, nach erfolgreicher Authentifizierung des Resource Owners und dessen Autorisierung für den entsprechenden Client.

Der Client stellt also für den Resource Owner Anfragen (HTTP-Requests) an den Resource Server. Damit der Client auf die Daten des Resource Owners zugreifen kann, muss dieser in der Anfrage ein Access Token mitschicken. Das Access Token autorisiert den Client, auf diese Daten zuzugreifen. Der Resource Server kommuniziert dann mit dem Authorization Server, um sicherzustellen, dass das gesendete Access Token valide ist.

Der Authorization Server und der Resource Server sind oftmals das gleiche System. Der Check der Access Tokens geschieht also über interne Methodenaufrufe. Bei größeren APIs kann der Authorization Server auf einem getrennten System laufen. Mit welchem Protokoll dann die Resource Server auf den Authorization Server zugreifen, ist allerdings implementierungsspezifisch. Die OAuth-2.0-Spezifikation hat diesen Teil nicht näher beschrieben.

OAuth 2.0 aus der Vogelperspektive

OAuth 2.0 definiert mehrere so genannte *Protocol Flows*. Diese Flows sind notwendig, da unterschiedliche Clients unterschiedliche Anforderungen und Möglichkeiten haben. Beispielsweise kann eine serverseitige Webapplikation einen gemeinsamen Schlüssel geheim halten – der Schlüssel ist Teil des Codes auf dem Server und kann normalweise nicht einfach ausgelesen werden. Anders verhält es sich bei den immer populärer werdenden clientseitigen HTML5-Applikationen, die diverse APIs direkt per JavaScript aus dem Browser heraus aufrufen. Der JavaScript-Code kann sehr einfach eingesehen werden. Somit sollte auch kein gemeinsamer Schlüssel darin abgelegt werden. Die vier Flows haben Folgendes gemeinsam:

- Zuerst wird immer ein Authorization Grant eingeholt. Der Authorization Grant ist die Zustimmung des Resource Owners, dass ein Client auf die Daten des Resource Owners zugreifen kann. Dieser Grant kann ein Code, also ein Stück Text sein, den der Client nach einer erfolgreichen Authentifizierung und Autorisierung des Resource Owners durch den Authorization Server geschickt bekommt. Er kann aber auch implizit dadurch gegeben werden (siehe Resource Owner Password Flow), dass der Resource Owner dem Client Username und Passwort übergibt.
- Im zweiten Schritt wird der Grant durch ein Access Token eingetauscht. Damit hat der Client nun die Möglichkeit, auf den Resource Server im Namen des Resource Owners zuzugreifen.
- Der Zugriff erfolgt letztlich mit dem Access Token. Um das Access Token nicht anderen Servern im Internet zu offenbaren, setzt OAuth 2.0 auf sichere Verbindungen per HTTPS-Protokoll.

Im einfachsten Fall können die ersten beiden Schritte (Authorization Grant besorgen und in ein Access Token eintauschen) in einem HTTP-Request/Response-Paar erledigt werden. Das ist beispielsweise beim Resource Owner Password Credentials Flow der Fall.

Das Access Token

Um ein API, das per OAuth 2.0 geschützt ist, zu nutzen, muss der Client durch die Zustimmung des Nut-

zers (Resource Owners) vom Authorization Server ein Access Token erlangen. Ein Access Token ist ein Stück Text, der typischerweise im HTTP Authorization Header (Authorization: Bearer <*token*>) geschickt wird. Facebook und andere Services erlauben es jedoch auch, das Access Token als URL-Parameter anzuhängen (*access_token*-Parameter). Dadurch wird der Zugriff auf Ressourcen, die per HTTP-GET-Methode erreichbar sind, direkt aus dem Browser heraus gerade für kurze Tests vereinfacht. Die Kommunikation sollte – egal, ob das Token in den URL-Parametern oder im Header transportiert wird – immer per HTTPS-Protokoll verschlüsselt sein.

Typischerweise haben die OAuth 2.0 Access Tokens eine bestimmte Lebensdauer, gemessen in Sekunden seit der Ausstellung des Tokens. Dem Client wird diese Gültigkeit zusammen mit dem Access Token mitgeteilt. Somit hat der Client bei jedem Request die Möglichkeit, schon davor zu überprüfen, ob ein Access Token noch gültig ist. Je nach Flow kann ein abgelaufenes Access Token dann durch ein Refresh Token in ein neues, frisches Access Token umgetauscht werden. Auch wie ein abgelaufenes Access Token in ein neues getauscht wird, soll gezeigt werden.

Die Endpoints

Um ein Token zu bekommen, muss man natürlich wissen, wo es sie gibt. Und wohin muss man den Anwender überhaupt schicken, um den Client für den Resource Owner zu autorisieren? Diese beiden „URLs“ sind die Token und Authorization Endpoints:

- Der Authorization Endpoint identifiziert den Resource Owner und erlaubt ihm dann, der Autorisierungsanfrage des Clients zuzustimmen. Typischerweise schickt der Client den Anwender per HTTP 302 Redirect zum Authorization Endpoint, also einer Webseite, die vom Authorization Server verwaltet und geschützt wird. Dort muss sich der Anwender einloggen (meist ganz normal, also per Basic Authentication – und hoffentlich per HTTPS) und dann den Client autorisieren (z. B. durch einen Klick auf einen Button). Sobald dies passiert ist, erhält der Client einen Authorization Grant. Im Falle des Authorization Code Flows erhält er diesen über einen erneuten Redirect zurück zur Clientwebsite. Der Grant wird in diesem Fall als Parameter an den Redirect-Link angehängt.
- Der Token Endpoint händigt die Access Tokens aus und benötigt dazu einen Grant. Diese Zustimmung ist im Falle des bekannten Authorization Code Flows ein Stück Text, ähnlich wie das Access Token selbst. Es kann jedoch auch ein implizierter Grant sein, wie beispielsweise der echte Username und das Passwort des Anwenders.

Wie Sie nun sicherlich bemerkt haben, gibt es unterschiedliche Wege zum Ziel. Damit der Token Endpoint das begehrte Access Token aushändigt, benötigen wir einen Grant. Dieser kann auf vier verschiedenen Wegen erlangt werden.

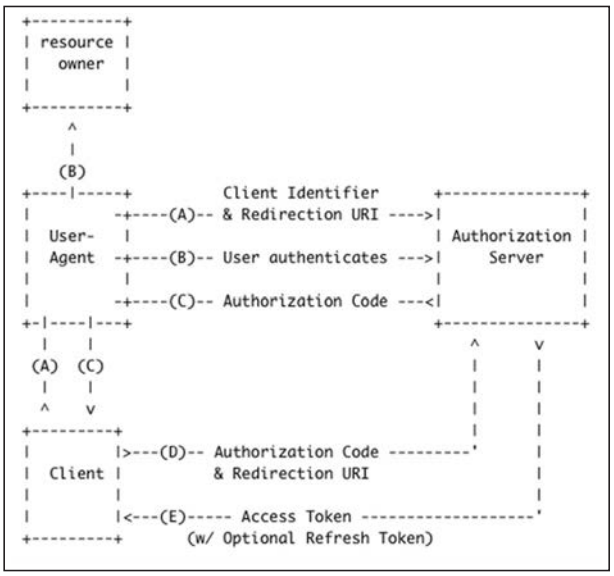


Abb. 1: OAuth 2.0 Flow mit dem Authorization Code Grant, Quelle: OAuth 2.0 Spec [3]

Client ID und Client Secret

Damit ein Client für den Resource Owner auf dessen Daten zugreifen kann, muss der Client beim Authorization Server registriert sein. Öffentliche APIs wie Facebook oder Google bieten dazu spezielle Registrierungsseiten an. Letztendlich erhält der Entwickler jedoch für seinen Client eine ID und ein Passwort (*client_id* und *client_secret*). Das Passwort, das Client-Secret, darf natürlich nicht in fremde Hände gelangen. Je nachdem, welcher Flow später zum Einsatz kommt, muss man während der Registrierung einen Redirect-URL angeben. Dieser ist für den Authorization Code und Implicit Flow unerlässlich, da durch den Redirektions-Mechanismus entweder der Authorization Code oder direkt ein kurze Zeit gültiges Access Token an den Client übermittelt wird.

Authorization Code Grant

Kurze Wiederholung: Der „Grant“ ist die Zustimmung des Anwenders. Es gibt unterschiedliche Arten, wie diese Zustimmung eingeholt werden kann. Die bekannteste Art ist dabei der Authorization Code Grant. Oftmals wird auch vom Authorization Code Flow gesprochen – gleichbedeutend mit „OAuth 2.0 unter Erlangung des Grants durch den Authorization Code“. Flow bezeichnet damit das Wechselspiel aus Request und Response. Die schematische Darstellung ist in **Abbildung 1** zu sehen.

Um den Authorization Code – die Zustimmung des Anwenders – zu bekommen, wird dieser über einen Browser-Redirect zum Authorization Endpoint geschickt. Dort muss sich der Anwender einloggen, falls er nicht bereits eingeloggt ist. Er muss dann zustimmen,

Um OAuth 2.0 besser verstehen zu können, sollte man die Rollen dieses Autorisierungsprotokolls verstanden haben. Insgesamt gibt es vier verschiedene Rollen.

dass der anfragende Client auf die Daten zugreifen darf. Sobald diese Zustimmung gegeben wurde, wird der Browser des Anwenders erneut über einen Redirect zurück zur ursprünglichen Webapplikation gebracht. Am URL des erneuten Redirects kann aber nun der anfragende Client den Authorization Code auslesen (als Parameter). Er nutzt den Code, um ein Access Token vom Token Endpoint zu erlangen. Diese zweite Anfrage geschieht im Hintergrund und normalerweise ohne dass dies der Anwender mitbekommt.

Gehen wir diesen Flow Schritt für Schritt durch: Zu Beginn wird der Anwender per Browser-Redirect zum Authorization Endpoint geschickt (A). Der Client weist also durch einen HTTP 302 Redirect den User-Agent (Browser) an, die Webseite des Authorization Servers aufzurufen (typischerweise */oauth/authorize*). Bei diesem Aufruf müssen auch ein paar Parameter mitgeschickt werden. Dank HTTPS ist die Verbindung verschlüsselt und kann damit also nur vom Authorization Server entschlüsselt werden. Folgende URL-Parameter werden mitgeschickt:

- *response_type*: Der Wert dieses Parameters muss „code“ sein, da wir dem Authorization Endpoint hiermit anzeigen, dass wir den Authorization Code Grant benutzen.
- *client_id*: Bei der Registrierung des Clients wurde die *client_id* vom OAuth-2.0-Provider festgelegt. In unseren Beispielen benutzen wir die *client_id 'mobile_android'*.
- *client_secret*: Ebenso wie die *client_id* wurde der Wert des Parameters *client_secret* bei der Registrierung vom Provider festgelegt. In unseren Beispielen benutzen wir ein *client_secret* mit dem Wert „secret“.
- *redirect_uri*: Dies ist typischerweise ein URL, zu dem nach einer erfolgreichen Autorisierung des Anwenders der Browser erneut umgeleitet wird. An diesen Redirect URI wird später der Authorization Code als URL-Parameter angehängt. Der *redirect_uri* muss mit dem bei der Registrierung beim OAuth-2.0-Provider angegebenen Wert übereinstimmen, ansonsten wird der Authorization Endpoint die Anfrage mit einem Fehler beantworten.
- *scopes*: eine durch Leerzeichen getrennte Liste der Bereiche, zu denen ein Client Zugriff haben will. Das Facebook-OAuth-2.0-API nutzt leider keine Leerzeichen, sondern trennt die Scopes per Komma. Die *scopes* geben bei Facebook beispielsweise an, auf welche Bereiche ein Client mit dem Access Token später zugreifen kann: *user_likes*, *user_checkins*, *friends_likes* etc.
- *state*: Ein optionaler Parameter, der aber die Sicherheit des Protokolls weiter erhöht. Als *state* sollte ein Pseudozufallswert gesetzt werden, der später beispielsweise durch Abrufen aus der HTTP-Session des Users verglichen werden kann. Durch Nutzung des *state*-Parameters können effektiv CSRF-Attacken (Cross-site Request Forgery [4]) vermieden werden.

Listing 1

```
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'https://<my_server>/oauth2_callback'

def scopes = [
  'customer'
]

def state = new Random(System.currentTimeMillis()).nextInt().toString()
request.getSession(true).setAttribute('state', state)

redirect "https://<server>/oauth/authorize?client_id=${client_id}&redirect_uri=
                ${URLEncoder.encode(redirect_uri, 'UTF-8')}&response_type=code&scope=
        ${URLEncoder.encode(scopes.join(' '), 'UTF-8')}&state=${URLEncoder.encode(state,
                                                                'UTF-8')}"
```

Der OAuth-2.0-Neuling ist zunächst vielleicht etwas von der Fülle dieser Parameter überwältigt, aber in (Groovy-)Code aus einem Gaelyk-Controller (Listing 1) sieht dieser Redirect nicht mehr so bedrohlich aus.

Der Anwender (Resource Owner) loggt sich nun auf der Authorization Page des Authorization Servers ein und autorisiert dann den Client, auf die angefragten Scopes zugreifen zu dürfen (B). Das Login selbst, die Authentifizierung, kann beispielsweise per Basic Authentication erfolgen. Die Zustimmung wird danach meist über den Click auf einen Link oder Button eingeholt: „Möchten Sie Client *mobile_android* Zugriff auf folgende Daten geben: *scopeA*, *scopeB*...?“

Nach der Zustimmung des Anwenders wird der Browser erneut per Redirect zurück zur ursprünglichen Webapplikation des Clients geleitet (C). An den *redirect_uri* werden jedoch zwei wichtige Parameter vom Authorization Server angehängt:

- *code*: Dies ist der Authorization Code, der vom Client zur Erlangung eines Access Tokens später verwendet wird.
- *state*: Diesen Parameter sollte der Client mit dem in der Session vorhandenen Wert vergleichen, um CSRF-Attacken zu verhindern.

Der Client tauscht den Authorization Code nun in einem Request zum Token Endpoint gegen ein Access Token aus (D). Betrachten Sie Listing 2.

Auch bei dieser Anfrage müssen wieder einige Parameter mitgeschickt werden:

- *grant_type*: Dem Token Endpoint wird hiermit mitgeteilt, dass wir einen Authorization Code gegen ein Access Token tauschen möchten.
- *code*: Natürlich müssen wir den Authorization Code mitschicken, der ja unser Grant ist.
- *redirect_uri*: Etwas unerwartet müssen wir auch in diesem Request den *redirect_uri* mitschicken – dieser wird allerdings nicht für einen Browser-Redirect verwendet, sondern wird vom Token Endpoint nur erneut mit dem für die *client_id* hinterlegten Wert verglichen.
- *client_id* und *client_id* werden erneut zur Überprüfung des Clients mitgeschickt.

In unserem Codebeispiel geben wir die Antwort des Token Endpoints dann einfach aus. Wie genau diese Antwort mit dem Access Token aussieht, sehen wir gleich (E). Geschafft. Wir haben nun die Antwort des Token Endpoints, ein laut OAuth-2.0-Spezifikation JSON-formatierter String, der u. a. das Access Token beinhaltet:

```
{"access_token":"865b3ecf-54d3-413c-951e-fae0b00c906f",
        "token_type":"bearer","refresh_token":"addc590a-c79c-4e7e-a1a7-
                                        f98921765c20","expires_in":43192}
```

Leider liefern derzeit nicht alle Token Endpoints JSON zurück. Facebook gibt diese Parameter beispielsweise als URL-Encoded String zurück. Es gibt also hier und da noch kleine Unterschiede, das Parsing der Rückgabe ist jedoch meist recht einfach.

Neben dem Access Token bekommen wir auch die Info, dass das Access Token als Bearer Token zu verwenden ist. Dies gibt dem Client einen Hinweis, wie das Token in den Requests zu verwenden ist: nämlich als HTTP Header im Request. Hier ein Beispiel:

```
def con = "https://<server>/rest/v1/electronics/customers/current".toURL().
                                                        openConnection()
con.setRequestProperty("Authorization", "Bearer 865b3ecf-54d3-413c-951e-
                                                        fae0b00c906f")
println con.inputStream.text
```

Mittels *expires_in* kann der Client selbst herausfinden, ob ein gespeichertes Token noch gültig ist. Sollte das Token abgelaufen sein, so kann er mittels des *refresh_tokens* (welches meist deutlich länger gültig ist) ein neues Access Token anfordern. Dazu später noch mehr.

Hervorzuheben ist an dieser Stelle, dass sich der Authorization Code Flow besonders für serverseitige Webapplikationen eignet. Dies liegt darin begründet, dass das *client_secret* sicher im Code bzw. in der Konfiguration der Webapplikation abgelegt werden kann. Für den Anwender stellt die Umleitung im Browser zum Authorization Endpoint ebenso kein Hindernis dar.

Listing 2

```
import com.google.appengine.api.urlfetch.*
import groovy.json.*
import java.net.URLEncoder

def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'https://<my_server>/oauth2_callback'

if (!params.code)
{
  out << "User denied access."
  return
}

if (!params.state)
{
  out << "State parameter missing, something is wrong..."
  return
}

if (!session)
{
  out << "No session, how weird!"
  return

}

if (params.state != session.state)
{
  out << "State does not match! (${params.state} != ${session.state})"
  return
}

def code = params.code

//exchange code for real oauth token
URL tokenURL = "https://<server>/rest/oauth/token".toURL()
HTTPResponse res = tokenURL.post(deadline: 30, payload:"code=${code}&client_
            id=${client_id}&client_secret=${client_secret}&redirect_uri=${URLEncoder.
             encode(redirect_uri, 'UTF-8')}&grant_type=authorization_code".getBytes())

out << res.text
```

Aus alt mach' neu: Refresh Tokens

Zwei der vier OAuth 2.0 Flows sehen vor, dass der Authorization Server Refresh Tokens an den Client liefert: Dies sind der Authorization Code Flow sowie der Resource Owner Password Credentials Flow (wird im 2. Teil der OAuth-2.0-Serie vorgestellt). Der Client kann damit ein abgelaufenes Access Token in ein neues eintauschen. Zwingend vorgeschrieben ist das nicht, aber gerade beim Resource Owner Password Grant, der gerne von offiziellen mobilen Clients benutzt wird, bringt die Benutzung von Refresh Tokens den Vorteil, dass der Client nicht erneut nach dem Usernamen und Passwort des Anwenders fragen muss. Anhand der Token Response kann der Client feststellen, wie lange ein Access Token gültig ist:

```
{ ... "expires_in": 42381 }
```

Der Client kann also pro-aktiv handeln und ein abgelaufenes Access Token vor einer wegen mangelnder Gültigkeit des Access Tokens fehlschlagenden Anfrage in ein neues Access Token umtauschen. Die Gültigkeit desselben wird durch den Key *expires_in* in Sekunden angegeben. Sobald die Gültigkeit abgelaufen ist, kann der Client versuchen, mittels des Refresh Token ein neues Access Token zu bekommen. Wie lange das Refresh Token gültig ist, ist von Server zu Server unterschiedlich. Betrachten wir abschließend den Token Refresh Flow anhand eines Beispiels (Listing 3).

Diese Anfrage wird wieder an den Token Endpoint geschickt. Der Endpoint weiß, dass der Client ein Refresh Token in ein neues Access Token (und Refresh Token) umtauschen will, da der *grant_type*-Parameter auf „refresh_token" gesetzt wird. Weitere Parameter sind:

- *client_id* und *client_secret*, um den Client zu authentifizieren
- *redirect_uri*: Meist wird vom Authorization Server auch der hinterlegte Redirect URI verlangt, der vom Server zusätzlich auf Übereinstimmung geprüft wird.
- Das Refresh Token durch den *refresh_token*-Parameter.

Listing 3

```
def client_id = 'mobile_android'
def client_secret = 'secret'
def redirect_uri = 'http://<my_server>/oauth2_callback'

def token = params.token

URL tokenURL = "https://<server>/oauth/token".toURL()
HTTPResponse res = tokenURL.post(deadline: 30, payload:"refresh_token=
        ${token}&client_id=${client_id}&client_secret=${client_secret}&redirect_uri=
    ${URLEncoder.encode(redirect_uri, 'UTF-8')}&grant_type=refresh_token".getBytes())

out << res.text
```

Die Response ist erneut ein JSON-formatierter String mit den Keys für das Access Token, Expiry, Refresh Token und Token Type.

Zusammenfassung und Ausblick

In diesem Teil der Serie haben wir OAuth 2.0 aus Sicht des Clients betrachtet und den Authorization Code Grant vorgestellt. Dies ist der wichtigste Grant, der u. a. von Google, Facebook oder Pinterest benutzt wird. Hierbei wird der Anwender zum Authorization Endpoint per Browser-Redirect geschickt, loggt sich dort ein und autorisiert den Client. Ein erneuter Redirect zurück zur Webapplikation des Clients liefert den Authorization Code, der vom Client in einem zweiten und für den Anwender unsichtbaren Request in das Access Token eingetauscht wird. Dieser Flow ist vor allem für serverseitige Webapplikationen geeignet, da das *client_secret* im serverseitigen Code sicher untergebracht ist.

Im zweiten Teil der Serie betrachten wir die drei weiteren OAuth 2.0 Grants. Diese weichen von dem vorgestellten Authorization Code Grant teils stark ab und richten sich an clientseitige Webapplikationen (also JavaScript-Clients), mobile Clients sowie beliebige Clientapplikationen, die auf eigene Ressourcen (nicht die Ressourcen eines Resource Owners) zugreifen wollen. Im dritten und letzten Teil betrachten wir dann die Implementierung eines OAuth-2.0-Servers anhand des OAuth-2.0-Moduls von Spring Security.

Sven Haiges arbeitet als Technology Strategist bei der Hybris GmbH in München. Neben Groovy & Grails beschäftigt er sich dort derzeit auch mit HTML5 und Android. Sven lebt mit seiner Familie in München. Ihm kann gerne unter @hansamann auf Twitter gefolgt werden.

Links & Literatur

[1] http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/

[2] http://gaelyk.appspot.com/

[3] http://tools.ietf.org/html/draft-ietf-oauth-v2-31

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

Anwendungen für die Google App Engine entwickeln

In Google we trust?

Eine kleine Idee für eine Applikation im Kopf, die Entwicklungsumgebung starten, die Idee umsetzen, die Applikation deployen und zusehen, wie die Userzahl wächst – geht das? An einem Tag? Sicher nicht, wenn man sich zunächst damit auseinandersetzen muss, wo man die App installiert, welche Zielinfrastruktur bzw. Zielplattform man zur Verfügung hat und welche Kosten anfallen – egal, ob die Anwendung ein Hit wird oder ein Flop. Möglich wäre es aber, wenn man sich nur auf den Inhalt konzentrieren müsste. Und Werkzeuge hätte, die gut in die bevorzugte Entwicklungsumgebung integriert sind. Kann dies die Google App Engine leisten?

von Michael Seemann

Angefangen hat die App Engine einmal mit Python, da diese Sprache offensichtlich von Google intern verwendet wurde, um etwa die Applikationen für das Werbesystem (AdWords und AdSense) zu erstellen. Etwa ein Jahr nach der Veröffentlichung kam dann im April 2009 Java als unterstützte Sprache hinzu. Die Unterstützung von Java dürfte der Unternehmensstrategie von Google geschuldet gewesen sein, auf eine weit verbreitete, typisierte Sprache zu setzen. Das schlug sich dann auch im Google Web Toolkit nieder und wird nicht zuletzt beim Einsatz von Java für die Android-Entwicklung sichtbar. Zu jener Zeit war noch nicht absehbar, dass Oracle Sun übernimmt und sich damit ein neues Feld von Patentstreitigkeiten ergibt.

Als letzte unterstützte Sprache kam Googles Go [1] hinzu. Diese Sprache ist besonders für Cluster und Cloud-Computing-Systeme optimiert – also ein idealer Kandidat für eine Plattform wie die Google App Engine. In diesem Artikel werden wir jedoch ausschließlich Java betrachten. Ein leichtgewichtiges Groovy Toolkit für die Google App Engine gibt es ebenfalls: Gaelyk [2].

Die Google App Engine ist momentan als Public-Cloud-System positioniert. Alle Applikationen werden in der Infrastruktur von Google betrieben. Dazu wird entweder eine App unter der Domain appspot.com eingerichtet oder man verknüpft die neu zu erstellende App mit einer Google Apps Domain [3]. Besonders in den USA wird dieser Google-Dienst häufig eingesetzt, da er eine sehr günstige Alternative zu einer eigenen Groupware-Lösung bietet. In Deutschland herrscht in dieser Hinsicht deutlich mehr Zurückhaltung, die nicht zuletzt auf Datenschutz- und Datensicherheitsaspekte zurückzuführen ist. Nicht zuletzt deshalb würde man sich wünschen, dass die GAE auch als Private Cloud verfügbar ist. Eine direkte Unterstützung von Google gibt es in dieser Hinsicht derzeit jedoch nicht. Allerdings versucht beispielsweise das Open-Source-Projekt *appscale* [4] eine Laufzeitumgebung für GAE-Applikationen zur Verfügung zu stellen, in die man seine Apps deployen kann. Das Projekt wird aktiv weiterentwickelt, stößt aber natürlich auch an seine Grenzen: Zum einen hängt es den Google APIs prinzipbedingt immer etwas hinterher, und zum anderen sind nicht alle Services verfügbar, die die GAE zur Verfügung stellt, da es sich häufig um Dienste handelt, die ausschließlich von Google betrieben und zur Verfügung gestellt werden (z. B. Google Cloud Storage). Für diese müssen dann jeweils vergleichbare Dienste in der *appscale*-Plattform umgesetzt werden – oder sie bleiben außen vor und sind nicht verfügbar.

Insgesamt sind die Hürden für das Aufsetzen einer neuen App aber denkbar niedrig. Im Prinzip kann jeder bis zu zehn Apps auf der Plattform kostenlos betreiben. Erst, wenn die kostenlosen Quoten überschritten werden, muss man das Abrechnungsmodul aktivieren. Dann zahlt man für den tatsächlichen Verbrauch an Speicherplatz (derzeit 0,24 US-Dollar pro GB und Monat), CPU-Zeit (0,08 US-Dollar pro Stunde), ausgehendes Datentransfervolumen (0,12 US-Dollar pro GB) und Aufrufe der Service-APIs [5]. Ein GB Speicher und etwa fünf Millionen Seitenaufrufe pro Monat sind in den kostenlosen Quoten enthalten. Werden diese Grenzen überschritten, werden Anfragen an die Applikation von der App Engine nicht mehr bearbeitet. Für einen Versuchsballon ist das jedoch allemal ausreichend, und Miniapplikationen ohne großen Datenspeicher kommen mit diesen Grenzen sehr gut zu recht.

Wie sieht es mit der Verfügbarkeit der App Engine aus? An dieser Stelle ist man mit seiner App auf Gedeih und Verderb Google ausgeliefert. Gibt es Ausfälle oder

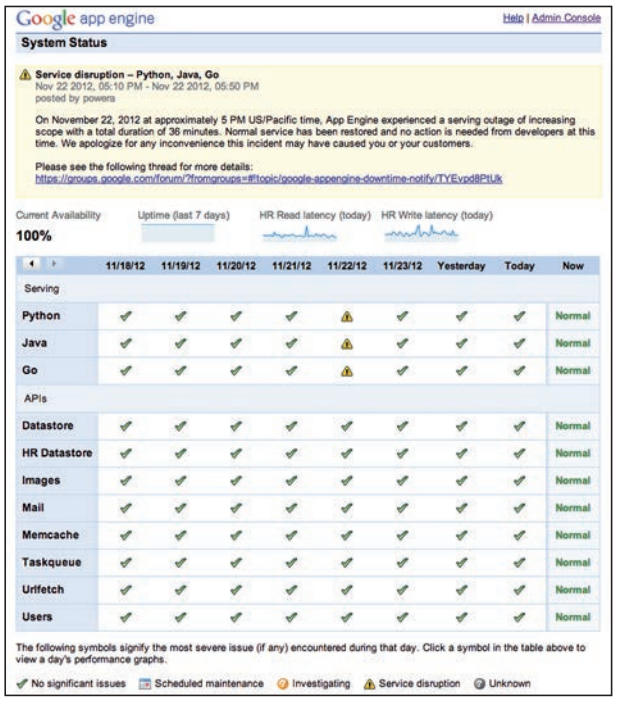


Abb. 1: GAE-Systemstatus

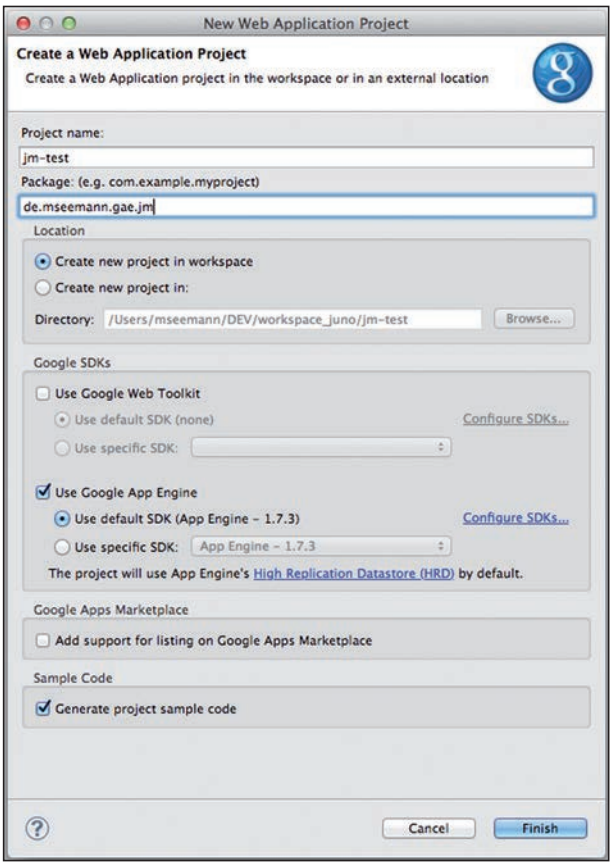


Abb. 2: GAE-Projekt erstellen

sind Wartungsarbeiten erforderlich, steht die eigene App nicht zur Verfügung, und es gibt auch keine Möglichkeit, mal eben schnell den Anbieter zu wechseln. Über den Status kann man sich unter [6] informieren. Bei Problemen erhält man dann nur ein „Wir bitten um Entschuldigung“ (**Abb. 1**). Derartige Fälle treten aber nicht allzu häufig auf.

Entwicklung für die GAE

Wenn es darum geht, Applikationen für die GAE zu erstellen, hat Google im Lauf der über vierjährigen Entwicklungszeit einiges getan, um den Einstieg so einfach wie möglich zu machen. Das SDK enthält zwar für alle erdenklichen Aufgaben – im Zusammenhang mit der Google App Engine – Kommandozeilenwerkzeuge; Google stellt aber auch einen umfangreichen Satz an Plug-ins für Eclipse zur Verfügung. Hat man die Plug-ins installiert, kann man sofort mit seiner ersten App beginnen. Dazu legt man ein neues Web Application Project an und aktiviert die Option Use Google App Engine (**Abb. 2**).

Nachdem das Projekt angelegt wurde, führt man Run As Web Application aus und kann seine erste App für die GAE unter http://localhost:8888/ bewundern. Zugegeben, die Ausgabe von *Hello App Engine!* reißt niemanden vom Hocker, aber man kann sich dem grundsätzlichen Aufbau einer App-Engine-Applikation leicht nähern. Es handelt sich nämlich um eine typische Webapplikation für einen Servlet-Container. In der *web.xml* ist ein Servlet konfiguriert und mit einem URL-Pattern verknüpft. In der generierten *index.html* ist ein Link auf dieses Servlet gesetzt. Dieses Servlet tut ebenfalls nicht viel: Es gibt lediglich *Hello World!* aus. Die Applikation muss übrigens mindestens unter Java 1.5 kompiliert werden. Die App Engine selbst führt die Applikation unter Java 1.6 aus.

Um diese App in der GAE zu deployen, muss man unter [7] eine Applikation einrichten. Dazu legt man zunächst eine *ID* für die Applikation fest. Für dieses Beispiel verwenden wir *jm-artikel*. Damit steht die Applikation später unter dem URL jm-artikel.appspot.com zur Verfügung. Die *ID* muss also als Subdomain für appspot.com noch verfügbar sein. Das prüft die GAE für uns. Dann gibt man der Applikation noch einen Titel und klickt auf den Button Create Application (alle anderen Einstellungen belassen wir in der Standardeinstellung). Die GAE erstellt eine leere Hülle für die Applikation. Jetzt muss unsere erste App nur noch deployt werden.

Wieder zurück in Eclipse wird die *Application ID* in den Projekteinstellungen hinterlegt. Sollte das beim Deployen noch nicht der Fall sein, fordert das Plug-in dazu auf. Anschließend erreicht man über das Kontextmenü des Projekts den Menüpunkt Goolge | Deploy to App Engine und klickt in dem dann erscheinenden Dialog auf Deploy. Nach wenigen Sekunden ist die Applikation in die GAE übertragen und steht dort unter dem URL http://jm-artikel.appspot.com zur Verfügung – überzeugen Sie sich selbst! Das Ganze hat nicht einmal zehn Minuten gedauert.

Der lokal gestartete Webserver bildet übrigens in weiten Teilen die Laufzeitumgebung der App Engine nach. Auch hier ist der *DataStore* verfügbar, die User-Services werden emuliert, und auch das Sandbox-Verhalten wird

nachgebildet. Nicht möglich ist bspw. der Versand von E-Mails, und wenn man *Google Cloud SQL* verwenden möchte, muss man sich um das Aufsetzen einer lokalen MySQL-Datenbank selbst kümmern.

Am wichtigsten ist aber sicher das nachgebildete Sandbox-Verhalten. Die Applikationen laufen nämlich auf der GAE in einer Sandbox, in der keine Schreibzugriffe auf das Dateisystem möglich sind. Auch lassen sich keine Sockets öffnen. Für Zugriffe über HTTP muss man den *URL Fetch* Service benutzen. Auch JNI-Aufrufe stehen nicht zur Verfügung. Nicht alle Klassen des JRE werden unterstützt – eine Liste der verwendbaren Klassen kann man unter [8] einsehen. Dadurch gibt es auch Probleme mit einigen Frameworks wie ICEfaces oder Seam [9].

Auch in Bezug auf die Performance einer Applikation gibt es Beschränkungen. So muss jeder Request innerhalb von 60 Sekunden bearbeitet werden, wenn die App auf einer so genannten Frontend-Instanz läuft. Seit einiger Zeit es auch möglich, Code auf Backend-Instanzen auszuführen, für die diese Beschränkungen nicht gelten. Des Weiteren gibt es Beschränkungen, wie groß übermittelte Datenmengen sein dürfen – ein Response darf beispielsweise nicht größer als 32 MB sein, eine Entität im *DataStore* darf nicht größer als 1 MB sein und nicht mehr als 5000 Werte haben. Alle diese Beschränkungen muss man kennen, um die Funktionsweise der Apps darauf abzustimmen. Da ist es ein erheblicher Vorteil, wenn man diese Beschränkungen auch lokal testen kann. Abschrecken lassen sollte man sich von diesen Beschränkungen aber auch nicht. Sie sind dem Prinzip der GAE geschuldet: maximale Verteilung der App für sehr große Nutzerzahlen. Dazu müssen die Daten gut verteilbar sein, und es muss sehr einfach sein, neue Prozesse zu starten, in denen die Applikationen laufen können. Schließlich ist dieser Prozess vollautomatisch und richtet sich nach der Last auf der Applikation.

Der DataStore der GAE

Eine Besonderheit ist der Datenspeicher der Google App Engine. Sie verfügt nicht über eine herkömmliche relationale Datenbank, auch wenn Google mit *Google Cloud SQL* inzwischen eine solche Lösung anbietet. Diese ist momentan aber noch nicht für den produktiven Einsatz gedacht und lässt sich auch nicht ohne Bereitstellung eines Datenbankschemas betreiben.

Kriterium	Eigenschaft
IDE-Plug-in	Die Entwicklung und Administration von Anwendungen in der Google App Engine wird nahezu vollständig durch mehrere Eclipse-Plug-ins unterstützt.
Kommandozeilenwerkzeug	Ja. Zur Administration der Anwendungen in der App Engine und in der lokalen Entwicklungsumgebung.
Build-Tool-Support	Ant-Skripte müssen manuell erstellt werden.
Public/Private	Public Cloud. Private Cloud z. B. über appscale [4] mit Einschränkungen möglich.
Status Java	Bis auf wenige Ausnahmen bietet das Java-SDK den gleichen Funktionsumfang wie das Python-SDK. Einschränkungen auf White List von Klassen [8] und damit auch bei einigen Frameworks [9].
Rechenzentren	USA und Europa. Es ist teilweise möglich festzulegen, ob die Daten der Anwendungen auf Knoten in den USA oder in Europa gespeichert werden.
Unterstützte Sprachen	Python, Java, Go
Java-EE-Unterstützung	nur Servlet-Container
Relationale Datenbanken	Google Cloud SQL – allerdings momentan noch im Status „experimentell“.
Zusätzliche Services	Z. B. Blobstore, um Dateien bis zur Größe von 2 GB abzulegen, Image Service, um Bilder zu skalieren und zu rotieren, XMPP für Chat-Applikationen, Search, um eine Freitextsuche zu integrieren. Google Cloud SQL, um eine relationale Datenbank anzubinden.
Flexibilität	Es ist kaum möglich, die Plattform zu konfigurieren. Innerhalb gewisser Grenzen lassen sich einige Services konfigurieren (z. B. Queue, DataStore).
Skalierung	Automatisch. Für die Skalierung sorgt die Plattform. Automatisch werden weitere Instanzen der Anwendungen in eigenen JVMs gestartet.
Redundanz	Keine Eingriffe erforderlich. Instanzen werden automatisch gestartet. Falls eine HTTP-Session verwendet wird, muss nur dafür gesorgt werden, dass die Daten serialisierbar sind.
Kostenmodell	Nach Ressourcenverbrauch: benutzte Bandbreite, benutzter Speicherplatz, versendete E-Mails etc.
Lock-in	Solange man keine Google-spezifischen Dienste verwendet, lässt sich eine Anwendung grundsätzlich in jeden Web-Container deployen. Da das aber kaum möglich ist, lässt sich ein späterer Wechsel auf eine andere Plattform nur mit programmatischen Anpassungen bewerkstelligen.
Tutorial	siehe [10]

Tabelle 1: Eigenschaften der Google App Engine

Die volle Flexibilität bietet nur der *DataStore* der GAE. Dabei handelt es sich um einen schemalosen Datenspeicher, in dem Entitäten unter einem Schlüssel abgelegt werden können. Diese Entitäten können bis zu 5000 Eigenschaften haben. Unterstützt werden diverse Datentypen für diese Eigenschaften. Es können sogar mehrere Werte pro Eigenschaft gespeichert werden.

Durch hierarchische Beziehungen der Entitäten kann man festlegen, dass diese auf einem Knoten in der Infrastruktur der GAE gespeichert werden. Das ist wichtig, wenn man sicherstellen will, dass Änderungen an mehreren Entitäten durch eine Transaktion zusammengefasst werden. Auf diese Weise wird erreicht, dass sich die Daten leicht replizieren lassen und trotzdem ein hohes Maß an Datenintegrität sichergestellt ist. Außerdem muss man sich um die Problematik der Schemaevolution keine Gedanken machen, man kann jederzeit weitere Entitäten oder Eigenschaften hinzufügen. Dadurch lässt sich die Entwicklung von Prototypen beschleunigen, und die Aktualisierung der Anwendungen ist einfach möglich. Natürlich sind damit auch Nachteile verbunden:

- So ist es nicht trivial, wenn man referenzielle Integrität zwischen Entitäten herstellen möchte. Dazu müssen diese die gleiche übergeordnete Entität haben.
- Außerdem sind Anfragen Beschränkungen unterworfen. So sind beispielsweise Anfragen mit Ungleichheitsfiltern nur für eine Eigenschaft zulässig.
- Für alle Abfragen müssen Indices vorhanden sein. Bei einigen kann die GAE diese selbst erstellen, bei anderen müssen sie vorher definiert werden. Dabei unterstützt auch hier der Entwicklungsserver, sodass sich diese Forderung relativ leicht erfüllen lässt.

Dieses hohe Maß an Flexibilität erfordert andererseits aber auch, dass man das mögliche Datenchaos beherrschbar macht. Der *DataStore* verhindert nicht, dass Werte nicht *null* sein dürfen oder dass die gleiche Eigenschaft in unterschiedlichen Entitäten verschiedene Datentypen hat. Will man sich hier etwas typsicherer bewegen, bietet es sich an, auf *JPA* oder *JDO* zu setzen. Diese Frameworks sind in das SDK der GAE integriert und ermöglichen so auf Java-Ebene die Typsicherheit herzustellen. Außerdem lässt sich so der manuell zu erstellende Code für eine Persistenzschicht deutlich reduzieren. Bei Bedarf ist es aber trotzdem jederzeit möglich, das Low-Level-API zu verwenden.

Services der GAE

Bereits in den Anfängen der GAE wurden einige Dienste zur Verfügung gestellt, die bei der Umsetzung von An-

wendungen unterstützen. So muss man sich nicht um ein eigenes Usermanagement kümmern, sofern man damit leben kann, dass alle Anwender über einen Google-Account verfügen müssen. Die Plattform übernimmt dann die Authentifizierung und stellt der Applikation ein *User*-Objekt mit E-Mail-Adresse, Nickname und User-ID zur Verfügung.

Ein weiterer Dienst, der von Anfang verfügbar war, ist der Mail-Service. Dieser ermöglicht es, über die Plattform E-Mails zu versenden und zu empfangen. Für den Empfang von Mails muss man ein Servlet zur Verfügung stellen. An dieses leitet Google eintreffende E-Mails dann weiter.

Für verteilte Anwendungen besonders wichtig ist ein verteilter Cache. Dieser wird durch den *Memcache*-Service zur Verfügung gestellt, der wie eine Hashmap funktioniert. Man legt Werte (bis zu 1 MByte Größe) unter einem Schlüssel ab und kann bei Bedarf zusätzlich ein Verfallsdatum angeben.

Prinzipbedingt musste anfänglich jede Aufgabe mit einem Request verbunden sein. Inzwischen bietet die GAE einen Mechanismus, mit dem sich Aufgaben im Hintergrund ausführen lassen. Dazu kann man entweder die Task Queues verwenden oder eine eigene Backend-Instanz starten lassen. Die Task Queues simulieren eigentlich nur Requests, die normalerweise von Anwendern kommen und stoßen letztendlich lediglich die Abarbeitung von Requests an. Auf Backend-Instanzen kommt man ohne diese Requests aus und kann auch Threads starten, die länger laufen, als eine Request-Bearbeitung dauern darf (60-Sekunden-Grenze).

Wer Fotos seiner Nutzer verarbeiten möchte, der findet im Image Service Unterstützung für das Skalieren, Drehen und Verbessern von Bildern.

Zu den später integrierten Diensten zählt der *Blobstore*. Er stellte eine erste Version eines Speichers für größere Datenmengen dar. Inzwischen steht ein weiterer Service für das Speichern großer Daten als Experimentierversion zur Verfügung, der so genannte *Google Cloud Storage*. Die Interaktion zwischen GAE und *Google Cloud Storage* erfolgt über ein REST-API. Der Dienst steht auch ohne GAE zur Verfügung.

Möchte man eine Freitextsuche in seine Applikation integrieren, gibt es auch dafür inzwischen einen Dienst: *Search*. Auch dieser Dienst ist noch nicht für den produktiven Einsatz gedacht. Er hat nichts mit der eigentlichen Google-Suche zu tun. Vielmehr ähnelt er dem Umgang und den Möglichkeiten, wie sie etwa Lucene bietet. Die Daten werden in Dokumenten organisiert und anschließend einem Index hinzugefügt oder wieder aus diesem Index gelöscht. Auf solch einem Index kann dann nach Wörtern, Wortgruppen oder nach Feldinhalten gesucht werden. Die Suche unterstützt auch das Speichern und Suchen geografischer Punkte inklusive einer Abstandssuche.

Wie insbesondere an den letzten beiden Diensten und der SQL-Integration zu erkennen ist, baut Google die Möglichkeiten der GAE ständig aus. So kommt es, dass man manchmal das Gefühl hat, der aktuellen Entwicklung ständig hinterherzulaufen. Außerdem sollte man vor dem Einsatz von Diensten genau prüfen, ob Google diese als stabil deklariert hat. Andernfalls muss man seine Anwendung ständig anpassen und mit Ausfallzeiten rechnen.

Fazit

Nimmt man die doch nicht unerheblichen Einschränkungen und die Tatsache in Kauf, dass die entstehenden Anwendungen nur auf der Google-Plattform laufen, dann ist es mithilfe der Eclipse-Plug-ins und der Google App Engine durchaus möglich, kleinere Anwendungen in wenigen Tagen umzusetzen und Kunden zur Verfügung zu stellen. Der Klemmer ist die Tatsache, dass Anwendung und Daten bei Google liegen. Auch wenn es Projekte gibt, die bestrebt sind diese Restriktion aufzuheben, so bleiben diese jedoch naturgemäß immer hinter dem Entwicklungsstand der GAE zurück, und die Integration ist lückenhaft. Deshalb muss man sich entweder auf Google verlassen oder die Finger von der App Engine lassen.

Mit der Einführung der SQL-Datenbank bietet die GAE inzwischen allerdings auch eine Plattform, die sich für typische Geschäftsanwendungen eignet. Vielleicht wird Google die GAE ja ebenso wie die Google-Suche zumindest als Appliance anbieten und ermöglicht Unternehmen so, die Daten und Anwendungen in eigenen Rechenzentren zu hosten.

Michael Seemann ist Diplom-Wirtschaftsinformatiker. Er arbeitet als selbstständiger Softwareentwickler, Berater und Fachautor. Sie erreichen ihn unter seemann@mseemann.de.

Links & Literatur

[1] http://code.google.com/p/go/

[2] http://gaelyk.appspot.com/

[3] http://www.google.com/enterprise/apps/business/

[4] http://code.google.com/p/appscale/

[5] https://cloud.google.com/pricing/

[6] http://code.google.com/status/appengine

[7] https://appengine.google.com/

[8] https://developers.google.com/appengine/docs/java/jrewhitelist

[9] http://code.google.com/p/googleappengine/wiki/WillItPlayInJava

[10] https://developers.google.com/appengine/docs/java/gettingstarted/

Anzeige

Anzeige

Quellcode zum Download

Mit J2ME ins Internet

Kleine Plattform, großer Markt

Nachdem wir uns im ersten Teil der Serie mit dem Thema GUI für J2ME beschäftigt haben, folgt nun eine detaillierte Analyse der im Hintergrund befindlichen Logik. Dabei gehen wir besonders auf den Internetzugriff und das lokale Speichern von Daten ein – Themen, die für Anfänger oft problematisch sind.

von Tam Hanna

In einigen Bereichen „tickt" J2ME komplett anders als der große Bruder. Der Grund dafür ist die minimale Performance der damaligen Mobilcomputer – wir haben im ersten Teil der Serie festgestellt, dass J2ME samt einiger MIDlets in weniger als 5 MB Speicher untergebracht werden kann.

Record Management System

Das Realisieren eines Dateisystems stellt einen nicht unwesentlichen Overhead dar. Alte Handcomputer verwendeten deshalb statt FAT eine Art Datenbanksystem, das die Daten der Anwendungen flach (also ohne Ordner) in einzelnen Datenbankeinträgen vorhielt.

Bei J2ME ging Sun mit dem Record Management System, kurz RMS, einen ähnlichen Weg. Der unter javax.microedition.rms findbare Namensraum besteht aus einer Klasse und vier Interfaces. Am wichtigsten ist die *RecordStore*-Klasse, die wie die von Symbian her bekannte *FileServer*-Klasse den Zugriff auf das zugrunde liegende Dateisystem regelt.

Dabei kommen fünf Regeln zum Einsatz: Die erste besagt, dass ein MIDlet nur auf seine eigenen Daten zugreifen darf. Es ist also nicht möglich, auf die von einer anderen MIDlet-Suite erstellten Record Stores zurückzugreifen, um so applikationsübergreifende Workflows zu realisieren.

Die Dateinamen der einzelnen Record Stores müssen auf MIDlet-Suite-Ebene einzigartig sein. Sie umfassen bis zu 32 Unicode-Zeichen und sind – auch hier hält man sich streng an die Java-Konventionen – Case-sensitive. Jeder Zugriff auf den Record Store inkrementiert die Versionsnummer desselben, die einzelnen Records werden durch einzigartige IDs identifiziert. Ärgerlicherweise ist der Record Store nicht Thread-safe. Wer ein multithreaded MIDlet realisiert, muss sich also selbst um die Synchronisierung kümmern.

Das Interface *RecordComparator* definiert eine Hilfsklasse, die die einzelnen Records der Applikation miteinander vergleicht. Das ermöglicht Sun das Anbieten von generischen Algorithmen – die Vergleichsintelligenz wird vom Entwickler durch eine eigene Methode beigesteuert. Der *RecordFilter* arbeitet nach demselben Schema – er ermöglicht es, beim Durchlaufen einer Datenbank nur die relevanten Einträge anzuzeigen. Das eigentliche Durchlaufen ist die Aufgabe der *RecordEnumeration*.

Zu guter Letzt erlaubt der *RecordListener* das Überwachen einer Datenbank. Seine Methoden werden immer dann aufgerufen, wenn sich etwas ändert – so kann ein Prozess auf einen anderen achten.

Datenbank herbei!

Doch damit erst genug der Theorie – unsere erste Datenbank will erstellt werden. Das Beispiel *SuSJ2ME4* besteht aus mehreren Commands und einer Textbox – der Button Create | Open database ist mit dem Code in Listing 1 verdrahtet.

Listing 1

```
try
    {
      myRS=RecordStore.openRecordStore("TmgnStore", true);
      txt.setString("Record Store offen");
    }
    catch (Exception e)
    {
      myRS=null;
    }
```

Das Identifizieren der einzelnen Record Stores erfolgt – wie schon in der Einleitung beschrieben – durch einen Dateinamen. Der zweite Parameter weist *openRecordStore* an, die Datenbank zu erstellen, so sie noch nicht existiert – wäre dieser auf „False“, so würde die Methode bei nicht existierender Datenbank eine abfangbare Exception werfen. Zum Schließen greifen wir auf die im vorigen Teil des Artikels besprochene Lifecycle-Methode *destroyApp* zurück (Listing 2).

Listing 2

```
public void destroyApp(boolean unconditional)
{
  if(myRS!=null)
  {
    try
    {
      myRS.closeRecordStore();
    }
    catch (Exception e)
    {
    }
  }
}
```

Listing 3

```
String toStore=myTextBox.getString();
byte[] myData=toStore.getBytes();
try
{
  int recordID=myRS.addRecord(myData, 0, myData.length);
  txt.setString("Successfully created record!" + Integer.toString(recordID));
}
catch (Exception e)
{
  txt.setString("Error creating");
}
Display.getDisplay(this).setCurrent(aForm);
```

Listing 4

```
try
{
StringBuffer outBuffer=new StringBuffer();
RecordEnumeration anEnum=myRS.enumerateRecords(null, null, true);
while (anEnum.hasNextElement())
{
  outBuffer.append(new String(anEnum.nextRecord()));
  outBuffer.append("\n");
}
txt.setString(outBuffer.toString());
}
catch (Exception e)
{
}
```

Hier ist eigentlich nur wichtig, dass die Implementierung den Record Store nicht beim ersten Aufruf von *closeRecordStore* schließt. Stattdessen läuft im Hintergrund ein Zähler mit, der bei jedem Aufruf von *open* inkrementiert und bei jedem Aufruf von *close* dekrementiert wird. Das eigentliche Schließen erfolgt erst dann, wenn dieser Zähler den Wert Null annimmt. Im nächsten Schritt sollten wir unsere Datenbank um einen Record erweitern. Zum Abholen des Inhalts verwenden wir diesmal eine TextBox, die bildschirmfüllend angezeigt wird:

```
myTextBox=new TextBox("New record", "", 50, TextField.ANY);
myTextBoxOK=new Command("OK", Command.OK, 0);
myTextBox.addCommand(myTextBoxOK);
myTextBox.setCommandListener(this);
Display.getDisplay(this).setCurrent(myTextBox);
```

Auch die TextBox wird als Member-Variable der *Midlet*-Klasse angelegt. Beim Erstellen fragt der Konstruktor neben dem Beschreibungstext und einer Vorlage auch nach der maximalen Eingabelänge und einer Konstante, die die einzugebenden Daten beschreibt. Dabei kommen die im vorigen Artikel besprochenen Konstanten aus dem *TextField*-Objekt zum Einsatz – das Übergeben von *All* weist das System dazu an, jegliche Art von Eingabe zu akzeptieren.

Nach dem üblichen Parametrieren der diversen Commands rufen wir *setCurrent* auf, um die TextBox bildschirmfüllend darzustellen. Der Benutzer interagiert mit ihr und schließt sie danach über unser Command. Dessen Event Handler enthält die eigentliche Intelligenz (Listing 3).

Auch hier verbirgt sich keine allzu komplexe Logik. Im ersten Schritt beschaffen wir uns den String, der vorher vom Benutzer in die TextBox eingegeben wurde. Danach fügen wir dem Record Store einen neuen Eintrag hinzu, der die ermittelten Daten enthält.

Dabei verlangt die Methode insgesamt drei Parameter. Der erste ist ein Zeiger auf ein *byte[]*-Array, in dem die zu speichernden Rohdaten liegen (das Datenformat ist für RMS von keinerlei Belang).

Wie die meisten *RecordStore*-Methoden wirft auch *addRecord* von Zeit zu Zeit eine Exception. Aus diesem Grund ist es erforderlich, die Anweisung in einem Try-catch-Block zu vergraben.

Zu beachten ist, dass die einzelnen Records einer Datenbank außer der Record-ID keinerlei besonderen Eigenschaften haben. Es ist also nicht möglich, einen Record als „wichtig“ zu kennzeichnen – das Durchlaufen aller Records ist das täglich Brot des J2ME-Entwicklers.

Durchsuchen der Records

Genau das wollen wir uns im nächsten Schritt ansehen. Der Befehl *Output all* durchläuft alle Einträge der Datenbank und gibt ihre Inhalte am Ende in die TextBox des Formulars aus (Listing 4).

Die Methode *EnumerateRecords* gibt uns einen Record Enumerator zurück, der uns beim eigentlichen

Durchlaufen der Records assistiert. Die ersten beiden Parameter erlauben das Übergeben eines Filters und/oder einer Sortierklasse – ihre Aufgabe ist in der Einleitung beschrieben. Der dritte Parameter legt fest, ob die Enumeration bei Änderungen in der zugrunde liegenden Datenbank aktualisiert werden soll oder nicht.

Zu beachten ist, dass das Löschen eines Records nicht zum Recycling seiner Record-ID führt. Es ist also durchaus möglich, eine Datenbank mit den Record-IDs 1, 2, 3 und 55 vorzufinden. Aus diesem Grund ist es nicht ratsam, zum Enumerieren der Einträge auf eine klassische *for*-Schleife zurückzugreifen – es könnte sein, dass die Datenbank Tausende von „unbelegten" Record-IDs enthält.

File I/O

Die J2ME-Plattform legt nicht fest, wie die RMS-Implementierung ihre Daten speichert. In der Praxis – User wollen ja auch Klingeltöne, Emoticons und Bilder speichern – greifen die Hersteller deshalb trotzdem auf ein hierarchisches Dateisystem zurück. Im Laufe der Zeit entschied man sich bei Sun, den Zugriff auf dieses auch für MIDlets freizugeben.

Um das API nicht zwangsweise zum Teil der Plattform zu machen, wurde es in die JSR 75 ausgelagert. Diese auch *File Connection* genannte Schnittstelle erlaubt den Zugriff auf Dateisysteme – Zeit, sie sich im Beispiel *SuSJ2ME5* näher anzusehen. Der erste Befehl prüft, ob das vorliegende Telefon die JSR75 überhaupt unterstützt (Listing 5).

Listing 5

```
String versionId = System.getProperty("microedition.io.file.FileConnection.version");
if(versionId.equals("1.0"))
{
  txt.setString("JSR gefunden");
}
else
{
  txt.setString("JSR fehlt oder inkompatibel");
}
```

Der hier gezeigte Code ist relativ primitiv. Wir benutzen *getProperty*, um die Versioneigenschaft des *FileConnection*-Namespaces auszulesen. Bis dato gibt es nur die Version 1.0 – wenn der zurückgegebene String passt, so kennt das Telefon das API für den Dateizugriff. Je nach Telefon gibt es ein oder mehrere Laufwerke. Diese werden in *FileConnection* als Roots bezeichnet. Die Auflistung aller verfügbaren Roots erfolgt mit dem Code in Listing 6.

Auch hier findet sich keine Raketenwissenschaft. Der Aufruf von *FileSystemRegistry.listRoots()* liefert uns

eine Liste der Roots, die wir danach wie einen gewöhnlichen Iterator auswerten. Der Code zum Durchforsten beliebiger Laufwerke bzw. ihrer Unterordner folgt einem ähnlichen Schema (Listing 7).

Das *FileConnection*-API reagiert auf das Öffnen von URLs mit dem Präfix *file://*. Aus diesem Grund basteln wir aus dem „derzeitigen Ort" einen Pfad zusammen, der danach mit einer *FileConnection* „enumeriert" wird.

Damit bleibt nur noch die Frage nach der Darstellung. Wir öffnen nach einem Klick auf Browse eine ListBox, in der die von den oben erstellten Methoden zurückgegebenen Vektoren angezeigt werden:

```
Display.getDisplay(this).setCurrent(myLBox);
updateDisplay(getRoot());
```

Das „Einschreiben" in das aktive Display entspricht dem von Alerts und Textboxen bekannten Schema. Die Methode *updateDisplay* handhabt das Aktualisieren der in der Liste gespeicherten Werte (Listing 8).

Wichtig ist, dass die ListBox einen eigenen Speicher für Elemente mitführt. Das Einpflegen und Entfernen von Items erfolgt direkt durch Methoden der Liste – das unter Android zwingend erforderliche Anbieten eines Modells entfällt unter J2ME komplett. Zu guter Letzt noch ein Blick auf das Command Handling der Liste:

```
if(c==myCmdSelect)
{
  myLocation=myLocation.concat(myLBox.getString(myLBox.
                                                getSelectedIndex()));
  updateDisplay(getList(myLocation));
}
```

Auch dieser Code entspricht fast 1:1 dem im ersten Teil des Artikels verwendeten Event Handler. Wir ergänzen den „lokalen Pfad" um das vom Benutzer ausgewählte Verzeichnis und aktualisieren danach den Inhalt der Liste. Zu beachten ist, dass wir Commands theoretisch zwischen mehreren Displayables recyceln können – der hier nicht abgefragte *d*-Parameter des Event Handlers erlaubt die Korrelierung. Die ListBox selbst wird wie ein Alert im Konstruktor der MIDlet-Klasse erstellt:

```
myLBox=new List("Browser", List.IMPLICIT);
myCmdSelect=new Command("Traverse", Command.OK, 0);
myLBox.addCommand(myCmdSelect);
myLBox.setCommandListener(this);
```

An dieser Stelle ist wichtig, dass J2ME mehrere Typen von ListBox kennt. Das Verwenden der Option *Implicit* sorgt dafür, dass immer nur ein Element auf einmal markierbar ist.

Debugging J2ME

Beim Entwickeln komplexerer Anwendungen ist es früher oder später soweit: Ein Programmfehler tritt auf. Wer den von Sun bereitgestellten Emulator verwendet, darf zur Abwehr des Problems auf eine Vielzahl verschiedener Methoden zurückgreifen. Am einfachsten ist es, Debugger-Informationen in die Konsole von NetBeans auszugeben. Dazu kommt der vom PC bekannte Namespace *System.out* zum Einsatz – der Rest entspricht 1:1 der in jedem Lehrbuch auffindbaren Vorgehensweise. Bei komplexeren Problemen ist es oft wünschenswert, mit Breakpoints zu arbeiten. Auch das ist in NetBeans problemlos möglich – wichtig ist in diesem Fall nur, dies der IDE im Voraus mitzuteilen.

Listing 6

```
public Vector getRoot()
{
Vector dirList = new Vector();

Enumeration e = FileSystemRegistry.listRoots();
while (e.hasMoreElements())
{
  String anElement=(String) e.nextElement();
  dirList.addElement(anElement);
}
return dirList;
}
```

Listing 7

```
public Vector getList(String currLocation)
{
Enumeration e = null;
FileConnection fileCon = null;
try {
  fileCon = (FileConnection) Connector.open("file:///" + currLocation);
  e = fileCon.list();

  if (fileCon != null) {
    fileCon.close();
  }
}
catch (Exception ex) {}
Vector dirList = new Vector();
while (e.hasMoreElements())
{
  dirList.addElement((String) e.nextElement());
}
return dirList;
}
```

Listing 8

```
void updateDisplay(Vector aVect)
{
  myLBox.deleteAll();
  for(int i=0;i<aVect.size();i++)
  {
    myLBox.append((String)aVect.elementAt(i), null);
  }
}
```

Zum Starten des Programms verwenden Sie statt des Play-Symbols das Debug-Symbol, das in der Toolbar direkt links vom normalen Icon ist. Beim Start des Emulators wirft er eine Meldung nach dem Schema *Starting emulator in debug server mode* in die Konsole.

Beim Entwickeln von Anwendungen auf Basis der File Connection ist es sinnvoll, auf die Dateisysteme des Emulators zuzugreifen. Jedes emulierte Telefon hat dabei sein eigenes Verzeichnis – der Standardemulator auf der Maschine des Autors sucht sein Dateisystem im Pfad *C:\Users\TAMHAN\javame-sdk\3.2\work\JavaMEPhone1\appdb\filesystem*. Dort finden Sie in der ersten Ebene die Roots, eine Ebene darunter die jeweiligen Inhalte.

PIM

Es gibt in JSR75 auch ein API zum Zugriff auf die am Telefon befindlichen PIM-Daten. Das bedeutet, dass Ihre Applikationen mit den in Adressbuch, Kalender und To-do-Listen gespeicherten Records interagieren dürfen. Auch dafür bauen wir ein kleines Beispiel – *SuS-J2ME6* demonstriert das richtige Verwenden dieses API. Als Erstes auch hier ein Blick auf die Methode zum Prüfen der Unterstützung – der Unterschied zu vorher liegt nur im anderen String:

```
if(c==myCheckJSR)
{
  String versionId = System.getProperty("microedition.pim.version");
  if(versionId.equals("1.0"))
  {
    txt.setString("JSR gefunden");
  }
```

Zu beachten ist, dass auch das Vorhandensein des API nicht für die Existenz der jeweiligen Datenbanken garantiert. Es ist also durchaus möglich, dass ein Telefon nur ein Adressbuch enthält – das Öffnen der Kalender und To-do-Listen schlägt in diesem Fall fehl.

Das Zählen der in der Kontaktdatenbank befindlichen Einträge erlaubt das Vorführen einiger grundlegender Konzepte zum Zugriff auf PIM-Daten (Listing 9).

Der Zugriff auf PIM-Daten erfolgt immer über das PIM-Objekt. Dabei handelt es sich um ein Singleton, weshalb wir unsere Instanz durch den Aufruf von *getInstance()* beleben. Das PIM-Objekt erlaubt uns das Interagieren mit den eigentlichen Kontaktdatenbanken – der Aufruf von *openPIMList* liefert je nach Parameterlage eine Kontakt-, eine Adress- oder eine To-do-Liste zurück. Als Erstes bekommen Sie eine generische PIM-List zurückgeliefert. Erst durch einen Cast ist der Zugriff auf die spezifischen Elemente möglich, der Rest des Codes ist Standard-Java.

Beim Einpflegen eines neuen Eintrags tritt eine Besonderheit auf. Da die Anzahl der unterstützten Felder je nach Implementierung anders ist, muss die Methode auch dies berücksichtigen (Listing 10).

Nachdem wir Zugriff auf die Kontaktliste erhalten haben, erstellen wir mit der Methode *createContact()* einen neuen Kontakt. Er wird am Ende der Routine durch Aufruf von *commit()* in den Remanent-Speicher geschrieben.

Der Name des Kontakts ist als *StringArray*-Eigenschaft definiert. Aus diesem Grund erstellen wir als Erstes ein String Array, dessen implementationsabhängige Größe durch Aufruf von *stringArraySize* ermittelt wird. Danach prüfen wir bei jedem Element, ob es der Implementierung bekannt ist. Ist das der Fall, so folgt die

Listing 9

```
else if (c==myCountAdresses)
{
  PIM myAccessor=PIM.getInstance();
  PIMList aList;
  try
  {
    aList = myAccessor.openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE);
    ContactList cList=(ContactList) aList;

    Enumeration anEnumerator=cList.items();
    int counter=0;
    while(anEnumerator.hasMoreElements())
    {
      counter++;
      anEnumerator.nextElement();
    }
    txt.setString("Items: " + Integer.toString(counter));
    aList.close();
  }
  catch (PIMException ex)
  {
    txt.setString("PIM error");
  }
}
```

Listing 10

```
else if (c==myAddAdresses)
{
  PIM myAccessor=PIM.getInstance();
  PIMList aList;
  try
  {
    aList = myAccessor.openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE);
    ContactList cList=(ContactList) aList;

    Contact newContact=cList.createContact();
    String[] name = new String[cList.stringArraySize(Contact.NAME)];
    if (cList.isSupportedArrayElement(Contact.NAME, Contact.NAME_FAMILY))
      name[Contact.NAME_FAMILY] = "Hanna";
    if (cList.isSupportedArrayElement(Contact.NAME, Contact.NAME_GIVEN))
      name[Contact.NAME_GIVEN] = "Tam";

    newContact.addStringArray(Contact.NAME, PIMItem.ATTR_NONE, name);
    newContact.commit();

    aList.close();
  }
```

Zuweisung eines Werts. Zu guter Letzt schreiben wir das String Array in den Kontakt.

App ins Netz

Das Aufkommen der ersten J2ME-Handys war zeitlich eng mit dem Erst-Deployment von GPRS verbunden. Dieser Funkdienst erlaubte erstmals das Versenden und Empfangen von Paketdaten – das bis dato verwendete CSD-System baute wie ein klassisches Telefonmodem eine Verbindung auf, die nach Minuten verrechnet wurde. Trotz der damals extrem hohen Preise wurde dadurch erstmals richtige Datenkommunikation möglich. Sun trug dieser Tatsache durch die *HTTPConnection* Rechnung – in *SusJ2ME7* ist sie wie in Listing 11 eingebunden.

Auch in diesem Code verbirgt sich keine Raketenwissenschaft. Wir verwenden die Connector-Klasse, um ein *HTTPConnection*-Objekt zu erzeugen. Danach öffnen wir einen *InputStream* und lesen seinen Inhalt in einen *StringBuffer* ein.

Natürlich zeigt dieses Beispiel nur die Basis. In der Praxis wäre es sinnvoll, das Einlesen der Daten in einen Hintergrund-Thread auszulagern. Auch ist zu beachten, dass sowohl der Strom als auch das *HTTPConnection*-Objekt über *Close()*-Methoden verfügen, deren Aufruf mit Sicherheit sinnvoll ist. Damit nicht genug: J2ME bietet auch eine Vielzahl anderer Netzwerkschnittstellen, darunter sogar Sockets. Es ist also durchaus legitim, den Netzwerk-Stack als durchaus gut entwickelt zu bezeichnen.

Manifestationen

Beim Deployment einer Applikation liest die auf dem Telefon befindliche Managementsoftware als Erstes die Manifest-Datei aus. Bevor wir unser erstelltes Beispiel ausführen können, müssen wir erst die notwendigen Permissions deklarieren. Dazu klicken wir das Projekt rechts an und wählen die Option Properties. Im daraufhin erscheinenden Dialog wechseln wir in die Rubrik Appliation Descriptor | API Permissions und fügen die erforderlichen API-Klassen hinzu. Während der Programmausführung würde das Telefon den Benutzer trotzdem fragen, ob dieser Zugriff gewünscht ist – das ist einer der lästigen Aspekte der Verwendung von J2ME. Je nach Telefontyp lässt sich diese Abfrage durch eine Signatur verhindern – das Behandeln dieses sehr undankbaren Themengebiets würde den Rahmen des Artikels endgültig sprengen.

Listing 11

```
HttpConnection    aConn = null;
DataInputStream    inStream = null;
StringBuffer      aBuffer = new StringBuffer();

try
{
  aConn = ( HttpConnection ) Connector.open( "http://www.google.com" );
  inStream = new DataInputStream( aConn.openInputStream() );
  int ch;
  while ( ( ch = inStream.read() ) != -1 )
  {
    aBuffer = aBuffer.append( ( char ) ch );
  }
}
catch(Exception e)
{

}
finally
{
  txt.setString(aBuffer.toString().substring(0, 50));
}
```

Kleine App ganz groß

Der Blu-Ray-Standard war vor einigen Jahren groß in Mode. Der DVD-Nachfolger erlaubt das Einbinden von interaktiven Inhalten in die optischen Medien. So ist es Filmstudios möglich, ihren Film z. B. mit Mini-Games zu versehen. Dabei kommt eine BD-J genannte Technik zum Einsatz. Die XLet genannten Prögrämmchen werden in Java geschrieben und haben ein eng mit dem J2ME-Lebenszyklus verwandtes Zustandsdiagramm.

Je nach „Profile“ des Players gibt es mehr oder weniger Funktionen wie z. B. ein lokales Dateisystem oder den Zugriff auf Internetinhalte. Statt der klassischen LCDUI kommt ein HAVI genannter GUI-Stack zum Einsatz, der Zugriff auf die Medieninhalte erfolgt durch ein spezielles API [1].

Fazit

Damit endet unsere kleine Reise durch die Welt von J2ME. Die Technologie ist mit Sicherheit nicht allzu hip – bietet aber insbesondere in Entwicklungsländern gigantische Reichweiten. Denn: Auch wenn die Anzahl der Smartphone-User rasch wächst – noch sind die Besitzer von „Dumbphones“ mit Abstand in der Überzahl. Der beispiellose Erfolg von Opera Mini zeigt, dass auch diese Zielgruppe Interesse an Software hat – sie ist oftmals nur nicht in der Lage, dafür direkt zu bezahlen.

Bei der „professionellen“ J2ME-Entwicklung ist es Usus, auf externe Bibliotheken zurückzugreifen. Die meisten Telefonhersteller haben eigene Libraries zum Zugriff auf spezifische Hardwareerweiterungen; zusätzlich buhlt eine Vielzahl von General Purpose Libraries um die Gunst der Entwickler.

Die Möglichkeit der Expansion in Richtung Blu-Ray ist ein weiteres Argument für die Beschäftigung mit J2ME – und auf dem Lebenslauf sieht Erfahrung mit dieser Technologie sicherlich nicht schlecht aus.

Tam Hanna befasst sich seit der Zeit des Palm IIIc mit Programmierung und Anwendung von Handcomputern. Er entwickelt Programme für diverse Plattformen, betreibt Onlinenewsdienste zum Thema und steht unter tamhan@tamoggemon.com für Fragen, Trainings und Vorträge gern zur Verfügung.

Links & Literatur

[1] http://www.cojug.org/downloads/BluRay-Development.pdf

Die hohe Kunst des agilen Testens

Yoga für Fortgeschrittene

Mangelnde Qualität und schlechte Anforderungen können schnell zu ausufernden Projektlaufzeiten führen und vorhandene Budgets sprengen. Daher setzten clevere IT-Verantwortliche auf solides Testmanagement, bei dem die methodische Planung und Durchführung entscheidend ist. Daneben müssen Testabteilungen in Zeiten der Agilität über Test-Engineers mit dem richtigen Mix an Eigenschaften verfügen. Bereits vor der Ära der agilen Softwareentwicklung war es nicht einfach, qualifizierte Test-Engineers mit der richtigen Kombination an Skills zu finden – heute, in Zeiten von Scrum, ist dies ungleich schwieriger. Umso wichtiger ist es, die benötigten Qualifikationen zu kennen, um geeignete Mitarbeiter zu finden, und das bestehende Team zu entwickeln. Eine besondere Rolle spielt dabei vor allem in agilen Vorgehensmodellen das Mantra des Yoga: Ausgewogenheit.

von Sven Schirmer

Zugegeben, Software-Testing und Yoga haben auf den ersten Blick nicht viel gemeinsam – schaut man jedoch genauer hin, wird eine grundlegende Parallele erkennbar: Wie im Yoga, kommt es beim erfolgreichen Software-Testing in agilen Projekten auf die Ausgewogenheit an. Der Yogi, der einen Kopfstand lernen möchte, wird am Anfang seiner Praxis öfter umfallen – erst wenn ihm klar wird, wie wichtig die Ausgewogenheit von verschiedenen Eigenschaften ist, wird er Erfolg haben und sein Ziel erreichen. Ähnlich müssen Testmanager und Test-Engineers ihre Eigenschaften Flexibilität, Stärke und Balance sinnvoll einsetzen, um das Testing in agilen Projekten erfolgreich durchzuführen. Wenn das Augenmerk auf die richtigen Fähigkeiten gelegt wird und diese bewusst weiterentwickelt werden, werden sich die Mitarbeiter einer Test-Factory auch in agilen Projekten beweisen.

Anhand des Vorgehensmodells Scrum werden die Fähigkeiten aufgezeigt, die ein professioneller Test-Engineer in der modernen Softwareentwicklung mitbringen sollte, um erfolgreich zu sein. Der Artikel beschreibt und begründet, auf welche Eigenschaften im Bereich der Softskills es vor dem Hintergrund der Agilität ankommt.

Was ist Scrum eigentlich?

Der Grundgedanke hinter Scrum ist, Komplexität in Entwicklungsprojekten durch drei Prinzipien zu reduzieren:

1. Transparenz: Der Fortschritt und die Hindernisse eines Projekts werden täglich und für alle sichtbar festgehalten.
2. Überprüfung: In regelmäßigen Abständen werden Produktfunktionalitäten geliefert und beurteilt.
3. Anpassung: Die Anforderungen an das Produkt werden nicht vorab festgelegt, sondern nach jeder Lieferung neu bewertet und bei Bedarf angepasst.

Scrum verkörpert die Werte des Agilen Manifests. Das Ziel dieser Projektvorgehensweise ist eine möglichst schnelle, kosteneffiziente und gleichzeitig qualitativ hochwertige Entwicklung einer Software zu gewährleisten, die einer im Vorfeld klar definierten Vision entspricht.

Im Gegensatz zu gängigen Vorgehensmodellen wie dem Wasserfallmodell erfolgt bei Scrum die Entwicklung in kurzen sich wiederholenden Intervallen, den so genannten Sprints. Sämtliche Aktivitäten sind eng terminiert und dienen der unmittelbaren Vorbereitung der jeweils nächsten Aktivität. User Stories ersetzen die

Das agile Manifest

- **Individuen und Interaktionen** sind mehr als Prozesse und Werkzeuge
- **Funktionierende Software** ist mehr als umfassende Dokumentation
- **Zusammenarbeit mit dem Kunden** ist mehr als Vertragsverhandlung
- **Reagieren auf Veränderung** ist mehr als das Befolgen eines Plans

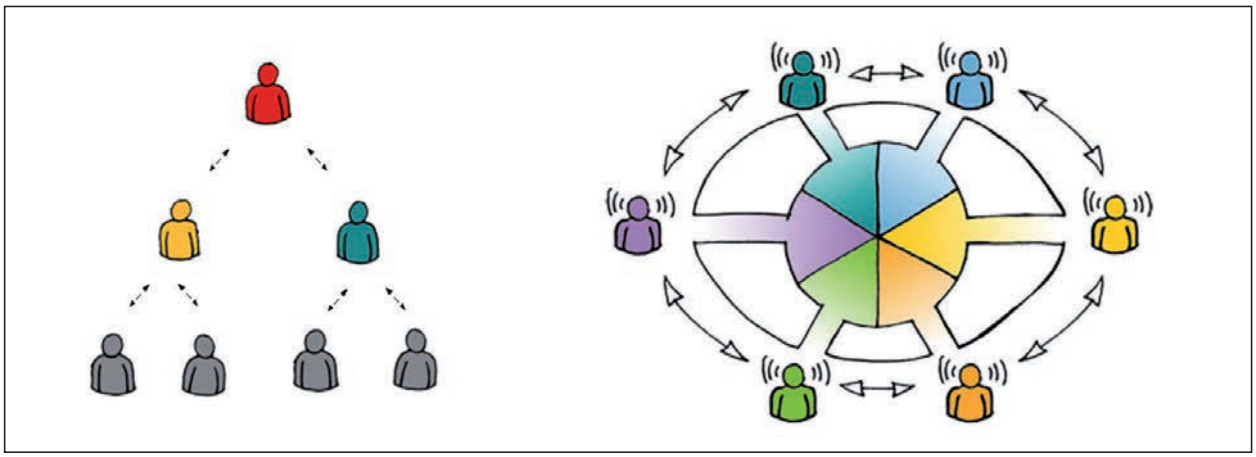


Abb. 1: Im klassischen Projekt ist das Testteam hierarchisch organisiert, im Scrum-Team ist der Tester ein Teammitglied zusammen mit Entwicklern

sonst üblichen Lasten- und Pflichtenhefte. Nach jedem zwei bis vier Wochen umfassenden Sprint steht bei Scrum die Lieferung einer lauffähigen Software. Eine Auslieferung an den Kunden sollte grundsätzlich möglich sein.

Der größte Unterschied zwischen klassischen und agilen Modellen ist die Art und Weise wie die Teams im Projekt aufgebaut sind. Während in hierarchischen Modellen die Koordination des Teams von oben nach unten gestaltet ist, liegt der Hauptfokus bei agilen Teams in der Kooperation der Teammitglieder untereinander. Eine wichtige Rolle im Scrum-Team ist die des Scrum Masters. Er ist dafür verantwortlich, dass das Team möglichst ungestört innerhalb eines Sprints arbeiten kann und auftretende Behinderungen möglichst schnell aus dem Weg geräumt werden. Eine zweite zentrale Rolle ist die des Product Owner. Seine Aufgabe ist es, die Anforderungen aus Kundensicht in Form von User Stories zu formulieren und die Software nach jedem Sprint abzunehmen.

Ein zentraler Punkt ist damit das Zusammenspiel im Team. Hierbei ist darauf zu achten, dass sich das Team selbst organisiert. Scrum Master und Product Owner sollen das Team unterstützen, haben aber nicht die klassische Rolle des Projektleiters. Wann welche Aufgaben innerhalb eines Sprints erledigt werden, entscheidet das Team.

Der Scrum Master achtet darauf, dass niemand in den Prozess der Selbstorganisation eingreift, das Team in seiner Arbeit beeinflusst oder Verantwortlichkeiten beansprucht, die ihm nicht zustehen. Und genau hier beginnen die Herausforderungen für die Testorganisation.

Die Gefahren bei der Unausgewogenheit von Flexibilität, Stärke und Balance

- **Balance und Flexibilität ohne Stärke** bedeutet, dass Testaspekte bei den Planungen und innerhalb der Sprints vermeintlich wichtigeren Entwicklungsaspekten zum Opfer fallen.
- **Flexibilität und Stärke ohne Balance** bedeutet, dass zu viel oder auch zu wenig Testautomatisierung umgesetzt wird und die Testabdeckung und damit letztendlich die Qualität leidet.
- **Stärke und Balance ohne Flexibilität** bedeutet, dass die Akzeptanz des Testings im Scrum-Team leidet, weil Themen adressiert aber danach nicht vom Testspezialisten umgesetzt werden.

Die zentrale Herausforderung für den Test

In klassischen Projekten verantwortet der Testmanager die Planung, das Controlling und die Koordination des Testing. Er ist zentraler Ansprechpartner für den Kunden und letztendlich verantwortlich für die Qualität des Softwareprodukts. Das Testteam agiert innerhalb von einem Projekt als eigenständiges Teilprojekt, das sich meist über den Testmanager oder in verschiedenen Meetings mit den anderen Teilprojekten abstimmt. Der Test-Engineer innerhalb dieses Teilprojekts ist dem Testmanager untergeordnet und operativ tätig. Er leitet beispielsweise Testfälle aus Anforderungen ab, erstellt Testautomatisierungsskripte oder generiert für die Durchführung erforderliche Testdaten.

Bei Scrum allerdings verschwimmen die Rollen des Testmanagers und des Test-Engineers aufgrund der Teamorganisation stark. Zusätzlich zu den fachlichen und technischen Anforderungen an die Test-Engineers im Scrum-Team gewinnen bestimmte Softskills signifikant an Bedeutung.

Da gute Test-Engineers schwer zu finden sind und diese nun bei Scrum-Projekten vor neuen Herausforderungen stehen, ist es umso wichtiger, die Fähigkeiten der eigenen Mitarbeiter zu evaluieren und durchdachte Weiterbildungsmaßnahmen zu etablieren. Welche Eigenschaften sind nun essenziell in der agilen Softwareentwicklung?

Wie beim Yoga stützt sich ein exzellenter Test-Engineer in der agilen Softwareentwicklung auf drei Fähigkeiten: Flexibilität, Stärke und Balance.

Flexibilität – Das Agile Manifest als Herausforderung für den modernen Testmanager

Wie in den einzelnen Übungsabläufen im Yoga ist auch für die Mitglieder des Scrum-Teams Flexibilität eine wichtige Eigenschaft. Bedingt durch die geringe Größe des Teams müssen die Mitglieder flexibel verschiedene Rollen einnehmen können, um alle erforderlichen Aktivitäten im Sprint abdecken zu können. Neben operativen Themen ist es wichtig, sich ergänzend mit Fragen des Testmanagements auseinanderzusetzen, um gegebenenfalls steuernd mit einzugreifen.

Mangelnde Flexibilität führt dazu, dass auf Unvorhergesehenes nicht angemessen und zeitnah reagiert werden kann. Fällt beispielsweise der Test-Engineer im Team durch Krankheit aus, so muss ein anderes Teammitglied dessen Rolle übernehmen können, damit der Sprint nicht gefährdet wird. Wichtig ist, dass jeder im Team bereit ist, eine ihm nicht völlig vertraute Rolle zu übernehmen, um die Qualität im Sprint voranzubringen.

Gerade die Werte des „Agilen Manifests“ können für Testmanager, die schon große Projekte begleitet und verantwortet haben, eine Herausforderung an die eigene Arbeitsweise sein. Hier kommt es auf die Flexibilität der Aufgaben an, die dieser in einem Scrum-Team übernehmen kann. So ist es dort selbstverständlich, dass sich der Test-Engineer nicht nur mit planerischen und methodischen Fragen um das Thema Test beschäftigt, sondern auch selbst die Testfälle erstellen und anschließend durchführen kann. Daher ist es wichtig, dass ein Testspezialist im agilen Umfeld sowohl die Aspekte aus dem operativen Testing als auch aus dem Testmanagement beherrscht und diese flexibel, den aktuellen Umständen im Sprint entsprechend einsetzen kann.

Stärke – Konsequente Einbindung des Testings von Beginn an

Die zweite, zentrale Eigenschaft eines Test-Engineers in Zeiten der Agilität ist Stärke im Sinne von Durchsetzungsvermögen. In der Yogapraxis ist dieses Durchsetzungsvermögen vor allem gegenüber dem eigenen Ego wichtig. Dieses verleitet einen häufig dazu, die Yogastunde sehr niedrig zu priorisieren und lieber andere Dinge zu tun. Danach wundert man sich, warum man im Yoga nicht vorankommt.

In Scrum-Projekten verleiten die kurzen Entwicklungszyklen gerne dazu, Testanforderungen niedrig priorisiert hinten anzustellen. Hier ist der Test-Engineer gefordert, das Thema Testing zu platzieren und darauf zu achten, dass die erforderlichen Aktivitäten bei den Planungen für die Sprints mit den entsprechenden Aufwänden berücksichtigt werden. Diese Fähigkeit hat direkten Einfluss auf das Ergebnis des Sprints. Fehlt dem

Test-Engineer die notwendige Stärke, so leidet darunter die Qualität der Software und das Team wundert sich, warum das Ergebnis vom Product Owner zurückgewiesen wird.

Balance – Testautomatisierung vs. manuelle Tests

Gerade der Testautomatisierung wird ein hoher Stellenwert innerhalb der agilen Softwareentwicklung zugemessen. Hier kommt der Aspekt der Balance ins Spiel, denn es ist weder effizient noch realistisch alle erforderlichen Tests zu automatisieren. Der Test-Engineer muss innerhalb des Sprints „ausbalancieren", welche Tests sinnvoll zu automatisieren sind, beziehungsweise wo das Kosten-Nutzen-Verhältnis für eine Automatisierung nicht passt. Wenn es einem Yogi an der notwendigen Balance fehlt, wird er unweigerlich bei bestimmten Gleichgewichtsübungen, wie dem Baum, umfallen. Fehlt es dem Test-Engineer an der notwendigen Balance, geht übermäßig Zeit für das Erstellen oder die Wartung von Automatisierungsskripten verloren, was sich negativ auf die Testabdeckung auswirkt. Bei einem Übermaß an automatisierten Tests passiert es häufig, dass wichtige Tests gestrichen werden, da diese nicht mehr im gegebenen Zeitrahmen automatisiert werden können. Gerade bei komplexen fachlichen Abläufen empfiehlt es sich oftmals, statt in eine Automatisierung besser in manuelle Tests zu investieren.

Die bewusste Entscheidung über die Balance zwischen automatisierten und manuellen Testes ist ein wesentlicher Faktor in agilen Projekten, da es ohne Testautomatisierung nur unter sehr hohen Aufwänden möglich ist, Regressionstests innerhalb der Sprints durchzuführen und so die Stabilität der Software zu gewährleisten.

All dies zeigt, welch hohe Erwartungen an die Mitarbeiter der Testabteilung in agilen Projekten gestellt werden. Der ideale Test-Engineer, der über das notwendige fachliche Wissen, Teamfähigkeit und alle drei angesprochenen Eigenschaften verfügt, ist selten. Fehlt dazu die Möglichkeit, sich in diesen Aspekten weiterzuentwickeln, werden die wenigen gut ausgebildeten Test-Engineers bald zu den „bedrohten Arten" zählen. Es empfiehlt sich, stets zwei Personen für das Testing im Team einzuplanen, um gegenseitig von den Fähigkeiten des Anderen zu profitieren und zu lernen. Weitere Möglichkeiten sind ein regelmäßiger Austausch über Projektgrenzen hinweg, qualifizierte Aus-/Weiterbildungen auch im Bereich der Softskills und Zertifizierungen sowohl im Testing als auch im agilen Vorgehen.

Was bedeutet das für die Testorganisation der Zukunft?

Testmanagement und operative Testaufgaben vermischen sich zusehends. Eine klare Trennung von Testmanagement und Testentwicklung, wie sie in „klassischen" Projekten zu beobachten ist, wird es innerhalb von Scrum-Teams in dieser Form nicht mehr geben.

Zudem wird das Thema Testautomatisierung weiter an Bedeutung gewinnen. Je nach Organisationsform sollte man sich bewusst entscheiden, ob die erforderliche Testautomatisierung innerhalb der Sprints umgesetzt oder dort „nur" definiert wird, welche Tests zu automatisieren sind. Falls dies der Fall ist, wird die Testautomatisierung außerhalb des Scrum-Teams, im besten Fall parallel, umgesetzt. Die automatisierten Testskripte können dann im nächsten oder übernächsten Sprint für die Regressionstests verwendet werden.

Auch die technischen Fähigkeiten der Test-Engineers werden in agilen Vorgehensmodellen immer wichtiger. So werden sich „Tester mit Entwicklergenen" in einem Scrum-Team sicher besser behaupten können, als solche, die nie selbst programmiert haben.

Dies bedeutet aber nicht, dass nun alle Aufgaben, so wie sie zuvor gelebt wurden, falsch waren und abgeschafft werden sollten. Denn nur in den wenigsten Unternehmen wird heute ausschließlich nach Scrum gearbeitet, zumeist handelt es sich um Mischformen von klassischen und agilen Modellen.

Fazit – Scrum ist vor allem eine Chance

Das Thema Scrum stellt IT-Verantwortliche vor neue Herausforderungen. Die beschriebenen Risiken dürfen nicht unterschätzt werden, dennoch sollte Scrum in erster Linie als Chance gesehen werden, vor allem um Software-Testing und -entwicklung näher zusammenzubringen. In Projekten mit eigenständigen Test- und Entwicklungsteams bilden sich vielfach verhärtete Fronten, die oft nur schwer zu überwinden sind.

Scrum bietet nicht nur die Möglichkeit, diese Bereiche wieder zu einen, sondern darüber hinaus auch qualitativ hochwertige Software zu liefern. Zugegeben, die Herausforderungen wachsen, vor allem da die Aufgaben und Anforderungen zunehmend breiter werden. Wer aber die notwendigen Eigenschaften und Fähigkeiten moderner Test-Engineers im Scrum-Umfeld kennt und bei der Auswahl und Weiterentwicklung berücksichtigt, wird auch zukünftig erfolgreich sein. Wichtig ist, den Mitarbeitern die Chance zu geben, sich zu entwickeln. Ähnlich dem Yoga kommt es letztendlich auf die Ausgewogenheit an: Die Eigenschaften Flexibilität, Stärke und Balance müssen alle vorhanden sein und in einem sinnvollen Verhältnis zueinander stehen.

Sven Schirmer ist Bereichsleiter Testmanagement bei Maiborn-Wolff et al. Er kann auf mehr als dreizehn Jahre Erfahrung in der IT-Branche, sowohl aus Sicht der Entwicklung und als Projekt- und Testmanager zurückblicken. In kritischen Großprojekten verantwortet er das Testmanagement oder berät die Projektleitung bei methodischen oder strukturellen Fragen in Testthemen. Weitere Schwerpunkte sind Beratung und Coaching von Testorganisationen und der Aufbau sowie die Optimierung von Testabteilungen in Unternehmen oder großen Projekten. In seiner Freizeit praktiziert er schon seit mehreren Jahren Yoga.

Sind Benutzerbedürfnisse die wahren Anforderungen?

Für den Benutzer

Wann haben Sie sich zuletzt über ein schlecht bedienbares System geärgert? Wahrscheinlich ist das noch nicht so lange her, und Sie haben es auch so schnell nicht mehr benutzt. Dabei werden viele Systeme doch für Benutzer entwickelt. Woran liegt es also, dass sie so schlechte Usability aufweisen?

von Dirk Schüpferling und Monika Popp

Schlechte Usability liegt am häufigsten daran, dass die Entwickler die zukünftigen Benutzer und ihre Bedürfnisse und Verhaltensweisen gar nicht kennen. Sie wissen nicht, wie, warum und in welcher Umgebung Menschen Systeme überhaupt benutzen. Dazu kommt auch, dass die Disziplin Requirements Engineering oft als reine Ermittlung von Funktionalitäten eines Systems betrachtet wird. Wird der Fokus zu früh auf die Systemfunktionalitäten gelegt, wird der Usability-Aspekt häufig vernachlässigt. Diese Problemstellung kann mithilfe einer systematischen Integration von Usability-Engineering-Methoden in das Requirements Engineering gemildert werden (im Weiteren auch als „kombinierte Variante" bezeichnet). Eine Möglichkeit dazu besteht darin, die Benutzer genau zu analysieren, ihre Bedürfnisse und Verhaltensweisen zu extrahieren, um daraus anschließend Anforderungen an das System abzuleiten (**Abb. 1**).

Um das Ganze nicht so abstrakt zu beschreiben nun ein Beispiel. Für eine Restaurantkette soll ein System entwickelt werden, das die Tätigkeiten des Kellners größtenteils ersetzt (z. B. Bestellung entgegennehmen, Weinempfehlung aussprechen). Neben den Tätigkeiten des Kellners soll das System auch administrative Tätigkeiten der weiteren Angestellten systemisch unterstützen (z. B. Lebensmittel bestellen, Urlaub planen).

Bedürfnisse und Verhaltensweisen der Benutzer

Im klassischen Requirements Engineering werden alle Stakeholder (Projektbeteiligte) mithilfe der Stakeholder-Checkliste ermittelt und befragt, da eine unvollständige Stakeholder-Analyse zu einer lückenhaften Anforderungsspezifikation führen kann. Die Stakeholder im Restaurantbeispiel sind zum einen die Mitarbeiter des Restaurants sowie alle Gäste, aber auch Wartungskräfte und Entwickler. Bei der Befragung der Stakeholder im klassischen Requirements Engineering wird der Fokus meist weniger auf die Bedürfnisse und Verhaltensweisen der Benutzer, als auf Systemanforderungen gelegt.

Anders dagegen das Vorgehen im Usability Engineering. Dort werden zu Beginn der Systementwicklung Markt- und Designforschung betrieben. Denn bevor ein Produkt den Kundenwünschen nach entwickelt werden kann, müssen erst verschiedene Daten der potenziellen und tatsächlichen Benutzer des Produkts vorliegen. So steht in der so genannten Research-Phase (Forschungsphase) die Suche nach Bedürfnissen, Ansichten und Zielen der Benutzer im Mittelpunkt [1]. Auf das Restaurantsystem bezogen, werden die Benutzer des Systems (Mitarbeiter und Gäste) nach ihren Aktivitäten, Zielen, technischen Einstellungen, kognitiven Fähigkeiten, ihrem Umgang mit IT-Systemen aber auch ihren Ängsten befragt. Besonders wichtig ist das mentale Modell der Benutzer – ihre Vorstellung wie das Restaurantsystem funktioniert – zu erforschen. Denn liegt die technische Umsetzung des Restaurantsystems später nahe bei den Vorstellungen der Benutzer, können diese das System intuitiv bedienen. Wichtig bei der Beobachtung und Befragung von Benutzern ist es, dass sie in ihrer Umgebung, dem so genann-

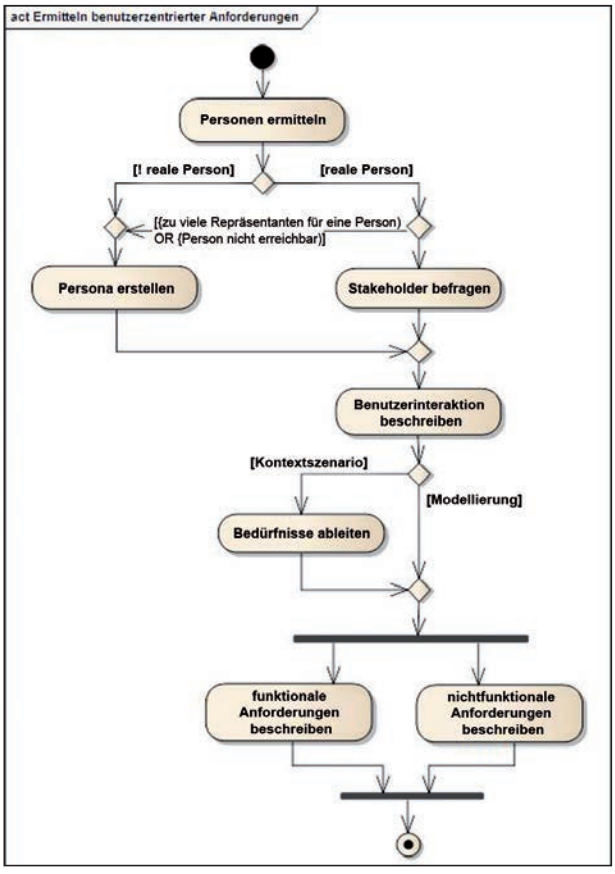


Abb. 1: Ablaufdiagramm zum Ermitteln benutzerzentrierter Anforderungen

ten „Nutzungskontext“, betrachtet werden. So wird der Koch bei seinen täglichen Arbeitsschritten in der Küche beobachtet und gegebenenfalls dazu befragt.

Die Ergebnisse der Research-Phase werden mithilfe von Personas, einem Modell von Alan Cooper [2], zusammengefasst und visualisiert. Personas sind keine realen Personen, sondern User-Archetypen, die die verschiedenen Ziele und beobachteten Verhaltensmuster der pozentiellen Benutzer beschreiben. Mithilfe von Personas kann im Projektteam über verschiedene Benutzertypen und ihre Bedürfnisse kommuniziert werden, um dann zu entscheiden, welcher für das Design von Form und Verhalten des Systems am wichtigsten ist.

Im Restaurantbeispiel wurden mithilfe der Personas verschiedene Typen von Gästen identifiziert, zum Beispiel „Stammgast“ und „Single“. Die Persona „Stammgast“ besucht regelmäßig das Restaurant, reserviert meist für zwei Personen und genießt am liebsten die regionale Küche. Die Persona „Single“ besucht alleine das Restaurant und erfreut sich am hohen Unterhaltungswert (Livemusik), da sie bei diesen Veranstaltungen leicht mit anderen Gästen ins Gespräch kommt. Hier wurden Personas erstellt, da die Rolle „Gast“ von zu vielen realen Personen repräsentiert wird, die nicht alle befragt werden können. Sind Stakeholder sehr schwer erreichbar oder existieren im Moment noch keine realen Stakeholder für ein Interview (zukünftiger Markt), so werden auch in diesen Fällen Personas erstellt. In allen anderen Fällen werden keine Personas erstellt, sondern alle realen Stakeholder, wie im klassischen Requirements Engineering, befragt.

Kombinieren wir nun die gezeigten Vorgehen der Disziplinen Requirements Engineering und Usability Engineering, so gelangen wir zu einer vollständigen und fundierten Benutzeranalyse, aus der im nächsten Schritt die Benutzerbedürfnisse extrahiert werden können. Nicht nur aus den Persona-Dokumenten, sondern auch aus den geführten Stakeholder-Interviews können nun die Bedürfnisse und Verhaltensweisen der Benutzer abgeleitet werden.

Im klassischen Requirements Engineering werden nun die Benutzeraktivitäten – das sind die Tätigkeiten des Benutzers – ihre Häufigkeit und ihr Umfang zum Beispiel zu einem Use-Case-Diagramm modelliert. In unserem Beispiel verwenden wir ein Use-Case-Diagramm, da es eine gute Basis für die weiteren Schritte im Requirements Engineering ist. So wird anschließend jeder Use Case mithilfe der Use-Case-Schablone genauer beschrieben. Die Beschreibung enthält standardmäßig folgende Attribute: Name, Kurzbeschreibung, Akteure, Vorbedingungen, fachlicher Auslöser, Normalfall und Ergebnis [3]. Tabelle 1 zeigt ein Beispiel einer Use-Case-Beschreibung für den Use Case „Urlaub planen“ des Kochs.

Das Usability Engineering verwendet Kontextszenarien, um die Benutzeraktivitäten mit dem System darzustellen. Kontextszenarien sind knappe, erzählerische Beschreibungen im Fließtext, die die ideale Interaktion mit dem System aus Sicht der Persona beschreibt.

Use Cases und Kontextszenarien sind Methoden, die die Interaktion von Benutzern mit einem System beschreiben und sich nur in der Dokumentationsform

Tabelle 1: Use Case „Urlaub planen“

Name:	Urlaub planen
Kurzbeschreibung:	Der Koch hat die Möglichkeit, Urlaub zu planen. Dazu muss er die gewünschten Urlaubstage in den Urlaubskalender eintragen.
Akteure:	Koch
Vorbedingungen:	Koch ist am System angemeldet.
Fachlicher Auslöser:	Der Koch hat den Wunsch, Urlaub zu planen.
Normalfall:	Koch ruft Urlaubskalender auf. System zeigt Urlaubskalender mit Urlaubseinträgen der anderen Mitarbeiter an. Koch wählt die Darstellung „Schulferien einblenden“ aus. System zeigt die Schulferien im Urlaubskalender an. Koch wählt Urlaubstage aus. Koch bestätigt die Eingabe. System prüft den Urlaubseintrag.
Ergebnis:	Koch hat die Urlaubstage in den Urlaubskalender eingetragen und wartet auf eine Benachrichtigung des Systems.

Tabelle 2: Extraktion von Bedürfnissen

Szenariotext	Bedürfnisse
Der Koch Tommaso Zanolla möchte mit seiner Familie zwei Wochen am Stück in Urlaub fahren.	Möglichkeit bieten, Urlaubstage zusammenhängend zu wählen
Da seine beiden Kinder schulpflichtig sind, orientiert sich Tommaso an den Schulferien.	Möglichkeit bieten, sich an Schulferien zu orientieren
Die Schulferien werden im Urlaubskalender eingeblendet.	Schulferien in Urlaubskalender einblenden
Tommaso wählt zwei Wochen in den Sommerferien aus und bekommt einige Stunden später vom System die Rückmeldung, dass sein Urlaub genehmigt wurde.	Urlaub auswählen Rückmeldung vom System liefern Urlaubsgenehmigung anzeigen

unterscheiden. Dadurch werden in der kombinierten Variante beide Methoden eingesetzt, um aus Benutzeraktivitäten funktionale Anforderungen abzuleiten. Der Ablauf (= Normalfall) einer Use-Case-Beschreibung kann nun entweder modelliert oder mit einem Kontextszenario beschrieben werden. Kontextszenarien sind in einigen Fällen sehr nützlich, um mit dem Nutzer noch einmal seine Aktivitäten und Bedürfnisse durchzusprechen, da sie den Ablauf leicht verständlich und sehr konkret darstellen. In dem Fall, dass ein Kontextszenario erstellt wurde, werden daraus anschließend Bedürfnisse abgeleitet.

Persona:	**Koch**
Mentales Modell:	**Bedürfnisse:**
Vorstellung des Systems; Der Koch hat die Vorstellung, dass das System wie ein Stapel Papier aufgebaut ist. In dem Stapel Papier befinden sich alle Tabellen und Listen, die es jetzt in Papierform auch gibt.	Das System soll alle Tabellen und Listen, die genutzt werden, digital visualisieren.
Umgebung (physisch):	
Geräuschkulisse; In der Küche läuft meistens Radio im Hintergrund. Bei verschiedenen Arbeitsschritten ist der Geräuschpegel lauter (Spülen, Braten, Geschirr einräumen, Mixen …). Außerdem gibt es einige Geräte, die Töne von sich geben (Gefriertruhe, falls zu lange offen, Backofen, nach eingestellter Zeit …).	Das System soll Geräte, die Töne von sich geben, übertönen. Das System soll das Radio übertönen. Das System soll die Geräusche von Arbeitsschritten, bei denen der Geräuschpegel überdurchschnittlich laut ist, übertönen.
Arbeitsplatzumgebung; Der Koch hat in der Küche verschiedene Arbeitsplätze. Diese sind in Lebensmittel- und Schmutzbereich unterteilt. Im Lebensmittelbereich müssen verschiedenste Hygienemaßnahmen durchgeführt werden, um mit Lebensmitteln zu arbeiten. Flüssigkeiten, Fettspritzer und extreme Temperaturen stellen eine Gefahr für Endgeräte dar. Der Koch legt bestimmte Wege in der Küche zurück.	Das System soll im Lebensmittelbereich keinen Schmutz verursachen. Das System soll im Lebensmittelbereich funktionieren. Das System soll unter Einfluss von Flüssigkeiten, Fettspritzern und extremer Temperaturen (Hitze, Kühlung) funktionieren.
Physische und kognitive Eigenschaften:	
Verarbeitung von Informationen; Der Koch ist im visuellen Lernen sehr gut. Er kann somit Sachverhalte und Abläufe, bei denen er zusieht, sehr gut und schnell verarbeiten und auch adaptieren.	Das System soll Sachverhalte und Abläufe in Form von Videos oder Animationen darstellen können.
Fertigkeiten und Kenntnisse:	
Bildungsstand; Der Koch besitzt eine abgeschlossene Berufsausbildung.	Das System soll für Benutzer mit einer abgeschlossenen Berufsausbildung benutzbar sein.
Umgang mit dem Computer; Der Koch besitzt privat einen Computer und nutzt diesen zur täglichen Administration (E-Mails, Word, Excel, Fotos hochladen). Außerdem nutzt der Koch unregelmäßig soziale Netzwerke. Bei kleineren technischen Problemen braucht er keine fremde Hilfe.	Das System soll für Benutzer, die den PC zur täglichen Administration nutzen, gut bedienbar sein. Das System soll für Benutzer, die unregelmäßig soziale Netzwerke nutzen, gut bedienbar sein.
Mobiltelefon; Der Koch besitzt ein Smartphone und benutzt dieses öfters als seinen Computer, um ins Internet zu gehen.	Das System soll für Benutzer, die ein Smartphone haben und dieses zum Surfen im Internet nutzen, gut bedienbar sein.
Ziele:	
Zentrale Einkaufsliste; Der Koch möchte zu jeder Zeit auf die „Lieferantentabelle“ restaurantübergreifend zugreifen können, um schneller arbeiten zu können.	Das System soll die Lieferantentabelle jederzeit restaurantübergreifend anzeigen.
Vorratshaltung anzeigen; Der Koch möchte, dass der Vorratsbestand vom System angezeigt wird.	Das System sollte den Vorratsbestand anzeigen.
Alle Spezialitäten anzeigen; Der Koch möchte seine bisherigen Spezialitäten angezeigt bekommen, um sich seiner Kreativität bewusst zu werden.	Das System soll die bisherigen Spezialitäten anzeigen. Das System soll dem Benutzer dabei ein gutes Gefühl geben.
Preisvergleich beim Einkauf; Der Koch möchte beim Einkauf einen Preisvergleich der Produkte angezeigt bekommen, um günstiger einkaufen zu können.	Das System soll beim Einkauf einen Preisvergleich der Produkte anzeigen.

Tabelle 3: Extraktion von Bedürfnissen aus der Persona „Koch“

Was sind Bedürfnisse und wie werden sie extrahiert?

Bedürfnisse können auch mit dem Begriff „Kundenwünsche“ gut übersetzt werden, da es sich dabei um noch nicht geprüfte Anforderungen handelt. Bedürfnisse sind somit eine Vorstufe zu Anforderungen, wie sie im Requirements Engineering definiert sind. Um Bedürfnisse aus einem Kontextszenario zu extrahieren, wird eine Tabelle mit zwei Spalten angelegt. In der linken Spalte befindet sich pro Zeile ein Satz des Kontextszenarios. In der rechten Spalte werden die daraus erhobenen Bedürfnisse stichpunktartig dokumentiert. Tabelle 2 zeigt die Extraktion von Bedürfnissen aus dem Kontextszenario „Urlaub planen“ des Kochs.

Die extrahierten Bedürfnisse werden anschließend mit den Ergebnissen der Modellierung auf neue Funktionalitäten hin überprüft. Gehen neue Funktionalitäten aus den Bedürfnissen hervor, so werden diese mit aufgenommen. Danach startet das klassische Requirements-Engineering-Vorgehen, das aus Funktionalitäten funktionale Anforderungen erstellt. Dabei spielt es keine Rolle, ob die Aktivitäten zuvor als Szenarien oder als Use Cases dokumentiert wurden.

Neben funktionalen Anforderungen gibt es im Requirements Engineering auch nichtfunktionale Anforderungen (NFA), die in folgende Kategorien eingeteilt werden: technische Anforderungen, Anforderungen an die Benutzerschnittstelle, Qualitätsanforderungen, Anforderungen an Lieferbestandteile, Anforderungen an durchzuführende Tätigkeiten und rechtlich-vertragliche Anforderungen. Zum Beispiel in der Kategorie „Anforderungen an die Benutzerschnittstelle“ wird eine Menge von Faktoren beschrieben, die sich mit dem visuellen, akustischen oder auch haptischen Erscheinen und der Bedienung des Produkts befassen. Sie ergänzen die funktionalen Anforderungen – zumindest diejenigen, die sich an der Oberfläche zeigen – und spezifizieren, wie die funktionalen Anforderungen zu erscheinen haben. Darunter fallen Bedienkonzept, Rahmenbedingungen für die Gestaltung der Benutzungsoberfläche und Bedienelemente.

Im Usability Engineering werden nun alle restlichen Attribute der Personas, wie mentales Modell, physische Umgebung, Ängste und Ziele auf Bedürfnisse überprüft. Das Vorgehen ist hier das gleiche wie bei den Kontextszenarien. Es wird eine Tabelle mit zwei Spalten angelegt, in der links pro Zeile die Attribute eingetragen und rechts die Bedürfnisse daraus extrahiert werden. Hilfreich ist es hierbei, wenn die Bedürfnisse mittels ganzer Sätze formuliert werden. Der Sinneszusammenhang kann dadurch besser wiedergegeben werden, was zu weniger Missverständnissen führt. In Tabelle 3 sind Ausschnitte extrahierter Bedürfnisse aus der Persona Koch zu sehen.

Im kombinierten Vorgehen werden nicht nur die restlichen Attribute der Personas auf nichtfunktionale Anforderungen hin überprüft, sondern auch die gesamten Interviewdokumente und alle bereits verwendeten Informationen (Benutzeraktivitäten, Kontextszenarien, Use-Case-Beschreibungen). Im Usability Engineering werden wie gezeigt normalerweise aus den restlichen Persona-Attributen Bedürfnisse abgeleitet. Der Dokumentationsschritt dazu fällt in der kombinierten Variante weg,

Tabelle 4: Ausschnitt eines NFA-Dokuments

Technische Anforderungen:
Das System soll ein Gewicht von max. 200 g aufweisen. (Quelle: Koch)
Das System muss bei Temperaturen von min. -15°C bis max. +50°C fehlerfrei funktionieren. (Quelle: Koch, physische Umgebung)
Das System soll unter Einfluss von Flüssigkeiten und Fettspritzern fehlerfrei funktionieren. (Quelle: Koch, physische Umgebung)
Anforderungen an die Benutzerschnittstelle:
Die Benutzeroberfläche des Systems soll in den Sprachen Deutsch, Englisch und Italienisch zur Verfügung gestellt werden. (Quelle: Koch, Stammgast)
Die Bedienung des Systems soll konform zu MS Office gestaltet sein. (Quelle: Koch, Fertigkeiten und Kenntnisse)
Qualitätsanforderungen:
Das System soll für Benutzer, die den PC zur täglichen Administration nutzen, innerhalb von max. 10 Minuten erlernbar sein. (Quelle: Stammgast)
Das System soll so gestaltet sein, dass die Gesundheit von Menschen in keinster Weise gefährdet wird. (Quelle: Geschäftsführer)
Das System soll unabhängig vom lauten Geräuschpegel gut zu bedienen sein. (Quelle: Koch, physische Umgebung)
Das System soll dem Benutzer bei der Bedienung ein gutes Gefühl vermitteln. (Quelle: Stammgast)
Anforderungen an Lieferbestandteile:
Das Betriebshandbuch muss ein Datensicherungskonzept enthalten. (Quelle: Geschäftsführer)
Das Onlinehilfesystem soll Bilder und Videos zur Veranschaulichung der funktionalen Beschreibungen enthalten. (Quelle: Koch, physische und kognitive Eigenschaften)
Anforderungen an durchzuführende Tätigkeiten:
Der Auftragnehmer soll den Auftraggeber über das für die Softwareentwicklung verwendete Vorgehensmodell umfassend informieren. (Quelle: Geschäftsführer)
Der Auftragnehmer soll alle Projektrisiken identifizieren und einschließlich ihres Gefährdungspotenzials (Eintrittswahrscheinlichkeit, Schadenshöhe) in einem Risikokatalog dokumentieren. (Quelle: Geschäftsführer)
Rechtlich-vertragliche Anforderungen:
Alle Dokumente sind in deutscher Sprache nach den Regeln der neuen deutschen Rechtschreibung zu liefern. (Quelle: Geschäftsführer)
Das System soll so gestaltet sein, dass das Jugendschutzgesetz eingehalten wird. (Quelle: Jugendschutzbeauftragter)

um ein Ergebnisartefakt einzusparen. Somit werden gleich nichtfunktionale Anforderungen formuliert, die anschließend anhand der sechs NFA-Kategorien aus dem Requirements Engineering eingeteilt und auf Vollständigkeit geprüft werden. Die NFA-Kategorien dienen dabei als Checkliste, da sie generisch wiederverwendbar sind. Die erhobenen nichtfunktionalen Anforderungen werden im nächsten Schritt noch nach Requirements-Engineering-Methoden qualitätsgesichert. Tabelle 4 zeigt einen Ausschnitt eines NFA-Dokuments.

Nun ist ersichtlich, dass die klassische Anforderungserhebung durch die Integration von Usability-Engineering-Methoden optimiert wird, sodass eine qualitativ hochwertige Anforderungsspezifikation erreicht werden kann. Die beiden Disziplinen ergänzen sich in ihrer Vorgehensweise perfekt, indem das Usability Engineering an vielen Stellen immer wieder die Kreativität des Analytikers anregt und das Requirements Engineering gleichzeitig die nötige Struktur liefert. Speziell die Benutzeranalyse liefert essenzielle Daten wie Ziele, Verhaltensweisen und Bedürfnisse der Benutzer, die im klassischen Requirements Engineering nicht so detailliert erhoben werden. Die Analyse und Modellierung von Benutzern ist somit eine wichtige Anforderungsquelle für funktionale und nichtfunktionale Anforderungen, da sie bei Entwicklungs- und Designentscheidungen hilft und die Sichtweise erweitert. Die Kombination der beiden Disziplinen erweitert nicht nur den Blickwinkel auf das System, sondern bringt Vollständigkeit für die weiteren Entwicklungsphasen und sichert die Akzeptanz der Nutzer. Außerdem fördert eine Verschmelzung der beiden Disziplinen das Miteinander der Projektteilnehmer langfristig.

Dirk Schüpferling (www.sophist.de/dirk.schuepferling) ist bereits seit 2001 als Berater und Trainer bei den SOPHISTen. Er forscht auf dem Gebiet der Dokumentationstechniken und löst hier Problemstellungen aus der Praxis, indem er Neuerungen und Anpassungen an etablierten Tools und Dokumentationsarten vornimmt.

Monika Popp (http://www.sophist.de/monika.popp) unterstützt als Beraterin Kunden bei der Ermittlung, Dokumentation und Verwaltung von Anforderungen. Ergänzend zur beratenden Tätigkeit erforscht sie, wie sich die Disziplinen Usability Engineering und Requirements Engineering sinnvoll ergänzen und kombinieren lassen.

Links & Literatur

[1] Goodwin, Kim: „Designing For The Digital Age“, Wiley Publishing, Inc., Indianapolis, 2009

[2] Cooper, Alan/Reimann, Robert/Cronin David: „About Face – Interface und Interaction Design“, mitp, Heidelberg, 2007

[3] Rupp, Chris/die SOPHISTen: „Requirements-Engineering und -Management“, Hanser, Nürnberg, 2009

Dokumentenverarbeitung mit dem docx4j-Framework

What's up, Doc?

„docx4j" ist ein Java-Open-Source-Framework, das hauptsächlich von der Plutext Pty Ltd. mit Sitz in Melbourne (Australien) entwickelt wird. Es dient in erster Linie als Schnittstelle zwischen einer in Java geschriebenen Geschäftsanwendung und den XML-basierten Microsoft-Dokumentformaten (Open XML). Dabei wird zur Erstellung der Java-Klassen auf JAXB zurückgegriffen. Das ermöglicht es Entwicklern, bequem auf Dokumente zuzugreifen, Informationen auszulesen, Dokumente zu manipulieren und diese auch in verschiedenen Formaten abzulegen. Die Komplexität der Verarbeitung kann dabei von der programmatischen Erstellung von Serienbriefen bis hin zu komplexen Geschäftsberichten oder Vertragsdokumenten reichen.

von Lars Drießnack

Im Gegensatz zu automatisierten Textverarbeitungsprozessen außerhalb einer Geschäftsanwendung können mit der Verwendung von docx4j als Bibliothek der Applikation alle Datenquellen genutzt werden, die die etwaige Geschäftsanwendung selbst ebenfalls nutzt, da die Verarbeitung an Ort und Stelle durchgeführt wird (potenzielle Verwendung, **Abb. 1**). Es entfallen somit aufwändige Import/Export-Prozesse sowie Schnittstellen zur Datenbereitstellung nach außen. Die Modellierung und Formatierung des gewünschten Ausgabedokuments obliegt grundsätzlich dem Entwickler, kann aber durch die Verwendung von Vorlagen vereinfacht und gesteuert werden. Die aktuelle Version 2.7.1 wurde im Oktober 2011 veröffentlicht. Sie unterstützt die Versionen MS Word 2007 und 2010, da ab Word 2007 Dokumente im XML-basierten Containerformat *docx* gespeichert werden können. Jason Harrop, CEO von Plutext und Chefentwickler, betreibt die Internetseite www.docx4java.org, die Foren, Beispielprojekte und Blogs rund um das Framework enthält.

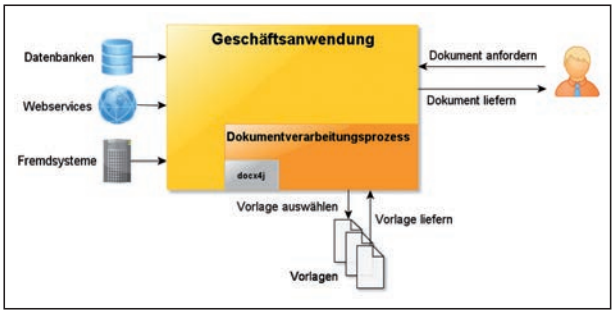


Abb. 1: Potenzielle Verwendung von docx4j

Wichtige Klassen

Im docx4j-Framework sind grundsätzlich alle Elemente der WordML als Klassen repräsentiert. Es empfiehlt sich, bei Verwendung spezieller Klassen zunächst ein Beispieldokument zu erstellen und sich den XML-Code des Elements und die notwendigen Bestandteile anzusehen. Im Folgenden werden einige wesentliche Klassen und deren Verwendung erläutert.

WordprocessingMLPackage ist die docx4j-Repräsentation des *docx*-Dokuments (im Folgenden nur als *wmlPackage* bezeichnet). Sie kann entweder durch Laden eines bestehenden Dokuments oder durch Erzeugung einer neuen Repräsentation eines noch zu definierenden Dokuments erstellt werden. Laden und Erzeugen einer Instanz dieser Klasse erfolgt über die statischen Methoden *load(File docxFile)* und *createPackage()*. Die Methode *createPackage()* übernimmt dabei die Teilschritte:

- Erzeugung des *wmlPackage*-Objekts
- Erzeugung eines *MainDocumentPart*-Objekts
- Erzeugung eines *Body*-Objekts mithilfe der *ObjectFactory*
- Erzeugung eines *Document*-Objekts mithilfe der *ObjectFactory*
- Zusammenstellung der Objekte zum *wmlPackage*

Die Speicherung eines *wmlPackages* erfolgt über die Methode *save(File docxFile)* der jeweiligen Instanz.

Sdt-Klassen sind in MS Word besser bekannt als „Inhaltssteuerelemente“ und können zur Ausgabe und Formatierung im Dokument verwendet werden. Je nach Kontext, in dem ein Inhaltssteuerelement verwendet wird, kann es verschiedene Ausprägungen (Java-Klassen) annehmen. Die geläufigsten sind:

- *SdtBlock*
- *SdtRun*
- *CTSdtCell*
- *CTSdtRow*

In der Version 2.6.0 des Frameworks konnten diese Klassen nicht über eine gemeinsame Basisklasse oder ein Interface angesprochen oder verarbeitet werden und stellten somit trotz der Tatsache, dass es sich immer um Inhaltssteuerelemente handelte, vollkommen eigenständige Klassen dar. Im Zuge des Projekts zur Integration von Word 2007 in die bestehende Geschäftsanwendung bei der Sächsischen Aufbaubank – Förderbank hat die Saxonia Systems AG einige Änderungen und Erweiterungen des Frameworks angeregt, die zu großen Teilen übernommen wurden. Eine dieser Änderungen war die Einführung des *SdtElement Interfaces*, das ab der Version 2.7.0 von allen *Sdt-Klassen* implementiert wird. Dadurch wird die Verarbeitung von Inhaltssteuerelementen deutlich vereinfacht.

CTCustomXml-Klassen stellen in Microsoft Word alle die Elemente dar, die nicht Bestandteil der WordML sind und daher auch nicht Bestandteil von Microsoft Word. Elemente dieser Art können durch das Einbinden eines externen, selbst definierten *xsd*-Schemas in Word verwendet werden. Dies kann mithilfe der Entwicklertools in MS Word problemlos realisiert werden. Zur Verwendung der Bestandteile im Java-Code muss das Schema als solches nicht bekannt sein; jedoch sind zur Verarbeitung dieser Elemente Kenntnisse über die jeweiligen Attribute zwingend notwendig.

Analog zu den *Sdt*-Klassen werden diese Klassen je nach dem Kontext der Verwendung durch unterschiedliche Klassen in docx4j repräsentiert, beispielsweise als *CTCustomXMLBlock* oder *CTCustomXMLRun*. Diese Klassen konnten bis zur Version 2.6.0 ebenfalls nicht einheitlich verarbeitet werden. Daher wurde auch hierfür ein entsprechendes Interface durch die Saxonia Systems AG vorgeschlagen und mit der Version 2.7.0 in docx4j übernommen.

„Hello World“ – Ein einfaches Dokument bearbeiten

Wie bereits erwähnt, besteht mit docx4j die Möglichkeit, ein Dokument quasi aus dem Nichts zu erzeugen. Dieses Verfahren ist jedoch enorm aufwändig und geht weit über das Erzeugen und Zusammenstellen von Java-Objekten hinaus. Um ein funktionierendes Word-Dokument vom Java-Quellcode her anzulegen, müssen neben Optik und Funktionalität auch Richtlinien der WordML beachtet werden, die im Framework selbst keine Berücksichtigung finden.

Methoden des Interface „SdtElement“

- *get/setSdtPr* und *get/setSdtEndPr* für die Klassen *SdtPr* und *CTSdtEndPr*, die Eigenschaften der *Sdt*-Klassen enthalten
- *getContent* für die Klasse *SdtContent*, die über eine gleichnamige Methode verfügt und so den einheitlichen Zugriff auf die Liste der in der *Sdt-Klasse* enthaltenen Objekte ermöglicht

Methoden des Interface „CTCustomXmlElement“

- *get/setCustomXmlPr* für die Klasse *CTCustomXmlPr*, die Eigenschaften der *CTCustomXml*-Klassen enthält
- *get/ setUri* zum Auslesen bzw. Festlegen des Schema-URL
- *get/setElement* zum Auslesen bzw. Festlegen des Elementtyps; dieser wird normalerweise durch das XML Schema vorgegeben
- *getContent* – diese liefert eine Liste der im Element enthaltenen Objekte

Best Practice ist es daher, einen Workflow zu entwerfen, der als Grundlage eine mit MS Word erstellte Vorlage verwendet und diese anhand von Daten bearbeitet und gestaltet. Im folgenden Beispiel wird anhand einer Datenquelle entschieden, ob als Anrede „Herr“ oder „Frau“ in einem Dokument mit einem Inhaltssteuerelement namens „Anrede“ ausgegeben werden soll.

Neben Inhaltssteuerelementen können natürlich auch alle anderen Elemente der Word ML beliebig manipuliert werden. Beispielsweise könnte man den Ausgabetext auch statt in ein Inhaltssteuerelement in einen Text, einen Paragraphen oder in eine Tabellenzelle schreiben.

Listing 1

```
private void setSalutation(int gender){
  // loading the template file
  File docxFile = new File("myTemplate.docx");
  WordprocessingMLPackage template = WordprocessingMLPackage.load(docxFile);
  // initialisation of SdtUtility
  SdtUtility sdtUtil = new SdtUtility(template);
  // find SdtElement "Anrede"
  SdtElement anrede = sdtUtil.getSdtElementByTagName("Anrede");
  // insert text
  if (gender == 0) {
    sdtUtil.setContentText(anrede, "Herr");
    return;
  } else {
    sdtUtil.setContentText(anrede, "Frau");
    return;
  }
}
```

Listing 2

```
private void fillTable(final List<MyData> data) {
  File docxFile = new File("myTemplate.docx");
  WordprocessingMLPackage template = WordprocessingMLPackage.load(docxFile);
  // initialisation of SdtUtility
  SdtUtility sdtUtil = new SdtUtility(template);
  // row counter
  int currentRow = 1;
  // get row of table
  SdtElement zeile = sdtUtil.getSdtElementByTagName("zeile");
  // get parent of row (should be the table)
  final Child parent = (Child)XmlUtils.unwrap(sdtAusgabeElement.getParent());
  // get content of row parent
  List<Object> parentContent = Docx4jUtility.getParentContent(zeile);
  // get index of row in parent content (in this simple example it should be 0)
  int indexOfContent = Docx4jUtility.getElementIndex(zeile);
  // fill table
  for (MyData date : data) {
// in case of the first iteration we do not need to copy something,
// in all other cases we have to copy the row
    if (currentRow != 1) {
      // create copy of row with tag suffix
      zeile = sdtUtil.copySdtElement(zeile, currentRow);
      zeile.setParent(parent);
      indexOfContent++;
      parentContent.add(indexOfContent, zeile);
      Docx4jUtility.fixParentRelationship(zeile, true);
    }
    insertRowData(template, data, currentRow);
    currentRow++;
  }
}

private WordprocessingMLPackage insertRowData
      (final WordprocessingMLPackage template,
       final MyData data, final int currentRow) {
  String index = "";
  if (currentRow != 1) {
    index = "#" + currentRow;
  }
  // initialisation of SdtUtility because content has changed
  SdtUtility sdtUtil = new SdtUtility(template);
  // get SdtElements of the current row
  SdtElement name = sdtUtil.getSdtElementByTagName("name" + index);
  SdtElement a = sdtUtil.getSdtElementByTagName("A" + index);
  SdtElement b = sdtUtil.getSdtElementByTagName("B" + index);
  // set values to the elements
  sdtUtil.setContentText(name, data.getName());
  sdtUtil.setContentText(a, data.getA());
  sdtUtil.setContentText(b, data.getB());
  return template;
}
```

Der wesentliche Vorteil von Inhaltssteuerelementen besteht aber darin, dass sie über die Attribute *Titel* und/oder *Tag* eineindeutig identifiziert werden können.

Iteratives Füllen einer Tabelle

Während beim oben genannten Beispiel viel an die Funktionalität des Serienbriefes in MS Word erinnert, wird im Folgenden beschrieben, wie eine Liste beliebiger Länge vom Java-Code in ein Word-Dokument überführt werden kann. Hierfür ist zunächst ein entsprechendes Template anzulegen, das die Tabelle und eine Musterzeile enthält.

In **Abbildung 2** sind die Inhaltssteuerelemente *tabelle*, *zeile*, *name*, *A* und *B* zu sehen. Ziel des Codebeispiels in Listing 2 ist es, die Tabelle entsprechend der vorhandenen Daten zu befüllen. Die Werte für *name*, *A* und *B* sind dabei als Attribute in der Klasse *MyData* enthalten.

Wie im Beispiel in Listing 2 zu sehen, ist es notwendig, die neu erzeugten Inhaltssteuerelemente mit einem

Die Klasse „Docx4jUtility“

Diese Klasse ist ebenfalls nicht im docx4j-Framework enthalten. Sie bietet einige grundlegende Funktionalitäten für alle Elemente der WordML. So beispielsweise das Auffinden des Elternelements und dessen Inhalts. Die wichtigste Funktionalität ist die Methode *fixParentRelationship()*. Diese „repariert“ die Parent-Child-Beziehungen, die durch die Manipulation des Dokuments verloren bzw. gelöscht worden sein könnten.

Die Klasse „SdtUtility“

Die in Listing 1 verwendete Klasse *SdtUtility* ist nicht im docx4j-Framework enthalten. Sie wurde von der Saxonia Systems AG entwickelt und stellt einige Basisfunktionen zur Verarbeitung von *SdtElementen* zur Verfügung. Bei Initialisierung mit einem *wmlPackage* wird dieses auf *SdtElemente* durchsucht. Die gefundenen Elemente werden in einer *Map* gehalten, sodass der Zugriff darauf sehr komfortabel möglich ist. Weiterhin können verschiedene Manipulationen – wie die im Beispiel zu sehende Veränderung des Inhalts eines *SdtElements* – vorgenommen werden.

Listing 3

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="textFieldExpression">
  <xs:complexType>
    <xs:attribute name="target" type="xs:string" use="required"/>
    <xs:attribute name="condition" type="xs:string" use="required"/>
  </xs:complexType>
  </xs:element>
</xs:schema>
```

Index zu versehen. Die verwendete Methode *copySdtElement(…)* der *SdtUtility* nutzt die im Framework vorhandene Klasse *XmlUtils* und deren Methode *deepCopy(…)*, um das Element zu kopieren. Zusätzlich wird der entsprechende Index an das Tag des Inhaltssteuerelements angehängt. Würde man diese Indexierung weglassen, würde neben der Tatsache, dass das Element in der *SdtUtility* nicht auffindbar wäre, zunächst kein Fehler auftreten. Jedoch würden beim Öffnen des Dokuments mit einer Word-Instanz sämtliche gleichnamigen *SdtElemente* gelöscht. An die Stelle der Inhaltssteuerelemente würde dann deren Inhalt rücken, was zu unerwünschten Nebeneffekten führen kann. Eine weitere Erweiterungsmöglichkeit dieses Beispiels ist die bedingungsabhängige Ausgabe von Zwischensummenzeilen sowie einer Summierungszeile am Ende.

tabelle Zeilenbezeichnung	Wert A	Wert B
zeile name name name	A a A	B b B zeile tabelle

Abb. 2: Eine Tabelle mit Inhaltssteuerelementen in einem MS-Word-Dokument

Sehr geehrte textFieldExpression anrede anrede anrede ...,

...

Abb. 3: Begrüßungszeile mit CustomXml- und Inhaltssteuerelement in einem MS-Word-Dokument

Bedingte Ausgabe von Daten mittels „CustomXml“

Wie der Name schon sagt, sind *CustomXml*-Elemente nutzerspezifisch, das heißt jeder kann diese definieren und verwenden, wann und wie er möchte – beispielsweise zum Einbinden von anderen Dokumenten in das bestehende, um einen modularen Aufbau der Dokumente zu erreichen, oder zur bedingten Formatierung oder Ausgabe von Daten. Um einen kurzen Einblick in die Möglichkeiten zu geben, wird das „Hello World“-Beispiel ein wenig abgewandelt.

Nun soll die Befüllung des *SdtElement* nicht im Java-Code geschehen, sondern der Wert soll anhand einer Bedingung aus einem *CustomXml*-Element definiert werden. Hierzu müssen Typ und Attribute des *CustomXml*-Elements zunächst in einem Schema festgelegt werden (Listing 3).

Damit man das Element *textFieldExpression* während der Erstellung des Word Templates verwenden kann, muss das Schema in der Word-Instanz über Entwicklertools | Schema | Schema hinzufügen eingebunden werden. Diese Verwendung könnte dann wie in **Abbildung 3** aussehen.

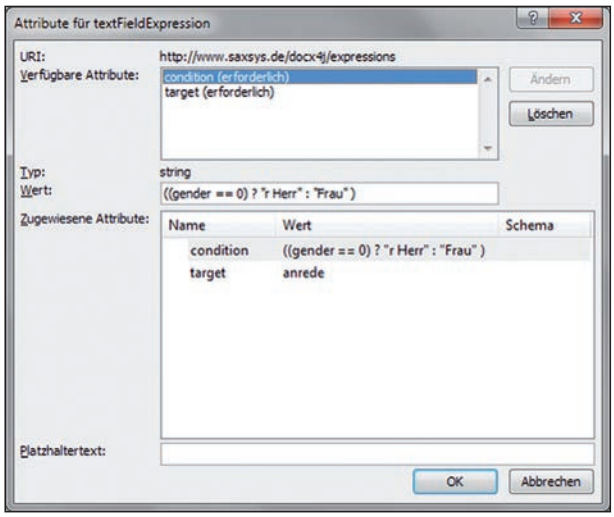


Abb. 4: Attributdefinition für CustomXml-Elemente in einem MS-Word-Dokument

Die in der *xsd*-Datei vergebenen Attribute können dann ebenfalls in Word definiert werden (**Abb.4**). Für dieses Beispiel gilt:

- *condition*: Java-Code, der zur Laufzeit ausgewertet werden soll
- *target*: Tag des *SdtElements*, das mit dem Ergebnis befüllt wird

Listing 4

```
private void proceedTextFieldExpressions(final Map<String, Parameter>parameter) {
  File docxFile = new File("myTemplate.docx");
  WordprocessingMLPackage template = WordprocessingMLPackage.load(docxFile);
  // initialisation of CustomXmlUtility
  CustomXmlUtility customXmlUtil = new CustomXmlUtility(template);
final List<CTCustomXmlElement> expressions =
        customXmlUtility.getExpressions();
  // processing of expression
  for (CTCustomXmlElement expression : expressions) {
  if (expression.getElement().equals(TEXTFIELDEXPRESSION)) {
    SdtUtility sdtUtil = new SdtUtility(template);
    // get target SdtElement
    String targetName =
      customXmlUtility.getCustomXmlAttributValue(expression, TARGET);
    SdtElement target = sdtUtil.getSdtElementByTagName(targetName);
    // evaluate expression
    String condition =
      customXmlUtility.getCustomXmlAttributValue(expression, CONDITION);
    ExpressionEvaluator evaluator =
       new ExpressionEvaluator(condition, parameter);
    String expResult = evaluator.evaluateString();
    sdtUtil.setContentText(target, expResult);
    }
  }
}
```

Unabhängig von der Elementdefinition im Schema sind mit dem docx4j-Framework zunächst alle *CustomXml*-Bestandteile gleich zu behandeln. Daher muss vor der Verarbeitung stets geprüft werden, um welchen Elementtyp es sich handelt (Listing 4).

Fazit

Das Framework ist in seiner gegenwärtigen Form ein gutes Hilfsmittel für die programmatische Verarbeitung von MS-Word-Dokumenten. Da *docx* selbst als Containerformat sehr komplex ist, sind die Möglichkeiten zur Manipulation von Dokumenten ebenso vielfältig wie unüberschaubar. Das volle Potenzial kann daher nur ausgeschöpft werden, wenn der Entwickler über umfangreiche Kenntnisse in Sachen WordML verfügt.

Wie in den Beispielen zu sehen, liegen die Schwächen des Frameworks ganz klar in fehlenden Utility-Klassen und Klassenhierarchien. Das liegt daran, dass die Entwicklung aufgrund der Nische, in der sie sich bewegt, eher langsam vorankommt, und an der WordML, die als Basisspezifikation Berücksichtigung finden muss.

Weiterhin sind einige Bestandteile von docx4j derzeit erst rudimentär ausgeprägt. Zu nennen ist hier in erster Linie die Speicherung von Dokumenten im HTML- und PDF-Format als auch der Zugriff und die Bearbeitung von Kopf- und Fußzeilen oder die Formatierung von Listen im Dokument. Das bedeutet nicht, dass eben genannte Funktionen nicht zur Verfügung stehen, sondern nur, dass diese nicht out of the Box vom Framework angeboten werden und sofort funktionieren. Hier ist schlicht zusätzlicher Programmieraufwand nötig, um fehlende Funktionalität umzusetzen. Hat man diesen Aufwand einmal betrieben – siehe oben genannte Utility-Klassen und Hilfsmittel – lassen sich sehr schnell sehr gute Ergebnisse erzielen.

Lars Drießnack ist Java Consultant bei der Saxonia Systems AG. Seine Schwerpunkte sind XML-basierte Dokumentenverarbeitung, Java-Persistence-Frameworks und JSF2-Frontend-Technologien.

Die Klassen „CustomXmlUtility“ und „ExpressionEvaluator“

Diese beiden Klassen dienen in erster Linie der Verarbeitung von *CustomXml*-Elementen und sind nicht Bestandteil des docx4j-Frameworks. Die Klasse *CustomXmlUtility* stellt dabei, analog zur Klasse *SdtUtility*, Basisfunktionalitäten rund um die *CustomXml*-Elemente bereit. Die Klasse *ExpressionEvaluator* wertet den Inhalt des *condition*-Attributs zur Laufzeit aus und liefert je nach Typ der Expression einen booleschen oder einen String-Wert zurück.

Wissen fürs Projekt oder für die Schublade?

Software-Design-Dokumente

Software Design Descriptions/Documents (SDD) [1] sind für die erfolgreiche Entwicklung und Wartung von Software unentbehrlich. Wirklich? Wer oder was beeinflusst deren Sinn und Einsatz im Projekt? Was ist eigentlich zu beachten, damit sie nicht schon während des Projekts zum Papiertiger werden und einen Nutzen darstellen können? Eine pragmatische Übersicht.

von Berthold Schulte

Ein Dokument dient der schriftlichen Vermittlung von Information und Wissen zwischen Menschen. Software-Design-Dokumente beinhalten im Speziellen das Wissen über eine abstrakte Lösung zum Aufbau eines Stücks Software. Die darin dargestellte Information ist allerdings nur einer sehr kleinen Leserschaft zugänglich – da Wissen aus dem Bereich der Softwareentwicklung noch nicht zum Allgemeinwissen gehört, technisch komplex ist und eine eigene Sprache besitzt.

Die Zielgruppen

Wer gehört zu dieser Leserschaft, welche Erwartungen hat sie und welchen Einfluss hat sie auf das Dokument? Neben den Softwareentwicklern sind Tester, Requirement Engineers, Businessanalysten (BA) sowie der Kunde, die Projektleitung und die IT-Abteilung potenzielle Leser, wenn auch mit einem unterschiedlichen Fokus auf das, was sie aus dem Dokument erfahren wollen.

Softwareentwickler und Dev-Peers: Ein Softwareentwickler hat in der Regel die „engste" Beziehung zu einem SDD. Einerseits erstellen Softwareentwickler das Design und die Dokumente oft selbst, andererseits arbeiten sie aktiv und ständig damit, da es ihre Arbeitsvorlage ist. Eben diese direkte Beteiligung an der Umsetzung verpflichtet den Entwickler auch zur Wahrung der Aktualität des SDDs. Hält er Änderungen am Design für nötig oder ergeben sich zwingend Gründe dafür, muss er sie in irgendeiner Form niederschreiben. Ansonsten gehen diese Änderungen in der Menge des Codes unter. Arbeitet er mit einem SDD eines zum Beispiel erfahreneren Entwicklers, so muss eine gute und ständige Kommunikation zwischen beiden herrschen. Der Autor des Designs muss auch bereit sein, Änderungen in sein Design aufzunehmen. Diese Art der Kommunikation und aktiven Arbeit mit dem SDD gewährleistet erst die erfolgreiche Dokumentation von Softwaresystemen, die langfristig wartbar sind.

Qualitätssicherung: Tester arbeiten eher passiv mit einem SDD. Ein SDD enthält für einen Tester die ersten Hinweise zum technischen Hintergrund dessen, was er testet. Für reine Blackbox-Tests eines grafischen User Interfaces mag dies nicht weiter relevant sein. Für die Erstellung von umfangreichen Systemtests, die eine Komponente und ihr Zusammenspiel mit anderen Komponenten eines Systems testen sollen, allerdings schon. So können in dem SDD unter anderem die Verbindungen und Abhängigkeiten zu anderen Systemen beschrieben sein, die bei der Testabdeckung und -erstellung auch mitbedacht werden können.

Requirements Engineer: Dem Requirements Engineer vermittelt das SDD einen ersten Eindruck von der zukünftigen Realisierung und Komplexität eines Stücks Software, einer Komponente oder eines ganzen Systems. Er wird je nach Umfang und Detaillierungsgrad des SDD Teile des Systemkontextes, des Datenmodells sowie die wesentlichen Geschäftsabläufe dort wiederfinden und Abweichungen zu seinen Anfor-

derungen feststellen können. Zum Beispiel während eines Reviews gemeinsam mit dem Autor des Designdokuments.

Fachabteilung, Analyst: Der BA wird der erste sein, der das Dokument nur mit fachlichem Background betrachtet. Ihn werden technische Details ebenso wenig interessieren, wie der Zusammenhang im Systemkontext. Ein Blick sowohl in die Zusammenfassung am Anfang des Dokuments und in das Datenmodell als auch in die Anmerkungen zu möglichen Tests und fachlichen Einschränkungen stehen für ihn eher im Vordergrund. Das Umreißen der Anforderungen durch einen anderen Autor und dies in anderen Worten, wird daher für den BA der größte Nutzen aus dem SDD sein können.

Je unflexibler ein Vorgehensmodell ist und Phasen strikt voneinander trennt, desto stabiler ist in der Theorie auch das SDD.

Projektleitung, Management, Kunde: Der Projektleitung oder auch dem Kunden gibt das SDD einen Einblick in die Komplexität des Systems und ermöglicht unter Umständen Schlussfolgerungen für den weiteren Projektverlauf und für weitere Timelines. Die IT-Abteilung des Kunden wird ebenso Interesse bekunden, zum Beispiel um zu erfahren, welche weiteren Systeme die zukünftige Software benötigt. Für die Unternehmensführung steht die Begutachtung von SDD sicherlich nicht im Fokus. Allerdings wird in kleineren Unternehmen vom Management gerne mal ein Blick in die laufende Projekte geworfen. Insbesondere wenn die Designphase lange dauert, wirken konkrete Ergebnisse in Schrift und Form beruhigend auf die Geschäftsführung.

Einflüsse von Vorgehensmodellen

Das Vorgehensmodell gibt den Plan vor, wie und in welchen Phasen die Beteiligten zusammenarbeiten. Es unterteilt den Entwicklungsprozess dabei zeitlich und inhaltlich in Phasen, die mehr oder weniger voneinander abgegrenzt sind. Diese Trennung wird dabei je nach Modell entweder forciert oder abgelehnt. Eine sehr ausgeprägte Abgrenzung der Entwicklungsphasen gibt beispielsweise das Wasserfallmodell vor, wohingegen agile Ansätze eher auf einen dynamischen Prozess mit sich wiederholenden Iterationen setzen. Das Vorgehensmodell prägt aber nicht nur die Zusammenarbeit während des Entwicklungsprozesses, sondern auch die Ausgestaltung des SDD und den Sinn und Zweck, den das SDD während der Entwicklung einnehmen kann.

Wasserfallmodell: Bei diesem Vorgehen wird dem Erstellen des Designs und damit dem Schreiben des SDD eine eigene, zeitlich begrenzte Phase nach der Analyse und vor der Implementierung zugeordnet. Danach wird das SDD nur noch während der Implementierung abgearbeitet. Die Designphase ist dabei ein Übergang von kundengetriebenen, fachlichen Aussagen in der Analysephase zu entwicklergetriebenem, technischem Verständnis in der Implementierungsphase. Da dieser Ablauf in der Theorie streng sequenziell ist, muss das SDD als Übergangsdokument wasserdicht sein und darf nicht zu Fragen oder Ungereimtheiten in der darauffolgenden Phase führen. Das heißt, das Dokument muss komplett und exakt sein. Ansonsten landet man, ähnlich der Abfahrt mit der Wasserbahn auf dem Jahrmarkt, im kalten Wasser. Ein Vorgehen nach dem Wasserfallmodell benötigt daher die ausgefeiltesten und umfangreichsten SDD im Vergleich zu den folgenden Modellen.

Iterative Modelle: Bei iterativen Modellen werden die einzelnen Arbeitsschritte, wie zum Beispiel Analyse, Design, Test und Deployment, mehrmals durchlaufen und die Software sukzessive erstellt. Die Schritte sind dabei angelehnt an die Phasen des Wasserfallmodells. Insbesondere liegt das Design zeitlich immer zwischen einem Analyse- und einem Implementierungsabschnitt. Somit wird sich die Designbeschreibung Schritt für Schritt mitentwickeln müssen, wobei die vorherige Version des SDD nicht als komplett angesehen werden muss.

Agile Vorgehensweise: Den wohl am schwersten zu beherrschenden Einfluss üben agile Ansätze aus. Sie setzen eher auf den lauffähigen Code als auf ausgefeilte Designdokumentationen. Daher wird auf die Erstellung einer Designdokumentation gerne explizit verzichtet. Eine Begründung lautet oft: „Der Code ist die Dokumentation und die Anforderungen ändern sich so schnell und oft, dass die Anfertigung und Aktualisierung detaillierter Designdokumentationen den Aufwand nicht rechtfertigt." Die Qualität des Codes und das letztendliche Softwaredesign sollen sich bei diesen Ansätzen durch ständiges Refaktorisieren ergeben. Was leider dazu führen kann, dass der Überblick für das Ganze verloren geht und einige Designentscheidungen nicht mehr oder nur bruchstückweise nachvollziehbar sind.

Kosten und Nutzen

Bei wasserfallartigen oder iterativen Vorgehensmodellen ist, wie beschrieben, eine eigene Phase für das Designen und damit das Erstellen eines SDD vorgesehen. Der Umfang und die Länge der Projektabschnitte müssen geplant und ihr Aufwand geschätzt werden. Man hat damit einen guten Überblick über die zu erwartenden Kosten für die Designphase(n), da diese beim initialen Erstellen des Designs und dessen Dokumentation anfal-

len. Läuft das Projekt aus dem Zeitplan und sind umfangreiche Änderungen nötig, hängt es natürlich von dem dann geplanten Vorgehen ab, welche Kosten entstehen. Hier übertragen sich sowohl die Vor- als auch die Nachteile des jeweiligen Modells auf das SDD. Generell gilt: Je unflexibler ein Vorgehensmodell ist und Phasen strikt voneinander trennt, desto stabiler ist in der Theorie auch das SDD. Die Kosten für Anpassungen des Dokumentes sind gleich Null, da sie im Modell nicht vorgesehen sind? Demgegenüber entstehen bei sehr flexiblen, agilen Vorgehen erst gar keine Kosten für das SDD, weil dessen Erstellung gar nicht vorgesehen ist?

Unter der Annahme, dass auch bei agilen Methoden Designbeschreibungen [2] entstehen und dokumentiert werden, können und sollten Sie die Zeiten, die für das Designen, dessen Dokumentation und das Einpflegen von Änderungen benötigt werden, erfassen und in Abständen von ein paar Tagen oder Wochen auswerten. Wird das Designen und Dokumentieren zu sehr geliebt und besteht beispielsweise die Gefahr von Over-Engineering, sind Anpassungen im Vorgehen immer noch möglich. Wohingegen ein weitergehender Nutzen von SDD neben der letztendlichen Spezifikation auf Papier nur schwer messbar ist, aber durchaus gegeben sein kann.

Reflexion: Das Erstellen und Dokumentieren des Designs hat zweierlei „reinigende“ Wirkungen. Zum einen reflektiert der Autor des Dokuments während des Aufschreibens die Ideen und damit das Design noch einmal. Zum anderen können weitere Personen einen Einblick in die Vorstellung vom Design erhalten und ihre Erfahrung miteinbringen. So werden zum Beispiel Design-Patterns gerne zu oft angewendet und unter Umständen auch noch falsch. In einem vom Entwickler erstellten Diagramm lassen sich sowohl dessen Intention im Vorfeld erkennen und so Missverständnisse aufdecken, als auch Wissen und Erfahrung teilen. Bei richtiger und sinnvoller Anwendung solcher Patterns und dem Austausch von Erfahrung können andere Entwickler ebenso davon profitieren. Und dies ohne eine teure Schulung besuchen zu müssen.

Antizipation von Bugs: Kann ein SDD zukünftige Bugs im System antizipieren? Fehler bezüglich der Umsetzung der Anforderungen resultieren oft aus falsch erfassten oder verstandenen Anforderungen und gesetzten Schwerpunkten bei der Entwicklung. Da solche Fehler in frühen Phasen des SDLC günstigster zu beheben sind als in späten, dient ein SDD immer auch der Verifizierung des Geforderten. Oft werden fachliche Ungereimtheiten und Abhängigkeiten erst bei der genauen technischen Ausgestaltung sichtbar, da das Wesen der Implementierung maximale Exaktheit voraussetzt. Zudem dient das Niederschreiben der Entscheidungsfindung für bestimmte Problemlösungen der späteren Nachvollziehbarkeit und deckt kommende technische Probleme oft erst auf. So können fehlende, nicht funktionale Anforderungen (NFA), zum Beispiel Schätzungen des zu erwartenden Datenaufkommens oder der Systemlast, zu Designfehlern führen. Das heißt, werden die NFA beim Design der Software nicht berücksichtigt, führt dies später im Betrieb der Software oft zu bösen Überraschungen.

Weisen Sie daher auf fehlende oder ungenaue Requirements im SDD hin, deren Klärung eine exakte technische Umsetzung erst ermöglichen. Und dokumentieren Sie die zu dem Zeitpunkt gemachten Annahmen zur Umsetzung der Lücken. Sie sichern sich als Dienstleister oder Zulieferer damit unter Umständen besser ab, wenn das SDD Teil des Werksvertrages wird und es zu Unstimmigkeiten mit dem Auftraggeber kommen sollte.

Umfang und Ausprägungen

Die Spanne der Ausprägung von Designbeschreibungen kann von losen Skizzen und Diagrammen als Diskussionsgrundlage bis hin zu ausgefeilten Konzepten und 100-seitigen Spezifikationen als Anleitung zur Implementierung einer Software reichen.

Diskussionsgrundlage: Die Skizze auf Papier oder einem Whiteboard [3] dient natürlich nur als Diskussionsgrundlage, um mit Kollegen über einen oder mehrere Designansätze zu sprechen. Hier stehen eher der kreative Findungsprozess im Vordergrund und dessen mehr oder weniger formale Notation. Zudem bekommen Sie einen ersten Eindruck von der zukünftigen technischen Komplexität des Systems oder der Komponente. Sie können Vor- und Nachteile abwägen, notieren und in einem nachfolgenden ausgearbeiteten SDD aufführen. Ein Leser wird es Ihnen danken zu erfahren, wie es zur abschließenden Entscheidung kam. Er muss die Idee nicht noch einmal komplett durchdenken und erkennt sofort, wo er sich anders entschieden hätte oder Optimierungsbedarf sieht.

Konzept: Sind die ersten Gedanken einmal sortiert, formalisiert und niedergeschrieben, können Sie das Dokument zunächst als ersten Plan zur Umsetzung betrachten. Dieses Konzept kann nun schon einer größeren Runde von Entwicklern und am Projekt Beteiligten vorstellt werden. Es wird die wesentlichen Komponenten beschreiben und Risikoeinschätzungen sowie die

Zielgruppen-Jargon: Beispiel

Sie entwickeln zwei Komponenten, die über verschiedene Schnittstellen kommunizieren. Um diesen Teil der Applikation zu testen, sind in der Regel sowohl Komponententests als auch Integrationstests von Interesse. Die Begriffe „Komponententest“ und „Integrationstest“ sind Fachtermini aus dem Bereich der Softwarequalitätssicherung und werden von Testern sofort verstanden, wenn Sie diese Art von Begriffen im Kapitel „Test der Komponente“ verwenden. Die Intention ist hier natürlich nicht dem Tester vorzuschreiben, welche Teststrategien er verwenden soll, aber dennoch sind sinnvolle Hinweise im jeweiligen Jargon eine gute Diskussionsgrundlage während eines Design-Reviews.

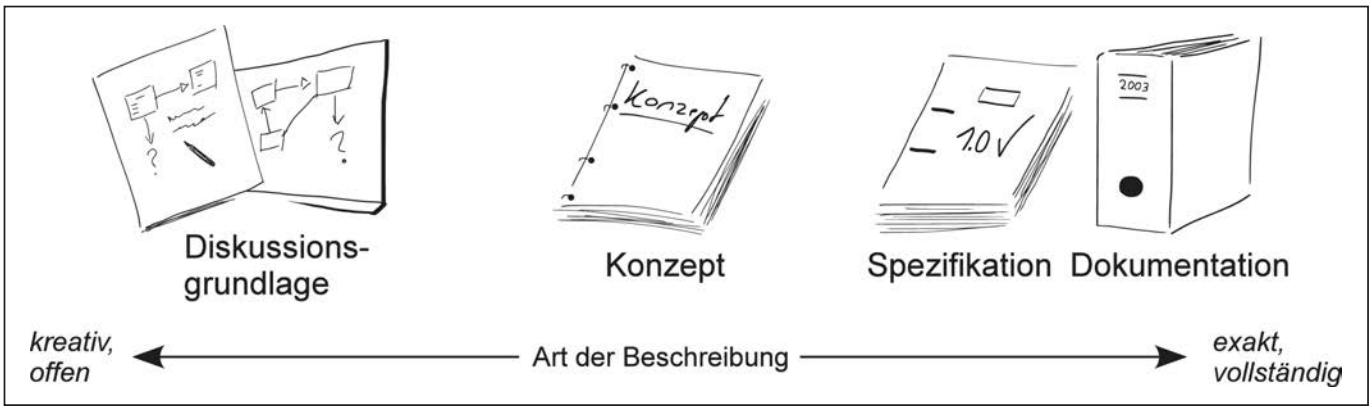


Abb. 1: Die Möglichkeiten ein Design zu erläutern, reichen von kreativ skizzierend bis hin zu exakt dokumentierend

Entscheidungsfindung beinhalten. Wenn dieses erste Konzept steht, ist der kreative Part (**Abb. 1**) des Designens, also die Anforderungen in ein Design zu übertragen, größtenteils abgeschlossen. Der Übergang von einem Konzept zu einer Spezifikation stellt dabei eine Gradwanderung dar. So können zum Beispiel Teile des Konzepts schon soweit ausgearbeitet sein, dass diese eine detailliertere Spezifikation nicht mehr benötigen. Aber andere Teile können immer noch im Stadium einer Ideensammlung feststecken und Markierungen wie „noch offen, todo, tbd." etc. enthalten.

Spezifikation: Das technisch exakte Spezifizieren löst den kreativen Teil des Designprozesses nun endgültig ab. Ein SDD, das die Anforderungen an eine detaillierte technische Spezifikation erfüllt, sollte das gesamte Stück Software wie zum Beispiel eine Komponente eines größeren Systems beschreiben und kann es bis auf exakte Schnittstellen runterbrechen. Das heißt, technische Abhängigkeiten, wie zum Beispiel zu Datenbanken und weiteren externen Systemen, gehören in dieses Dokument als auch zu verwendende Frameworks, Patterns, Programmiersprachen bis hin zu Codeformatierungsrichtlinien und Vorgaben zur Codedokumentation. Schließlich muss in diesem Stadium die Konsistenz des Inhalts gegeben sein und die bereits erwähnten, als offen gekennzeichneten Stellen im Dokument dürfen nicht mehr auftauchen.

Dokumentation: Ist die Software fertiggestellt, getestet und abgenommen, können die bis dahin entstandenen SDD zunächst als Dokumentation für die Maintenance-Phase Verwendung finden. Allerdings hängt dies stark von der bis dahin erzeugten Ausprägung und der Aktualität der Dokumente ab. Ein paar lose Skizzen für die Wartungsphase der Applikation auf dem Schreibtisch liegen zu haben, zeugt dabei natürlich nicht von Professionalität.

Je nach Vorgehensmodell entstehen verschiedene Typen von Dokumenten. Beim Wasserfallmodell mit einer eigenen Designphase werden eher umfangreiche und ausgereifte Dokumente entstanden sein. Im iterativen oder agilen Prozess entstehen eher kleine Artefakte wie einzelne Diagramme mit kurzer Erläuterung und die Dokumentation im Code. Der Aufwand, eine Dokumentation für die Maintenance-Phase zu erstellen, ist daher bei einem agilen Vorgehen unter Umständen höher als bei wasserfallartigen Ansätzen.

Inhalt

Sprache und Ausdruck: Ein SDD beschreibt technische Zusammenhänge, Abhängigkeiten und Anforderungen. Die Ausdrucksweise ist daher eher nüchtern und ohne literarische Ausschweifungen. Der Text muss kurz und knapp auf den Punkt kommen und zwar auf verständliche Art und Weise. Sicherlich eine der größten Herausforderungen bei der Erstellung eines SDD. Die Intention des Aphorismus „So wenig wie möglich und so viel wie nötig" ist dabei oft nicht einfach einzuhalten, da sie oft nur für eine Zielgruppe erfüllbar ist und nicht immer für jede Zielgruppe ein ganz neues Dokument erstellt wird. Fachliche abstrakte Anforderungen in etwas technisch Konkretes zu übersetzen und dies wieder so zu beschreiben, dass es je nach Zielgruppe auch ein technischer Laie nachvollziehen kann, wäre natürlich die Kür, ist aber eher selten zu leisten.

Versuchen Sie die Sprache im Dokument ausgewogen und konsistent zu halten. Sie können aber Kapitel in das Dokument einführen, die explizit für andere Stakeholder, wie zum Beispiel die Kollegen aus der Qualitätssicherung oder die Analysten, gedacht sind. Versuchen Sie dabei deren Sprache zu verwenden. Begeben Sie sich auf den Jahrmarkt und lernen Sie die Sprachen der Schausteller. Wenn Sie in dem entsprechenden Kapitel zum Beispiel die jeweiligen Fachtermini der Zielgruppe verwenden, werden deren Vertreter das Kapitel mit Interesse lesen und Feedback geben – den richtigen Gebrauch des Jargons vorausgesetzt (Kasten: „Zielgruppen-Jargon: Beispiel"). Im Glossar des Dokumentes können dazu nicht geläufige Fachtermini kurz erläutert werden.

Struktur: Neben einer verständlichen und knappen Sprache ohne literarische Ausschweifungen und Wiederholungen gilt: Halten Sie die Struktur der entstehenden Dokumente konsistent. Wird Ihnen keine Struktur vorgegeben, können Sie sich an Standards oder Strukturen, die sich als Best Practice erwiesen haben, orientieren.

Als eine grobe strukturelle Trennung zugunsten der Übersichtlichkeit ist beispielsweise die Differenzierung

von High-Level-Design und Low-Level-Design möglich. Ein High-Level-Design-Dokument stellt eher die architektonische Sicht [4] auf das ganze System mit seinen Komponenten dar. Neben der Komponentenübersicht enthält es unter anderem Kapitel zur Erläuterung von Laufzeitszenarien sowie Vorgaben für die Realisierung von nicht funktionalen Anforderungen. Dokumente, die die Softwarearchitektur beschreiben, unterliegen dabei nicht so häufigen Änderungen. Die Architektur enthält elementarste System- und Designentscheidungen, die oft nicht einfach zu ändern sind und daher eher selten geändert werden. Im Gegensatz dazu beschreibt ein Low-Level-Design eine spezifische Komponente zum Beispiel durch ein Klassenmodell mit Methoden und deren Signaturen detaillierter. Je nach Fortschritt des Designs können Sie sich aber auch in der Low-Level-Designdokumentation zunächst auf die zentralen Aspekte beschränken. Stellen Sie nur die wesentlichen Methoden einer Klasse dar – wenn überhaupt. UML-Tapeten sind auch hier fehl am Platze.

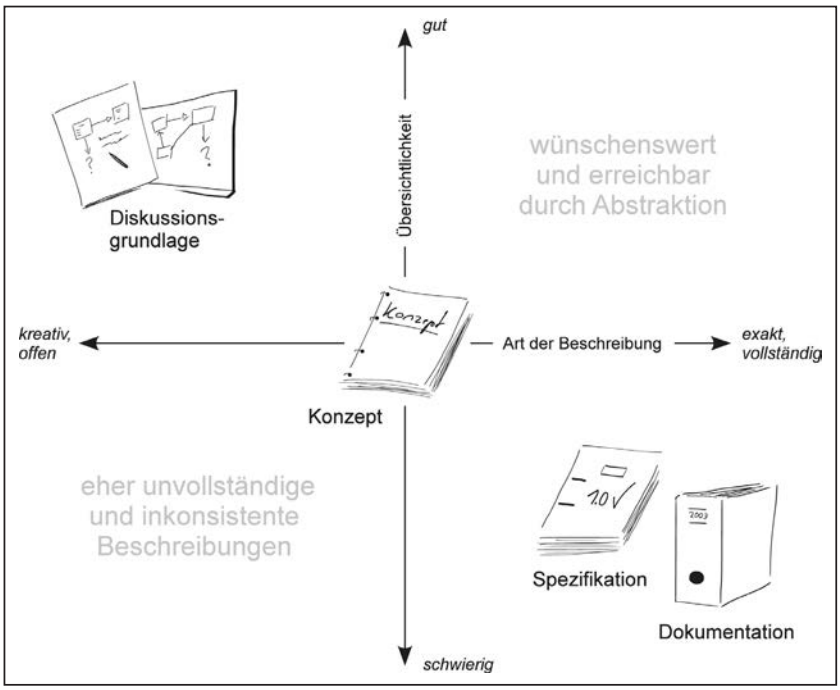


Abb. 2: Ein Design übersichtlich und exakt zu beschreiben, kann durch die Fokussierung auf das Wesentliche gelingen

Die Idee hinter den propagierten Strukturen ist dabei immer das Wissen in versteh- und überschaubare Stücke zu zerlegen und dies anhand einer mehr oder weniger nachvollziehbaren Logik. Diese Logik basiert aber auch auf dem geplanten Design der Software; steht und fällt also auch mit den Designentscheidungen. Je mehr Sie eine Entscheidung beschreiben müssen, desto unlogischer kann die Entscheidung gewesen sein. Oder aber umso wichtiger!

Lässt sich das Design nicht in einfachen und klaren Sätzen und Diagrammen erläutern oder müssen Sie zu weit ausholen, sollten Sie Ihren Ansatz vielleicht noch einmal überdenken. Schließlich soll die Dokumentation einen einfachen und schnellen Einstieg in das Design ermöglichen und von der darunterliegenden Komplexität der Software abstrahieren.

Der Schlüssel zum Code

Um diese Komplexität greifbar zu machen, müssen Sie sich zwangsläufig auf ein Abstraktionsniveau einlassen, das die zentralen technischen Ideen zur Realisierung der Software möglichst übersichtlich (**Abb. 2**) vermitteln kann und an Ihr Vorgehen angepasst ist.

Je höher das Abstraktionsniveau ist, desto mehr Komplexität wird verdeckt, aber umso verständlicher kann die Beschreibung sein. Das bewusste Weglassen von Information, in unserem Falle technischer Details, ist eine der Ideen hinter dem Begriff Abstraktion [5]. Dies gelingt insbesondere durch das Herausarbeiten und Verallgemeinern von grundlegenden Faktoren einer Begebenheit, zum Beispiel eines technischen Problems bzw. einer technischen Lösung. Letztendlich sind einfache und abstrakt gehaltene SDD änderungsstabiler und in der Wartung kostengünstiger als SDD, die zu sehr ins Detail gehen und Aspekte erläutern, die besser im Code zu erfahren sind.

Betrachten Sie daher die Designdokumentation als Schlüssel zum Code der Software und denken Sie an den Leser, der ihn aufschließen will. Den Einwand, ein Entwickler hätte nur den funktionierenden Code abzuliefern, würde ich stets hinterfragen. In einem Autohaus werden die Fahrzeuge schließlich auch mit Schlüssel ausgeliefert, und es wird im Vorfeld darüber gesprochen.

Berthold Schulte ist seit dreizehn Jahren als Softwareentwickler und Berater im Java-Umfeld tätig. Er arbeitet überwiegend in internationalen Projekten und befasst sich mit der Konzeption und Migration von global aufgestellten E-Commerce-Plattformen hin zu Cloud-basierten Ansätzen.

Links & Literatur

[1] IEEE 1016-2009, PDF, ISBN 978-0-7381-5925-6

[2] http://www.agilemodeling.com/essays/agileDocumentation.htm

[3] http://www.agilemodeling.com/essays/whiteboardModeling.htm

[4] Effektive Softwarearchitekturen, Ein praktischer Leitfaden, G. Starke, ISBN 978-3-446-42728-0, 5. Auflage

[5] http://en.wikipedia.org/wiki/Abstraction

Test-driven Development mit Android und Maven

Grüne Männchen testgetrieben

Wer in gewöhnlichen Java-Enterprise-Projekten eine Infrastruktur für Test-driven Development aufbaut, kann auf langjährig bewährte Werkzeuge zurückgreifen. In Android-Projekten sieht die Sache jedoch anders aus: Da die Plattform jung und nicht pures Java ist, begegnet man auf dem Weg zur ausreichenden Test Coverage einigen Stolpersteinen.

von Eugen Seer

Die testgetriebene Entwicklung ist eine der Grundlagen für agile Praktiken, wie Continuous Integration (Kasten). Der Entwickler erstellt zuerst die Unit Tests und beginnt danach mit der Implementierung. Diese gilt als abgeschlossen, sobald die Unit Tests erfolgreich durchlaufen. Die Konsequenzen daraus sind:

1. Die Entwicklung der Schnittstellen aus Test- bzw. Kundensicht führt zu besserem Design.
2. Die konsequente Abdeckung des Quellcodes durch Unit Tests macht implizite Annahmen des Entwicklers explizit und überprüfbar. Die Abhängigkeit vom ursprünglichen Entwickler und das Risiko von Änderungen sowie Refaktorisierungen werden reduziert, Collective Code Ownership wird möglich.

Warum für Android-Apps?

Es liegt in der Natur von Android als Plattform für mobile Endgeräte, dass sich der Umfang von Android-Apps im Gegensatz zu beispielsweise Enterprise-Software in Grenzen hält. Dies hat folgende Ursachen:

- Die knappen Hardwareressourcen von mobilen Endgeräten setzen dem Spielraum der Entwickler enge Grenzen.
- Android-Apps fokussieren sich auf eine Hauptaufgabe, das reduziert die Anzahl der Bildschirmmasken.
- Die beschränkte Displaygröße reduziert wiederum den Umfang der einzelnen Bildschirmmasken.
- Die Geschäftslogik wird meistens auf einem Backend-System ausgeführt.
- Das komfortable Android-API senkt den Bedarf an Boilerplate-Code.

Dies führt im Entwickleralltag nun leider dazu, dass Themen wie Build-Prozess und Unit-Test-Abdeckung zu wenig Beachtung finden, weil ihre Bedeutung unterschätzt wird. Da die Komplexität bei Android-Apps aber verborgen in der Fragmentierung der Plattform und den hohen Benutzererwartungen liegt, können sich diese technischen Schulden später rächen, wenn es darum geht, eine hochqualitative App auf einer Unzahl an Geräten in Produktion zu nehmen.

Das Fundament: Build-Prozess und Build-Server

Für die Konfiguration des Build-Prozesses empfiehlt sich Maven. Es bietet ein Plug-in für Android und außerdem die folgenden Merkmale:

- Dependency Management
- Wiederverwendung des lokalen Build auf einem Build-Server
- Ausführung von Unit Tests
- Quellcodeanalyse
- Generierung von Dokumentation

Als Build-Server stehen kommerzielle Produkte, wie Bamboo oder Teamcity, als auch freie, wie Hudson oder Jenkins, zur Verfügung. Sie sorgen dafür, den Build-Prozess permanent auf einer zentralen Umgebung, abgeschottet von Entwicklerrechnern zu integrieren und auszuführen.

Continuous Integration

Continuous Integration [1] ist eine der Hauptvoraussetzungen für den Qualitäts- und Produktivitätsschub der agilen Methode. Sie geht u. a. zurück auf Martin Fowlers Zitat „If it hurts, do it more often“ [2]. Das heißt schwierige manuelle Prozesse werden automatisiert und so oft ausgeführt, bis sie einfache Routine werden. Die wichtigsten Bestandteile dabei sind:

1. Versionsverwaltung
2. Abdeckung des Quellcodes durch Unit Tests
3. Automatisierter Build
4. Automatisierte Verteilung
5. Kurze Test- und Feedback-Zyklen

Dadurch werden die Kompilierbarkeit des Quellcodes und das Durchlaufen der Tests andauernd überprüft. Probleme werden schnell erkannt, Rückschlüsse auf deren Ursache können daher einfach gezogen werden, weil der Fehler immer nur kurz zuvor passiert sein kann und die Änderungen in den Köpfen der Entwickler noch sehr präsent sind. Für die Erzeugung eines Android-Projektes mit Maven [3] sollten die folgenden Voraussetzungen erfüllt sein:

- Maven 3 ist installiert und im Systempfad enthalten.
- Das Android-SDK mit API-Level 8 ist installiert, die Umgebungsvariable *ANDROID_HOME* zeigt darauf.

Nun ist das folgende Kommando auszuführen, dadurch wird ein Projekt mit dem Namen „android-test-driven" erstellt. Im Beispiel wurde dabei Android-API Level 8 gewählt, denn damit können ca. 95 Prozent der derzeit ausgelieferten Geräte erreicht werden [4]:

```
mvn archetype:generate \
  -DarchetypeArtifactId=android-quickstart \
  -DarchetypeGroupId=de.akquinet.android.archetypes \
  -DarchetypeVersion=1.0.8 \
  -DgroupId=com.sourcecommander.javamagazin \
  -DartifactId=android-test-driven
```

Das Projekt kann mit folgendem Befehl, ausgeführt im Projektverzeichnis, gebaut werden:

```
mvn install
```

Optional: Import des Projekts in Eclipse

Wer das Projekt unter Eclipse bearbeiten möchte, gibt diese Zeile ein:

```
mvn eclipse:eclipse
```

Danach ist die Datei *.project* wie in Listing 1 anzupassen. Jetzt kann das Projekt in Eclipse importiert werden, indem man es in Eclipse mit dem Menüeintrag *File->Import->Existing Maven Projects* auswählt.

Unit Tests: nur scheinbar einfach

Nun existiert ein Android-Projekt, das mit Maven gebaut und mit Eclipse bearbeitet werden kann. Bevor der Entwickler Unit Tests schreiben kann, muss er sich für die Ablaufumgebung der Tests entscheiden. Zur Auswahl stehen:

- **Die Java Virtual Machine**: Ein Unit Test für die Java Virtual Machine wird wie gewohnt unter Zuhilfenahme der Bibliothek JUnit 4.8 geschrieben. Die Probleme beginnen, sobald man auf eine Klasse aus einem der *android.**-Pakete wie beispielsweise *Context* zugreift. Dieser Unit Test schlägt auf einer Entwicklermaschine fehl, weil die Android-spezifischen Klassen hier lediglich Stubs sind, die bei Verwendung Exceptions werfen. Daher sind diese Unit Tests lediglich für Klassen geeignet, die keine direkte oder indirekte Abhängigkeit zur Android-Plattform aufweisen. Erfahrungsgemäß sind diese jedoch sehr wenige, denn auch wenn die reinen Algorithmen vom Glue Code getrennt sind, scheidet ein Test der Benutzeroberfläche immer noch komplett aus.
- **Android Emulatoren oder Devices**: Es ist möglich, Unit Tests in einem Emulator oder auf einem Mobile Device auszuführen. Der Nachteil ist, dass der Emulator dafür zu langsam und zu instabil ist. Ein Mobile Device wiederum lässt sich nur schwer an einen Build-Server anschließen, in der Cloud schon gar nicht.

Es ergibt sich also folgende Situation, wie in Tabelle 1 veranschaulicht.

Beide Lösungen sind also nicht zufriedenstellend, wünschenswert wäre eine Lösung, die sowohl stabil auf der lokalen und der Integrationsumgebung läuft als

Listing 1

```
<!-- zwei Android Build Commands hinzufügen -->
  <buildCommand>
    <name>com.android.ide.eclipse.adt.ResourceManagerBuilder</name>
    <arguments></arguments>
  </buildCommand>
  <buildCommand>
    <name>com.android.ide.eclipse.adt.PreCompilerBuilder</name>
    <arguments></arguments>
  </buildCommand>
<!-- Android Project Nature für Eclipse setzen -->
  <nature>com.android.ide.eclipse.adt.AndroidNature</nature>
```

Listing 2

```
<!-- Diese Dependencies sollten unter denen für Android stehen ! -->
<dependency>
  <groupId>com.pivotallabs</groupId>
  <artifactId>robolectric</artifactId>
  <version>1.0-RC1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>
```

Ablaufumgebung	JVM	Android	Der dritte Weg?
Stärken	schnell und stabil	*android.** Packages verfügbar	schnell, stabil und *android.**
Schwächen	Exceptions bei *android.**	langsam und instabil	keine

Tabelle 1: Stärken und Schwächen der Unit-Test-Ablaufumgebungen

auch Android-spezifische Abhängigkeiten sinnvoll auflösen kann. Weiterhin sollte die Lösung eine Messung der Code Coverage erlauben, um Qualitätsrichtlinien überprüfen zu können.

Der dritte Weg: Robolectric

Glücklicherweise hat die Open-Source-Community mit Robolectric einen dritten Weg vorgezeichnet. Robolectric ist ein Unit-Test-Framework, das die Abhängigkeiten zu Android in der JVM auflöst. Es kann in der Maven-POM-Datei, wie in Listing 2 beschrieben, eingebunden werden [5].

Falls notwendig, sollte man nun im Projektverzeichnis das Verzeichnis *src/test/java* anlegen. Ebenso ist sicherzustellen, dass die Klasse *R.java* existiert. Falls dem nicht so ist, hilft folgender Befehl:

```
mvn android:generate-sources
```

Ein Unit Test mit Robolectric wird wie gewöhnlich bei Maven im Verzeichnis *src/test/java* abgelegt. Er wird jedoch nicht mit dem JUnit Runner ausgeführt, sondern mit dem RobolectricTestRunner annotiert. Im Beispiel in Listing 3 wird auf die Android-String-Ressourcen zugegriffen und der Namen der App validiert.

Es ist aber noch viel mehr möglich, auch die Benutzerschnittstelle, deren Ereignisse und die Abfolge von Activities kann wie in Listing 4 angesteuert und getestet werden.

Listing 3

```
// Package und Import Statements ...
@RunWith(RobolectricTestRunner.class)
public class AppNameTest
{
   @Test
   public void shouldHaveAppName() throws Exception
   {
      String appName = new
          HelloAndroidActivity().getResources().getString(R.string.app_name);
      Assert.assertEquals("android-test-driven", appName);
   }
}
```

Listing 4

```
// Package und Import Statements ...
@RunWith(RobolectricTestRunner.class)
public class GuiAndEventsActivityTest
{
   private HelloAndroidActivity activity;
   private Button pressMeButton;

    @Before
   public void setUp() throws Exception
   {
      activity = new HelloAndroidActivity();
      activity.onCreate(null);
      pressMeButton = (Button) activity.findViewById(R.id.press_me_button);
   }

   @Test
   public void shouldHaveAButtonThatSaysPressMe() throws Exception
   {
      assertThat((String) pressMeButton.getText(), equalTo("Tests Rock!"));
   }

   @Test
   public void pressingTheButtonShouldStartTheListActivity() throws Exception
   {
      pressMeButton.performClick();

      ShadowActivity shadowActivity = shadowOf(activity);
      Intent startedIntent = shadowActivity.getNextStartedActivity();
      ShadowIntent shadowIntent = shadowOf(startedIntent);
      String intentClassName = shadowIntent.getComponent().getClassName();
      assertThat(intentClassName, equalTo(NextActivity.class.getName()));
   }
}
```

Fazit: keine Ausreden mehr

Es ist nicht nur möglich, testgetrieben in Android zu entwickeln, es ist vor allem notwendig. Wer seinen Quellcode mit Test Coverage schützt, spart sich nicht nur den sonst notwendigen manuellen Testaufwand nach jeder Änderung, sondern kann diese automatisierten Tests auch noch viel schneller ausführen. Dies ermutigt dazu, notwendige Änderungen durchzuführen und die strukturelle Qualität durch Refaktorisierungen aufrechtzuerhalten. Klug ist, wer das von Anfang an macht, denn jede Ausführung der Test Suite trägt zur Amortisation der anfänglichen Investition bei.

Es sei abschließend folgender Ausflug in die Medizin erlaubt: Es ist heute eine Selbstverständlichkeit, dass ein Chirurg mit sterilen Händen operiert. Als jedoch der Arzt Ignaz Semmelweis [6] im 19. Jahrhundert das Auftreten von Kindbettfieber auf die mangelnde Hygiene des Krankenhauspersonals zurückführte, wurden seine Erkenntnisse als spekulativer Unfug abgetan. Man kann sich heute Quellcode mit Unit Tests so wie saubere Hände bei einer Operation vorstellen. Und wer die heute schon hat, der hat morgen noch einen quicklebendigen Androiden.

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.

Links & Literatur

[1] http://www.martinfowler.com/articles/continuousIntegration.html

[2] http://martinfowler.com/bliki/FrequencyReducesDifficulty.html

[3] http://www.thoughtsonmobile.com/2012/06/android-apps-mit-maven-bauen.html

[4] http://developer.android.com/about/dashboards/index.html

[5] http://pivotal.github.com/robolectric/maven-quick-start.html

[6] http://de.wikipedia.org/wiki/Ignaz_Semmelweis

NEUE SERIE

Reichen Java und Linux aus, um Android-Apps zu entwickeln?

Zwei Seiten einer Medaille?

Erschwingliche Datenflatrates, eine echte Vielfalt verfügbarer und preiswerter Smartphones und Tablets und eine Vielzahl an Inhalten für jeden Geschmack haben den Siegeszug des mobilen Internets ermöglicht. Waren es früher noch die etwas sperrigen Ökosysteme von BlackBerry oder Nokia, die das mobile Internet für eine vergleichsweise kleine Gruppe an Nutzern abdeckten, ist dies heute – gerade über den privaten Konsum getrieben – ein Allgemeingut geworden und damit wird dieser Punkt wieder interessant für Unternehmen.

von Jörg Pechau

Unternehmen verschieben Geschäftsprozesse in das mobile Internet. Sie reichern diese mit mobilen Anwendungen an, sie finden neue Geschäftsprozesse für das mobile Internet, sie frischen alte auf. Oder sie haben es vor und dann stellt sich fast automatisch die Frage: Wir benötigen auch eine App, aber wie, auf welcher Plattform und wer macht es?

Die Argumentationslinie geht verallgemeinert häufig so: Machen wir es selber oder kaufen wir eine Dienstleistung ein? Dienstleister sind teuer, wir haben doch selber Experten. Wir können doch Java und wir entwickeln ohnehin für Unix oder Linux – richtig? Smartphones sind doch auch nur kleine Computer, Android ist Linux, Java ist Java – wie schwer kann es sein, für Android zu entwickeln? Also machen wir es selbst.

Diese Argumentationslinie ignoriert offensichtlich wirtschaftliche Aspekte wie z. B. auf welcher Plattform wir unsere Zielgruppen finden, wer Kunde und wer Anwender sein wird, ob eine Plattform überhaupt ausreichend ist usw. Unterstellen wir einfach für unseren Fall,

Artikelserie

Teil 1: Einleitung, notwendiges Vorwissen
Teil 2: UI-Entwicklung

dass sich jemand diese Gedanken gemacht hat, anderenfalls würde wahrscheinlich ein Artikel nicht ausreichen, die einzelnen Aspekte zu betrachten, die in einen derartigen Entscheidungsprozess einfließen.

Zurück zum Thema: Android ist ein Akteur auf dem Markt für Mobile, dessen „Software-Stack" kompatibel zu den IT-Infrastrukturen in Unternehmen ist – sie sind allem Anschein nach Enterprise-tauglich.

Ziel dieses Artikels ist, die Annahme „Java und Linux reichen aus, um für Android zu entwickeln" in einigen Aspekten näher zu betrachten.

Einleitung

Die Überschrift allgemeingültig einzulösen, ist naturgemäß schwierig, doch für die Java-basierten unternehmenskompatiblen Software- und Prozess-Stacks haben wir ausreichend Erfahrungen und Beobachtungen aus unserem Alltag.

Ein Großteil der Java-basierten Softwareentwicklung in Unternehmen zielt darauf ab, Software für die serverseitige Funktionalität zu entwickeln, als Teil von Applikationen, deren Interaktionen mit den Anwendern häufig mit Webtechnologien realisiert werden. In diesem Fall kommt dann neben Java eine Kombination aus HTML, CSS und JavaScript zu tragen, um in Browsern einen Rich-Client-Eindruck zu erzeugen. Die „klassische" Fat-Client-Entwicklung für den Desktop findet heutzutage kaum noch statt.

In dieser Konstellation steht Server für eine Kombination aus Middleware-Produkten: einem Applikationsserver oder Servlet-Container, einem Webserver und einer relationalen Datenbank – zuweilen auch einer NoSQL-Datenbank. Vervollständigt wird diese Zusammenstellung je nach Anwendungsfall zum Beispiel durch Messaging-Systeme oder die Integration von weiteren Servern. Beim Betriebssystem wird aus Kostengründen – sei es Lizenzkosten oder Wartung – oft auf ein Unix- oder Linux-Derivat gesetzt.

Auf der Applikationsebene hat sich etabliert, auf die Erfahrung der Entwicklergemeinde durch die Nutzung von etablierten Frameworks zu setzen: Weit verbreitet sind Application-Frameworks wie Spring oder objektrelationale Mapper wie Hibernate. Auf der Ebene der Benutzeroberflächen (UI) kommen dann noch verschiedenste Webframeworks hinzu und all das mehr oder weniger auf Basis des sehr umfangreichen Java-Standards oder der Java Enterprise Edition.

Als Konsequenz hat sich in vielen Unternehmen, die wir kennen lernen durften, eine typische Aufgabenteilung ergeben: Das Gros der Softwareentwickler programmiert in Java für die Serverseite und bindet diese Funktionalität an ein mehr oder weniger vorgegebenes UI. Der Entwurf des UI kommt in dieser Aufgabenteilung dann von (Web-)Designern. Das heißt, der eigentliche Entwurf von UIs betrifft die meisten Softwareentwickler nur mittelbar, insofern sie für die Anbindung an das Backend verantwortlich, manchmal auch für die Ausprogrammierung des Entwurfs zuständig sind.

Diese Aufgabenteilung spiegelt sich im Vorgehen, in der Entwicklungsinfrastruktur und in den Grundannahmen beim Entwurf von Softwaresystemen wider: Es gibt die verschiedensten Ansätze, um den Softwareentwicklungsprozess zu verbessern und die Softwarequalität zu erhöhen. Sei es Test-driven Development [1] oder Scrum [2] oder einer der vielen anderen Vertreter. Oder die Vielzahl an freien und kommerziellen Werkzeugen, die Entwickler bei ihrem täglichen Streben nach Qualität unterstützen. Entwickler können sich beim Entwurf ihrer Systeme auf erprobte Muster verlassen und solchen Grundannahmen folgen, dass Ressourcen wie Speicher oder Performance kein Problem sind. Bandbreite ist stets ausreichend vorhanden. Es ist keine „geheime Kunst" mehr, Anwendungen zu optimieren – durch Caches, Lastverteilung, horizontale Skalierung etc., um hohe Performanz bei geringen Latenzen auch bei vielen parallelen Nutzern und großen Datenmengen zu erzielen.

Das heißt, wir können die Situation des durchschnittlichen Java-Entwicklers vereinfachend wie folgt zusammenfassen: Linux und Java sind vertraut, Webtechnologien teilweise auch. Benutzungsoberflächen zu entwerfen und umzusetzen, ist meistens nicht die zentrale Aufgabe. Über technische Restriktionen braucht man sich wenig Gedanken machen.

Blick in die mobile Welt

Die gute Nachricht ist – im Prinzip ändert sich eigentlich nicht sehr viel. Die schlechte Nachricht – wenn wir es so nennen wollen – ist, dass sich zwar nicht viel ändert, wir aber einige unserer lieb gewonnen Grundannahmen und Prinzipien überarbeiten müssen. Grob gesagt müssen wir uns mit zwei großen Themenblöcken auseinandersetzen:

- Wir müssen wieder lernen, ressourcenschonend Software zu entwickeln.
- Wir müssen uns intensiv mit Benutzeroberflächen auseinandersetzen.

Kurz: Wir müssen uns mit sehr anderen nicht funktionalen Anforderungen und dem Umgang damit beschäftigen, als wir es bisher gewohnt waren.

Ressourcen spielen wieder eine Rolle

Charakteristisch für die Handhabung von Smartphones ist, dass wir sie in der Regel mit uns „spazieren" tragen. Banal? Vielleicht ja, doch das ist der wesentliche Unterschied zu den Endgeräten, gegen die wir im nicht mobilen Bereich entwickeln, aus dem sich einige gravierende Einschränkungen ableiten.

Hardware kann sehr einschränkend sein

In der Vermarktung der aktuellen Android-Geräte überbieten sich die Hersteller gegenwärtig in dem Bemühen, darzustellen, wer das leistungsfähigere Smartphone anbietet. Aus einem Datenblatt eines aktuellen Smartphones [3] vom oberen Ende der Leistungsskala:

Wir müssen wieder lernen, ressourcenschonend Software zu entwickeln. Und wir müssen uns intensiv mit Benutzeroberflächen auseinandersetzen.

- CPU: Quad-Core-Prozessor, 1,4 GHz, ARM
- Datenspeicher/Arbeitsspeicher: 32 GB Speicher, davon 1 GB Arbeitsspeicher
- Connectivity: WiFi IEEE 802.11 a/b/g/n; UMTS/HSPA+/HSUPA; GPRS/EDGE
- GPS, NFC
- Display: 4,7“, 1280 x 720

Das liest sich zunächst einmal gut. Werfen wir zum Vergleich einen Blick auf die entsprechenden Kenndaten eines ordentlich ausgestatteten Laptops [4]:

- CPU: Quad-Core-Prozessor, 3,6 GHz, Intel i7
- Datenspeicher/Arbeitsspeicher: diverse Festplatten ab 512 GB Speicher, 16 GB Arbeitsspeicher, 6 MB L3 Cache
- Connectivity: WiFi IEEE 802.11 a/b/g/n
- Display: 15,4“, 1440x900

Im direkten Vergleich schneidet das Smartphone zumindest vordergründig recht gut ab. Wenn wir genauer hinschauen, ist die Situation für uns Entwickler leider nicht mehr so luxuriös.

Heap

Von der desktop- oder serverseitigen Java-Entwicklung sind wir es gewöhnt, dass wir zum einen den Heap der JVM anpassen können und dass wir zum anderen in den Rechnern Speicher nachrüsten können und auch darüber die maximale Heap-Größe beeinflussen können. Die Möglichkeit, die Heap-Größe anzupassen, haben wir unter Android mit der Dalvik Virtual Machine (DVM) nur eingeschränkt. Die Option, den Arbeitsspeicher aufzurüsten, haben wir für mobile Geräte in der Regel gar nicht.

Beginnen wir mit dem Heap, d. h. dem Arbeitsspeicherbereich für die eigene App. Im Prinzip wird die Heap-Größe durch die Gerätehersteller festgelegt, dabei galten in der Regel bisher pro Prozess, in dem meine Apps liefen, folgende Größenordnungen:

- Android 1.x: 16 MB
- Android 2.x: 24 oder 32 MB
- Android 3.x/4.x: 48 MB oder 64 MB

Ab Android 3.x gibt es die Option *android:largeHeap="true"* [5], die im Manifest einer Android-App zu setzen ist. Damit kann das 48-MB-Limit pro App heraufgesetzt werden, doch auch hier limitieren dies viele Geräte zumeist auf Werte zwischen 128 und 256 MB. Damit gelangen wir immerhin in den unteren Bereich der uns von der Serverseite vertrauten Heap-Größen.

Auf älteren Android-Versionen, „Gingerbread“ macht Stand November 2012 [6] noch über 50 % des Markts aus, haben wir – ohne das Android-Gerät zu „rooten“ – keine Möglichkeit die Heap-Größe unter DVM zu beeinflussen. All dies steht natürlich unter dem Vorbehalt, dass ein Gerät überhaupt in der Lage ist, soviel Heap bereitzustellen. Treffen wir hier zu optimistische Annahmen über die Speicherverwendung, laufen wir Gefahr, zu viele Geräte von der Nutzung meiner App auszugrenzen.

Warum ist das überhaupt ein Problem? Sobald ich den Heap ausreize, was bei datenintensiven Apps wie z. B. Bildverarbeitung schnell der Fall sein kann, wird eine Garbage Collection (GC) ausgelöst. Soweit ist das natürlich nichts Neues, und serverseitig löst dies heutzutage kaum noch Schrecken aus, da wir zum einen über die Heap-/Speichergröße die Häufigkeit einer GC beeinflussen können und zum anderen der GC-Mechanismus auf der verfügbaren Hardware effektiv und schnell arbeitet.

Auf Android-Smartphones möchte man dieses gerne vermeiden, weil GC sich im Laufzeitverhalten der App bemerkbar machen kann. Da Apps üblicherweise Anwendungen mit direkter Benutzerinteraktion sind, kann sich dies sowohl in der Benutzbarkeit (Usability) als auch in der User Experience niederschlagen. Anwender registrieren Verzögerungen ab ca. 100 bis 200 ms Wartezeit. Reagiert das UI einer App nicht innerhalb von 5 s, greift das Betriebssystem ein und es erscheint der bekannte „Application not Responding“-Dialog.

Anwender sind tendenziell ungeduldig. Smartphones werden meist in einem mobilen Kontext verwendet, d. h. unterwegs und zwischendurch. Wenn ich also nur „eine Minute“ Zeit habe und in dieser Zeit erst einmal warten muss, bis das Gerät nach einem Start seinen Speicher aufgeräumt hat, dann schaue ich mich als Nutzer schnell nach Alternativen um. Eine Studie von 2010 [7] zeigt, dass zu dem Zeitpunkt nur 26 Prozent aller heruntergeladenen Apps mehr als zehn Mal gestartet wurden und ca. 50 Prozent der Apps spätestens nach dem dritten Start wieder gelöscht wurden. Das heißt, unsere App sollte sofort interagieren und damit die Chance bieten, einen Anwender sofort von sich zu überzeugen und nicht erst nach einer „Gedenkpause“. Abgesehen davon sollten wir generell mit der Zeit unserer Nutzer verantwortungsvoll umgehen.

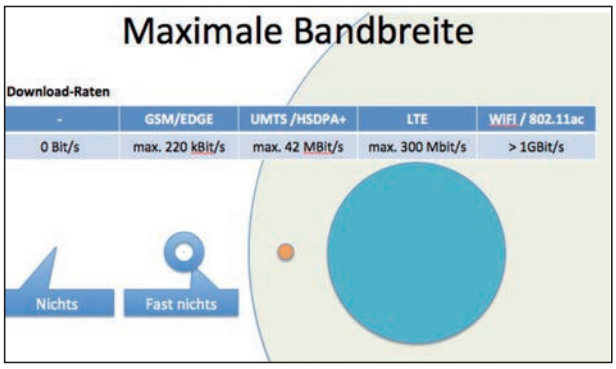


Abb. 1: Bandbreite und Downloadraten

Wir wissen, dass wir alles, was mehr Zeit in Anspruch nehmen kann, aus dem UI-Thread auslagern sollten. Wir sollten aber auch ansonsten innerhalb des UI-Threads Komplexität vermeiden. Komplexe und vor allem potenziell speicherintensive UI- oder Applikationsframeworks, wie wir sie gerne serverseitig verwenden, sind in diesem Sinne keine gute Idee. „Keep it simple, keep it plain" sei unsere Devise.

Gerätespeicher

Smartphones haben meistens einen Speicher zwischen 8 und 64 GB. Der Arbeitsspeicher hat eine Größe von bis zu 1 GB und hängt bei den meisten Geräten vom zur Verfügung stehenden Speicher des Geräts ab. Das Betriebssystem beansprucht seinen Teil im Speicher und alle anderen installierten Apps und allgemein gespeicherte Daten ebenfalls. Nicht jedes Gerät bietet die Option, den Speicher über Speicherkarten zu erweitern.

Den bis zu 1 GB großen Arbeitsspeicherbereich von Smartphones müssen sich alle laufenden Apps teilen. Der bis zu 32 GB große Speicher muss alle Daten auf dem Telefon speichern. Für alles, was wir direkt auf dem Gerät benötigen, muss das reichen.

Das bedeutet, dass wir uns mit Informationsdesign [8] auseinandersetzen müssen: Welche Informationen benötigen wir wirklich auf dem Gerät? In welcher Granularität benötige ich die Daten? Kann ich Informationen sinnvoll aggregieren? Wie lange muss ich die Daten vorhalten?

Abhängig von meiner Antwort kann es bedeuten, dass ich auch serverseitig meine Schnittstellen anpassen muss. Denn was auf der Serverseite bei ausreichendem Massenspeicher und genügend Bandbreite gut funktioniert, lässt sich nicht so ohne Weiteres auf den mobilen Bereich übertragen: Das Verarbeiten großer Nachrichtenmengen, z. B. das Aufbereiten von Karten inklusive Overlays, kostet Zeit und geht auf Kosten der Batterielaufzeit und des Speichers. Elegant wäre es also, nur das zu übertragen, was auch wirklich auf dem Gerät zu dem Zeitpunkt benötigt wird.

Bandbreite

Erinnern Sie sich noch an die Zeit, als Bandbreite eine Rolle spielte? Heutzutage sprechen wir im LAN von über 1 bis 10 GBit Bandbreiten und darüber hinaus. Selbst im WiFi sind wir im GBit-Bereich angelangt, was Übertragungsraten betrifft. 16 MBit bei DSL ist zumindest in Ballungsräumen Standard. Unternehmen kaufen sich noch deutlich „dickere" Leitungen ein. Das heißt für die meisten praktischen Fragen spielen Bandbreiten meistens keine Rolle mehr, häufig sind die dahinter stehenden Anwendungen langsamer als das, was wir über die Leitungen transportieren können. Im mobilen Bereich könnte unsere Bandbreite wie in **Abbildung 1** dargestellt aussehen.

Könnte? Die Realität sieht für die meisten Nutzer anders aus: WiFi kostet häufig Geld und ist nicht unbedingt immer die sichere Wahl. Das Thema Handover zwischen verschiedenen Hotspots, sodass ich mich WiFi nutzend z. B. durch eine Stadt bewegen kann, ist immer noch ein Draft. Abgesehen davon ist die Vision von Städte abdeckenden WiFi-Netzen leider auch noch genau das: eine Vision.

Wir sind also auf die Mobilfunknetze angewiesen und hier sehen wir, dass sich LTE (Long Term Evolution, der neueste eingeführte Standard) im Ausbau befindet. LTE ist nicht für alle verfügbar, es gibt erst wenige Geräte, die es nutzen können, wenige Anbieter, die sich die Nutzung auch mit relativ hohen Preisen „vergolden" lassen. Das bedeutet für die meisten Anwender EDGE oder 3G, gedeckelt durch Datenflatrates auf 100 MB und 5 GB Traffic.

Auch mit UMTS könnten wir gut klar kommen, die Abdeckung ist prinzipiell gut, wenn meine verfügbare Bandbreite nicht auch noch abhängig wäre von:

- der Anzahl eingebuchter Geräte pro Funkzelle
- meiner Bewegung innerhalb der Funkzelle
- meiner Lage zum Funkmast
- Energiesparmaßnahmen des Endgeräts
- eventuellen Bandbreitenfiltern der Operator (Flatrate)
- der Bandbreite, mit der ein Anbieter seinen Mobilfunkmast an seine Infrastruktur anbindet. Gerade hier gibt es gigantische Unterschiede, sodass ich zwar eine sehr gute UMTS-Verbindung, aber trotzdem praktisch keinen Datendurchsatz haben kann.

Den verbleibenden Rest an Bandbreite können wir nutzen. Unglücklicherweise können wir die meisten dieser Punkte nicht beeinflussen. Wir können aber darauf eingehen, indem wir defensiv gegen mögliche schlechte Datenverbindungen entwickeln.

Das bedeutet zum einen ein geschicktes Informationsdesign, wie schon zuvor beschrieben. Zum anderen sollte jegliche Server-App-Kommunikation asynchron erfolgen, sodass die Anwendung „responsive" bleibt.

Es lohnt sich darüber nachzudenken, ob die Anwendung auch offline nutzbar sein soll, und es lohnt sich ggf. über Synchronisationsstrategien nachzudenken, z. B. dass größere Datenmengen nur bei bestehender 3G- oder WiFi-Verbindung gezogen werden sollen. Schlussendlich sollten wir – wenn wir versuchen die Nutzung der Bandbreite zu optimieren – auch noch ei-

nen Blick auf verwendete Protokolle und Standards werfen. Als Beispiel zeigt **Abbildung 2** die Payload einer SOAP- [9] und einer REST-basierten [10] App-Server-Kommunikation.

Ohne Whitespace ergibt sich hier im Vergleich JSON zu XML ein Verhältnis von 359 Bytes zu 527 Bytes. Das heißt, wir können in diesem Fall ca. 30 % der zu übertragenden Daten einsparen. Absolut mag es nicht viel klingen, doch wenn wir uns beispielsweise Edge mit seinen maximalen 220 kBit/s vor Augen führen, sehen wir, dass gerade in diesem Fall jedes Byte zählt.

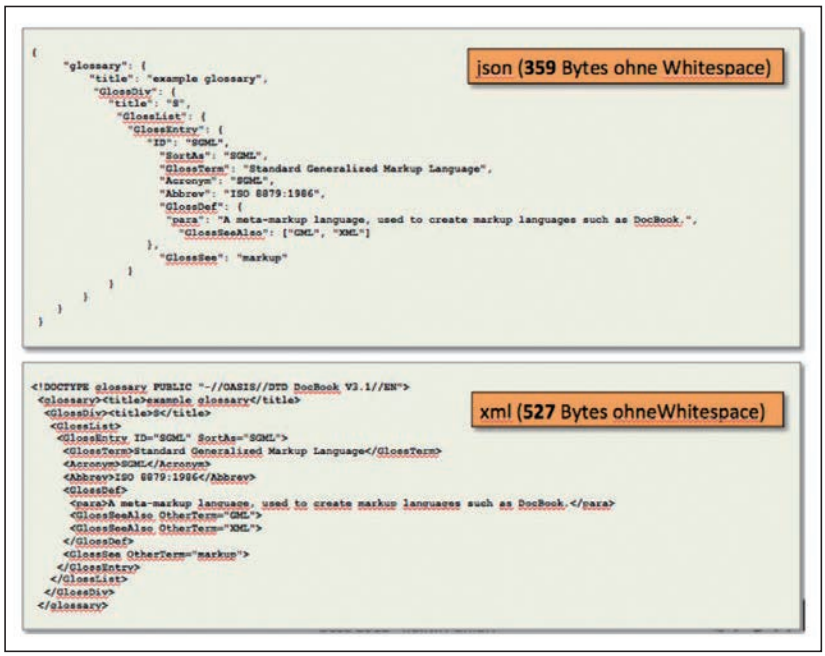


Abb. 2: Payload einer SOAP- und einer REST-basierten App-Server-Kommunikation (Quelle: json.org)

Learnings Ressourcen

Was können wir also für das Thema „Ressourcen" mitnehmen? Im „Informatiker-Sprech" reden wir hier über nicht funktionale Anforderungen: Wir sollten Ressourcen schonen. Im Einzelnen heißt dies, da die Dalvik- deutlich langsamer als die Hotspot-VM ist, dauert auch Garbage Collection länger. Hinzu kommt, dass wir häufig weniger Speicher als gewohnt zur Verfügung haben. Es ist also nicht mehr egal, wie viele Exemplare erzeugt werden, also versuchen wir nur dann Exemplare anzulegen, wenn wir sie benötigen und ggf. schauen wir, welche wir wieder verwenden können. Da Reflection langsam ist, sollten wir sie vermeiden.

Manche Architekturmuster greifen nicht mehr: SOAP und XML ist deutlich langsamer als RESTful APIs und JSON. Hibernate und Co. sind auf dem Desktop in Ordnung, auf mobilen Geräten müssen die Frameworks einfacher sein, das bedeutet, dass wir entweder darauf verzichten oder auf die leichtgewichtigeren Varianten zugreifen. Andere Programmiermodelle stehen nicht oder nur eingeschränkt zur Verfügung.

Da wir in Android mit einem Komponentenmodell arbeiten, das sehr lose gekoppelt ist, mit ungewohnten Lebenszyklen, kann das OS zugunsten der Kerntelefonieaufgaben jederzeit in den Programmablauf eingreifen.

Wir sollten also anstreben, „konservativ" zu entwickeln, was den Ressourcenverbrauch betrifft. Unsere Anwendung darf nicht blockierend sein, „wenn es mal wieder länger dauert". Um die Bandbreite gut auszunutzen, müssen wir den „Protocol Overhead" und die übertragenen Daten minimieren. Die Kommunikation sollte wieder aufsetzen können, sodass wir nicht mehrmals die gleichen Daten übertragen. Wenn wir schon mit Webservern kommunizieren, sollte der „if-modfied-since"-Header konsequent genutzt werden, um redundante Abfragen zu minimieren.

Im Design unserer Informationen können wir versuchen, Daten schon auf dem Server zu minimieren, statt zu viel auf das Gerät zu transportieren und erst dort vergleichsweise teuer weiterzuverarbeiten.

Im zweiten Teil dieser Serie werden wir auf den Aspekt der UI-Entwicklung eingehen, eine der wesentlichen Aufgaben, wenn man eine mobile App entwickelt.

Jörg Pechau ist Geschäftsführer der I·C·N·H GmbH, die sich auf mobile Apps und das mobile Internet fokussiert. Zuvor hat er als Softwarearchitekt, Projektmanager und Entwicklungsleiter für Klein- und Großunternehmen gearbeitet. Im Bereich mobiler Services und Apps ist er seit 2003 tätig. Bei I·C·N·H entwirft er mobile Anwendungen und Systeme, berät, schult und hält Vorträge zum Thema.

Links & Literatur

[1] http://www.agiledata.org/essays/tdd.html

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

[3] http://www.samsung.com/de/consumer/mobile-device/mobilephones/smartphones/GT-I9300MBDDBT-spec

[4] http://www.apple.com/de/macbook-pro/specs/

[5] http://developer.android.com/guide/topics/manifest/application-element.html#largeHeap

[6] http://de.statista.com/statistik/daten/studie/180113/umfrage/anteil-der-verschiedenen-android-versionen-auf-geraeten-mit-android-os/

[7] http://techcrunch.com/2011/03/15/mobile-app-users-are-both-fickle-and-loyal-study/

[8] http://en.wikipedia.org/wiki/Information_design

[9] http://www.w3.org/TR/soap/

[10] http://www.ibm.com/developerworks/webservices/library/ws-restful/

Schneller, flüssiger, responsiver: Performanceverbesserungen in Android 4.1

Alles in Butter

Android musste sich in den letzten Jahren immer wieder die Kritik gefallen lassen, dass die Apps deutlich langsamer seien als z. B. in iOS. Insbesondere regelmäßige Aussetzer bei der Oberflächenbedienung fielen bei regelmäßiger Benutzung beider Betriebssysteme tatsächlich auf. Mit Android Jelly Bean (also 4.1) hat Google daher das Project Butter ins Leben gerufen, um diese Performanceprobleme anzugehen. Aber was hat sich tatsächlich getan und was kann der App-Entwickler gegen Performanceprobleme tun?

von Lars Röwekamp und Arne Limburg

Warum verhalten sich iOS-Anwendungen deutlich performanter als ihre Android-Pendants? Google hat dafür gleich mehrere Gründe ausgemacht und diese mit Android 4.1 (Jelly Bean) im Rahmen des Projects Butter behoben [1]. Dieses geht vor allem zwei Low-Level-Themen im Android-Kern an: die Animationen und die Touch-Events.

Flüssigere Animationen

Ein Standard-Android-Display läuft mit mindestens 60 Hz, d. h. 60 Bildern pro Sekunde. Die erste Maßnahme, um Android-Animationen flüssiger erscheinen zu lassen, war also, alle Animationen mit 60 Bildern pro Sekunde abspielen zu lassen. Das hat allerdings zur Folge, dass erhöhte Anforderungen an die Erstellung der einzelnen Bilder der Animation gestellt werden: Für jedes bleiben nur 16 Millisekunden. In dieser Zeit muss zunächst die CPU die Bilddaten aufbereiten, damit die GPU sie dann dem Display in einem Grafikbuffer zur Anzeige zur Verfügung stellen kann. Bis das geschehen ist, zeigt das Display einen anderen Buffer mit dem vorherigen Bild an. Diese Technik (also ein Buffer zum Beschreiben und einer zum Anzeigen) nennt man *Double Buffering*. Damit das Display den angezeigten Buffer erst wechselt, wenn er von der GPU komplett beschrieben ist, wird seit jeher eine Technik namens VSync angewendet, die GPU und Display synchronisiert. Wenn der Zyklus CPU-GPU-Display nicht in das 16-ms-Zeitfenster passt, wird ein Bild ausgelassen, was sich beim Benutzer als leichtes Ruckeln bemerkbar macht. Leider kann die CPU in Ice Cream Sandwich (und in früheren Android-Versionen) jederzeit mit der Berechnung beginnen. Da sind aber in der Regel schon wertvolle Millisekunden von den 16 zur Verfügung stehenden verstrichen, wodurch deutlich weniger Zeit pro Bild für GPU und Display zur Verfügung steht. Es kommt daher häufig zu besagtem Auslassen von Bildern. In Jelly Bean werden jetzt nicht nur Display und GPU, sondern vorgeschaltet auch die CPU via VSync synchronisiert. Das hat zur Folge, dass die CPU immer direkt am Anfang der 16 ms startet und für den Zyklus CPU-GPU-Display die vollen 16 ms verwendet werden können. Ausgelassene Bilder gibt es daher seltener, weil pro Bild mehr Zeit zur Verfügung steht.

Wenn dennoch ein Bild ausgelassen wird, kommt es zu einem weiteren Problem: In diesem Fall findet kein Wechsel der Buffer des Double Bufferings statt, sondern der alte Buffer wird weiterhin angezeigt, während die GPU einen weiteren Zyklus lang den nicht angezeigten Buffer befüllt, um das ausgelassene Bild nachzuholen. Viel schlauer wäre es allerdings, das Bild nicht nachzuholen, sondern tatsächlich wegzulassen. Dies ist beim Double Buffering aber nicht möglich, da alle Buffer belegt sind (der eine wird angezeigt, der andere mit dem alten Bild befüllt). Das ist der Grund für die Einführung des Triple Buffers in Jelly Bean: Kommt es zum Auslassen eines Bildes, wird einfach das Nachfolgebild in den dritten Buffer geschrieben und kann dann rechtzeitig angezeigt werden. Das alte Bild kann dann ausgelassen werden. Animationen werden nicht mehr verzögert, sondern lassen schlimmstenfalls ein Bild aus, was das Auge als weniger störend empfindet.

Flüssigere Touch-Bedienung

Auch um die Touch-Bedienung flüssiger werden zu lassen, hat Google sich ein paar Dinge einfallen lassen. Als erste Maßnahme wurde auch die Verarbeitung der Touch-Events mit VSync synchronisiert, damit sie nicht „querschießt“, sondern sich geschmeidig in den Verarbeitungsprozess einfügt. Zusätzlich wurde ein Mechanismus eingebaut, der antizipiert, an welcher Stelle sich der Finger zum Zeitpunkt des Screen-Refreshs befinden wird, um auch hier eine Verzögerung zu vermeiden.

Eine weitere, davon unabhängige Verbesserung ist der so genannte Input-Boost. Moderne Handy-Prozessoren legen sich schlafen, wenn sie nichts zu tun haben, um Strom zu sparen. Der Input-Boost sorgt dafür, dass, sobald der Benutzer den Bildschirm berührt, der Prozessor sofort aufgeweckt wird und mit voller Geschwindigkeit die Touch-Events verarbeiten kann.

Systrace

So weit, so gut. Aber was bedeutet Project Butter für den Entwickler? Und was kann er tun, um seine Apps möglichst responsiv zu entwickeln? Die gute Nachricht ist, dass Project Butter selbst keine direkte Auswirkung auf die Entwicklung eigener Apps hat. Den Performancegewinn bekommt man automatisch geschenkt. Und viel wichtiger: Auch Apps, die für ältere Android-APIs gebaut sind, laufen unter Jelly Bean flüssiger. Man muss also nicht auf das Jelly-Bean-API wechseln, um von den Performancegewinnen profitieren zu können.

Was kann man aber darüber hinaus tun, um seine App performanter zu gestalten? Mit dem SDK für Jelly Bean liefert Google zu diesem Zweck ein zusätzliches Tool namens Systrace [2] aus. Es dient dazu, die Performance der eigenen App im Zusammenspiel mit dem Betriebssystem und so z. B. auch mit der Grafikausgabe zu analysieren.

Systrace liefert eine grafische Repräsentation aller laufenden Threads der Applikation und des Betriebssystems über die Zeit. Hier kann man also z. B. Threads finden, die überdurchschnittlich lange ausgeführt werden, was immer problematisch ist.

Sucht man speziell nach Problemen mit der Grafikausgabe (z. B. unrunde Animationen), lohnt ein Blick auf den Prozess „SurfaceFlinger". Dieser sollte während einer Animation in regelmäßigen (kurzen) Abständen laufen, um das Display zu aktualisieren. Wenn der Benutzer der App nichts tut und auch keine Animation abläuft, sollte der SurfaceFlinger-Prozess auch nichts tun. Problematisch sind die Bereiche, in denen der Prozess zwar etwas tut, aber die Abstände zwischen den Ausführungen größer sind als normal. Das deutet dann auf die oben erwähnten ausgelassenen Bilder in der Animation hin, und hier lohnt sich ein genauerer Blick auf die anderen Prozesse, die parallel in Betriebssystem und Applikation laufen. Findet man zum Beispiel einen zur gleichen Zeit laufenden eigenen Thread, der teure Speicheroperationen o. ä. durchführt, hat man den Übeltäter gefunden.

Fazit

Mit Project Butter ist Google tatsächlich ein großer Schritt gelungen, um Android deutlich flüssiger und responsiver zu gestalten. Alle genannten Änderungen bekommt der Android-App-Entwickler geschenkt, d. h. auf allen Geräten, die Jelly Bean installiert haben, fühlen sich alle Apps automatisch schneller an.

Das Tool Systrace bietet dem Entwickler darüber hinaus die Möglichkeit zu analysieren, wo es in der eigenen App weiteres Optimierungspotenzial gibt. Mit dem in dieser Kolumne vorgestellten Know-how sollte sich die ansonsten doch etwas kryptische *systrace*-Ausgabe deutlich besser lesen lassen.

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] Project Butter (Präs.): http://video.golem.de/handy/8504/google-io-project-butter-wird-vorgestellt.html

[2] Systrace: http://developer.android.com/tools/debugging/systrace.html

[3] Jelly Bean: http://www.android.com/about/jelly-bean/

[4] Funktionsweise von Project Butter: http://www.androidpolice.com/2012/07/12/getting-to-know-android-4-1-part-3-project-butter-how-it-works-and-what-it-added/

Vorschau auf die Ausgabe 3.2013

Come Play with us!

Mit dem Play-Framework lassen sich hochskalierbare Enterprise-Anwendungen mit Scala und Java bauen. Gestartet als „Just another web framework“ hat Play spätestens mit der Übernahme in den Scala-Stack der Firma Typesafe einen wahren Siegeszug im Java-Webumfeld angetreten. Grund genug, Play im Detail unter die Lupe zu nehmen. Lernen Sie in unserem Tutorial, wie Sie selbst Ihre ersten Play-Anwendungen erstellen können.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 6. Februar 2013

Querschau

eclipse MAGAZIN — Ausgabe 1.2013 | www.eclipse-magazin.de

- IoT, M2M, MQTT: Eclipse im Internet der Dinge
- CSI und RCP: Eclipse für Chemiker
- DI und e4: Integration von OSGi Blueprint Services in Eclipse 4

entwickler magazin — Ausgabe 1.2013 | www.entwickler-magazin.de

- Licht 2.0: Post Processing als Revolutionsführer
- Agil: Requirements Engineering in Scrum
- Sozial: Social Project Management

MOBILE TECHNOLOGY — Ausgabe 1.2013 | www.mobile360.de

- Nicht nur für Entwickler: Das ist neu in iOS 6
- Money, Money, Money: App Store Optimization
- Android-Push: Alternative mit XMPP und ActiveMQ

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: 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:
Caroline Buck, Lars Drießnack, Thilo Frotscher, Werner Gross, Sven Haiges, Tam Hanna, Martin Heimrich, Marek Iwaszkiewicz, Arne Limburg, Bernhard Löwenstein, Daniel Lübke, Michael Müller, Jörg Pechau, Monika Popp, Lars Röwekamp, Sven Schirmer, Berthold Schulte, Dirk Schüpferling, Jens Schumann, Michael Seemann, Eugen Seer, János Vona, Martin Winandy

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