Wolken verdecken die Sonne

Im Jahr 2009 standen auf der Oracle OpenWorld Scott McNealy, Chairman von Sun Microsystems, und Oracle-CEO Larry Ellison gemeinsam auf der Bühne, um über die Übernahme von Sun durch Oracle zu reden. Angefangen von Java über das Solaris-Betriebssystem bis hin zur SPARC-CPU und den Sun-Speicher-Technologien sollte alles an Oracle gehen. Ellison bekräftigte sein Engagement für Sun als Hardware- und Open-Source-Software-Unternehmen und versprach, noch stärker in die SPARC-, Solaris-und MySQL-Produkte zu investieren als der bisherige Besitzer.

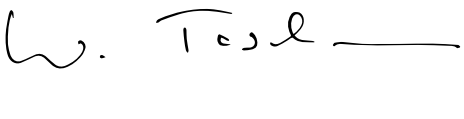
Von Java war an diesem denkwürdigen Abend weniger die Rede. So kam es auch, dass mit James Gosling, dem Urvater der Programmiersprache, zahlreiche Weggefährten dem Unternehmen den Rücken kehrten.

Auf der Hardware-Seite hat sich die Übernahme ausgezahlt. Der integrierte Stack aus Hard- und Software, der kurz davor in Kooperation mit Hewlett Packard durch die ersten Engineered Systems begonnen hatte, ist dank der Sun-Hardware ein großer Erfolg geworden. Die mit Betriebssystem, Datenbank, Middleware und Anwendungen, also allem, was der Kunde braucht, vorinstallierten und voll integrierten Exadata-Systeme sind heute in vielen Unternehmen im Einsatz.

In den letzten Jahren hat Larry Ellison Oracle auf die „Alles in die Cloud"-Strategie eingeschworen. Alles, was bis dahin war, wurde über Bord geworden, das Unternehmen bis auf den letzten Stein umgedreht und auf die Cloud eingeschworen. Kein Wunder, dass Sun-Juwelen wie Java hinter diesen Wolken verschwunden sind und von Oracle nur noch auf Sparflamme betrieben werden.

Von Larry Ellison ist überliefert, dass er sich in seinem Alter nur noch die Zahl „40" merken kann – 40 Milliarden Umsatz im Jahr und 40 Prozent Marge bei den Produkten. Jemand, der in 40 Jahren ein derart erfolgreiches Unternehmen aufgebaut hat, kann nicht viel falsch gemacht haben. Von daher ist es umso wichtiger, dass im Fall von Java eine aufmerksame und aktive Community den Lauf der Dinge überwacht und entsprechend beeinflusst.

Ihr



Wolfgang Taschner
Chefredakteur Java aktuell

Inhalt

Ein sehr hilfreiches Werkzeug, das die bestehenden Testmethoden ergänzt

Das passiert, wenn man einen Kicker digitalisiert und mit Amazon Alexa in einen Raum sperrt

3 Editorial

6 Das Java-Tagebuch

8 Update: Brauchen wir Java in der Cloud?
Andreas Chatziantoniou und Matthias Fuchs

12 Property Based Testing – eine Einführung
Nicolai Mainiero

16 Putting TDD to the test
Edwin Günthner

20 Continuous Database Integration mit Flyway
Sandra Parsick

25 Der sprechende Kickertisch
Marco Buss

31 JavaScript für Java-Developer – brauchen wir das wirklich?
Robert Rohm

Worauf bei der Einführung verteilten Arbeitens zu achten ist und wie man die Beteiligten schult

Continuous Integration und Feature Branches sind zwei allgegenwärtige Themen in der heutigen Software-Entwicklung

38 Coden von der Dachterrasse – in Rumänien
Steven Schwenke

43 Coding Continuous Delivery – Performance-Optimierung für die Jenkins-Pipeline
Johannes Schnatterer

47 „Es wird zurzeit alles getan, um Java in die richtige Richtung zu lenken …“
Andreas Badelt

48 Feature Branches und Continuous Integration sind kein Widerspruch
Sebastian Damm

54 Java-Architekturen dauerhaft sichern mit ArchUnit
Peter Gafert

59 Von Java Swing zu JavaFX – ein Textsystem macht sich selbstständig
Falk Tengler und Vasily Smeltsov

66 Alle iJUG-Mitglieder auf einen Blick

66 Impressum/Inserenten

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

22. November 2017

EE4J: Erste Project Proposals

In den letzten Wochen sind einige Project Proposals im Rahmen von EE4J bei der Eclipse Foundation gemacht worden, etwa zu Grizzly, JAX-RS, zu Jersey, Web Sockets und Tyrus (jeweils getrennt nach Spezifikation und Referenz-Implementierung) sowie jüngst zu Ozark (die MVC-RI). Wie der Name schon sagt, sprechen wir noch nicht vom eigentlichen Projektstart, aber im Rahmen der Proposals wird nicht nur der Scope geklärt, der während der Transition zu Eclipse in der Regel erst einmal feststeht (also das, was der jeweilige JSR beziehungsweise die Referenz-Implementierung mitbringt), sondern beispielsweise auch die Projektleiter und die Committer-Liste. Zu letzterer hieß es ja in den vergangenen Wochen, dass zunächst einmal die Expert-Group-Mitglieder eingebunden werden sollen – viele hatten allerdings noch nichts von Eclipse gehört. Das klärt sich nun hoffentlich alles. Wenn ich in den nächsten Wochen Langeweile haben sollte, mache ich mal einen Abgleich. Leider kann man die Liste der Proposals nicht nach dem anvisierten Top-Level-Projekt (EE4J) filtern.
https://www.eclipse.org/projects/tools/proposals.php

30. November 2017

Kotlin 1.2

Die von JetBrains (IntelliJ IDEA, WebStorm etc.) entwickelte Programmiersprache Kotlin folgt mit der neuen Version 1.2 dem Multi-Plattform-Pattern, wie es beispielsweise auch mit Scala/Scala.js möglich ist. Kotlin-Code soll sowohl für die JVM kompiliert als auch in JavaScript für den Browser umgewandelt werden können. Bei Kotlin kommt natürlich noch dazu, dass „JVM" durch die Kooperation mit Google auch „Android" bedeuten kann. Das neueste Release 2017.3 von IntelliJ IDEA bietet bereits Multi-Plattform-Unterstützung.
https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released

30. November 2017

Open Liberty Beta unterstützt jetzt EE 8 vollständig

IBM hat eine Beta der nächsten Version ihres Application-Servers WebSphere Liberty veröffentlicht, die jetzt Java EE 8 komplett unterstützt (Full und Web Profile). Weitere Beta-Versionen sollen in den kommenden Monaten folgen; wer jedoch nicht warten oder sowieso nicht die kommerzielle Version nutzen möchte, kann ja auch die täglichen Builds von Open Liberty nutzen, dem kürzlich freigegebenen Open-Source-Projekt, auf dem auch die kommerzielle Version basiert.
https://tinyurl.com/y9knvd6y

5. Dezember 2017

Switch-Expressions für Java

Java soll Switch-Expressions (zusätzlich zu den vorhandenen Switch-Statements) bekommen. Damit sollen in Zukunft mächtigere und besser lesbare Ausdrücke entstehen, inklusive Pattern Matching, wie man sie aus funktionalen Sprachen kennt. Die Arbeiten an verbessertem Pattern Matching für Java waren wohl der eigentliche Auslöser. Ein entsprechender Entwurf für ein „JDK Enhancement Proposal" (JEP) mit der Nummer 8192963 ist angelegt worden.
https://tinyurl.com/y7olxfx4

5. Dezember 2017

The Android Wars Reloaded

Oh nein, es geht schon wieder los! Oracle und Google haben sich im langjährigen Rechtsstreit um Android gestern erneut vor Gericht gesehen. Nachdem ein Bezirksgericht in San Francisco letztes Jahr Google recht gegeben hatte (Stichwort: „fair use" der Java-APIs in Android) und Oracle im Frühjahr dagegen angegangen ist, geht nun das neue Verfahren vor einem Bundes-Berufungsgericht los. Beim Nachlesen der bisherigen Historie habe ich zumindest eine kleine Anekdote erfahren, die mir wohl bislang entgangen war: Der Richter des vorherigen Verfahrens, William Alsup, hatte in Vorbereitung seines Einsatzes extra gelernt, in Java zu programmieren, um die Argumente der Parteien besser zu verstehen. So viel Einsatz würde ich mir hier auch öfter wünschen – bei manchen Verhandlungen mit IT-Bezug vor deutschen Gerichten fragt man sich ja schon, wo die Grenze zwischen entschuldbarer Ahnungslosigkeit und bewusster Ignoranz verläuft.
https://www.theregister.co.uk/2017/02/11/oracle_refuses_to_let_java_suit_die/

16. Dezember 2017

JBoss 7.1 EAP bleibt bei Java EE 7

JBoss 7.1 EAP ist raus, setzt aber weiterhin auf Java EE 7. Vielleicht will man sich bei Red Hat nicht die Mühe machen, den Application-Server erst für EE 8 und dann für das inhaltlich wohl mehr oder weniger identische erste EE4J-Release zu zertifizieren? Allerdings wird Letzteres noch ein wenig auf sich warten lassen. Wahrscheinlicher ist wohl, dass sie einfach noch nicht so weit sind, aber nicht noch länger mit einem neuen Release (unter anderem mit dem neuen Security-Subsystem „Elytron") warten wollten. Ein paar Einzelteile von EE 8 stecken jedoch schon drin: HTTP/2 wird zum Beispiel unterstützt – auch ohne den vollen Umfang von Servlet 4.0.
https://www.redhat.com/en/blog/red-hat-releases-jboss-eap-71

17. Dezember 2017

JCP: Öffentliches EC Meeting

Das letzte Meeting des JCP Executive Committee 2017 ist eine öffentliche Telefonkonferenz, der Mitschnitt ist jetzt auf „jcp.org" zu finden. Neben dem üblichen organisatorischen Geplänkel und Status-Reports zu Themen, die sich auf den JCP selbst beziehen, dreht es sich im Wesentlichen darum, die neuen Mitglieder im EC zu begrüßen und alle in der Runde nach ihren Schwerpunkten für 2018 zu fragen. Einige Mitglieder fehlen, auch die Eclipse Foundation ist nicht dabei, was natürlich gerade jetzt sehr schade ist. Von den Übrigen kommt zunächst mal nicht viel Überraschendes – bei den Firmen fällt es meist nicht schwer zu raten, wo sie im EC ihre Schwerpunkte setzen werden: Hazelcast bei Performance-Themen und JCache für EE4J, Azul bei den JSRs für das OpenJDK 10 und 11 und den neuen Prozes-

sen rund um den Release Train, JetBrains bei den Sprach-Features, Credit Suisse und Red Hat bei den Themen EE4J und Cloud, HP und Intel unter anderem beim Thema „Persistent Memory", IBM hängt fast überall mit drin … Auch das Stichwort „Java ME" fällt einige Male. Tatsächlich gibt es auch eine kleine, aber feine Runde von ME-Enthusiasten, die in der „Java ME Working Group" als einer Art Unterausschuss des EC das Thema weitertreiben. Die Meeting-Protokolle der Gruppe sind nur mäßig interessant; sollte jemand allerdings tatsächlich Interesse daran haben, sind dort auf jeden Fall die besten Ansprechpartner gelistet, die man sich wünschen kann.
https://community.oracle.com/community/java/jcp/executive-committee

19. Dezember 2017

EE4J PMC Meeting

Das Protokoll der letzten Sitzung des EE4J „Project Management Committee" (PMC) im Jahr 2017 ist veröffentlicht. Der (Marketing-)Name für das gesamte Projekt ist weiterhin ein Thema, die Diskussion wird jedoch auf 2018 verschoben. Ein anderer Aspekt der Namensgebung wird hingegen konkretisiert: Die vom JCP geerbten API-Projekte sollen nicht das „API" im Namen tragen, sondern ein „Project for", also „Eclipse Project for JAX-RS", während Implementierungsprojekte einfach das Eclipse voranstellen, also „Eclipse Jersey". Eine Entscheidung darüber, wie mit Spezifikationen umzugehen ist (der Fokus des JCP), wird auch verschoben. Interessanterweise steht da, dass das Konzept der Referenz-Implementierung (RI) aus dem JCP in EE4J „(noch) nicht" existiert. Eine RI wird ja im MicroProfile-Projekt explizit ausgeklammert. Also abwarten, ob EE4J sich eher an MicroProfile oder an den JCP annähert. Bis zu einer Entscheidung soll jedenfalls nur von „einer", nicht von „der" Implementierung geredet werden. Ein weiteres Thema ist die Projekt-Besetzung: Bezüglich der Einzelprojekte wird die Empfehlung ausgesprochen, möglichst je eine Doppelspitze zu besetzen. Als initiale Committer eines API-Projekts sollen wie bereits diskutiert alle Mitglieder des korrespondierenden JSR eingeladen werden. Für die Implementierungs-Projekte „kann" als Kriterium eine „minimum number of commits" des bisherigen Projekts in GitHub herangezogen werden. Frage für Detailversessene: „Sind die Projekte wirklich alle bereits in GitHub gewesen?".
https://www.eclipse.org/ee4j/minutes

20. Dezember 2017

Die ersten beiden EE4J-Projekte

Und jetzt gibt es auch die ersten beiden richtigen EE4J-Projekte, nicht nur Proposals: Es sind: Eclipse Yasson, die Referenz-Implementierung von JSON Binding – das „Eclipse" muss jetzt immer davor genannt werden, darauf legt die Eclipse Foundation großen Wert – und Eclipse EclipseLink. Ok, für das existierende Projekt, das nur unter ein neues Top-Level-Projekt kommt, reicht wohl ein „Eclipse". Die eigentliche Arbeit an EE4J kann dann zumindest für diese beiden Teile losgehen.
https://projects.eclipse.org/projects/ee4j.yasson

23. Dezember 2017

The Java Release formerly known as 18.3

JDK 10, das nun definitiv nicht mehr JDK 18.3 heißen wird, nimmt Formen an. Die Implementierung der Features – unter anderem „Local-Variable Type Inference", ein verbessertes API für Garbage Collectors und ein paralleler Full GC für den G1, um die „worst case"-Zeiten zu verbessern – ist abgeschlossen. Mitte Dezember wurde die „Rampdown Phase One" eingeleitet, es werden also von nun an nur noch Prio-1-bis-3-Bugs behoben. Ab dem 18. Januar sollen es dann nur noch „Showstopper"-Bugs sein („Rampdown Phase Two"). Am 20. März soll die Freigabe erfolgen (hoffentlich nicht „pünktlich wie ein ICE").
http://openjdk.java.net/projects/jdk/10

3. Januar 2018

MicroProfile 1.3

Nicht mehr ganz im geplanten Q4/2017, aber nur drei Tage später ist MicroProfile 1.3 nach einer kurzen Review-Phase veröffentlicht worden. Es enthält gegenüber der Version 1.2 drei neue APIs: OpenTracing 1.0 (ein Standard zum Verfolgen von Transaktionen oder Workflows innerhalb verteilter Systeme), OpenAPI 1.0 (früher als „Swagger" bekannt), und scheinbar relativ kurzfristig hinzugefügt auch einen „Type-safe REST Client 1.0". Config (1.2) und Metrics (1.1) sind gegenüber MicroProfile 1.2 aktualisiert worden. In den Release-Notes wird ausdrücklich darauf hingewiesen, dass für 1.3 keine Updates vorgenommen worden, um APIs auf die Java-EE-8-Spezifikation anzuheben (CDI 2.0, JAX-RS 2.1, JSON-P 1.1). Das soll im Release 2.0 geschehen, das für den 31. März 2018 angekündigt ist. Von einem Release 1.4 wird intern auch gesprochen, allerdings wohl eher als inoffizielles Projekt, das keine großen neuen Features bringt, sondern Arbeiten bündelt, die Entwicklern die Arbeit mit MicroProfile nahebringen sollen (Tutorials, Guides und Hands-on Labs, Beispiele, Videos etc.).
http://microprofile.io/blog/2018/01/eclipse-microprofile-1.3-available

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich in der DOAG Deutsche ORACLE-Anwendergruppe e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit dem Jahr 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Brauchen wir Java in der Cloud?

Andreas Chatziantoniou, DevTops GmbH und Matthias Fuchs, esentri AG

Seit einiger Zeit mehren sich die kritischen Stimmen: Java hat seine Zeit gehabt; JEE ist zu groß und zu schwer zu verstehen; kann man in der Cloud nicht auch ganz gut ohne Java (JEE) leben? Der Artikel betrachtet diese Entwicklung und geht dabei auf die verschiedenen Aspekte ein, um dann zu einer Antwort darauf zu kommen, ob JEE in der Cloud noch eine Daseinsberechtigung hat.

Java als Sprache kennt einige Facetten, und die Werbe-Aussage von Oracle meint, eine sehr große Anzahl von Geräten würden Java brauchen/ausführen/damit arbeiten. Warum also die Frage, ob Java noch zeitgemäß sei, wenn dies doch so eine Erfolgsgeschichte ist?

Mit der Anzahl der Geräte, die Java unterstützen, und insbesondere mit den Java Cards (Smart Cards auf denen Java läuft), aber gerade auch im Bereich IoT erscheint es so, als ob niemand an der Daseinsberechtigung – oder gerade der Dominanz – von Java im Bereich Mobile/BluRay zweifelt. Die Autoren sind daher davon überzeugt, dass die Diskussion sich nicht an Java als Sprache (in verschiedenen Ausprägungen) orientiert, sondern vielmehr die Frage nach der Zukunft eines Java-basierten Application-Servers gestellt wird. In Bezug auf das JEE-Modell *(siehe Abbildung 1)* können wir diese Kritik verstehen.

Nach vielen Jahren in diversen Projekten, bei denen JEE eingesetzt wurde, lässt sich sagen, dass JEE sehr vollständig ist, aber nur selten alle Komponenten genutzt werden. Nachfolgend eine Übersicht (bei Weitem nicht vollständig) mit Fragen, die sich ein JEE-Application-Server gefallen lassen muss:

- *Präsentation*
  Viele Möglichkeiten, aber nutze ich alle?
- *Business Logic*
  Servlets, JSP, EJBs – sinnvoll, aber alle gleichzeitig?
- *Common Services*
  - JAX-WS, RMI, JTA, JDBC, JMS, JMX, JAAS, JNDI
  - Wahrscheinlich haben 80 Prozent und mehr aller Anwendungen nur JDBC, JMS und EJB
  - JAAS? weblogic/welcome1
- *Backend*
  - Database – ja, gerne
  - Web Services
  - Directory
  - CORBA – in 2017?

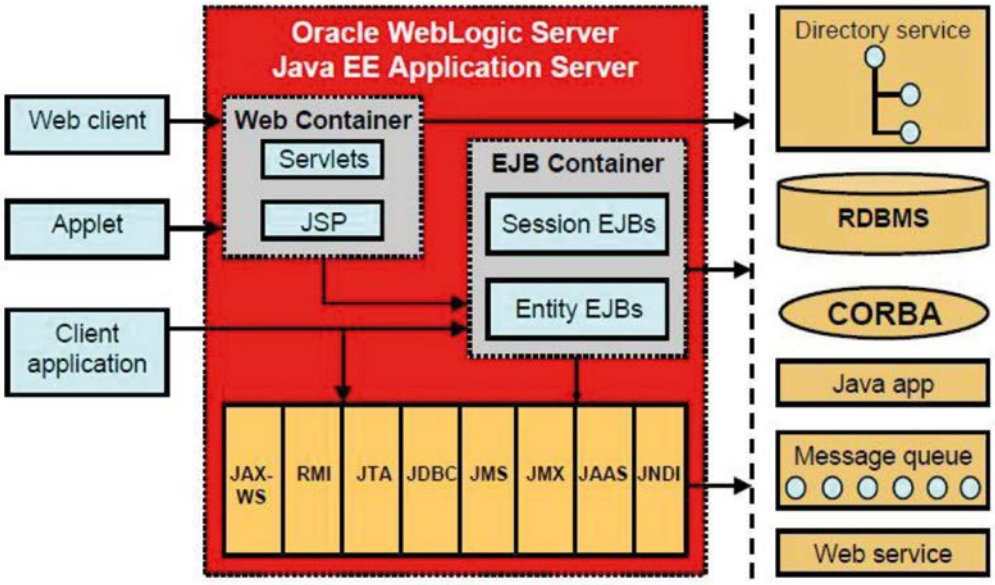


Abbildung 1: Überblick JEE

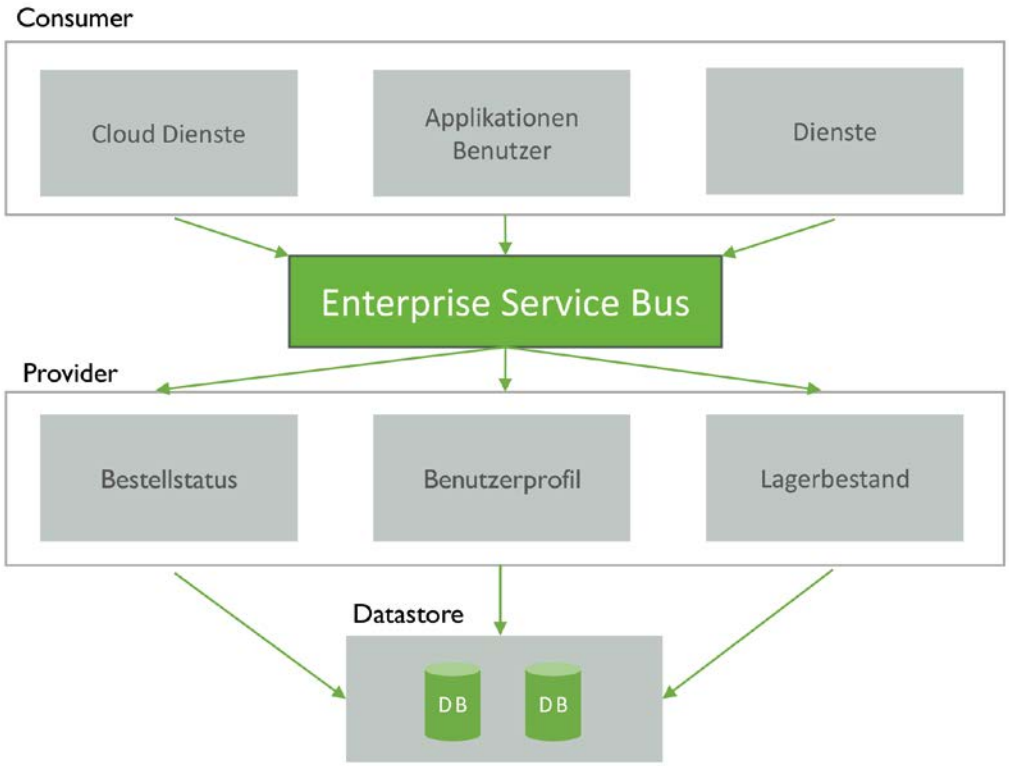


Abbildung 2: SOA-Architektur

Hier wird nach Meinung der Autoren deutlich, aus welcher Richtung die Kritik kommt. Tatsächlich ist die heutige Aufstellung eines Application-Servers darauf fokussiert, alle möglichen Aspekte zu bedienen. Dies hat zur Folge, dass der Umfang des Gebotenen sehr groß ist. Das bedeutet natürlich auch eine dementsprechende Lernkurve. Weiterhin ist der Fußabdruck eines solchen Application-Servers relativ groß und bis alle Komponenten geladen und initialisiert sind, können schon mal ein paar Minuten verstreichen. Wir sollten also auf die Suche gehen, um Lösungen zu finden, die genau den Umfang haben, den ein Projekt braucht, trotzdem skalierbar sind und gegebenenfalls Erweiterungen zulassen, falls sich die Anforderungen ändern.

Von SOA bis Microservice

Serviceorientierung (SOA) war in den letzten Jahren häufig die Basis der Software-Architektur. Es ist eine Sichtweise, wie verteilte Dienste und Services genutzt werden können. SOA-Services werden von unterschiedlichen Bereichen verantwortet und betrieben. Die Dienste sind unabhängig und können über eine Schnittstelle angesprochen werden. Um bei einer höheren Zahl von Diensten die Übersicht zu behalten, werden diese meist über einen zentralen Vermittler verbunden, die Middleware. Hier kommt meist ein Servicebus zum Einsatz, etwa der Oracle-Servicebus. Sowohl der Servicebus als auch die einzelnen Services werden oft auf Application-Servern entwickelt. Datenbanken sind über den Application-Server angesprochen, somit verwenden unterschiedliche Serviceaufrufe oftmals gleiche Datenbanken *(siehe Abbildung 2)*.

Der Microservice-Ansatz geht noch einen Schritt weiter. Ziel ist es, kleine Prozesse zu entwickeln, die unabhängig lauffähig und unabhängig einrichtbar sind. Eine Grundannahme von Microservices ist es, Datenbanken oder andere Storage-Komponenten nicht gemeinsam zu nutzen. Es ergeben sich daher unabhängige Services, die komplett unabhängig entwickelt werden können. Eine zentrale Komponente wie ein Servicebus entfällt *(siehe Abbildung 3)*.

Das Microservice-Design findet in der heutigen Zeit immer mehr Zuspruch. Im Rahmen der Services wird nur ein API bereitgestellt. Die Datenverarbeitung in den Services oder die Entwicklungssprache ist nicht vorgeschrieben. Zusätzlich sind die Services zustandslos, damit es möglich ist, beliebig viele Instanzen einzelner Microservices zu starten. Eine Kommunikation zwischen den Services ist nicht vorgesehen, somit sind Funktionen auf Basis eines Clusters oder eines Remote-Calls, wie sie in Application-Servern vorhanden sind, nicht notwendig.

Java und Microservices

In einem Microservice-Design werden die einzelnen Komponenten völlig unabhängig entwickelt, wie es im Rahmen von agilen Entwicklungsprozessen häufig gefordert wird. Die Kommunikation erfolgt über definierte Schnittstellen (APIs). Eine gemeinsame Basis wie eine gleiche Programmiersprache ist nicht notwendig (Polyglott). Zusätzlich sind die Services meist im Rahmen von Containern bereitgestellt, was problemlos auch unterschiedliche Betriebssysteme möglich macht. Je nach Ziel der Services verwendet man die Sprache oder die Implementation, die eine sinnvolle Umsetzung ermöglicht.

Java bietet sich als Programmiersprache an, ein Wechsel auf Go oder Python ist aber jederzeit möglich, da die Services komplett getrennt ent-

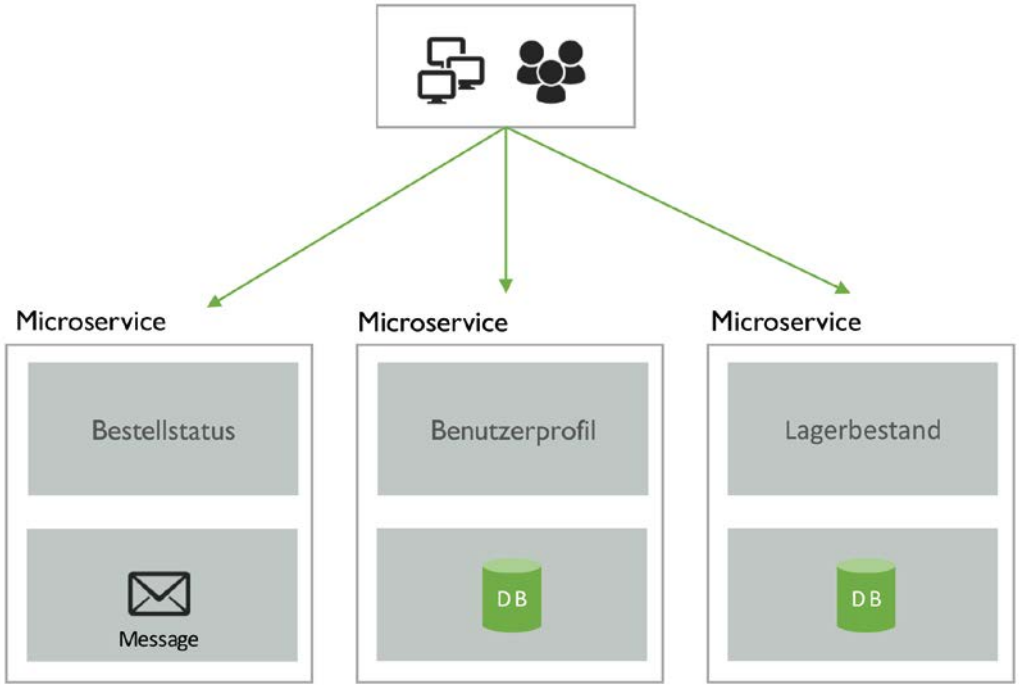


Abbildung 3: Microservice-Architektur

wickelt werden. Ebenso sollte bei Microservices die Datenhaltung getrennt für jeden Container beziehungsweise für jede Komponente erfolgen, um die Unabhängigkeit der Services zu gewährleisten. Dies bietet die Möglichkeit, die optimale Kombination aus Entwicklungsumgebung, Programmiersprache und Speicherpersistierung (Datenbank) zu wählen.

Microservices und Cloud

Es gibt mehrere Gründe, mit Microservices in der Cloud zu starten. Beim Wechsel von SOA-Architekturen auf Microservices ist ein großer Umbau der Infrastruktur notwendig; es ist sinnvoll, dies mit einem Wechsel in eine Cloud zu verbinden. Weiterhin ist das Wesen von Microservices, diese durch Erhöhen oder Verringern der Anzahl der Services (Container) der notwendigen Performance anzupassen. Diese Anpassung der Ressourcen ist in der Cloud sehr einfach möglich, da die Kosten der Infrastruktur teilweise sekundengenau berechnet werden können. Hier sehen wir zwei Richtungen entstehen, die in Zukunft die Einsatzmöglichkeiten von Java beeinflussen werden:

- Container
- Cloud

Im Bereich der Container-Technologie wird sichtbar, dass ein Problem der Application-Server- und Java-Technologie gelöst wird. Oft passiert es, dass auf der Strecke von der Entwicklung über Integration und Abnahme zur Produktion die Umgebungen nicht identisch sind. Dies hat viele Ursachen: Entwickler mit vielen Freiheitsgraden und Geltungsbedürfnissen; Systemadministratoren, die beim Bereitstellen etwas übersehen; manuelle und fehlerhafte Bereitstellungsprozesse; Topologie- und Hardware-Unterschiede.

Beim Einsatz von Containern ist diese Unzulänglichkeit behoben; der Container ist in allen Projektphasen identisch. Hierdurch verschwindet die „Configuration Drift" und damit auch eine Anzahl von Problemen. Nun kommt der Umfang des Application-Servers wieder zum Tragen. Ein vollständiger Application Server wie Oracle WebLogic Server ist im Container erhältlich. Man kann also relativ schnell eine bestehende Anwendung in einen Container packen und in Produktion nehmen.

Da aber im Container bezüglich Skalierung und Hochverfügbarkeit andere Möglichkeiten vorhanden sind, wäre der Übergang zu kleineren, modular aufgebauten Application-Servern denkbar. Kandidaten dafür wären WildFly Swarm, Payara Micro oder TomEE. Diese sind im Container-Umfeld eher anzutreffen als ein klassischer JEE-Full-Stack-Application-Server. Dies hat natürlich auch etwas mit der Lizenzpolitik des Application-Server-Herstellers zu tun. Insbesondere wenn der Application-Server-Anteil am Gesamtsystem eher klein ist, machen sich dessen Kosten schnell bemerkbar. Zusätzlich sind Cluster-Funktionalitäten, wie sie die großen Application-Server-Anbieter im Programm haben, aus Microservice-Sicht nicht notwendig. Application-Server mit geringerem Funktionsumfang bieten somit alle Möglichkeiten, die für eine Microservice-Architektur notwendig sind.

Der zweite Bereich ist die Cloud. Hier ist sowohl für Entwickler als auch für Betreiber ein Umdenken erforderlich. War man bisher vollständig in der Lage, alle Bereiche seiner Infrastruktur zu benutzen, zu konfigurieren, zu untersuchen oder zu integrieren, muss man sich nun mit einem (Web-basierten) User-Interface begnügen. Zugang zu den Logfiles aller Komponenten ist oft nur über Dashboards möglich, der Betrieb wird durch den Cloud-Dienstleister übernommen. Es geht sogar so weit, dass nur der Code beim Anbieter eingespielt wird; welche Server-

Technologie im Hintergrund arbeitet, ist nicht mehr relevant. Im Extremfall spielt man nur noch Funktionen ein (Serverless), somit entfällt sogar die Bindung an die Programmiersprache. Beim Oracle Java Cloud Service ist diese „Encapsulation" am weitesten umgesetzt; man kann demnach auch über Java-as-a-Service sprechen. Der eingebaute JEE-Application-Server (WebLogic) ist nicht mehr direkt sichtbar.

Warum Java und nicht ...

Trotz der Möglichkeit, andere Sprachen einzusetzen, wird für Microservices meist Java verwendet. Dies hat mehrere Gründe. Einerseits gibt es viele Entwickler, die Java beherrschen. Durch einen Wechsel der Sprache müsste man umlernen oder neue Entwickler einstellen. Zudem ist der Produkt-Support mit Java vorhanden und in vielen Projekten erprobt. JEE-Funktionalität geht meist verloren, da es aus Microservice-Infrastruktur- und Architektur-Sicht nicht benutzt wird. Es findet eine Verlagerung von Funktionalität aus dem Application-Server-Umfeld hin zur Infrastruktur statt.

Andere Sprachen wie gerade Python oder Go holen auf. Support ist möglich und es gibt bereits viele Projekte, die erfolgreich damit umgesetzt wurden. In Zukunft werden sich die Sprachen durchsetzen, die für den Use-Case am besten geeignet sind, es ist also eine Abschätzung pro Service notwendig, welche Sprache mit den vorhandenen Ressourcen wie Personal, Support und der geforderten Funktionalität das beste Ergebnis und vor allem eine schnelle und einfache Umsetzung verspricht. Hier ist insbesondere der Aspekt von großen Enterprise Applications zu beachten.

Der Charme von Microservices liegt zurzeit oft darin, leicht zu extrahierende Teile eines Monolithen neu zu bauen. Bieten diese Microservices jedoch den Umfang, den ein (weltweit tätiges) Unternehmen benötigt, und sind Komponenten wie Security oder Data Management darin abgebildet? Wir wissen, dass sich mit Microservices Anwendungen entwickeln lassen, wir wissen aber noch zu wenig davon, ob diese ein vollwertiger Ersatz von JEE-basierten Systemen sind. Mit anderen Worten: Selbst bei kurzen Lebenszyklen von Software (und dem Entwicklungsstatus von anderen Sprachen beziehungsweise Application-Servern) ist Java eine Langzeitgarantie, die für Unternehmen eine große Planungssicherheit bietet.

JEE ist tot ...

Zum heutigen Zeitpunkt (Java 9 wurde gerade veröffentlicht, Java 10 ist in Entwicklung) erscheint den Autoren die Aussage gerechtfertigt, dass Java eine gute Wahl ist, wenn es um den Bau von großen Enterprise Applications geht. Dies ist insbesondere dann wahr, wenn die Erwartung der Laufzeit mehrere Jahre übersteigt. Zudem kann man annehmen, dass Organisationen, die JEE-basierte Application-Server einsetzen und für ihre Anwendungen davon abhängig sind, auch noch mehrere Jahre damit arbeiten werden. In diesem Sinne ist JEE noch lange nicht abgeschrieben.

Da viele JEE-Anwendungen hauptsächlich auf EJB basieren, werden wir in der Zukunft Umbaumaßnahmen sehen, bei denen Spring oder ähnliche leichtgewichtige Frameworks statt EJB zum Einsatz kommen. Dies wird auch Auswirkungen auf den JEE-basierten Application-Server-Markt haben. Die traditionellen Hersteller dieser Application-Server schalten langsam aber sicher auf Angebote um, die entweder in der Cloud laufen (Platform as a Service) und damit die Komplexität zurückdrängen – oder bieten kleinere Application Server an, die in einer modularen Version im Container (generell Docker) lauffähig sind.

Fazit

Die Schlussfolgerung kann eigentlich nur sein: Java is alive and kicking! In den nächsten Jahren werden sowohl Java als auch JEE-basierte Application-Server im Enterprise-Bereich dominant sein und dabei immer öfter durch modernere Entwicklungen ergänzt werden. Konkurrenz durch andere Sprachen braucht Java nicht zu befürchten, Organisationen und Entwickler werden jedoch mehr Auswahl haben, wenn es um die Sprache und/oder den Application-Server geht.

Andreas Chatziantoniou
andreas.chatziantoniou@devtops.gmbh

Andreas Chatziantoniou ist seit dem Jahr 1988 in der IT unterwegs, seit dem Jahr 1998 ausschließlich im Oracle-Technologie-Umfeld. Hier liegt sein Fokus seit dem Jahr 2001 stark auf Application-Servern und der Integration von Systemen. Die Einführung solcher Systeme bei Organisationen und die Übergabe in den Betrieb sind hier immer wiederkehrende Themen. Seit einiger Zeit umfasst sein Arbeitsumfeld auch Container-Technologien und Cloud-Dienste.

Matthias Fuchs
matthias.fuchs@esentri.com

Matthias Fuchs ist seit über 10 Jahren im Umfeld von Oracle-Datenbanken sowie im Bereich Architektur und Infrastruktur von Enterprise-Anwendungen tätig. Er hat seit mehr als 5 Jahren Erfahrungen mit Oracle Engineered Systems wie Oracle Exadata, in Bezug auf Architektur, Tuning und Auditing. Die letzten Jahre erweiterte er sein Beschäftigungsfeld auf NoSQL-Datenbanken und widmete sich IT-Architekturen im Cloud-Umfeld. Momentan arbeitet er in Projekten mit Amazon Webservices oder der Oracle Cloud.

Property Based Testing – eine Einführung

Nicolai Mainiero, sidion

Testen ist schwer. Sämtliche Pfade durch einen Code-Abschnitt aufzuzählen, ist aufwändig und die Anzahl der Parameter lässt die möglichen Kombinationen für Eingaben in die Höhe schnellen. Dieser Artikel stellt ein sehr hilfreiches Werkzeug vor, das bestehende Testmethoden ergänzt und den erwähnten Schwierigkeiten begegnet. Property Based Testing ist eine Methode, um zu prüfen, ob eine Funktion für alle Eingaben korrekt ist. Dies wird erreicht, indem sie mit zufällig generierten Daten ausgeführt wird.

Testen ist nach wie vor die am weitesten verbreitete Methode, um Softwarequalität sicherzustellen, und mit dreißig bis vierzig Prozent der Gesamtkosten *[1]* ein nicht zu vernachlässigender Kostenfaktor in der Entwicklung. Man kann die Kosten reduzieren, indem man die Tests zu einem Großteil automatisiert. Die Testpyramide von Mike Cohn *[2]* liefert ein Modell dafür, wie die Tests über die verschiedenen Schichten einer Anwendung verteilt sein sollten. Der größte Anteil entfällt hierbei auf die sogenannten „Unit-Tests", die sich jeweils um einen sehr kleinen und lokalen Bereich kümmern und nur diesen testen.

Als verbreiteter Prozess für das Entwickeln von Unit-Tests hat sich das Test Driven Development und später darauf aufbauend das Behavior Driven Development entwickelt. Testgetriebene Entwicklung nach Kent Beck *[3]* gliedert sich in folgende drei Phasen:

1. Red: Schreibe einen Test, der ein neues zu programmierendes Verhalten (Funktionalität) prüfen soll. Fange mit dem einfachsten Beispiel an. Existiert die Funktion bereits, kann dies auch ein gemeldeter Fehler oder eine neu zu implementierende Funktionalität sein. Dieser Test wird vom vorhandenen Programmcode erst einmal nicht erfüllt und muss dementsprechend fehlschlagen.
2. Green: Ändere den Programmcode mit möglichst wenig Aufwand und ergänze ihn, bis er nach dem anschließend angestoßenen Testdurchlauf alle Tests besteht.
3. Clean up (Refactoring): Entferne Wiederholungen (Code-Duplizierung), abstrahiere, wo nötig, und richte ihn nach den verbindlichen Code-Konventionen aus. In dieser Phase darf kein neues Verhalten eingeführt werden, das nicht durch Tests schon abgedeckt ist. Nach jeder Änderung werden die Tests ausgeführt. Schlagen sie fehl, darf die offenbar fehlerhafte Änderung nicht übernommen werden. Ziel des Aufräumens ist es, den Code verständlich und leicht nachvollziehbar zu machen.

Diese drei Schritte werden so lange wiederholt, bis der Code die gewünschte Funktionalität liefert, es keine bekannten Fehler mehr gibt und dem Entwickler keine weiteren sinnvollen Tests mehr einfallen. Dieses eigentlich sehr gut strukturierte Vorgehen birgt jedoch Risiken. Wenn die Funktionalität nicht ausreichend beschrieben ist oder wenn in dem Refactoring-Schritt unsauber gearbeitet wird, kann es passieren, dass die Codequalität trotz Tests sinkt.

Folgendes einfaches Beispiel soll diese Problematik demonstrieren.

In den *Listings 1* bis *4* sind Tests und die dazu passenden Implementierungen für eine Addierfunktion aufgelistet. *Listing 2* zeigt die einfachste mögliche Implementierung, um den Test aus *Listing 1* zu bestehen. In *Listing 3* wird dann ein weiterer Testfall hinzugefügt. *Listing 4* zeigt wiederum den Code, der nötig ist, um auch diesen Testfall zu bestehen. In diesem Beispiel wurde auf das Refactoring verzichtet, um aufzuzeigen, dass es entscheidend ist, nicht nur die Anforderungen im Test und in der Implementierung stur zu erfüllen, sondern die Anforderungen auch zu verstehen, selbst wenn sie nur exemplarisch formuliert worden sind. Property Based Testing zwingt einen dazu, sich mit den Anforderungen an eine Funktion genauer auseinanderzusetzen.

QuickCheck

Die Idee von Property Based Testing geht auf die Veröffentlichung einer Bibliothek für Haskell zurück. Koen Claessen und John Hughes *[4]* stellten die folgenden Anforderungen an ein Werkzeug zum randomisierten Testen von Haskell-Programmen: Durch eine formale Spezifikation soll es automatisch feststellen können, ob ein Test erfolgreich ist oder nicht, und selbstständig Testdaten erzeugen können. Zu deren Erstellen soll es dem Tester eine Möglichkeit bieten zu bestimmen, wie Testdaten generiert werden. Außerdem soll es leichtgewichtig sein. Das Ergebnis war QuickCheck, das all diese Anforderungen erfüllen konnte. Dieses Werkzeug hat dann den Begriff „Property Based Testing" geprägt.

Property Based Testing

Im Gegensatz zum klassischen Unit-Testing werden hier keine konkreten Zusicherungen wie zum Beispiel in *Listing 1* gemacht, sondern es werden Eigenschaften der Methode gesucht, die für alle Eingabe-Parameter gelten. Diese Eigenschaften werden dann mit vielen automatisch generierten, randomisierten Daten überprüft, um die Implementierung zu testen.

Für die Erzeugung dieser Daten gibt es sogenannte „Generatoren". Sie kennen die Domäne und wissen, wie Eingabe-Parameter zu erzeugen sind. Damit das Fehlschlagen eines Tests nicht mit komplexen Eingaben nachvollzogen werden muss, existiert das sogenannte „Shrinking". Es ist ein wesentliches und zugleich das spannendste Feature von Property Based Testing. Damit wird eine Eingabe so lange vereinfacht, bis eine kleinstmögliche Eingabe gefunden ist, die den Test fehlschlagen lässt. Dadurch ist es meist sehr einfach, den Fehler in der Implementierung nachzuvollziehen und zu korrigieren.

JUnit-Quickcheck

Es gibt von QuickCheck mehr als fünfzig Implementierungen in mehr als vierzig Sprachen, unter anderem auch Clojure, Scala oder Java. Allein für Java existieren mindestens fünf Implementierungen, um Property Based Testing einzusetzen. Die nachfolgenden Beispiele sind alle mit JUnit-Quickcheck *[5]* realisiert. Sie lassen sich aber einfach auf andere Implementierungen übertragen.

Listing 5 zeigt, wie man mit der Bibliothek einen Property Based Test schreibt. Wichtig sind die Annotationen „@RunWith(JUnitQuickcheck.class)" und „@Property", mit denen JUnit angewiesen wird, einen speziellen Test-Runner zu verwenden. Dieser Test-Runner versteht neben den üblichen JUnit-Annotationen auch „@Property". Eine damit annotierte Methode wird mit Eingaben aufgerufen, die durch einen passenden Generator automatisch entstanden sind. In diesem Fall werden also zufällige Strings erzeugt und überprüft, ob die „length()"-Methode erwartungsgemäß funktioniert, also die Summe der Längen zweier Strings gleich der Länge des verketteten Strings ist. Standardmäßig wird die Testmethode hundertmal aufgerufen und die Zusicherung geprüft. Dies erfolgt wie in klassischen JUnit-Tests auch mit „assertEquals()".

In diesem Beispiel werden Basis-Typen als Parameter benutzt.

```
public class AdderTest {
  @Test
  public void testApp() {
     assertEquals(4, Adder.sum(1, 3));
  }
}
```

Listing 1

```
public class Adder {
  public static int sum(int a, int b) {
     return 4;
  }
}
```

Listing 2

```
public class AdderTest {
  @Test
  public void testApp() {
     assertEquals(4, Adder.sum(1, 3));
     assertEquals(7, Adder.sum(4, 3));
  }
}
```

Listing 3

```
public class Adder {
public static int sum(int a, int b) {
  switch (a){
     case 4:
        return 7;
     }
     return 4;
  }
}
```

Listing 4

```
import com.pholser.junit.quickcheck.Property;
import com.pholser.junit.quickcheck.runner.JUnitQuick-
check;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

@RunWith(JUnitQuickcheck.class)
public class StringProperties {
  @Property
  public void concatenationLength(String s1, String
s2) {
  assertEquals(s1.length() + s2.length(), (s1 + s2).
length());
     }
}
```

Listing 5

JUnit-Quickcheck kann damit von Haus aus umgehen und weiß, wie diese erzeugt werden müssen. *Listing 6* zeigt, wie man für andere Typen einen Generator implementiert, der auch Shrinking unterstützt. Dies ist nötig, um für den zu testenden Code geeignete Domänen-Objekte erzeugen zu können. Es müssen lediglich zwei Methoden überschrieben werden, wie im Beispiel zu sehen.

Um neue Werte zu erzeugen, wird zunächst „generate()" überschrieben und ein neuer Punkt mit einer zufälligen X-Y-Koordinate erzeugt. Das API stellt dafür einen Zufallsgenerator als Parameter zur Verfügung, der es einem ermöglicht, den Test bei Bedarf immer wieder mit denselben zufällig erzeugten Objekten aufzurufen. Dies kann beim Debuggen von Tests hilfreich sein.

Als Zweites sollte die Methode „doShrink()" implementiert sein, damit klar ist, wie vereinfachte Objekte aus einem bestehenden erzeugt werden können. Je nach Domäne bedeutet dieses Vereinfachen etwas anderes. Im Beispiel sind vereinfachte Punkte nur näher am Ursprung des Koordinatensystems. Man sieht auch, dass nicht nur ein kleinerer Punkt erzeugt wird, sondern vier neue Punkte. Je nach Domänen-Objekt können beliebig viele vereinfachte Objekte zurückgegeben werden. Jetzt lassen sich die Tests einfach mit JUnit-Quickcheck erstellen.

Auf den ersten Blick erscheint es schwierig, geeignete Eigenschaften für eine Methode zu finden, um Property Based Testing anwenden zu können. Aber es gibt eine ganze Reihe von Mustern, die einem helfen, solche Eigenschaften zum Testen zu formulieren.

Unterschiedliche Wege, dasselbe Ziel

Dem Muster „Unterschiedliche Wege, dasselbe Ziel" sind Eigenschaften zuzuordnen, deren Reihenfolge bei der Ausführung keine Rolle spielt. Man kennt dieses Muster auch unter dem Namen „Kommutativität", zum Beispiel bei der Addition. Hier spielt es keine Rolle, ob zuerst „+ 1" und danach „+ 2" gerechnet wird oder zuerst „+ 2" und danach erst „+ 1". Das Ergebnis ist in beiden Fällen dasselbe.

Abbildung 1 zeigt ein konkretes Beispiel für dieses Muster. In diesem Fall wird geprüft, ob die Liste korrekt sortiert wurde, indem wir die beiden Wege zum Ergebnis vergleichen. Das vorherige Anhängen von „Integer.MIN_VALUE" an die Liste und das darauffolgende Sortieren muss dasselbe Resultat zurückliefern, als wenn die Liste als Erstes sortiert und danach erst „Integer.MIN_VALUE" vorne an die Liste gesetzt wird.

Hin und zurück

Dieses Muster eignet sich für alle Operationen, für die eine Umkehrung existiert. Der bekannteste Vertreter dieser Art von Operationen ist die Codierung/Decodierung von Daten. Das ist jedoch nicht das einzige Paar, das geeignet ist. Zum Beispiel können auch folgende Paare überprüft werden: Addition/Subtraktion, Schreiben/Lesen oder auch Getter/Setter. Außerdem gibt es noch Operations-Paare, bei denen es sich zwar nicht um die Umkehrung der Operation handelt, die aber dennoch die Anforderungen dieses Musters erfüllen. Beispiele sind: Einfügen/Enthält oder Erzeugen/Existiert.

In *Abbildung 2* ist noch ein Sonderfall für dieses Muster dargestellt: Manche Operationen sind ihre eigene Umkehrfunktion – in diesem Fall die Methode zum Umkehren einer Liste. Eine wiederholte Anwendung der Methode führt wieder zur ursprünglichen Eingabe.

```
import java.awt.Point;

public class Points extends Generator<Point> {
  private static final Point ORIGIN = new Point();

  public Points() {
     super(Point.class);
  }

  @Override
  public Point generate(SourceOfRandomness random,
GenerationStatus status) {
     return new Point(random.nextInt(), random.nex-
tInt());
  }

  @Override
  public List<Point> doShrink(SourceOfRandomness ran-
dom, Point larger) {
     if (ORIGIN.equals(larger)) {
        return Collections.emptyList();
     }

     List<Point> shrinks = new ArrayList<>();
     shrinks.add(new Point(0, larger.y));
     shrinks.add(new Point(larger.x, 0));
     shrinks.add(new Point(larger.x / 2, larger.y));
     shrinks.add(new Point(larger.x, larger.y / 2));
     return shrinks;
  }
}
```

Listing 6

Manche Dinge ändern sich nie

Im nächsten Muster geht es um invariante Eigenschaften von Transformationen. Typische Vertreter sind beispielsweise die Größe einer Liste, die sich nicht ändert, wenn man jedes einzelne Element transformiert, oder der Inhalt einer Liste nach dem Sortieren. *Abbildung 3* zeigt dieses Muster anhand einer Liste, in der jedes einzelne Element transformiert wird. Es ist erkennbar, dass sich die Anzahl der Elemente nicht verändert hat.

Je öfter man sie anwendet, desto weniger machen sie aus

Diese Klasse von Eigenschaften zeichnen sich durch Idempotenz aus, eine mehrfache Ausführung der Operation liefert also das gleiche Ergebnis zurück wie eine einzige. *Abbildung 4* zeigt die „distinct"-Operation auf einer Liste. Es ist zu sehen, dass sich nach der ersten Anwendung das Ergebnis auch bei Wiederholung nicht mehr ändert. Dieses Prinzip lässt sich auf weitere Operationen wie Datenbank-Updates oder Nachrichtenverarbeitung ausweiten.

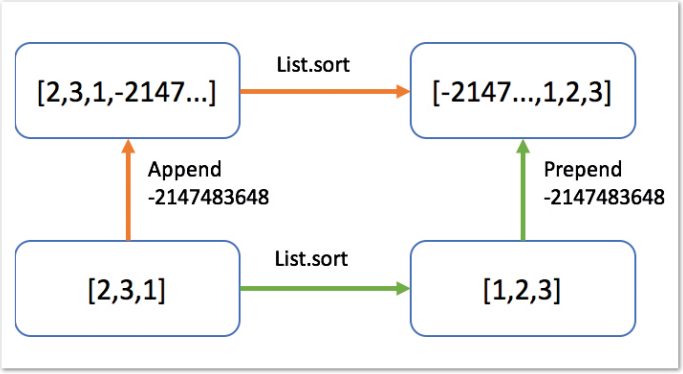


Abbildung 1: Ein Beispiel für „Unterschiedliche Wege, dasselbe Ziel"

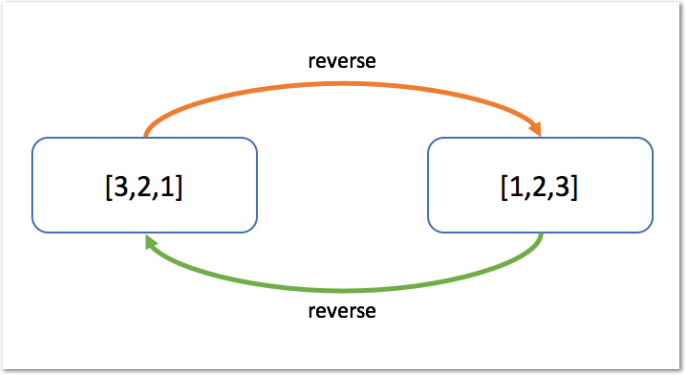


Abbildung 2: Ein Beispiel für „Hin und zurück"

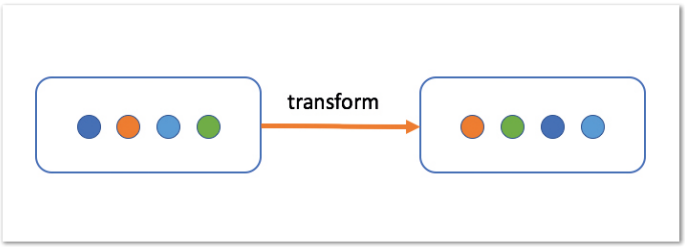


Abbildung 3: Ein Beispiel für „Manche Dinge ändern sich nie"

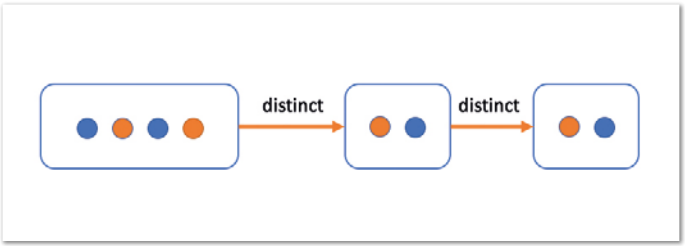


Abbildung 4: Ein Beispiel für „Je öfter man sie anwendet, desto weniger machen sie aus"

Schwer zu beweisen, einfach zu überprüfen

Dies ist eine Variante des Musters „Hin und zurück". Es kommt häufig vor, dass es viel schwieriger ist, die Lösung für ein Problem zu finden, als die Lösung zu prüfen. Das kennt jeder, der schon mal ein Sudoku gemacht hat. Zu prüfen, ob ein Sudoku korrekt ausgefüllt ist, geht sehr einfach, indem man die Zeilen, Spalten und 3x3-Blöcke nacheinander überprüft. Ein Sudoku korrekt auszufüllen, ist jedoch ungleich schwieriger. Weitere Probleme, die zu diesem Muster passen, sind zum Beispiel die Primfaktor-Zerlegung oder das Zerlegen eines Strings in einzelne Token. *Abbildung 5* zeigt ein weiteres Beispiel: Sortieren. Eine Sortier-Funktion ist komplex zu implementieren – zu prüfen, ob korrekt sortiert wurde, jedoch relativ einfach. Es muss für jedes Paar geprüft werden, ob die Paare korrekt geordnet sind.

Das Test-Orakel

Es bietet sich an, wenn eine alternative Implementierung bereits vorhanden ist und gegen diese validiert werden soll. Das kann zum Beispiel der Fall sein, wenn eine Single-Thread-Lösung durch eine Multi-Threaded-Lösung, die vermutlich schwieriger korrekt zu implementieren ist, ersetzt werden soll. Ebenfalls interessant ist dieses Muster bei umfangreichen Modernisierungen oder Refactorings. Die ursprüngliche Implementierung kann dabei als Orakel für eine Neu-Implementierung dienen und damit alle, auch nicht dokumentierte Eigenschaften der Anforderungen abdecken.

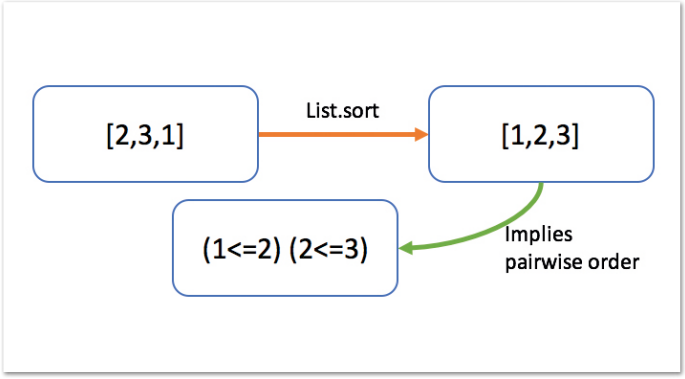


Abbildung 5: Ein Beispiel für „Schwer zu beweisen, einfach zu überprüfen"

Fazit

Property Based Testing ist eine nützliche Erweiterung des Werkzeugkastens, den ein Software-Entwickler zum Testen zur Verfügung hat. Es erfordert, dass die Anforderungen auf der Suche nach möglichen Test-Eigenschaften genauer und gründlicher analysiert und schlussendlich auch besser verstanden werden. Property Based Testing ist nicht dazu gedacht, das klassische Unit-Testing zu ersetzen, sondern es an geeigneter Stelle zu ergänzen. Es spricht nichts dagegen, beide Methoden zu kombinieren, sogar innerhalb eines Tests. Zu dem Trend, mehr und mehr Ansätze aus der funktionalen Programmierung in das Java-Ökosystem zu integrieren, passt Property Based Testing, das seinen Ursprung dort hat, in jedem Fall sehr gut.

Quellen

[1] M. Pol, T. Koomen und A. Spillner, Management und Optimierung des Testprozesses, dpunkt.verlag, 2002
[2] C. Mike, The Forgotten Layer of the Test Automation Pyramid: *https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid*
[3] K. Beck, Test Driven Development by Example, Addison-Wesley Verlag
[4] K. Claessen und J. Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs, ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, 2000
[5] P. Holser, JUnit-quickcheck: *http://pholser.github.io/junit-quickcheck/site/0.7*

Nicolai Mainiero
nicolai.mainiero@sidion.de

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Expert Software Developer bei der Firma sidion. Er entwickelt seit mehr als neun Jahren Geschäftsanwendungen in Java und PHP für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Aktorensysteme und reaktive Anwendungen.

Putting TDD to the test

Edwin Günthner, IBM Germany Development Lab

Der englische Ausdruck „Putting something to the test" bedeutet so viel wie „etwas auf den Prüfstand stellen." Genau darum geht es in diesem Artikel: den Nutzen der Methodik „Test Driven Development" (TDD) anhand von praktischen Erfahrungen aus dem realen Umfeld in einem Großunternehmen zu bestimmen und auszuloten. Dabei dient die persönliche Lernkurve des Autors als Ausgangsbasis für weiterführende Diskussionen rund um das Thema „TDD und Unit Testing".

Sie lesen richtig: Wir schreiben das Jahr 2017, und ich schreibe einen Artikel über TDD. Zu einem Zeitpunkt, an dem andere erklären, weshalb diese Methode in der Praxis versagt hat *[1]*. Heute zählen doch Agilität und die Fähigkeit, sich schnell wechselnden Anforderungen zu stellen. Da bleibt keine Zeit für rigide Strukturen beim Entwickeln – und überhaupt – sind Menschen nicht wichtiger als Prozesse? Sollte es nicht darum gehen, den Entwicklern zu vertrauen? Natürlich. Was dabei jedoch gerne unter den Tisch fällt: Damit das alles effektiv funktionieren kann, bedarf es auch des Vertrauens des Entwicklers in den geschriebenen Code.

Ohne dieses Vertrauen kommen wir in eine Situation, die Michael Feathers in „Working effectively with Legacy Code" wie folgt beschreibt: „What do you think about when you hear the term legacy code? If you are at all like me, you think of tangled, unintelligible structure, code that you have to change but don't really understand." Wir verstehen: Mit Legacy Code zu arbeiten, ist schwierig und unangenehm. Aber was zeichnet solchen Code aus? Feathers findet eine bestechend einfache und zugleich tiefgründige Antwort: „To me, legacy code is simply code without tests." Wer also vermeiden will, sich morgen über Code zu beschweren, den er gestern selbst geschrieben hat, für den lohnt sich auch heute die Beschäftigung mit dem Thema „Unit-Tests" und mit einer der zentralen Methoden zum Schreiben eben dieser Tests – TDD!

Dieser Artikel ist keine vollständige Einführung ins Thema „wie schreibe ich Unit-Tests mit TDD" – dazu gibt es schon genügend gute Bücher und Tutorials. Es ist allerdings dennoch wichtig sicherzustellen, dass jeder das gleiche Verständnis der verwendeten Begriffe hat.

Betrachten wir zunächst den Begriff „Unit Testing". Bei *[2]* findet sich: „In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use." Mit anderen Worten: Alles was man tut, um ein Programm irgendwie zu testen, ist ein Unit-Test. Eine so weit gefasste Definition erlaubt aber keinerlei Abgrenzung zu anderen Arten von Tests. Daher ist sie wenig hilfreich.

Dieser Artikel basiert stattdessen auf dem Verständnis guter Unit-Tests, wie sie die Website „ArtOfUnitTesting" *[3]* oder der Wikipedia-

Artikel zum Thema „Modultest" [4] beschreiben. Was man jedoch nicht vergessen darf: Die erstgenannte, unscharfe Definition ist durchaus gebräuchlich. Ich erlebe häufig, dass Kollegen über Unit-Tests sprechen, aber eigentlich Funktions- beziehungsweise Integrationstests meinen. In aller Regel ist es (leider) effektiver, sich selbst immer wieder diese unterschiedlichen Sichtweisen bewusst zu machen als zu versuchen, die eingefahrene Sichtweise anderer zu verändern.

Der Begriff „TDD" [5] ist eindeutig definiert: Im Kern geht es darum, dass eine Anforderung zuerst als ausführbarer Test spezifiziert wird. Erst danach erfolgt die eigentliche Implementierung. Diese ist erst abgeschlossen, wenn der neu hinzugefügte sowie alle anderen, bereits existierenden Tests erfolgreich durchlaufen worden sind. Bevor die nächste Anforderung bearbeitet wird, widmet man sich explizit der Qualität des neu geschriebenen Codes – man verbessert dann ausschließlich die Art und Weise, wie die Anforderung umgesetzt wird. Die existierenden Tests stellen sicher, dass die gewünschte Funktionalität durch die zusätzlichen Änderungen am Code nicht verändert wird. Das Ganze erfolgt in zyklischen Iterationen – wobei jeder einzelne Zyklus idealerweise nur wenige Minuten dauern sollte.

Nach Klärung dieser Begrifflichkeiten können wir uns jetzt den praktischen Aspekten zuwenden – wie schon angedeutet aus einer (zunächst) subjektiven Perspektive: Der erste Kontakt zu TDD hat sich für mich vor etwa fünf Jahren ergeben, als ich innerhalb der IBM zum Bereich „System Z Firmware" gewechselt bin. Sowohl im Kontakt zu Kollegen als auch bei Vorträgen beim Java Forum Stuttgart entstand der Eindruck, in TDD eine vielversprechende Methodik zu finden, deren Anwendung in kurzer Zeit die Qualität des geschriebenen Codes verbessern kann. Auch die Kollegen im neuen Team waren schnell überzeugt – das „wir machen das jetzt einfach" kam tatsächlich von Herzen. Nach wenigen Monaten stellte sich Ernüchterung ein: „Das hilft ja gar nicht". Konkret zeigte sich:

- Die meisten Tests entstanden nachträglich – sie wurden erst geschrieben, nachdem der Produkt-Code schon relativ weit fortgeschritten war.
- Die Tests waren umständlich, schwer zu verstehen – und vor allem: Sie haben mehr Probleme gemacht als gefunden.

Die direkte Konsequenz war eine spürbare Verringerung der Motivation einiger Team-Mitglieder, sich weiter mit der Thematik zu beschäftigen. Glücklicherweise blieb aber die Bereitschaft, die TDD-Aktivitäten Einzelner nicht einzuschränken.

Das bedeutet konkret, dass ich zunächst regelmäßig kleinere Aufgaben mit TDD bearbeitet habe. Im Jahr 2015 hat sich dann für mich die Möglichkeit ergeben, eine größere Problemstellung vollständig mit TDD umzusetzen. Ich arbeite innerhalb des Bereichs IBM Z und schreibe dort Java-Code für die Hardware Management Console (HMC) [6].

Die HMC ist das Management Interface für IBM Z Mainframes. Das Projekt „Dynamic Partition Manager" (DPM) [7] hat sich die Aufgabe gestellt, Teile dieser Management-Funktionalität in einer zeitgemäßen Art und Weise neu zu implementieren. Ein zentrales Element ist hierbei das Starten/Stoppen von Gast-Systemen auf dem Mainframe.

Mein Team hatte bereits einen Prototyp der neuen Start/Stopp-Funktion implementiert. Es zeigte sich allerdings, dass der neue Code erhebliche Defizite aufwies. Zum Beispiel fehlte eine durchgehende Fehlerbehandlung unter Berücksichtigung der notwendigen (kontextabhängigen) Rollback-Schritte. Da es auch keine Unit-Tests gab, waren selbst einfache Refactoring-Änderungen mit erheblichem Test-Aufwand verknüpft. Mit anderen Worten: Der neu geschriebene Code war innerhalb weniger Monate zu Legacy Code geworden. Da die notwendige Funktionalität überschaubar war, entschloss ich mich dazu, die ruhige Zeit am Jahresende zu nutzen, um eine komplett neue

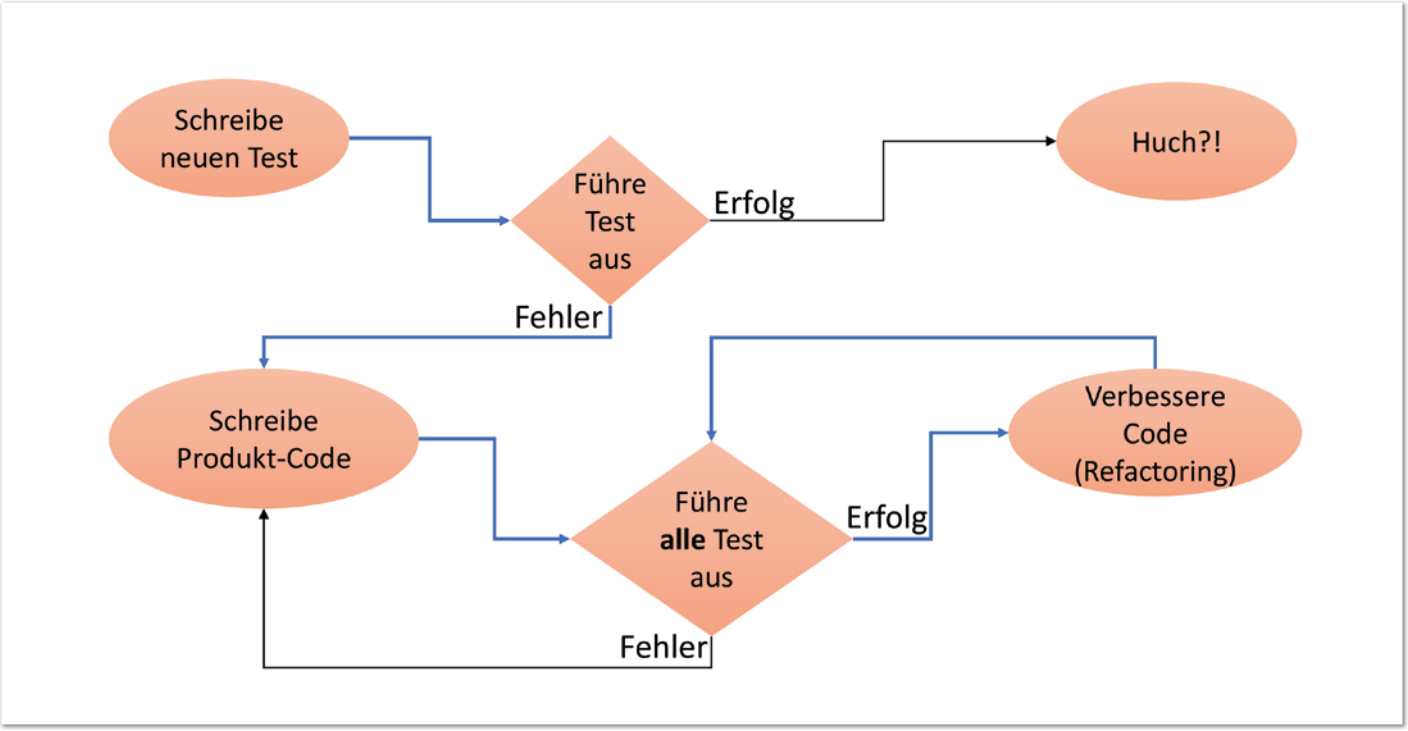


Abbildung 1: TDD im Überblick

Implementierung zu schreiben – und da ich es besser machen wollte, mit der Vorgabe, konsequent TDD einzusetzen. Die Erfahrungen, die ich dabei gesammelt habe, möchte ich im Folgenden darstellen.

Die erste und wichtigste Erkenntnis hat überraschenderweise nichts mit Technik oder Code-Qualität zu tun. Sie lautet nämlich: Konsequentes TDD macht Spaß. Rufen wir uns zunächst mit *Abbildung 1* in Erinnerung, wie TDD abläuft (die blauen Pfeile stehen für den normalen Geradeaus-Pfad beim Einsatz der Methodik).

Der erste Schritt besteht darin, eine neue Anforderung als ausführbaren Test zu formulieren. Danach ist es wichtig sicherzustellen, dass der neue Test beim Ausführen zu einem Fehler führt – falls der Test funktioniert, stimmt irgendetwas nicht – entweder ist der Test inhaltlich falsch oder aber unsere Erwartungshaltung. Anschließend wird Produkt-Code erzeugt (idealerweise so wenig wie möglich – es geht nur darum, diesen einen, neuen Test erfolgreich zu bestehen). Dann werden alle Tests ausgeführt (vorzugsweise: genau die Tests, die für den betroffenen Produkt-Code existieren).

Schlagen Tests fehl, gilt es die Ursache zu bestimmen und zu beseitigen. Wenn alle Tests erfolgreich sind, widmet man sich entweder der nächsten Anforderung – oder verwendet Zeit darauf, die Qualität des bisher geschriebenen Codes durch Refactoring zu verbessern. In Anlehnung an die grafische Darstellung von Tests in IDEs wie Eclipse spricht man auch vom „red/green/refactor"-Kreislauf. Wie bereits erwähnt, erfolgen diese Iterationen in sehr schneller Abfolge – ein einzelner Zyklus sollte nur wenige Minuten dauern (und den überwiegenden Teil dieser Zeit verbringt man idealerweise in der Refactoring-Phase). Diese Arbeitsweise führt beinahe automatisch zu einem positiven Arbeitsgefühl:

- Es ergibt sich eine kontinuierliche Serie von Herausforderungen (neuer, roter Test), auf die jeweils umgehend eine Belohnung (neuer Produkt-Code und grüne Tests) folgt.
- Gleichzeitig laufen diese Vorgänge stets im gleichen Kontext ab: in der Entwicklungsumgebung des Entwicklers.

Wenn neben der Anwendung dieser Vorgehensweise noch die Möglichkeit besteht, Störungen von außen zu minimieren oder (zumindest zeitweise) gänzlich zu eliminieren – dann kann sich sehr schnell ein echtes Flow-Erlebnis einstellen: Man kann sich mit voller Konzentration genau einer Aufgabe widmen. Kreative Ideen werden nicht nur sofort umgesetzt. Man erhält auch umgehend Feedback darüber, ob der neu geschriebene Code sich wie erwartet verhält.

Übrigens gilt auch die Umkehrung: Je länger man auf solches Feedback warten muss, desto unbefriedigender fühlt sich die eigene Arbeit an. In meinem Fall sind für echte Funktionstests die folgenden Schritte notwendig, um einen Patch zu testen:

1. Identifizieren genau derjenigen Java-Klassen, die von der Änderung betroffen sind
2. Erstellen eines Archivs mit allen zu Punkt 1 korrespondierenden, binären Artefakten
3. Einspielen des Archivs in einem (eventuell vorher zu erzeugenden) Test-System

Bestenfalls benötige ich für diese Schritte fünf bis zehn Minuten – bei veränderten Randbedingungen gerne auch dreißig Minuten oder mehr. Die Frage nach flüssigen Arbeitsabläufen stellt sich an dieser Stelle nicht mehr.

Diese drei Punkte habe ich nach ungefähr vier Wochen mit „purem TDD" abgearbeitet, um die Frage „Funktioniert der neue Code denn nun wirklich?" beantworten zu können. Überraschenderweise lautete die Antwort „nein". Bei der anschließenden Analyse zeigte sich schnell, dass die Ursache für den fehlgeschlagenen Versuch ein Fehler in Schritt 2 war – der erzeugte Patch war schlicht und ergreifend unvollständig. Neuer Patch, neuer Test – alles funktioniert wie erwartet.

Weiteres Nachdenken über den Fehlschlag führte zu der Erkenntnis: „Der neue Code hätte diesen Fehler anders verarbeiten müssen". Ich kam also nach vier Wochen intensiver Arbeit zu dem Schluss, dass im neuen Produkt-Code ein Design-Fehler steckt. Das Problem ließ sich aber wieder als Unit-Test darstellen und somit mit TDD bearbeiten.

In der Folge war es notwendig, etliche Klassen im Produkt-Code zu ändern. Nach wenigen Stunden waren alle zugehörigen Tests angepasst und konnten wieder erfolgreich ausgeführt werden. Und ich konnte mir sicher sein, dass sowohl das neue Problem behoben war, als auch, dass durch die vorgenommenen Änderungen keine anderen Fehler eingebaut wurden. Eine komplexe Änderung im Produkt-Code konnte also dank vollständiger Abdeckung der Code-Basis durch Unit-Tests in kürzester Zeit durchgeführt werden.

Nach etlichen weiteren Wochen Programmieren mit TDD waren alle Anforderungen implementiert. Das Ergebnis der anschließenden Test-Phase lautete: Keine Bugs gefunden. Im Detail:

- Keine Übertragungsfehler: Die neu geschriebene Komponente verhielt sich wie die alte Komponente.
- Die neu hinzugefügte Fehlerbehandlung (zur Bearbeitung komplexer Rollback-Szenarien) lieferte die erwarteten Ergebnisse. Gleiches gilt für neu eingebaute „Debug Support"-Funktionen.

Auch in den etwa achtzehn Monaten, in denen die neue Komponente genutzt wurde, wurden keine Fehler gefunden.

Bevor ich mich wieder der Theorie und der Frage nach den Voraussetzungen für den erfolgreichen Einsatz von TDD zuwende, ist es mir wichtig, einige Punkte zu benennen, die in meinem Experiment nicht funktioniert haben:

- Meine Abschätzung des Aufwands lag bei vier bis maximal acht Wochen. Tatsächlich habe ich eher sechzehn bis zwanzig Wochen benötigt.
- Es ist mir nicht gelungen, meine eigenen Erwartungen an Code-Qualität konsequent umzusetzen; zum Beispiel fanden Peer Reviews gar nicht beziehungsweise zu spät statt.

Aber: Die Methode TDD erhebt nicht den Anspruch, alle Probleme zu lösen. Sie gibt einen Rahmen vor, in dem man sehr kreativ und effizient agieren kann – mehr allerdings auch nicht.

Was unterscheidet diese Erfolgsgeschichte von den ersten Erfahrungen in meinem Team, die eher von Frustration gekennzeichnet waren? Die naheliegende Vermutung wäre: „Ich habe gelernt, bessere Tests zu schreiben." Das ist richtig, aber nur die halbe Wahrheit

– der wesentliche Punkt ist nämlich: „Ich habe gelernt, Produkt-Code zu schreiben, der testbar ist."

Es ist letztendlich sehr einfach: Wer eine Klasse schreibt, die zwanzig Felder hat und in der jede Methode fünf Parameter erwartet, um dann über mehr als hundert oder mehr Zeilen hinweg mehrere Dinge gleichzeitig tun – der muss sich nicht wundern, wenn diese Klasse gar nicht oder zumindest nur schwer und unvollständig testbar ist, und zwar unabhängig davon, ob Unit-Tests oder funktionale Tests am echten System zum Einsatz kommen. Im Umkehrschluss bedeutet das: Wenn ich eine Klasse mit Unit-Tests in den Griff bekommen will, sollten die folgenden Bedingungen für die Klasse beziehungsweise ihre Methoden erfüllt sein:

- Die Klasse kann in Isolation betrachtet werden
- Es gibt die Möglichkeit, die Klasse von ihren Abhängigkeiten zu entkoppeln
- Der Code implementiert genau eine Funktion

Oder, um es auf einen Nenner zu bringen: Der Produkt-Code muss mit dem Wissen über „Clean Code" geschrieben werden. Daraus folgt: Wer TDD und Unit Testing sinnvoll einsetzen möchte, muss sich nicht nur mit dem Themenkomplex „TDD" beschäftigen – also beispielsweise Bücher wie „JUnit-Profiwissen" (Michael Tamm) oder „xUnit Test Patterns" (Gerard Meszaros) intensiv bearbeiten. Nein: Darüber hinaus ist es unerlässlich, auch die Klassiker zur Code-Qualität – etwa „Clean Code" beziehungsweise „Agile Principles" (Robert Martin) oder „Refactoring" (Kent Beck) – zu verinnerlichen und immer wieder praktisch einzuüben.

Innerhalb unseres Teams hat es sich an dieser Stelle als besonders wertvoll erwiesen, regelmäßige Code Reviews durchzuführen, die ausschließlich die Clean-Code-Prinzipien *[8]* behandeln. Die kontinuierliche Beschäftigung mit beiden Themenkomplexen ist zwingend notwendig, um die initiale Frustration zu überwinden und zu echten Verbesserungen gelangen zu können. Übrigens ist das keine neue Erkenntnis – zum Beispiel hat Michael Feathers bereits im Jahr 2013 dazu den empfehlenswerten Vortrag „The deep synergy between testability and good design" *[9]* gehalten.

Eine kurzfristige Anordnung des Managements an die Entwickler: „Ihr macht jetzt mal zwei Monate TDD", am besten verbunden mit „und danach erwarten wir, dass die Fehler-Quote um x Prozent sinkt", kann also gar nicht funktionieren. Das Management muss sich vielmehr auf jahrelange Lernprozesse einstellen. Darüber hinaus wollen die Entwickler jetzt auch noch erheblichen Aufwand in das Schreiben von Tests investieren. Fragen wie: „Rentiert sich das überhaupt?" und „Werden wir nicht viel langsamer, wenn die Entwickler die Hälfte der Zeit mit dem Schreiben von Tests verbringen?", liegen dann natürlich nahe.

Wie ich jedoch bereits ausgeführt habe – das Gegenteil kann der Fall sein: TDD und Unit-Tests können dazu führen, dass die Entwicklung schneller vorangeht. Man muss sich klarmachen: Das Ziel bei der Entwicklung von Software ist nicht das Schreiben von Code. Das Ziel besteht vielmehr darin, am Ende ein Produkt liefern zu können, dass sowohl im Funktionsumfang als auch in der Qualität den Erwartungen der (zahlenden) Kunden entspricht.

Wenn ich als Entwickler zunächst m Zeilen Test-Code schreibe, um dann n Zeilen Produkt-Code folgen zu lassen, führt das genau dann zu einem positiven Return on Investment, wenn die m Zeilen Test-Code mir helfen, die erwartete Funktion insgesamt schneller zu liefern. Es geht nicht darum, die Anzahl der geschriebenen Zeilen Code zu minimieren, sondern darum, die Zeit bis zur Auslieferung der nächsten Funktion (mit hoher Qualität) möglichst kurz zu halten. TDD/Unit-Tests verkürzen diese Zeit deswegen, weil sie die Zeitspanne (massiv) verkürzen, die ich als Entwickler auf Feedback warte.

Aus dieser Beobachtung lässt sich ableiten, in welchen Fällen TDD/Unit Testing nicht oder nur bedingt hilfreich ist, nämlich genau dann, wenn die Entwickler über andere Wege schnelles Feedback erhalten können. Fred George beschreibt in *[10]* ein System, in dem die Entwickler Änderungen sofort ins Produktions-System einstellen können. Ist die neue Funktion gut, steigen die Gewinne wenige Sekunden nach dem Aktivieren der Änderung. Fallen die Gewinne jedoch, wird umgehend die alte Funktionalität wiederhergestellt. In diesem System benötigt man kein TDD, weil die meisten Änderungen gut sind – und weil jede Sekunde, die eine Änderung früher verfügbar ist, sofort als Gewinn in die Bilanz eingeht. Aber Entwickler, die in einem solchen System arbeiten, stellen vermutlich die große Ausnahme dar. Alle anderen, die schnelles Feedback erhalten wollen, können einstweilen auf TDD/Unit Testing zurückgreifen.

Fazit

TDD und Unit-Tests sind Methoden, mit deren Hilfe Entwickler schnelles Feedback über Code erhalten können. Um TDD sinnvoll zu nutzen, ist es weniger wichtig, sich stur an die Regeln der Methodik zu halten. Vielmehr geht es darum, das komplexe Zusammenspiel von Codequalität (gemäß Clean Code) und hilfreichen Unit-Tests regelmäßig einzuüben und durch geeignete Maßnahmen zu unterstützen. Dann kann TDD seine volle Wirkung entfalten – und gut gelaunte Entwickler dabei leiten, hochwertigen Produkt-Code zu schreiben.

Links

[1] *https://blogs.msdn.microsoft.com/ericgu/2017/06/22/notdd*
[2] *https://en.wikipedia.org/wiki/Unit_testing*
[3] *http://artofunittesting.com/definition-of-a-unit-test*
[4] *https://de.wikipedia.org/wiki/Modultest*
[5] *https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung*
[6] *http://ibm.biz/redbook-hmc*
[7] *http://ibm.biz/dpm-intro*
[8] *http://clean-code-developer.de*
[9] *https://www.youtube.com/watch?v=4cVZvoFGJTU*
[10] *http://www.se-radio.net/2016/03/se-radio-episode-253-fred-george-on-developer-anarchy*

Edwin Günthner
edwin.guenthner@de.ibm.com

Der Autor hat an der Universität Karlsruhe studiert und dort im Jahr 1999 den Abschluss als Diplom-Informatiker gemacht. Seit dem Jahr 2001 arbeitet er bei der IBM Deutschland Research & Development GmbH in Böblingen in wechselnden Rollen in den Bereichen zSystems-Hardware und Firmware-Entwicklung.

Continuous Database Integration mit Flyway

Sandra Parsick, Freiberufler

Skripte für relationale Datenbanken werden von Entwicklern gerne stiefmütterlich behandelt. Beim ersten Release können sie dank ORM-Frameworks generiert werden. Doch spätestens beim zweiten Release müssen Datenbank-Migrationsskripte geschrieben werden. Sie werden dann gerne an Tickets angehängt, per E-Mail verteilt oder in Release Notes versteckt. Irgendwann gibt es keinen Überblick mehr darüber, welche Datenbank-Skripte zu welcher Softwareversion gehören. Der Artikel zeigt, warum deren Einbindung in den Continuous-Integration-Prozess erstrebenswert ist, welche Voraussetzungen dafür geschaffen werden müssen und wie Flyway dabei helfen kann.

Wer kennt es nicht: Entwickler müssen sich eine Test-Datenbank teilen und kommen sich ständig in die Quere, da man Phantomfehlern hinterherjagt. Doch der eigentliche Grund war, dass sie sich gegenseitig die Datenbank-Struktur zerschossen haben. Fehler in der Produktion lassen sich nicht nachbilden, da sich die Test-Datenbank gravierend von der Datenbank in der Produktion unterscheidet. Dadurch haben auch Testläufe der Migrationsskripte keine Aussagekraft.

Werden verschiedene Test-Datenbanken gepflegt, dann weiß keiner so genau, welche Skripte gegen welche Datenbank liefen. Zu guter Letzt werden die Datenbank-Skripte in unterschiedlichen Systemen abgelegt und es ist nicht klar, in welcher Reihenfolge die Skripte ausgeführt werden müssen oder ob alle relevanten Skripte für den nächsten Livegang vorhanden sind. Diese Problematik möchte „Continuous Database Integration" angehen, indem ein Prozess geschaffen wird, bei dem zu jeder Zeit, wenn eine Änderung an den Datenbank-Skripten gemacht wurde, eine Datenbank mit ihren Testdaten erneuert werden kann.

Grundregeln für eine Continuous Database Integration

Um einen „Continuous Database Integration"-Prozess erfolgreich zu etablieren, sollten folgende Grundregeln eingehalten werden:

- Behandle den Datenbank-Code wie ganz normalen Source-Code
- Jeder Entwickler hat seine eigene Datenbank
- Die Test-Datenbanken ähneln den Produktions-Datenbanken
- Jede Änderung an der Datenbank ist nachvollziehbar

Um diese vier Grundregeln einhalten zu können, müssen folgende Maßnahmen ergriffen werden:

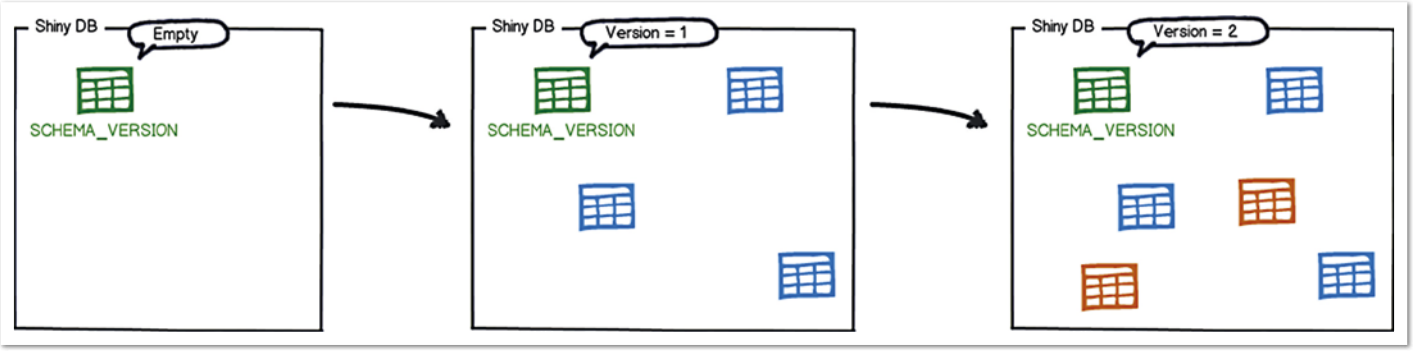


Abbildung 1: Arbeitsweise von Flyway bei einer leeren Datenbank (Referenz: „http://flyway.org")

schema_version

installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	Initial Setup	SQL	V1__Initial_Setup.sql	1996767037	axel	2016-02-04 22:23:00.0	546	true
2	2	First Changes	SQL	V2__First_Changes.sql	1279644856	axel	2016-02-06 09:18:00.0	127	true
3	2.1	Refactoring	JDBC	V2_1__Refactoring		axel	2016-02-10 17:45:05.4	251	true

Abbildung 2: Die Historien-Tabelle SCHEMA_VERSION (Referenz: „http://flyway.org")

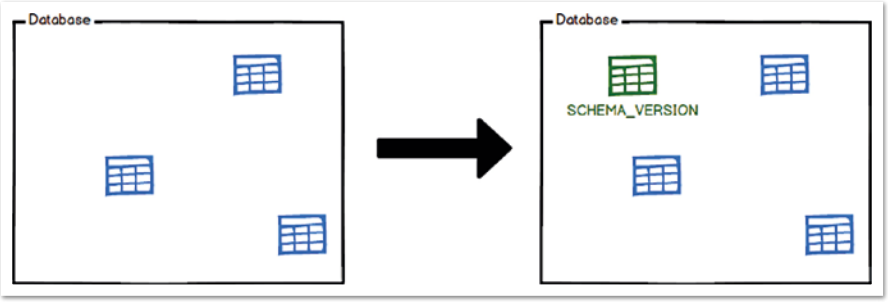


Abbildung 3: Flyway bei einer bestehenden Datenbank (Referenz: „http://flyway.org")

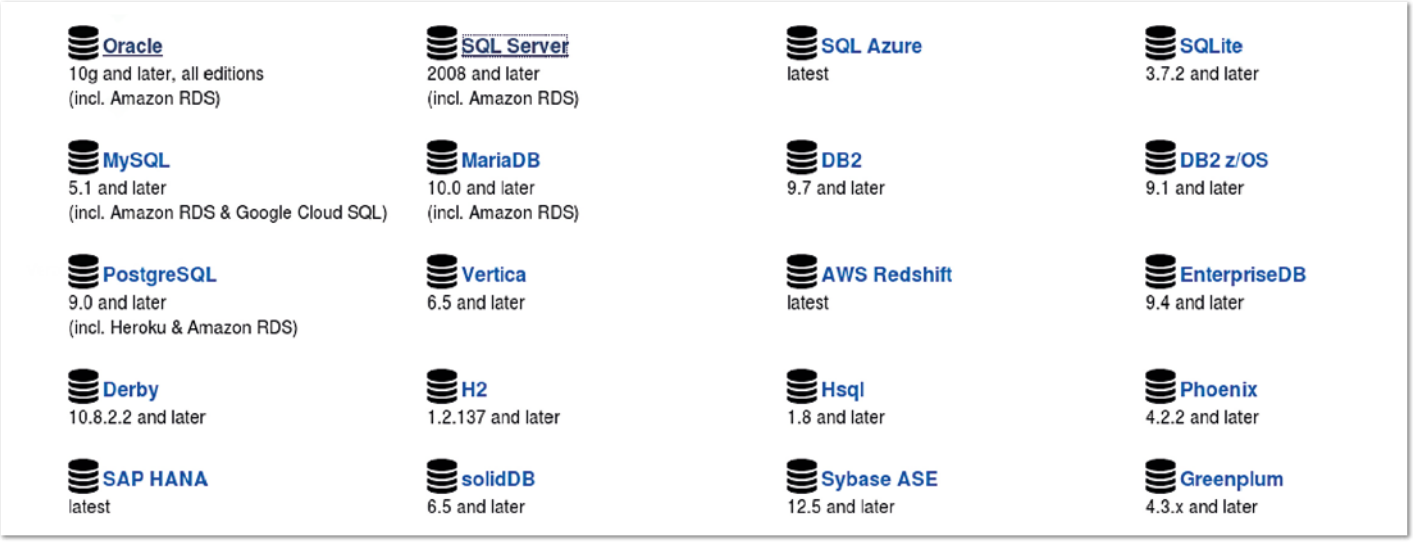


Abbildung 5: Unterstützte Datenbanken (Referenz: „http://flywaydb.org")

- Alle Datenbank-Artefakte wie DDL-Skripte, DML-Skripte, Konfigurationen, Testdaten, Skripte mit Stored Procedure oder Functions etc. gehören in ein Version-Control-System wie Subversion oder Git (Grundregel 1)
- Jede Änderung an den Datenbank-Artefakten wird getestet (Grundregel 1)
- Die Datenbank muss automatisiert aufsetzbar sein (Grundregel 2 und 3)
- Jede Änderung an der Datenbank wird in einer Historie protokolliert (Grundregel 4)

Diese Maßnahmen lassen sich gut mit Werkzeugen wie Liquibase, MyBatis oder Flyway umsetzen. Dieser Artikel zeigt, wie die Umsetzung mit Flyway aussehen könnte.

Flyway

Flyway *[1]* ist ein Migrations-Framework für relationale Datenbanken, geschrieben in Java. Das Ziel von Flyway ist, eine Datenbank „from scratch" erstellen zu können und den Stand der Datenbank zu verwalten. Dabei werden vier Migrationsmodi unterstützt: SQL- und Java-basierte sowie versionierte und wiederholbare Migrationen.

Die Grundfunktion von Flyway besteht darin, Migrationsskripte auszuführen („flyway migrate") und nach jedem erfolgreich ausgeführten Migrationsskript die Änderung in einer Tabelle in der betroffenen Datenbank zu protokollieren. Bei einer leeren Datenbank wird diese Historien-Tabelle („SCHEMA_VERSION") automatisch von Flyway angelegt; danach führt es die Migrationsskripte aus *(siehe Abbildung 1)*. In die Historien-Tabelle werden Informationen gespeichert *(siehe Abbildung 2)*:

- In welcher Reihenfolge die Skripte ausgeführt wurden („installed_rank")
- Welche Version die Skripte hatten („version")
- Eine Beschreibung dessen, was die Skripte machen („description")
- Um welche Art von Migration es sich handelt („type")
- Welches Skript ausgeführt wurde („script")
- Die Checksumme des Skripts, um nachträgliche Änderungen an den schon ausgeführten Skripten zu erkennen (nur bei SQL-basierten Skripten) („checksum")
- Von welchem Datenbank-Benutzer das Skript ausgeführt wurde („installed_by")
- Wann das Skript ausgeführt wurde („installed_on")
- Wie lange die Ausführung des Skripts dauerte („execution_time")
- Ob die Ausführung erfolgreich war („sucess")

	Versioniert	Wiederholbar
SQL-basiert	✔	✔
Java-basiert	✔	✔

Abbildung 4: Vier Möglichkeiten, um Migrationsskripte zu schreiben

Soll Flyway gegen eine schon bestehende Datenbank laufen, muss diese mit Flyway („flyway baseline“) vorbereitet sein. Dabei wird die Historien-Tabelle „SCHEMA_VERSION“ angelegt *(siehe Abbildung 3)*, danach kann die oben beschriebene Migration ausgeführt werden.

Vier Arten von Migrationsskripten

Wie am Anfang erwähnt, bietet Flyway vier Arten von Migrationsskripten an. Dabei sind immer zwei Arten miteinander kombiniert. *(siehe Abbildung 4)*.

Die versionierten Migrationsskripte haben immer eine eindeutige Version und werden im gesamten Lebenszyklus nur einmal ausgeführt. Typische Anwendungsfälle für diese Art von Skripten sind DDL-Änderungen (CREATE/ALTER/DROP für TABLES, INDEXES, FOREIGN KEYS etc.) oder einfache Datenänderungen.

Die wiederholbaren Migrationsskripte haben keine Version und werden immer dann ausgeführt, wenn sich ihre Checksumme ändert. Ihr Ausführungszeitpunkt ist, nachdem alle versionierten Skripte ausgeführt wurden. Typische Anwendungsfälle sind zum Beispiel die (Wieder-) Erstellung von Views, Procedures, Functions, Packages etc. oder ein Massen-Reimport von Stammdaten. Die SQL-basierten Migrationsskripte beinhalten SQL-Statements in einer Datenbank-spezifischen Syntax. Zusätzlich werden Platzhalter und Kommentare unterstützt *(siehe Listing 1)*.

Typische Anwendungsfälle für die SQL-Skripte sind DDL-Änderungen und einfache Datenänderungen. Die SQL-Syntax der Statements ist abhängig vom Datenbank-Anbieter, der eingesetzt wird. Momentan sind zwanzig Datenbank-Anbieter unterstützt *(siehe Abbildung 5)*. Der Dateiname der Skripte muss einem bestimmten Schema folgen, abhängig davon, ob die Skripte für eine versionierte oder eine wiederholbare Migration gedacht sind *(siehe Abbildung 6)*. Aus diesem Namensschema werden die Informationen für die Historien-Tabelle extrahiert („version“, „description“, „type“ und „script“).

```
/* Create a table for person */
Create table person (
  first_name varchar(128),
  last_name varchar(128)
);
GRANT SELECT, INSERT ON usermgnt.* TO 'technical-user'
@ '${address}' By '${password}';
```

Listing 1

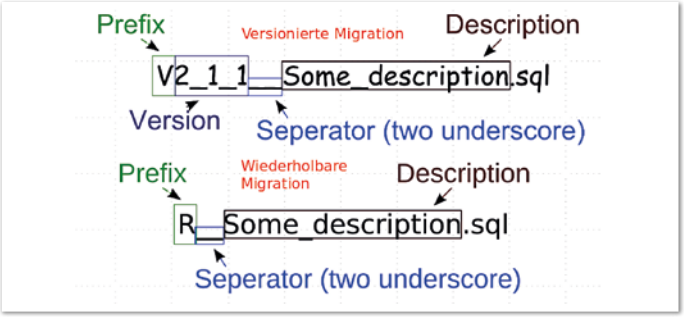


Abbildung 6: Bezeichnung der SQL-Skripte

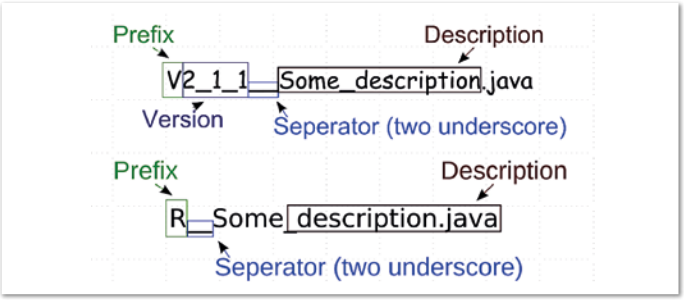


Abbildung 7: Bezeichnung der Java-Skripte

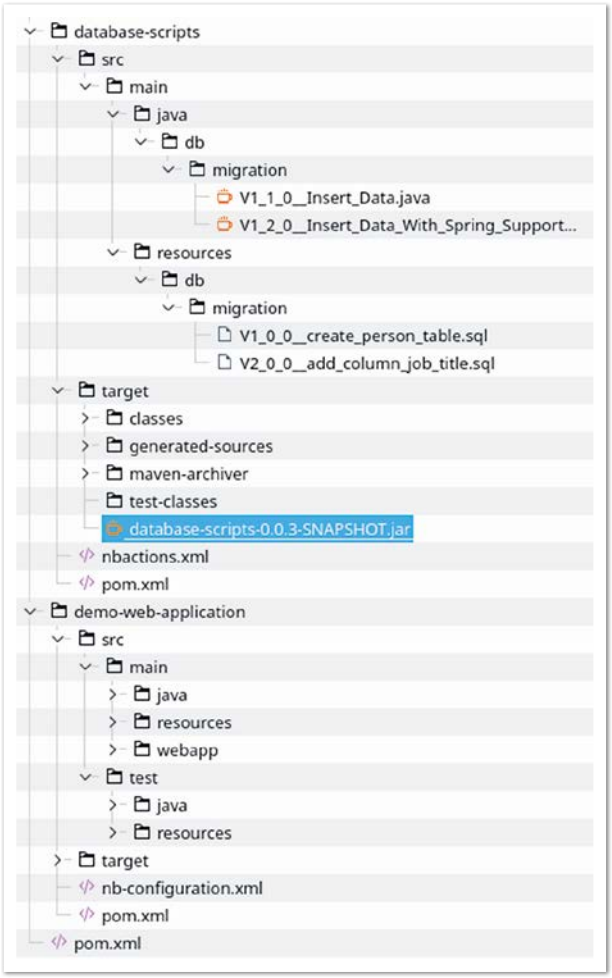


Abbildung 8: Ablageort der Migrationsskripte am Beispiel eines Maven-Projekts

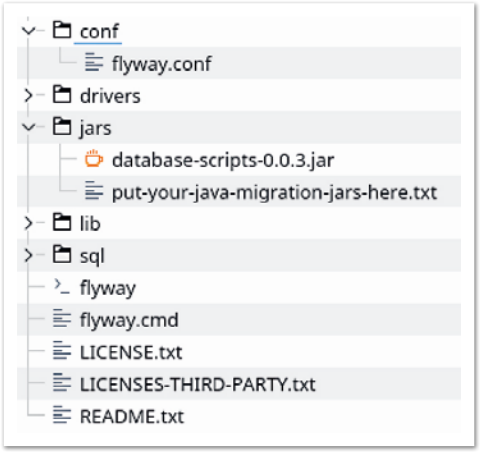


Abbildung 9: Ablageort der Migrationsskripte im Flyway-CLI für die Ausführung

Die Java-basierten Migrationsskripte sind Klassen, die entweder das Flyway-Interface „JdbcMigration" *(siehe Listing 2)* oder „SpringJdbcMigration" *(siehe Listing 3)* implementieren. Die Wahl ist abhängig davon, ob das „JdbcTemplate" von Spring oder lieber Plain JDBC benutzt werden soll.

Typische Anwendungsfälle für die Java-basierten Migrationsskripte sind BLOB- und CLOB-Änderungen oder fortgeschrittene Änderungen an Massendaten (Neuberechnungen, fortgeschrittene Formatänderungen usw.). Der Dateiname der Java-Klassen folgt demselben Schema wie bei den SQL-Skripten, nur dass die Dateiendung „java" ist *(siehe Abbildung 7)*.

Verwaltung der Migrationsskripte

Die Migrationsskripte werden wie normaler Quelltext behandelt und sollten zusammen mit den anderen Quelltexten des betroffenen Projekts abgelegt sein. Wenn man beispielsweise Maven als Build-Werkzeug benutzt, wird ein eigenes Maven-Modul im Projekt für die Migrationsskripte angelegt. Die Java-basierten Skripte sind unter „src/main/java" im Package „db.migration" abgelegt, die SQL-basierten Skripte unter „src/main/resources" *(siehe Abbildung 8)*.

Die während des Maven-Builds generierte Jar-Datei (im Beispiel „database-scripts-0.0.3.jar") ist im Flyway-CLI-Werkezeug unter „jars" abgelegt *(siehe Abbildung 9)*. Mit der passenden Konfiguration unter „conf/flyway.properties" *(siehe Listing 4)* können die Migrationsskripte per „flyway migrate"-Befehl ausgeführt werden *(siehe Abbildung 10)*.

Testen der Migrationsskripte während eines Builds

Wie auch bei Java-Quelltexten sollen auch die Migrationsskripte während eines Builds getestet werden. Da die Migrationsskripte

```
public class V1_1_0__Insert_Data implements JdbcMigra-
tion {

  @Override
  public void migrate(Connection connection) throws
Exception {
     try (Statement statement = connection.createS-
tatement()) {
        statement.execute("Insert into person (first_
name, last_name) Values ('Alice', 'Bob')");
     }

  }

}
```

Listing 2

```
sparsick@sparsick-ThinkPad-T430s ~/dev/NetBeansProjects/flyway-talk/flyway-4.0.3 $ ./flyway migrate
Flyway 4.0.3 by Boxfuse

Database: jdbc:mysql://192.168.33.10:3306/ (MySQL 5.5)
Successfully validated 3 migrations (execution time 00:00.014s)
Creating schema `flyway_demo` ...
Creating Metadata table: `flyway_demo`.`schema_version`
Current version of schema `flyway_demo`: 0
Migrating schema `flyway_demo` to version 1.0.0 - create person table
Migrating schema `flyway_demo` to version 1.1.0 - Insert Data
Migrating schema `flyway_demo` to version 2.0.0 - add column job title
Successfully applied 3 migrations to schema `flyway_demo` (execution time 00:00.051s).
sparsick@sparsick-ThinkPad-T430s ~/dev/NetBeansProjects/flyway-talk/flyway-4.0.3 $
```

Abbildung 10: Ausführung der Migrationsskripte mit Flyway-CLI

```
 T E S T S
-------------------------------------------------------
Running db.migration.DbMigrationITest
INFO  - ertyClientProviderStrategy - Found docker client settings from environment
INFO  - ckerClientProviderStrategy - Found Docker environment with Environment variables, system properties and defaults. Resolved:
    dockerHost=unix:///var/run/docker.sock
    apiVersion='{UNKNOWN_VERSION}'
    registryUrl='https://index.docker.io/v1/'
    registryUsername='sparsick'
    registryPassword='null'
    registryEmail='null'
    dockerConfig='DefaultDockerClientConfig[dockerHost=unix:///var/run/docker.sock,registryUsername=sparsick,registryPassword=<null>,registryEmail=<null>,regis

INFO  - DockerClientFactory        - Docker host IP address is localhost
INFO  - DockerClientFactory        - Connected to docker:
  Server Version: 17.05.0-ce
  API Version: 1.29
  Operating System: Linux Mint 18.2
  Total Memory: 19511 MB
       ℹ Checking the system...
       ✔ Docker version is newer than 1.6.0
       ✔ Docker environment has more than 2GB free
       ✔ File should be mountable
       ✔ Exposed port is accessible
INFO  - 🐳 [mysql:latest]          - Creating container for image: mysql:latest
INFO  - 🐳 [mysql:latest]          - Starting container with ID: 2668be66c2631e49b5bcb4e180665d223525ec896ea78034326076d5f9063d53
INFO  - 🐳 [mysql:latest]          - Container mysql:latest is starting: 2668be66c2631e49b5bcb4e180665d223525ec896ea78034326076d5f9063d53
INFO  - 🐳 [mysql:latest]          - Waiting for database connection to become available at jdbc:mysql://localhost:32769/test using query 'SELECT 1'
INFO  - 🐳 [mysql:latest]          - Obtained a connection to container (jdbc:mysql://localhost:32769/test)
INFO  - 🐳 [mysql:latest]          - Container mysql:latest started
INFO  - VersionPrinter             - Flyway 4.0.3 by Boxfuse
INFO  - DbSupportFactory           - Database: jdbc:mysql://localhost:32769/test (MySQL 5.7)
INFO  - DbValidate                 - Successfully validated 2 migrations (execution time 00:00.011s)
INFO  - MetaDataTableImpl          - Creating Metadata table: `test`.`schema_version`
INFO  - DbMigrate                  - Current version of schema `test`: << Empty Schema >>
INFO  - DbMigrate                  - Migrating schema `test` to version 1.0.0 - create person table
INFO  - DbMigrate                  - Migrating schema `test` to version 2.0.0 - add column job title
INFO  - DbMigrate                  - Successfully applied 2 migrations to schema `test` (execution time 00:00.133s).
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 13.9 sec
```

Abbildung 11: Ausführung der Tests für Migrationsskripte mit dem Flyway-Java-API und Testcontainers während eines Maven-Builds

```
public class V1_2_0__Insert_Data_With_Spring_Support implements SpringJdbcMigration {

  @Override
  public void migrate(JdbcTemplate jdbcTemplate) throws Exception {
     jdbcTemplate.execute("Insert into person (first_name, last_name) Values ('Charlie', 'Delta')");
  }

}
```

Listing 3

```
flyway.url=jdbc:mysql://192.168.33.10:3306
flyway.user=flyway
flyway.password=flyway
flyway.schemas=flyway_demo
# Unprefixed locations or locations starting with classpath: point to a package on the classpath and may contain
both sql and java-based migrations.
# Locations starting with filesystem: point to a directory on the filesystem and may only contain sql migrations.
flyway.locations=db/migration
```

Listing 4

Datenbank-spezifische SQL-Syntax enthalten können, muss auch eine Datenbank des entsprechenden Datenbank-Anbieters während des Builds vorhanden sein. Da beißen sich zwei Anforderungen an einen Build. Auf der einen Seite soll der Build von externen Anwendungen (wie einer Datenbank) unabhängig sein, zum anderen sollen auch die Migrationsskripte während eines Builds getestet werden, um so früh wie möglich eventuelle Fehler zu erkennen.

Aus diesem Dilemma kann die Java-Bibliothek „Testcontainers" *[3]* weiterhelfen. Sie hilft auf eine einfache Art und Weise, während eines JUnit-Tests Docker-Container *[4]* hochzufahren, und bietet für Datenbanken schon vorgefertigte JUnit-Rules an. Ein Test für die Migrationsskripte sieht dann so aus, dass ein JUnit-Test geschrieben wird, in dem die JUnit-Rule „MySQLContainer" (falls wie im Beispiel mySQL die Datenbank der Wahl ist) vom Test-Container eingebunden wird. Mit dem Flyway-Java-API können dann die Migrationsskripte gegen die Datenbank im Container ausgeführt werden *(siehe Listing 5)*. Dieser Test wird wie gewohnt während des Builds mit ausgeführt *(siehe Abbildung 11)*.

Die Grenzen von Flyway

Es soll nicht verschwiegen werden, dass Flyway auch seine Grenzen hat. Einer der meist genannten Kritikpunkte ist der fehlende Rollback-Mechanismus. Dies war eine bewusste Entscheidung bei Flyway *(siehe [5])*. Die Macher von Flyway empfehlen, bei einer fehlgeschlagenen Migration lieber das Backup der Datenbank wieder einzuspielen.

Wie bei der Vorstellung der SQL-Migrationsskripte erwähnt, unterstützt Flyway Datenbank-spezifisches SQL. Daraus folgt, dass, falls die Anwendung mit unterschiedlichen Datenbank-Anbietern laufen muss, die Migrationsskripte redundant für die jeweilige Datenbank geschrieben werden müssen. Dann ist es eine berechtigte Frage, ob Flyway das richtige Werkzeug für einen ist und ob dieser Anwendungsfall nicht besser mit Liquibase gelöst wird.

Fazit

Dieser Artikel zeigt, welche Grundregeln beachtet und welche Maßnahmen ergriffen werden sollen, wenn die Updates des Datenbank-Schemas kontrolliert und automatisiert ablaufen sollen. Anhand Flyway wird gezeigt, wie ein Werkzeug bei der Umsetzung der Maßnahmen helfen kann.

```
public class DbMigrationITest {
  @Rule
  public MySQLContainer mysqlDb = new MySQLContain-
er();
  @Test
  public void testDbMigrationFromTheScratch(){
     Flyway flyway = new Flyway();
     flyway.setDataSource(mysqlDb.getJdbcUrl(),
mysqlDb.getUsername(), mysqlDb.getPassword());

     flyway.migrate();
  }
}
```

Listing 5

Quellen

[1] Paul M. Duvall, Steve Matyas und Andrew Glover: Continuous Integration, Addison-Wesley (2007)
[2] *http://flywaydb.org*
[3] *https://www.testcontainers.org*
[4] *https://www.docker.com*
[5] *https://flywaydb.org/documentation/faq#downgrade*

Sandra Parsick
mail@sandra-parsick.de

Sandra Parsick, geb. Kosmalla, ist als freiberufliche Software-Entwicklerin und Consultant im Java-Umfeld tätig. Seit dem Jahr 2008 beschäftigt sie sich mit agiler Software-Entwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich der Enterprise-Anwendungen, agilen Methoden, Software Craftsmanship und in der Automatisierung von Software-Entwicklungsprozessen. In ihrer Freizeit engagiert sie sich in der Softwerkskammer Ruhrgebiet.

Der sprechende Kickertisch

Marco Buss, Opitz Consulting Deutschland GmbH

Was passiert, wenn man einen Kicker digitalisiert und mit Amazon Alexa in einen Raum sperrt? Dieser Artikel geht genau dieser Frage nach und zeigt an einem spaßigen Beispiel, wie man Alexa zu einer virtuellen Stadionsprecherin macht.

Der Tischkicker im Büro des Autors ist in seinem Unternehmen ein beliebter Pausentreffpunkt, an dem regelmäßig Matches und Turniere stattfinden. Die Punkte werden manuell gezählt – in der heutigen Zeit, in der alles digitalisiert wird, für den Autor ein unhaltbarer Zustand! Er begab sich also auf die Suche: Anfang 2016 wurde er auf die Amazon-IoT-Buttons aufmerksam und fand auch schon einige interessante und witzige Projekte im Internet, die mit diesen Buttons umgesetzt wurden. Bei einem dieser Projekte wurden von Alexa die Punkte beim Tischtennisspiel gezählt. Genau das war es, was ihm für den Kicker vorschwebte. Mitte Juni 2016 waren die IoT-Buttons auch in Deutschland verfügbar und er begann umgehend mit der Umsetzung.

Grobe Architektur-Skizze

Die Architektur der Lösung ist nicht ganz einfach: Die Punkte werden durch das Betätigen der IoT-Buttons gezählt. Ein in NodeJS geschriebener Server reagiert auf diese Events und interagiert mit dem Alexa-Voice-Service. Dieser persistiert die Punkte mithilfe eines Alexa-Skills als Lambda-Funktion in einer Dynamo DB und liefert eine Sprachantwort zurück. Die Antworten werden auf einem angeschlossenen Lautsprecher ausgegeben *(siehe Abbildung 1)*.

Ein Amazon Echo kann genutzt werden, um neue Spiele zu starten oder um den aktuellen Punktestand abzufragen. Der Aufbau war am Ende leider etwas umständlicher als vom Autor angenommen. Bevor er mehr über Alexa-Skills und den Alexa-Voice-Service wusste, dachte er, über die IoT-Buttons einen Event auslösen zu können und dann direkt auf einem Echo auszugeben. Das war allerdings so nicht umsetzbar.

AWS IoT

Mit der Amazon Internet of Things Plattform (AWS IoT) bietet Amazon einen komplett gemanagten Service für IoT-Anwendungen beliebiger Größe. Wie für Serverless-Angebote üblich, müssen sich die Anwender dabei keine Gedanken über den Betrieb und die Skalierung machen. All diese Aufgaben werden durch Amazon selbst abgedeckt. Es entstehen lediglich Kosten, wenn der Service auch wirklich verwendet wird.

Das Herzstück von AWS IoT ist ein MQTT-Broker für den Nachrichtenaustausch. Abgerechnet wird nach der Anzahl der Nachrichten und nach dem verbrauchten Datenvolumen. Beim Thema „Sicherheit", das ja für alle IoT-Anwendungen ein wichtiger Punkt ist, bietet Amazon hier die Authentifizierung und Autorisierung basierend auf Zertifikaten an. Ohne ein gültiges Zertifikat ist also keine Kommunikation möglich.

Neben diesen Features bietet die Plattform noch eine Rules Engine sowie sogenannte „Device Shadows". Mit der Rules Engine können Nachrichten der Devices bereits vorgefiltert werden. Die Device Shadows sind eine Möglichkeit, mit Devices zu kommunizieren, die nicht permanent online sind. Ist ein Device nicht online, kann sein Zustand trotzdem angepasst werden. In diesem Fall wird lediglich das Device Shadow verändert und es erfolgt eine Synchronisation des Zustands, sobald das Device wieder online ist *(siehe Abbildung 2)*.

So funktioniert das Zählen der Punkte im AWS IoT: Jeder IoT-Button besitzt eine eindeutige ID und ist vor der Benutzung einzurichten. Zum einen muss er in ein WLAN eingebunden sein, zum anderen müssen entsprechende Zertifikate gespeichert werden. Die benötigten Zertifikate hat der Autor einfach auf der AWS-IoT-Website erzeugt. Anschließend ist der Button direkt einsatzbereit. Bei jedem Klick wird eine Nachricht an „MQTT Topic iotbutton/<ID des Buttons>" versendet. *Listing 1* zeigt eine Beispielnachricht.

Neben der Seriennummer ist der „clickType" interessant. Unterschieden wird zwischen einem langen Druck auf die Taste, einem normalen Klick und einem Doppelklick. Für den Kicker wurde die NodeJS-Anwendung per MQTT mit AWS IoT verbunden. Sobald die Nachricht eines IoT-Buttons empfangen wird, startet die entsprechende Routine, je nachdem, ob der entsprechende Button von Spieler 1 oder von Spieler 2 betätigt wurde.

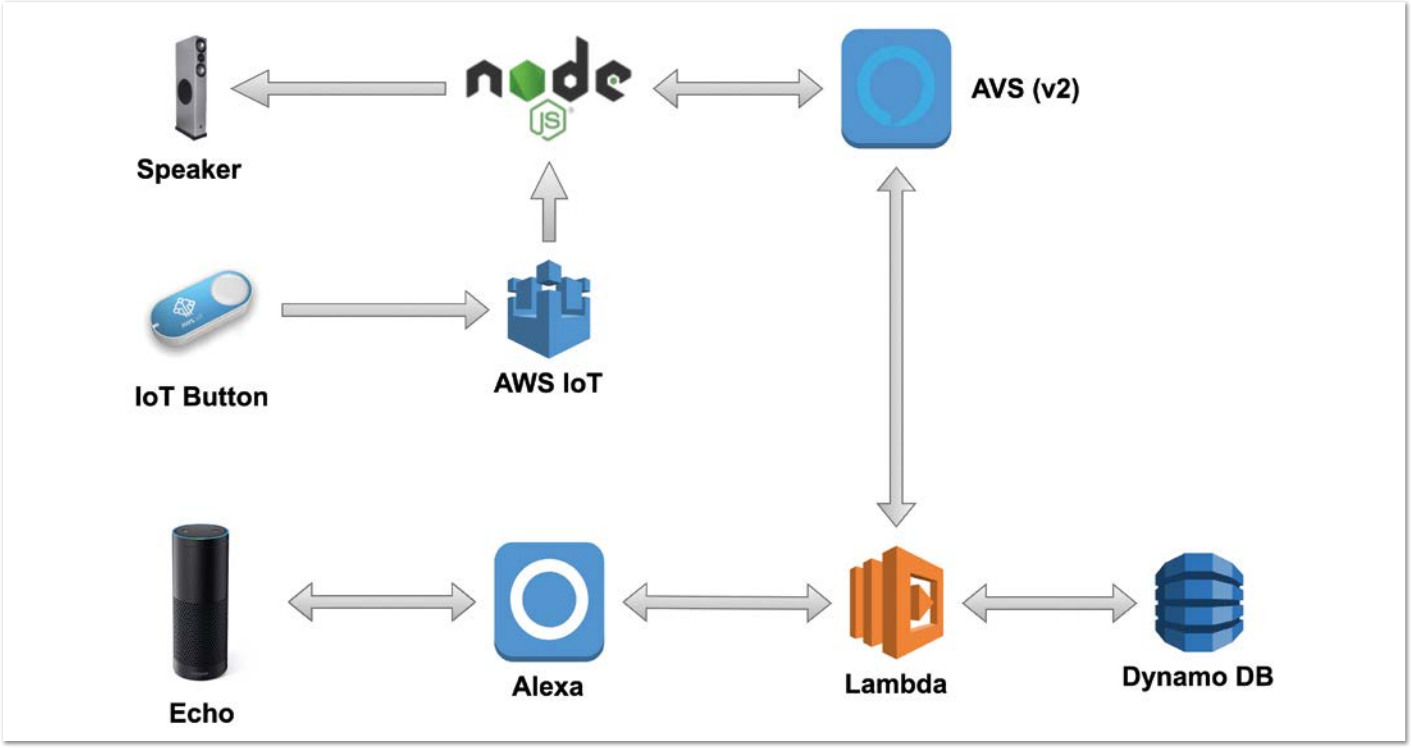


Abbildung 1: Die Architektur des Use Case „Sprechender Kickertisch"

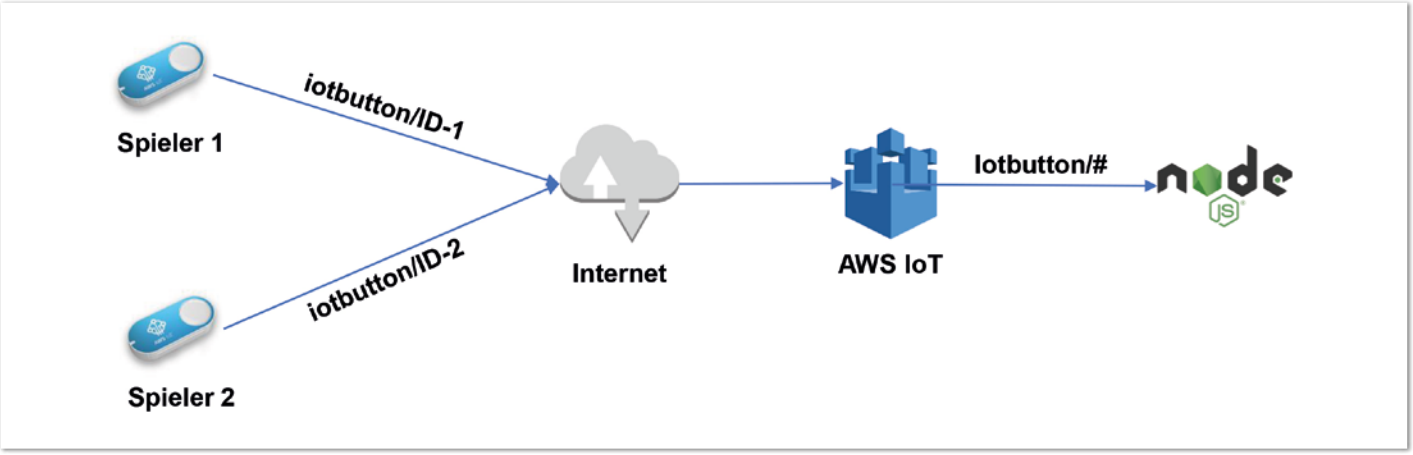


Abbildung 2: Aufbau des AWS IoT zur Punktezählung

Alexa Voice Service

Viele Anwender wissen nicht, dass Alexa nicht nur aus den Alexa-Skills besteht, sondern auch den Alexa-Voice-Service (AVS) beinhaltet. Dieser ist das eigentliche Herzstück jedes Echos und ist beispielsweise für die Auswertung der Sprache zuständig.

Anstatt diesen Service nur für eigene Produkte zu verwenden, bietet Amazon mit AVS für jeden Anbieter die Grundlagen, um Alexa in eigene Produkte zu integrieren. Diese Technik wird von immer mehr Herstellern genutzt: Seat plant beispielsweise, Alexa in einige Autos zu integrieren. Auch einige Lautsprecher-Hersteller haben eine Integration vor oder bereits entsprechende Produkte auf den Markt gebracht.

```
{
  "serialNumber": "ABCDEFG12345",
  "clickType": "SINGLE",
  "batteryVoltage": "2000 mV"
}
```

Listing 1: Nachricht des IoT-Buttons

Für den leichten Einstieg bietet Amazon ein C++-SDK an sowie detaillierte Anleitungen, wie dieses auf einem Raspberry Pi oder Linux/Mac-Rechner zu installieren ist. Für die Kommunikation mit AVS verwendet das SDK eine HTTP-basierte Schnittstelle. Mit deren Hilfe ist es möglich, diesen Service mit sehr vielen Programmiersprachen anzubinden. Neben dem C++-SDK existiert im Alexa-Git-Repository auch eine Java-Demo-Applikation *(siehe „https://github.com/alexa/alexa-avs-sample-app“).*

AVS-Security

Authentifizierung und Autorisierung erfolgen bei AVS Token-basiert. Für jede Interaktion ist ein Access-Token erforderlich. Um diesen zu erhalten, muss ein Device die Möglichkeit bieten, mit dem Dienst „Login with Amazon“ zu interagieren. Dafür stehen verschiedene Möglichkeiten zur Verfügung. Im einfachsten Fall kann man eine Companion Site oder App verwenden.

Durch „Login with Amazon“ ist auch die Verbindung vom Device zum Alexa-Nutzer festgelegt. Nach erfolgreicher Authentifizierung erhält die Companion Site oder App alle benötigten Tokens, um ein neues Access-Token anzufragen. Diese Tokens haben im Gegensatz zum

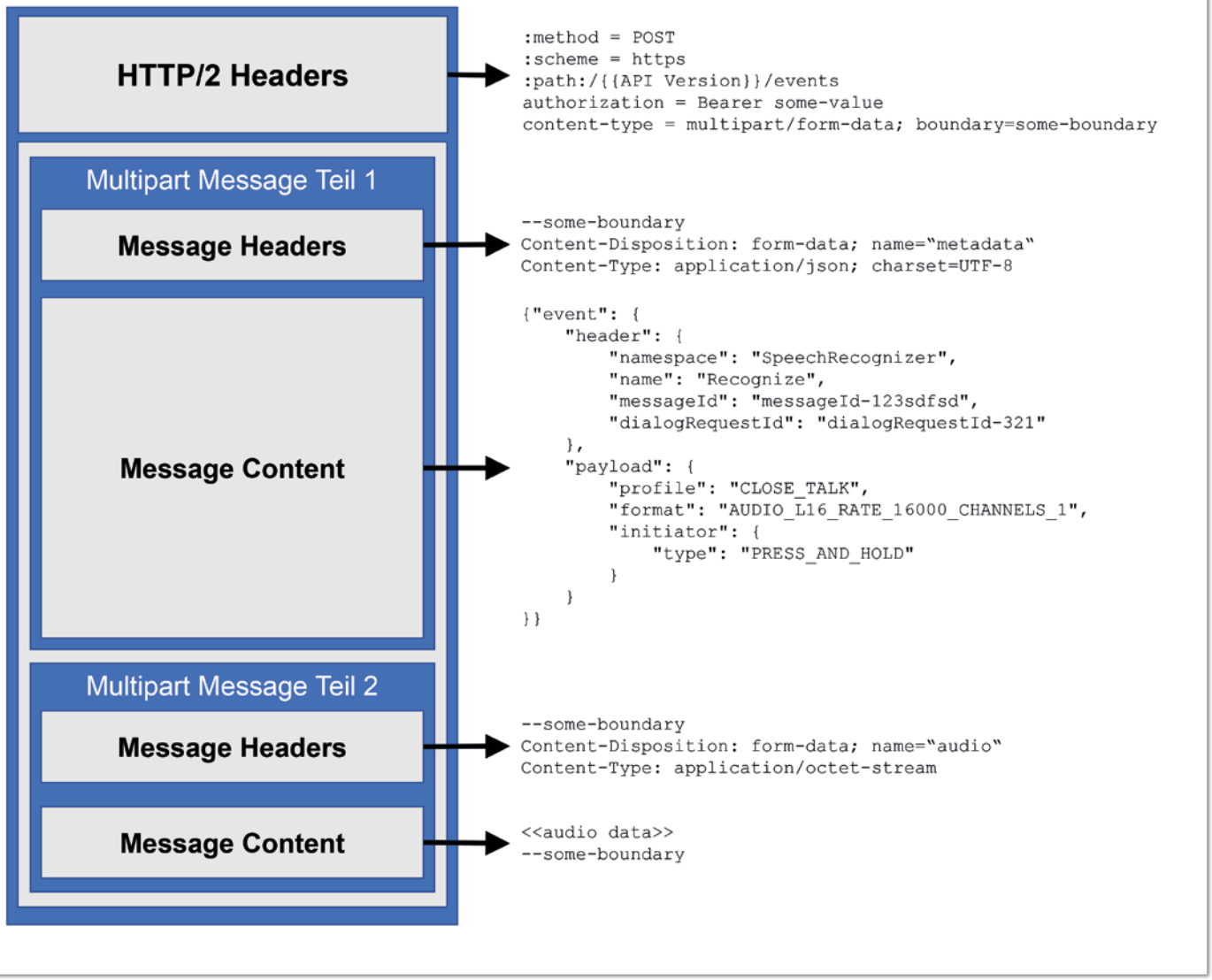


Abbildung 3: Aufbau einer AVS-Nachricht

Access-Token eine lange Lebenszeit und sollen auf dem Device gespeichert sein. Das Access-Token hingegen hält nur eine Stunde und ist daher in entsprechenden Abständen zu erneuern.

AVS v2

Anfang 2016 kam die Version 2 der AVS-Schnittstelle heraus. Sie brachte einen Umstieg auf HTTP/2 sowie eine signifikante Erweiterung des API. Während die Version 1 noch auf einem klassischen Request/Response-Prinzip basierte, führte Version 2 ein neues Kommunikationsmodell ein, das auf Events und Direktiven aufbaut.

Events sind dabei die Nachrichten, die vom Endgerät zum AVS-Service gesendet werden. Direktiven wiederum sind Anweisungen, die der AVS-Service initiiert. Diese Art der Kommunikation ist notwendig, um beispielsweise die neuen Notifications und die Steuerung des Endgeräts mit der Alexa-App zu ermöglichen.

Um die Schnittstelle zu nutzen, wird eine einzige HTTP/2-Verbindung zum AVS aufgebaut. Sobald die Verbindung steht, muss innerhalb von zehn Sekunden ein sogenannter „Downchannel-Stream" erzeugt werden. Dieser wird vom AVS verwendet, um die Direktiven zum Client zu senden. Der Stream befindet sich clientseitig in einem „half-closed State". Das bedeutet, dass der Client keine weiteren Daten sendet, aber weiterhin Daten vom Server empfangen kann.

Sobald der Downchannel-Stream erzeugt wurde, muss ein Sync erfolgen. Dieser teilt dem AVS den Zustand des Endgeräts mit. Er enthält beispielsweise Informationen zur aktuell eingestellten Lautstärke oder zum Mediafile, das gerade abgespielt wird. AVS-Messages sind HTTP/2-Multipart-Messages *(siehe Abbildung 3)*. Im HTTP/2-Header befinden sich allgemeine Informationen wie die Autorisierungsinformationen in Form des Access-Tokens.

Jeder Teil der Multipart Message besteht ebenfalls aus einem Header und dem eigentlichen Content. Im ersten Teil besteht der Content aus Informationen, die besagen, um welchen Event es sich handelt, und aus Parametern, die den Event näher beschreiben. Im zweiten Teil folgen dann die eigentlichen Nutzungsdaten wie im Beispiel ein Audio File.

```
{"intents": [
  {"intent": "StartGame"},
  {"intent": "Punktestand"},
  {"intent": "PlayerPoint",
     "slots": [
        {
           "name": "Player",
           "type": "SPIELER_PUNKT"
        }
     ]},
  {"intent": "AMAZON.HelpIntent"},
  {"intent": "AMAZON.StopIntent"},
  {"intent": "AMAZON.CancelIntent"},
  {"intent": "AMAZON.YesIntent"},
  {"intent": "AMAZON.NoIntent"}
]}
```

Listing 2: Intent-Schema des Kickertisch-Skills

```
Punktestand wie ist der punktestand
Punktestand wie steht es
Punktestand gib mir den punktestand
PlayerPoint Punkt für {PLAYER_POINT}
```

Listing 3: Sample Utterances

AVS und der Kicker

Nachdem die Grundlagen für den AVS nun geklärt sind, können wir uns der Frage nähern, wie der AVS genutzt werden kann, um die Punkte eines Kickerspiels zu zählen.

Wie bereits erwähnt, bekommt die NodeJS-Anwendung eine Nachricht per MQTT. Diese informiert sie darüber, welcher Button gedrückt wurde. Damit weiß sie, für welches Team der Punkt gezählt werden soll. An dieser Stelle kommt das Speech-Recognizer-Interface des AVS zum Einsatz; im Übrigen das einzige Interface des AVS, das für den Use Case benötigt wird. Es wird verwendet, um Sprachbefehle zu analysieren und die Befehle gemäß den Informationen des Alexa-Skills zu starten. Dafür benötigt das Speech-Recognizer-Interface ein Sprach-Sample. Das könnte beispielsweise „Alexa, wie spät ist es?" sein, wie es in der Interaktion mit einem Amazon-Echo üblich ist. Im Kickerspiel wäre ein Beispiel für ein Sprach-Sample „Punkt für Rot" oder „Punkt für Blau".

Das Speach-Recognizer-Interface interessiert nicht die Quelle des Sprach-Samples. Amazon Echo verarbeitet Sprach-Samples, die per Mikrofon aufgenommen wurden. Bei der Kicker-Anwendung wurde für jede Seite im Vorfeld ein entsprechendes Audiofile erzeugt und an den AVS gesendet. Die Samples müssen dabei folgende Eigenschaften erfüllen:

- 16 Bit Linear PCM
- 16 kHz Sample Rate
- Single Channel
- Little Endian Byte Order

Die verwendeten Sprach-Samples müssen übrigens nicht selbst gesprochen und entsprechend umgewandelt werden, dafür gibt es einen weiteren Service von AWS: Amazon Polly, ein Text-to-Speach-Service, den man beispielsweise in einen Blog einbinden kann, um die Posts vorlesen zu lassen, erzeugt auf einfache Weise die benötigten Dateien.

Im Gegenzug liefert der Speech-Recognizer-Service als Antwort ein Audiofile mit dem Ergebnis des Alexa-Skills. Beim Kickerbeispiel beinhaltet die Antwort unter anderem den aktuellen Spielstand. Die NodeJS-Anwendung gibt das Ergebnis über die angeschlossenen Lautsprecher aus.

Alexa-Skills-Kit

Die eigentliche Verarbeitungslogik ist im vorgestellten Beispiel als Alexa-Skill implementiert. Der AVS erkennt im Sprach-Sample, dass der Tischkicker-Skill aufgerufen werden soll, um den Punkt für einen Spieler zu zählen.

Alexa-Skills sind mit dem Alexa-Skills-Kit implementiert und bestehen aus zwei Teilen: einem Interaction Model und der eigentlichen Service-Funktion. Das Interaction Model enthält ein Intent-Schema und die Sample Utterances. Das Intent-Schema definiert, welche

Befehle der Skill unterstützt. Es wird zwischen eingebauten Intents und selbst definierten Intents unterschieden.

Listing 2 zeigt das Intent-Schema des Tischkicker-Skills. Die eingebauten Intents erkennt man am „AMAZON."-Präfix. Die restlichen Intents sind eigene Intents, die nur für den Skill gelten. Jeder Intent kann auch sogenannte „Slots" enthalten, wie beim „PlayerPoint"-Intent zu sehen ist. Slots definieren die Parameter eines Intent. Auf diese Parameter kann das System auch innerhalb der Service-Funktion zugreifen.

In den Sample Utterances sind Beispielsätze für die eigenen Intents abgelegt. Für jedes Intent sollten mehrere Beispielsätze definiert werden, damit der AVS-Service Intents und Slots entsprechend erkennen kann. In *Listing 3* sind die Sample Utterances für die beiden Intents „PlayerPoint" und „Punktestand" zu sehen.

Die Service-Funktion kann ein Web-Service sein, wird in den meisten Fällen jedoch als AWS-Lambda-Funktion implementiert. Für die Implementierung des Skills als Lambda-Funktion existieren verschiedene SDKs. Der Tischkicker-Skill ist in NodeJS implementiert.

Mithilfe eines Amazon Echo kann ein neues Spiel gestartet und der Punktestand unabhängig von einem gefallenen Tor erfragt werden. Da die Punkte ebenfalls mit diesem Skill gezählt werden, kann ein Tor auch ohne die IoT-Buttons mit den Kommandos „Punkt für Blau" oder „Punkt für Rot" gezählt werden.

Fazit

Der Artikel zeigt, wie einfach es sein kann, eine neue Idee mit sprachbasierten Systemen zu verwirklichen. Amazon bietet dafür mit seinen diversen Services – allen voran den Amazon-Voice-Services – eine gute Grundlage. Der Anbieter begünstigt zudem mit recht großzügigen Frei-Kontingenten für viele Services auch das Experimentieren mit den einzelnen Diensten.

Marco Buss
marco.buss@opitz-consulting.de

Marco Buss ist Senior Consultant bei Opitz Consulting und beschäftigt sich seit mehr als zehn Jahren mit der Java-Entwicklung im Enterprise-Umfeld. Seine aktuellen Themenschwerpunkte sind Cloud, Big Data und IoT.

Bist Du ein Macher-Typ, Weiter-Denker und Selbst-Bestimmer? Dann bewirb Dich jetzt bei NovaTec!

Wir sind kontinuierlich auf der Suche nach neuen Talenten und bieten Dir exzellente Einstiegsmöglichkeiten

WIR SUCHEN

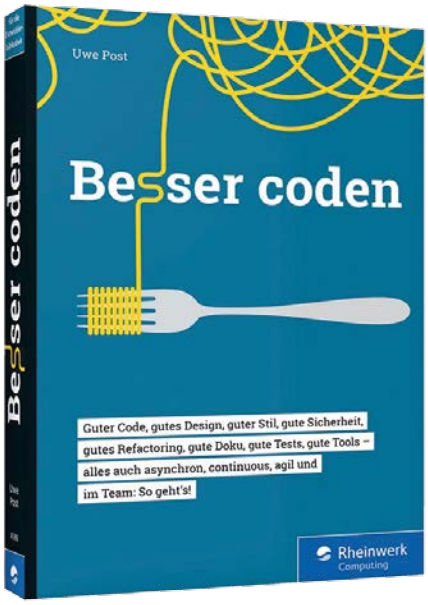
- **Young Professionals**
- **Studenten für Abschlussarbeiten**
- **Werkstudenten**
- **Praktikanten**

Wenn Du über Know-how und Begeisterungsfähigkeit verfügst und Spaß an Teamerfolgen hast, dann bieten wir Dir abwechslungsreiche Aufgaben, spannende Projekte und ein angenehmes und produktives Arbeitsklima.

Bewirb Dich jetzt und werde Teil unseres Teams!

karriere.novatec-gmbh.de

NOVATEC
make IT happen!



Besser coden

gelesen von Heiko Sippel

Uwe Post hat sein Buch „Besser coden" genannt, und das erscheint nach dem Durchlesen etwas ungenau, um den Inhalt zu beschreiben. „Besser entwickeln" wäre passender gewesen (aber vielleicht nicht so schön griffig), denn in dem Buch wird weit mehr behandelt als die Erstellung guten Codes. Auf mehr als 380 Seiten geht es in zwölf Kapiteln um viele Aspekte des Entwicklerberufs, was wieder einmal deutlich macht, wie viel mehr es zu einem guten Entwickler braucht als lediglich die Beherrschung einer oder mehrerer Programmiersprachen – was leider vielen IT-Verantwortlichen nicht bewusst ist, und vielleicht auch vielen Berufseinsteigern nicht.

Das Buch richtet sich daher im Wesentlichen an Programmierer, die nun den Schritt zum wirklichen Entwickler machen wollen. Daher wird vor allem Wert auf die Bedeutung verschiedener Technologien gelegt, die die Zusammenarbeit in Teams fördern; Einzelkämpfer sind nicht angesprochen. Der Inhalt bietet einen sehr guten Überblick über verschiedene Bereiche, darunter (kleine Auswahl):

- Code-Konventionen und -Metriken
- Continuous Integration
- Entwurfsmuster
- Versionskontrolle
- Testverfahren
- Entwicklungs-Tools
- Scrum und Kanban

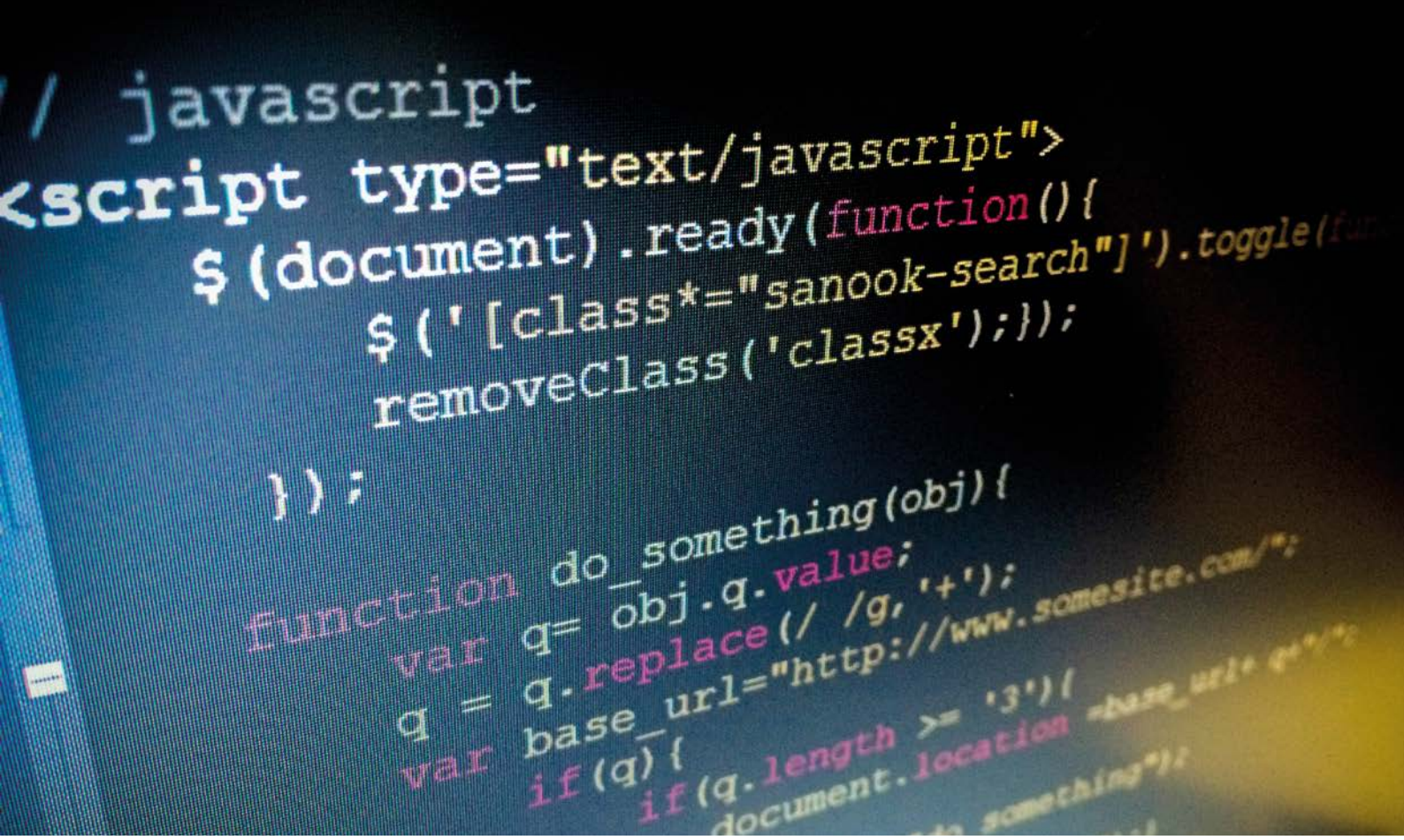
Alle Themen werden von Grund auf angegangen, sodass auch Neulinge in den jeweiligen Bereichen alles verstehen können. Dabei wird keine bestimmte Programmiersprache vorausgesetzt, obwohl Java und C/C++ im Vordergrund stehen. Die Kapitel sind unabhängig voneinander, man kann also Bekanntes auch überspringen, ohne den Faden zu verlieren. Natürlich sind viele Themen zu komplex, um sie vollständig auf einigen Dutzend Seiten zu behandeln, die Kapitel bieten jedoch einen sehr guten Einstieg. Es fehlen zwar Hinweise auf weitere Literatur oder Links, aber unter den Stichwörtern findet man genug im Internet. Die Liste der Tools ist sicher nicht vollständig – dafür wäre ein noch dickeres Buchs notwendig –, allerdings auf dem aktuellen Stand.

Man merkt dem Buch an, dass der Autor aus der Erfahrung heraus schreibt. Teilweise sind auch kleine Berichte aus seinem Berufsleben dabei, was den Text auflockert, der im Übrigen gut lesbar ist. Das ist zwar immer Geschmackssache und mag nicht jedem gefallen. Es ist kein akademischer Text. Über einige Ansichten kann man sicher auch diskutieren. So wäre etwa statt des Kapitels „Trollfütterung", in dem es um Gruppendynamik geht, ein Kapitel zu „DevOps" besser gewesen, denn dieses Thema wird immer wichtiger.

Ein kleiner Kritikpunkt ist der etwas schwache Druck der Abbildungen. Screenshots sind teilweise nur schwer zu entziffern, wie auch die Cartoons, die jedes Kapitel einleiten. Alte Hasen werden in dem Buch nichts Neues finden; für alle anderen gibt es eine Kaufempfehlung, auch wenn der Inhalt des einen oder anderen Kapitels bekannt sein sollte.

Heiko Sippel
heiko.sippel@infaktum.de

Titel:	**Besser coden**
Autor:	**Uwe Post**
Verlag:	**Rheinwerk Computing 2017**
Umfang:	**388 Seiten**
Preis:	**24,90 Euro, eBook 21,90 Euro, Bundle Buch + eBook 29,90 Euro**
ISBN:	**978-3-8362-4598-2**



JavaScript für Java-Developer – brauchen wir das wirklich?

Robert Rohm, Aeonium Software Systems

JavaScript ist „The World's Most Misunderstood Programming Language" [1] – und ein fester Bestandteil der Java-Laufzeit-Umgebung. Die JVM bringt die JavaScript-Engine Nashorn mit. In JavaFX besitzt die WebView-Komponente als Teil des eingebetteten WebKit-Browsers ebenfalls einen JavaScript-Interpreter, der auch aus Java heraus genutzt werden kann. Nur: Wofür soll das gut sein?

Mögliche Antworten auf diese Frage lassen sich auf zwei Wegen finden. Zum einen hilft das „Big Picture", wenn man sich vergegenwärtigt, wo JavaScript im Umfeld von Java-Anwendungen auftaucht *(siehe Abbildung 1)*. Wichtig ist, dass nicht nur der JavaScript-Interpreter, sondern auch die JavaScript-APIs mit ihm zusammen die JS-Laufzeitumgebung ausmachen. Im Web-Browser sind dies das DOM-API und die HTML5-JS-APIs. Standalone-Umgebungen wie „node.js" haben ihr eigenes API – und in der JVM kann Nashorn auf die Java-APIs zugreifen und die Webkit-Engine auf das dargestellte HTML. Andererseits gibt es gerade im Zusammenspiel mit der JVM und der Verwendung von Nashorn einiges zu beachten. Deshalb werden zunächst grundlegende Themen rund um die Nashorn-Engine näher beleuchtet.

JavaScript in der JVM

Nashorn, die JavaScript-Engine der JVM, kann auf zwei Wegen genutzt werden: entweder mit dem Konsolen-Tool „jjs" *[2]* oder über das Java-Scripting-API *[3]* in der Java-Anwendung. Mit „jjs" wird die Nashorn-Engine an der Kommandozeile aufgerufen; man kann entweder eine Skript-Datei angeben oder Nashorn interaktiv verwenden.

Als JavaScript-Ausführungsumgebung hat Nashorn seine eigenen Laufzeit-Bibliotheken, das sind neben dem Scripting-API alle Bibliotheken des JRE. Wer bereits mit JavaScript vertraut ist, kann also jede Java-Klasse in JavaScript nutzen. *Listing 1* zeigt dies am Beispiel einer Dateiverarbeitung mit dem NIO2- und Collections-API. Es geht von dem fiktiven Szenario aus, dass innerhalb großer Text-Dateien

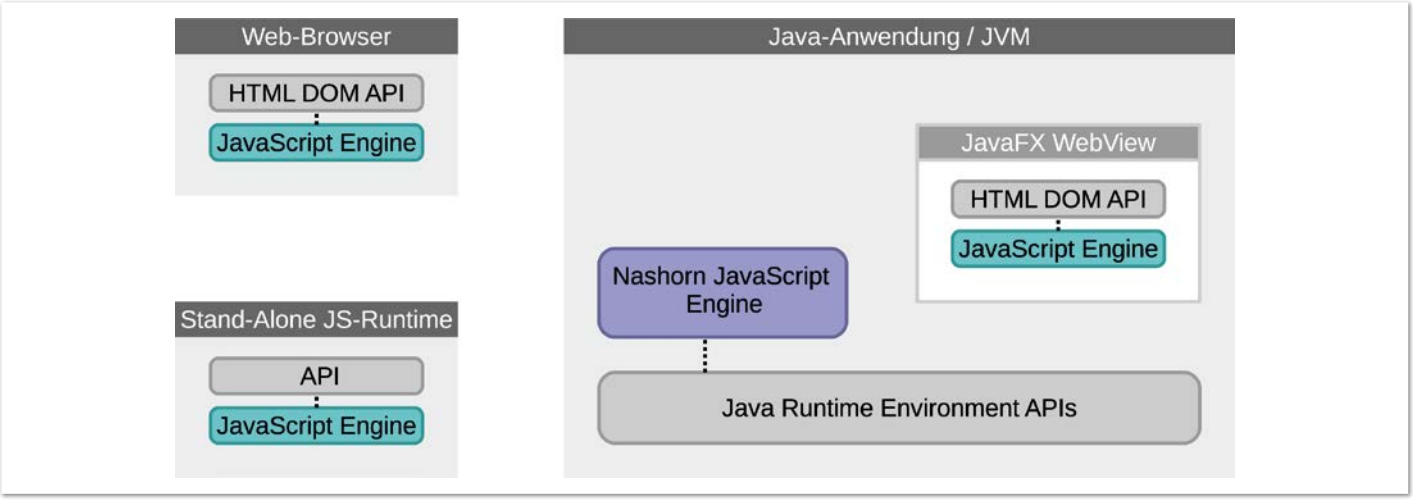


Abbildung 1: JavaScript im Umfeld von Java-Anwendungen

die maximale Zeilenlänge ermittelt werden soll – stellvertretend für anspruchsvollere Anforderungen. Man sieht neben dem Aufruf der Java-Methoden in JavaScript auch, wie Lambda-Ausdrücke in JavaScript abgebildet werden: mit Funktionsausdrücken beziehungsweise Closures.

Erfahrenere JavaScript-User werden einwenden, dass JavaScript ein eigenes Array-API *[4]* besitzt, das für diese Zwecke völlig ausreichend ist. Auch das ist kein Problem: Nashorn stellt in seiner Ausführungsumgebung das „Java"-Objekt bereit, das unter anderem Konvertierungsfunktionen besitzt, mit denen Java-Arrays in JavaScript-Arrays umgewandelt werden können – und umgekehrt. In *Listing 2* wird das String-Array mit „Java.from()" in ein JavaScript-Array umgewandelt, auf dem dann auch mit dem JavaScript-Array-API gearbeitet werden kann.

JavaFX auf Nashorn

Mit Nashorn lassen sich nicht nur sogenannte „Headless-Anwendungen" ohne GUI erstellen; auch Anwendungen mit Swing- oder JavaFX-Benutzeroberfläche können dank Nashorn in JavaScript geskriptet werden. Im Fall von JavaFX ist jedoch zu berücksichtigen, dass das JavaFX-Framework seinen eigenen Initialisierungszyklus besitzt. Die Start-Klasse einer JavaFX-Anwendung muss von „javafx.application.Application" ableiten und hat zwingend mehr oder weniger diese Grundstruktur *(siehe Listing 3)*.

Die Ableitung neuer Typen von Java-Klassen ist in Nashorn zwar möglich *[5]*, das würde jedoch nicht das Problem der Initialisierung lösen. Nashorn geht einen anderen Weg: Die Nashorn-Engine muss mit „jjs -fx" gestartet und der JavaScript-Code als Datei übergeben werden. Interaktives JavaFX-Scripting von der Konsole weg ist leider nicht möglich. Dieses Problem lässt sich umgehen, wenn der Script-Code dynamisch in einer bereits laufenden Java-JX-Anwendung ausgewertet wird.

Listing 4 zeigt, wie die Initialisierung einer JavaFX-Anwendung in JavaScript aussehen kann. Das Listing greift das Beispiel-Szenario wieder auf und visualisiert die Auswertung in einem JavaFX-Pie-Chart. Die Funktion „start(primaryStage)" ist übrigens verpflichtend. Sie ist auch in JavaScript mit Nashorn der Einsprungpunkt für JavaFX-Anwendungen.

```
var path = java.nio.file.Paths.get("/data/daten.csv");
var lines = java.nio.file.Files.readAllLines(path);

var maxLength = lines.stream().mapToInt(function(t)
{return t.length()}).max().getAsInt();
print("Max. Zeilenlänge: " + maxLength);
```

Listing 1: Java-Klassen in JavaScript verwenden

```
var path = java.nio.file.Paths.get("/data/daten.csv");
var lines = java.nio.file.Files.readAllLines(path);

// Umwandlung in ein JavaScript-Array
var linesArray = Java.from(lines);

// Verarbeitung mit der JavaScript-Array-API
var maxLength = linesArray.reduce(
  function(accumulator, value){
     return Math.max(accumulator || 0, value.length);
  }
);
print("Max. Zeilenlänge: " + maxLength);
```

Listing 2: Java-Arrays in JavaScript-Arrays konvertieren und verarbeiten

Die Umsetzung in JavaScript bringt gegenüber der Implementierung in Java einige Vereinfachungen; so dürfen Variablen, die in anonymen Funktionen oder Closures verwendet werden, durchaus erst unterhalb der Funktion deklariert sein. Auch die Notwendigkeit, dass solche Variablen „final" oder zumindest „effectively final" sein müssen, gibt es in JavaScript nicht.

Besseres Import-Management

Im *Listing 4* fällt die umständliche Deklaration der Variablen für die Java-Typen ins Auge. Für ein bequemeres und besseres Management der zu importierenden Klassen bietet Nashorn zwei Mechanismen: Einfacher geht es mit den Funktionen „importPackage" und „importClass" aus dem Nashorn-API *(siehe Listing 5)*.

Die Anweisung „importPackage(java.lang)" birgt ein Problem: Sie importiert Klassen als Konstruktor-Objekte in den globalen Scope der JavaScript-Engine. Darunter sind Standard-Typen wie „java.lang.Object", „java.lang.Boolean" etc. In JavaScript existieren eingebaute

Standard-Objekte mit dem Namen „Object“, „Boolean“, „Number“, „Array“ etc. Um Namenskollisionen zu vermeiden, wird statt der Import-Funktionen die Verwendung des „JavaImporter“ empfohlen. Mit ihm kann die Verwendung von Java-Typen auf einen definierten engeren Scope beschränkt werden. *Listing 6* zeigt, worauf beim Einsatz eines Importer in komplexeren Beispielen zu achten ist. So könnte die JavaFX-Anwendung mit „JavaImporter“ aussehen. Man sieht: Die „with(importer)“-Anweisung ist gegebenenfalls in inneren Funktionen zu wiederholen. Es reicht nicht, einen Scope mit Importer in der äußeren Funktion zu definieren.

```
import javafx.application.Application;
// …
public class MeineJavaFXApp extends Application {

  @Override
  public void start(Stage primaryStage) {
     // Aufbau des Fenster-Inhalts …
     VBox root = new VBox();
     root.getChildren().addAll( /* Fenster-Inhalt hier einfügen … */ );
     primaryStage.setScene(new Scene(root, 300, 250));
     primaryStage.show();
  }
  public static void main(String[] args) {
     Application.launch(args);
  }
}
```

Listing 3: Grundstruktur einer JavaFX-Anwendung

```
var Button = javafx.scene.control.Button;
var VBox = javafx.scene.layout.VBox;
var Scene = javafx.scene.Scene;
var PieChart = javafx.scene.chart.PieChart;
var PieChartData = javafx.scene.chart.PieChart.Data;
var FXCollections = javafx.collections.FXCollections;
var ObservableList = javafx.collections.ObservableList;

function start(primaryStage) {
  primaryStage.title = "Hello PieChart!";
  var button = new Button();
  button.text = "Datei laden ...";
  button.onAction = function () {
  var pieChartData = FXCollections.observableArrayList();
     var path = java.nio.file.Paths.get("/data/daten.csv");
     var lines = java.nio.file.Files.readAllLines(path);
     var n = 1;
     lines.stream().forEach(function(line){
        pieChartData.add(new PieChartData("Zeile " + n + ":\n" + line.length(), line.length()));
        n++;
     });
     chart.setData(pieChartData);
  };

  var root = new VBox();
  var chart = new PieChart();
  root.children.addAll(button, chart);
  primaryStage.scene = new Scene(root, 300, 250);
  primaryStage.show();
}
```

Listing 4: Mit Nashorn geskriptete Datenauswertung und -Visualisierung

```
// Dieses Skript muss zuerst geladen werden. Es stellt die Import-Funktionen bereit:
load("nashorn:mozilla_compat.js“);
// Import für Klassen und Packages
importClass(java.nio.file.Files);
importClass(java.nio.file.Paths);
importPackage(java.lang); // ACHTUNG: Hier gibt es Namenskollisionen!

var path = Paths.get("/data/daten.csv");
// ...
```

Listing 5: Vereinfachter Import für Klassen und Packages

```
var importer = JavaImporter(
            java.nio.file,
            javafx.scene.control,
            javafx.scene.layout,
            javafx.scene.chart.PieChart,
            javafx.scene.chart.PieChart.Data,
            javafx.scene.Scene,
            javafx.collections
        );

function start(primaryStage) {
  with (importer) { // Wichtig!

      primaryStage.title = "Hello PieChart!";
      var button = new Button();
      button.text = "Datei laden ...";
      button.onAction = function () {
         with (importer) { // Wichtig!
            var pieChartData = FXCollections.observableArrayList();
            var path = Paths.get(„/data/daten.csv“);
            var lines = Files.readAllLines(path);
            var n = 1;
            lines.stream().forEach(function (line) {
               pieChartData.add(new PieChart.Data("Zeile " + n + ":\n" + line.length(), line.length()));
               n++;
            });
            chart.setData(pieChartData);
         }
      };

      var root = new VBox();
      var chart = new PieChart();
      root.children.addAll(button, chart);
      primaryStage.scene = new Scene(root, 300, 250);
      primaryStage.show();
  }
}
```

Listing 6: Import mit „JavaImporter“, begrenzt auf engere Scopes

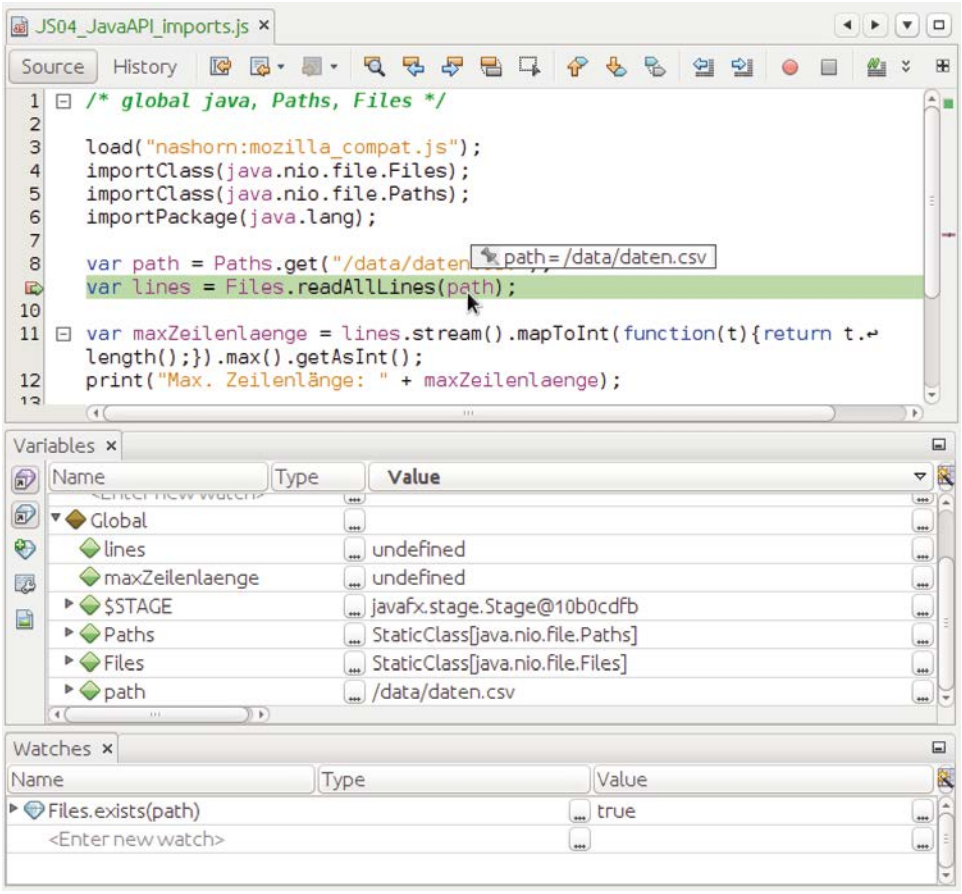


Abbildung 2: Debugging von JavaScript-Code mit NetBeans

Debugging mit Nashorn? Kein Problem

Die Fehlersuche mit klassischem Debugging funktioniert mit geeigneten IDEs wie NetBeans oder IntelliJ IDEA im Wesentlichen so wie in Java-Anwendungen. So kann bei Ausführung des JavaScript-Codes aus der IDE heraus auch wie gewohnt mit Break-Points und Debugging-Tools gearbeitet werden. *Abbildung 2* zeigt dies am Beispiel von NetBeans – hier hat man am Haltepunkt unter anderem Zugriff auf lokale Variablen, den globalen Scope, den Call Stack und auf überwachbare Ausdrücke.

Für tiefere Einblicke in die Arbeit von Nashorn ist es übrigens gut zu wissen, dass der Nashorn-Launcher „jjs“ mit der Option „-help“ zwar die wichtigsten (offiziellen) Kommandozeilen-Optionen beschreibt; der undokumentierten Aufruf „jjs -xhelp“ bringt dagegen die volle Palette zum Vorschein. So findet man hier nicht nur einige Optionen für die Erzeugung von Debug-Informationen, son-

dern auch für die Ausgabe des erzeugten Java-Bytecodes an der Konsole *(siehe Listing 7)*.

Das Java-Scripting-API

Bis hierher war nur die Ausführung von JavaScript in der Nashorn-Umgebung das Thema. Mit dem Java-Scripting-API kann die Nashorn-Engine dagegen auch aus Java-Anwendungen heraus genutzt werden. Dies erlaubt nicht nur die Integration von JavaScript-Code in Java-Anwendungen, sondern eröffnet auch zusätzliche Möglichkeiten. *Listing 8* zeigt zunächst den ersten Schritt, das Laden des Codes. Das Beispiel lässt sich gut mit dem Code von *Listing 2* in einer Datei „JS_Beispielcode.js" ausführen. Gibt man in dem Beispiel die verfügbaren JavaScript-Engines der JVM aus, fällt übrigens auf, dass der Name „Oracle Nashorn" lautet. Auch der Name „nashorn" kann für den „getEngineByName()"-Aufruf verwendet werden.

Für das Zusammenspiel von Java und JavaScript bietet das Java-Scripting-API drei wichtige Möglichkeiten: Zum einen können Java-Objekte und -Variablen in den Scope der ScriptEngine eingefügt werden. Zum anderen gibt der „engine.eval()"-Aufruf gegebenenfalls Objekte und Variablen aus dem Scope der ScriptEngine heraus. Außerdem lassen sich aus Java heraus Funktionen im Scope der ScriptEngine aufrufen. Dies erlaubt eine enge Verzahnung von Java- und JavaScript-Logik. Das Beispiel in *Listing 9* zeigt diese Möglichkeiten anhand einer fiktiven Business-Logik-Klasse „JavaLogic". Diese kann mit der Methode „engine.put()" in den JavaScript-Scope eingefügt und an einen Variablennamen gebunden werden.

Die „engine.get(…)"-Methode liest übrigens auch Variablen aus dem Engine-Scope heraus. Zudem ist in der Beispiel-Anwendung JavaScript-Code definiert, der Methoden auf der „JavaLogic"-Instanz aufruft. Um eine JavaScript-Funktion aus Java heraus aufrufen zu können, muss die Engine zunächst in eine aufrufbare „Invocable"-Engine gecastet werden. Hat der Rückgabe-Wert einen primitiven Typ, wird er in einen entsprechenden Java-Typ umgewandelt. Das gilt etwa für Strings. Das Beispiel zeigt jedoch, wie mit komplexen Rückgabe-Werten gearbeitet werden kann. JavaScript-Objekte lassen sich als Instanz von „jdk.nashorn.api.scripting.JSObject" ansprechen. Über dieses Interface ist auch der Zugriff auf die Eigenschaften und Methoden des Objekts möglich.

In der „JavaFX-WebView" gelten andere Rahmenbedingungen: Intern wird die WebKit Browser Engine *[6]* verwendet, die unter anderem auch in Apples Browser Safari zum Einsatz kommt. Das bedeutet für JavaFX gutes HTML-Rendering – aber auch einen eigenen, zweiten JavaScript-Interpreter. Aus verschiedenen Gründen konnte WebKits-JavaScript-Interpreter JavaScriptCore nicht ohne Weiteres gegen Nashorn ersetzt werden *[7]*. Man hätte damit rechnen müssen, gegebenenfalls einen eigenen Fork von WebKit pflegen zu müssen. Zudem implementiert die Klasse „javafx.scene.web.WebEngine" nicht dasselbe Interface „ScriptEngine".

Die WebView-Komponente eignet sich jedoch hervorragend zum Einbetten von HTML- und JavaScript-Inhalten. So können auch komplexere JavaScript-Anwendungen wie der Sourcecode-Editor „Ace" *[8]* oder Google Maps in Java-Anwendungen eingebettet werden.

Fazit

Mit den hier umrissenen Möglichkeiten kann JavaScript in der JVM zum einen sehr leicht sämtliche JRE-Bibliotheken für Scripting-

BROCKHAUS AG

SOFTWAREENTWICKLER - JAVA (M/W)

FESTANSTELLUNG, IN VOLLZEIT

IT-Beratung und Softwareentwicklung sind unsere Leidenschaft. Mit gut 30 Jahren Erfahrung sind wir als mittelständisches Unternehmen erfolgreich unterwegs und suchen zur Verstärkung unseres Teams einen JAVA-Entwickler.

IHRE HERAUSFORDERUNG

- Zu Ihren Hauptaufgaben gehört die Konzeption und Entwicklung von kundenspezifischen Softwarelösungen mit JAVA in einer JEE-Architektur, unter Verwendung von Technologien wie Spring, Hibernte, und objektorientierter Methoden (OOA, OOD, UML, OOP).
- Sie unterstützen das Team bei der Entwicklung von komplexen Schnittstellen, bei der Integration neuer Komponenten in bestehende Architekturen sowie bei der Weiterentwicklung der bestehenden Software.
- Die Integration neuer Komponenten in bestehende Architekturen sowie in unterschiedlichsten IT-Umgebungen
- Sie arbeiten in einem agilen Team und verwenden Entwicklungswerkzeuge und Laufzeitumgebungen, wie z. B. Eclipse, Maven, Apache Tomcat.

WANTED: INNOVATIVER TEAMPLAYER

BROCKHAUS AG | Pierbusch 17 | 44536 Lünen | +49 231 98 75 - 702
bewerbung@brockhaus-ag.de | www.brockhaus-ag.de/career

```
  --debug-lines (Generate line number table in .class files.)
     param: [true|false] default: true

  --debug-locals (Generate local variable table in .class files.)
     param: [true|false] default: false

  --debug-scopes (Put all variables in scopes to make them debuggable.)
     param: [true|false] default: false

  -doe, -dump-on-error (Dump a stack trace on errors.)
     param: [true|false] default: false
(…)
  -pc, --print-code (Print generated bytecode. If a directory is specified, nothing will
            be dumped to stderr. Also, in that case, .dot files will be generated
            for all functions or for the function with the specified name only.)
     param: [dir:<output-dir>,function:<name>]
(…)
  --print-mem-usage (Print memory usage of IR after each compile stage.)
     param: [true|false] default: false
```

Listing 7: Der Aufruf „jjs -xhelp" zeigt eine ganze Reihe interessanter Optionen

```
// …
public class JavaJSAnwendung1 {

  public static void main(String[] args) throws ScriptException, FileNotFoundException {

     // Ausgangspunkt: Der ScriptEngineManager
     ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

     // Zur Information: Ausgabe der Verfügbaren JS-Engines
     scriptEngineManager.getEngineFactories()
        .stream()
        .forEach((t) -> System.out.println(t.getEngineName()));

     ScriptEngine engine = scriptEngineManager.getEngineByName("JavaScript");
     // Direkte Übergabe von JavaScript-Code als String:
     engine.eval("print(\"Let's go!\");");
     // Oder besser: Einlesen des JavaScript-Codes aus einer Datei:
     engine.eval(new FileReader("/pfad/zu/JS_Beispielcode.js"));
  }
}
```

Listing 8: Ausführung von JavaScript mit Nashorn in einer Java-Anwendung

```
public class JavaJSAnwendung2 {

  /**
     * Java-Klasse, z.B. Business-Logik
     */
  public static class JavaLogic {

     public String doBusiness(String input) {
        return input + "done.";
     }
  }

  public static void main(String[] args) throws ScriptException, NoSuchMethodException {

     ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
     ScriptEngine engine = scriptEngineManager.getEngineByName("JavaScript");

     // Einfügen einer Instanz in den JavaScript-Scope
     engine.put("javalogic", new JavaLogic());
     // JavaScript-Logik, ruft Methode auf Java-Objekt auf und gibt ein Ergebnis-Objekt zurück
     engine.eval("function doWork(input) { "
           + " return {"
           + "          value: javalogic.doBusiness(input),"
           + "          status: \"OK\" "
           + "        };"
           + "}; ");

     // Aufruf einer JS-Funktion aus Java heraus:
     Invocable invocableEngine = (Invocable) engine;

     // Auswertung der Rückgabe als jdk.nashorn.api.scripting.JSObject:
     JSObject result = (JSObject) invocableEngine.invokeFunction("doWork", "lot of work ");
     System.out.println(result.getMember("value"));
  }
}
```

Listing 9: Gegenseitiger Aufruf von Java und JavaScript

Aufgabe nutzbar machen. Zum anderen bietet JavaScript als dynamische Sprache auch die Chance, dynamisches Verhalten innerhalb einer Java-Anwendung zu realisieren – auch wenn man neben den Chancen die Risiken sorgfältig diskutieren muss.

Neben der direkten Nutzung von JavaScript in der JVM mit Nashorn gibt es übrigens noch eine ganze Reihe weiterer interessanter Möglichkeiten, wie das Zusammenspiel von Java und JavaScript ein schlüssiges Gesamtbild ergibt – oder aber für bestimmte Architektur-Stile. Man denke nur an das WebSockets-API für die Kommunikation von JEE-Anwendungen und JavaScript-Clients, an AJAX- und REST-Kommunikation oder an Thin-Server-Architecture. Es ergibt also durchaus Sinn, sich als Java-Entwickler auch mit JavaScript zu beschäftigen.

Quellen

[1] http://crockford.com/javascript/javascript.html
[2] Oracle JDK 9 Documentation: *https://docs.oracle.com/javase/9/tools/jjs.htm*
[3] Oracle Java Platform, Standard Edition Java Scripting Programmer's Guide: *https://docs.oracle.com/javase/9/scripting/java-scripting-api.htm*
[4] Mozilla Developer Network, JavaScript Reference: *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array*
[5] Oracle Java Platform, Standard Edition Java Scripting Programmer's Guide: *https://docs.oracle.com/javase/9/scripting/using-java-scripts.htm*
[6] *https://webkit.org/*
[7] E-Mail von Richard Bair vom 9. Mai2013: *http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-May/007597.html*
[8] *https://ace.c9.io*

Robert Rohm
info@aeonium-systems.de

Robert Rohm arbeitet als selbständiger Entwickler, Berater und Trainer in Nürnberg. Mit einem starken Fokus auf der Architektur und Full-Stack-Entwicklung von mehrschichtigen Anwendungen steht er mit einem Bein im Frontend und mit dem anderen Bein im Backend. Als Software-Ingenieur (M. Eng.) und Lehrer ist ihm die Wissensarbeit und Wissensvermittlung nicht nur ein berufliches Anliegen, sondern auch eine persönliche Leidenschaft.

Coden von der Dachterrasse – in Rumänien

Steven Schwenke, msg DAVID

Verteiltes Arbeiten ist an sich nichts Neues, wird jedoch oft verhindert durch viele Missverständnisse und offene Fragen. Die Ursachen dafür liegen in fehlender Schulung der Beteiligten und in der Anpassung der Prozesse. Dieser Artikel gibt einen Überblick darüber, wie sich Entwicklungsarbeit in Nearshore verlagert, wie wir remote viel glücklicher arbeiten können, worauf man bei der Einführung verteilten Arbeitens achten muss und wie man die Beteiligten schulen kann.

Täglich acht Stunden auf der Projektfläche, eine kurze Mittagspause in einer schlechten Kantine, lieblos umrahmt von zwei Stunden Autofahrt, natürlich zur selben Zeit wie alle anderen auch. Diesen Tagesablauf kennt man, einige nennen ihn „Day Prison". Auch der Autor hat viele Jahre einen solchen Ablauf durchlebt. Ein neues Projekt brachte eine Kooperation mit rumänischen Kollegen und eine neue Rolle als TechLead. Beides führte zu wesentlich mehr Kommunikation mit Teammitgliedern und Architekten aus anderen Teams.

Gemeinsames Konzeptionieren, schneller Austausch sowie viele kleine Fragen und Gespräche bestimmen seitdem den Alltag. Entgegen der landläufigen Meinung, dass dies nur in einem gemeinsamen Büro möglich sei, erlebt der Autor tagtäglich das Gegenteil. Spätestens die vielen Skype-Gespräche würden in einem Großraumbüro einfach nicht funktionieren. Doch der Reihe nach. Was ist eigentlich „remote working" oder „verteiltes Arbeiten"?

Verteiltes Arbeiten hat viele Gesichter

Die Erfahrung hat gezeigt, dass verteiltes Arbeiten bereits dann anfängt, wenn Kollegen in einem anderen Stockwerk sitzen. Oft ist es in diesem Fall einfacher, eine Mail oder Kurznachricht zu schreiben, als sich selbst dorthin zu bewegen. Wenn die Kommunikation aufgrund dieser Raumsituation ins Digitale abdriftet, ist es praktisch auch egal, ob der Kollege im selben Gebäude sitzt, im Café um die Ecke oder eben in einem anderen Land. Das Ergebnis ist dasselbe: Schnelle Zwischendurch-Kommunikation wird digital abgehandelt, längere Treffen müssen organisiert werden.

Diese Verschiebung ins verteilte Arbeiten führt oft zu einem wesentlich höheren Freiheitsgrad. Sind die technischen und sozialen

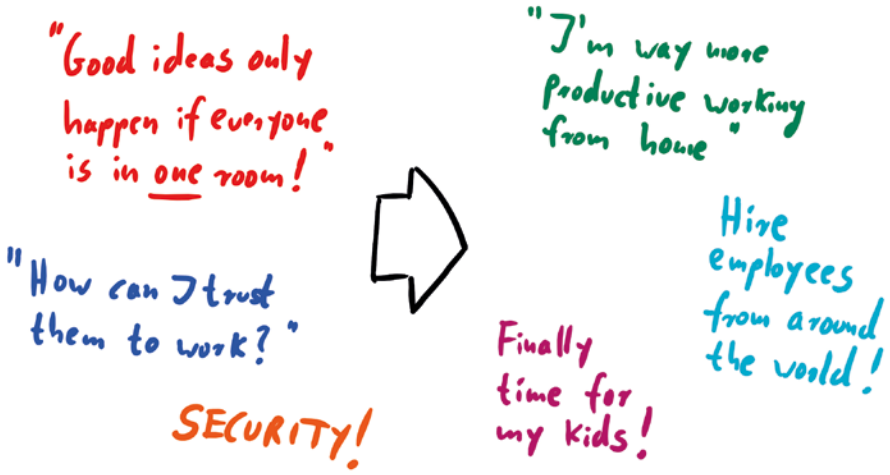


Abbildung 1: Es gibt viele Vorurteile und Annahmen, die den positiven Möglichkeiten entgegenstehen

Herausforderungen gemeistert und das Team eingespielt, kann jeder arbeiten, wo er möchte. Das wiederum führt oft auch zu zeitlicher Freiheit: Es kann gearbeitet werden, wann man möchte, solange man die gemeinsamen Termine einhält.

Die Kombination aus lokaler und zeitlicher Freiheit kann motivierte Arbeitnehmer dazu bewegen, dem Arbeitgeber die Premium-Stunden des Tages zu geben: Kann man sich nach den ersten Aufgaben nicht mehr konzentrieren, geht man in einer längeren Pause joggen. Danach ist man wieder frisch und geht gestärkt ans Werk. Am Nachmittag den Rasen mähen, wenn man von zuhause arbeitet, entspannt und bringt Zeit zum Nachsinnen über die aktuelle Aufgabe. Im Kontrast dazu sitzt man die Tiefphasen gern mal aus, wenn man in einem Single-Site-Team im gemeinsamen Büro arbeitet. Das bringt weder dem Projekt Fortschritt noch dem Arbeitnehmer ein gutes Gefühl – es kostet einfach nur Energie und Geld *(siehe Abbildung 1)*.

Natürlich ist das gerade beschriebene Szenario eine optimale, heile Welt. Aber sie ist möglich und kann funktionieren. Dafür ist eine aktive Auseinandersetzung mit den eigenen Prozessen, Tools und Mitarbeitern sowie in der Vertrauensbasis zwischen diesen notwendig.

Verteiltes Arbeiten will gelernt sein

Die erfolgreiche Einführung verteilten Arbeitens betrifft viele Bereiche *(siehe Abbildung 2)*:

- Mitarbeiter
- Soziales Umfeld des Mitarbeiters
- Häusliches Umfeld des Mitarbeiters
- Projektteam
- Firmeninterne Prozesse
- Kunde
- Unternehmen

Abbildung 2: Verteilt zu arbeiten erfordert Anpassung nicht nur bei sich selbst, sondern auch im Umfeld

Der wichtigste Faktor ist man selbst

Das Wichtigste gleich vorweg: Verteiltes Arbeiten funktioniert nicht für jeden Menschen, nicht für jede Rolle, nicht für jedes Team – und es führt nicht ohne Weiteres zu schnellerer Arbeit. Jeder muss für sich entscheiden, ob er zuhause arbeiten kann oder nicht. Falls man diesen Schritt geht, sollte man verstärkt auf sportliche Aktivitäten achten, da man sich sonst den ganzen Tag im eigenen Haus aufhalten würde und entsprechend weniger Bewegung bekommt.

Auch psychische Aspekte sind zu berücksichtigen: Der alte Witz des Entwicklers, der ohne Hosen aber mit gutem Hemd im Video-Chat sitzt, ist teilweise wahr. Sich zuhause gehen zu lassen, führt zu Unzufriedenheit. Besser ist es, sich wie üblich für die Arbeit zu kleiden, um den Arbeitsmodus im Kopf einzuschalten.

Das soziale Umfeld aktiv gesund halten

Ein nicht zu unterschätzender psychischer Faktor ist die drohende Vereinsamung: Wer nur zuhause arbeitet, hat durch das fehlende Pendeln zwar mehr Zeit – doch wie man diese gewonnene Zeit verwendet, ist ausschlaggebend. Eine gute Investition sind soziale Interaktionen wie Treffen mit Freunden, Spazierengehen mit der Familie oder neue Hobbies in Vereinen. Gute Kombination: Mannschafts-Sportarten. Die soziale Berührungsfläche mit Kollegen sollte ebenfalls gestärkt werden.

Der Autor organisiert anstehende Termine so, dass Bürotage mit Anwesenheit in der Firma möglichst gut ausgenutzt werden. Lücken zwischen diesen Terminen werden aktiv für spontane Gespräche genutzt. So wurden schon viele Probleme quasi nebenbei gelöst. Zusätzlich ist der Autor jeden Freitag zum Spiele-Nachmittag in der Firma, um mit Kollegen Freizeit zu verbringen.

Nur ein gutes häusliches Büro ermöglicht professionelles Arbeiten

Viele Entwickler möchten zuhause arbeiten, haben dort aber keine echte Arbeitsumgebung. Aus verschiedenen Gründen sollte ein eigener Raum als Büro dienen und es sollte hochwertige Ausstattung vorhanden sein. Das ist eine der für Remote Worker negativen Seiten: Wer zuhause arbeiten möchte, muss oft erst einmal investieren. Auch die Situation hinter dem eigenen Schreibtisch sollte professionell wirken, damit der Kunde im Video-Stream nicht unbedingt Einblick in die unaufgeräumte Wohnung erhält. Die einfachste Lösung ist hier die (saubere und Poster-lose) weiße Wand. Die während einer Besprechung über den Schreibtisch laufende Katze bringt zwar dem eigenen Team ein Lächeln und Gesprächsstoff, aber nicht jeder Kunde sieht das so positiv.

Die Kommunikation an den Rest der Familie ist entscheidend. Wer am heimischen Schreibtisch arbeitet, ist trotzdem nicht wirklich zuhause und jederzeit bedingungslos ansprechbar. Das sollte sehr klar kommuniziert werden. Hilfestellung bieten vereinbarte Signale wie Schilder an der Tür zum Heim-Büro oder vereinbarte Zeiten für gemeinsame Aktivitäten und „Bitte nicht stören"-Phasen.

Das Team muss „remote können" – mit allen Folgen!

Wird international gearbeitet, spielen Sprachkenntnisse eine entscheidende Rolle. Als ratsam hat sich sowohl eine Standard-Sprache als auch -Zeit herausgestellt. So muss nicht wiederholt gefragt werden, in welcher Sprache eine Ausarbeitung geschehen soll oder für welche Zeitzone eine zeitliche Angabe gilt. Die Sprache muss außerhalb des normalen Projekt-Alltags geschult werden, um Fehler in der Kommunikation zu vermeiden. In diesen Schulungen sind auch umgangssprachliche Phrasen zu behandeln, damit man zum einen näher am alltäglichen

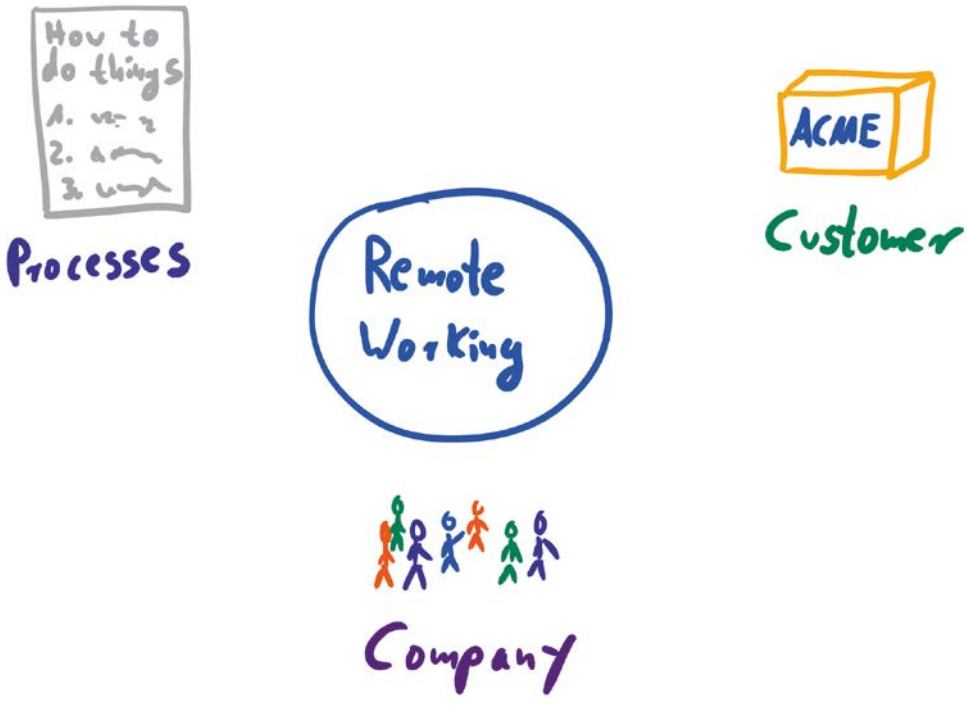


Abbildung 3: Auch die am Projekt beteiligten Stakeholder und Prozesse sind anzupassen

Sprachniveau arbeiten und zum anderen auch in lockeren Situationen souverän sein kann. Akzente und sprachliche Besonderheiten müssen offen angesprochen werden, um durch gezielte Schulungen ein hohes und von allen verständliches Sprachniveau zu erreichen. Es bringt wenig, wenn der durch amerikanische Filme geprägte, Hollywood-Englisch sprechende Inder sich mit dem auf Schul-Englisch-Niveau sprechenden Deutschen über komplexe Sachverhalte unterhält.

Doch nicht nur das gesprochene Wort ist Kommunikation, auch das Schreiben will gelernt sein. In Teams mit verschiedenen Zeitzonen spielen schriftliche Dokumente (im weiteren Sinne von User Stories, Fehlerbeschreibungen, Kommentaren für Code Reviews etc.) eine große Rolle. Je höher sowohl das Leseverständnis als auch die Fähigkeit des kompakten Schreibens ausgeprägt sind, desto leichter fällt der Wissenstransfer *(siehe Abbildung 3)*.

Prozesse sollten Arbeit unterstützen, nicht verhindern

Firmeninterne Prozesse sollten ebenfalls an das neue Setting angepasst sein, genauso wie der Software-Entwicklungsprozess selbst. Schon beim Aufsetzen eines neuen Projektteams sollte zwingend ein gemeinsames, physisches Treffen des gesamten Teams fest eingeplant und eingepreist werden. „Invest in beginnings" bedeutet, gerade zu Beginn ein Team stark aufzubauen und Grundregeln festzuziehen.

Einige Tage in dem Land, in dem der Großteil des Teams sitzt, inklusive jeder Menge gemeinsam verbrachter Freizeit, bauen soziale Brücken, die bei der Lösung späterer Probleme hilfreich sind. Bei so einer Zusammenkunft sollten auch gleich Grundregeln für das gemeinsame Miteinander festgelegt werden, und zwar durch das Team selbst. Spätestens bei jeder personellen Veränderung, mindestens aber zweimal im Jahr, sollte ein gemeinsamer Workshop wiederholt werden. Ein schönes Zitat einer Entwicklerin aus dem Team des Autors lautet: „You don't remember Skype-meetings. But you remember when the team met in real-life."

Als momentan sehr beliebte Entwicklungsmethodik spielt Agile in Verbindung mit verteiltem Arbeiten eine besondere Rolle. Ob Scrum oder Kanban – agile Methoden eignen sich hervorragend für verteilte Teams. Ein wichtiger Punkt ist hier die maximale Zeit, nach der auftretende Probleme auf jeden Fall besprochen werden. In einem gemeinsamen Büro bemerkt man, ob ein Teammitglied den ganzen Tag gefrustet vor einer Aufgabe sitzt, nicht weiterkommt und sich lauthals beschwert. Remote funktioniert das nicht.

In einem agilen Umfeld erfolgt spätestens nach 24 Stunden ein Treffen, in dem jeder Entwickler Probleme melden und Hilfe einholen kann. Selbst wenn dies, zum Beispiel von einem Junior-Entwickler, nicht getan wird, sollte es einem gut ausgebildeten Team mit einem aufmerksamen Scrum-Master auffallen, wenn der

DW

Made for minds.

freedom++

/* Ein unschlagbares Team: Deine Java-Skills und unsere Informationen gegen Fake News und Zensur – weltweit. Entwickle beim deutschen Auslandssender DW die digitalen Medien von morgen.

Informiere und bewirb dich auf dw.com/java-karriere oder triff uns auf der Javaland.

*/

junge Entwickler an zwei Tagen hintereinander keinen Fortschritt melden kann. Auch das Schneiden kleiner Aufgabenpakete und natürlich der Grundgedanke selbstorganisierender Teams unterstützen verteiltes Arbeiten.

Den Kunden aktiv dabeihaben und Verständnis schaffen

Der Kunde kann sehr unterschiedlich auf das Konzept verteilten Arbeitens reagieren. Es gibt Unternehmen, die zum Beispiel wegen des Kostendrucks Offshore-Teams erzwingen. Im Fachbereich kann es dennoch in der täglichen Arbeit zu Problemen kommen, wenn man es gewöhnt ist, die Entwickler im nächsten Büro zu finden und entsprechend starken Einblick in die tägliche Entwicklungsarbeit zu haben.

„Seeing is believing" ist ein guter Ansatz, um Problemen vorzubeugen. Im Projekt des Autors ist der Kunde zu den Daily Stand-ups eingeladen. Es wurde kommuniziert, dass dieser Termin weiterhin dem Team gehört und der Kunde dort Gast ist. Ziel ist eine möglichst große Sichtbarkeit des Teams mit seinen aktuellen Herausforderungen. Smalltalk vor und nach dem Daily dient auch dazu, kulturelle Besonderheiten wie Feiertage oder Feste anzusprechen und somit ganz nebenbei ein besseres Verständnis voneinander zu bekommen.

Im Unternehmen remote wagen und Chancen nutzen

Alle hier vorgeschlagenen Praktiken können nicht umgesetzt werden, wenn das eigene Unternehmen dies verhindert. Oft existiert eine falsche Vorstellung davon, was verteiltes Arbeiten mit sich bringt – sowohl an zusätzlichen Aufgaben und Kosten als auch an Chancen.

So ziemlich alle IT-Unternehmen suchen händeringend Mitarbeiter, finden sie aber nicht ausreichend und/oder nicht in ausreichender Qualität am gesuchten Standort. Erweitert man den eigenen Standort aber auf Landes- oder Europa-Ebene oder sogar weltweit, steigt die Wahrscheinlichkeit für das Finden wirklich guter Arbeitnehmer stark an. Dazu sind gleich mehrere Prozesse und Denkweisen zu ändern. Zum einen funktioniert das Marketing nicht mehr nur mit lokalen Stellenanzeigen oder dem Aushändigen von Giveaways auf lokalen Konferenzen.

Das Unternehmen muss in der ganzen Region sichtbar sein, in der es Mitarbeiter finden möchte. Das Sponsoring großer Konferenzen oder Schreiben von Artikeln in Fachzeitschriften ist da ein erster Schritt. Ziel dieser Maßnahmen sollte sein, dass potenzielle Bewerber regelmäßig darüber informiert werden, wie interessant die Arbeit dort ist, welche neuen Technologien eingesetzt und welche innovativen Projekte bearbeitet werden.

„Tu Gutes und sprich darüber" führt dazu, dass zum Beispiel durch das Unternehmen unterstützte Open-Source-Projekte in der Entwickler-Community gesehen werden, was wiederum Interesse erzeugt. Auch die Durchführung eigener und für Entwickler sinnstiftender Veranstaltungen erzeugt Sichtbarkeit und nach außen gestrahlte Professionalität. Werden diese Veranstaltungen noch live gestreamt oder als Webinar angeboten, wird die Nähe zu potenziellen Bewerbern aus anderen Regionen nochmal verringert. All diese Maßnahmen folgen dem Prinzip, die Chancen im verteilten Arbeiten zu sehen und zu nutzen. Es gibt eine Menge Gründe für Unternehmen, nicht verteilt zu arbeiten. Oft werden Vorurteile oder nicht ausreichende Information über das Thema mit diesen guten Gründen verwechselt.

Fazit

Damit verteiltes Arbeiten erfolgreich ist, müssen viele Aspekte berücksichtigt werden: Selbstdisziplin, Selbstmanagement und das Schaffen eines sozialen und sportlichen Ausgleichs sind die Basis für jeden, der von Zuhause und generell woanders arbeiten möchte. Ist das eigene Heim das neue Büro, sollte ein abgetrennter Arbeitsplatz eingerichtet werden. No-Go-Zones und -Zeiten für die Familie schaffen eine ruhige Arbeits-Atmosphäre.

Selbstorganisierte agile Teams sind ein fruchtbarer Nährboden für verteiltes Arbeiten und umgehen viele der auftretenden organisatorischen und sozialen Probleme wie fehlende Hilfegesuche, zu wenig Kommunikation und fehlende Identifikation mit dem Projekt und den Kollegen. Das eigene Unternehmen muss Finanzierung und Zeiten bereitstellen, damit sich virtuelle Teams regelmäßig an einem Ort austauschen und Zeit miteinander verbringen können.

Auch der Austausch zwischen dem Kunden und dem ganzen Team erhöht das gegenseitige Verständnis und hilft, aufkommende Probleme miteinander zu lösen, anstatt Aufgaben hin und her zu schieben. Trotz allem funktioniert verteiltes Arbeiten nicht für alle Rollen, Menschen und Teams. Die Erfolgswahrscheinlichkeit kann mit gezielten Schulungen und schnell durchgeführten Maßnahmen jedoch wesentlich erhöht werden. Die nachfolgenden Ausgaben der Java aktuell beleuchten zusätzliche Aspekte des verteilten Arbeitens. Weitere Informationen unter *„https://github.com/msg-DAVID-GmbH/RemoteWorking".*

Steven Schwenke
steven.schwenke@msg-david.de

Steven Schwenke ist Software Craftsman und liebt, was er tut. Als Technical Teamlead leitet er ein verteiltes Team und führt regelmäßig Inhouse-Workshops durch. Zusätzlich organisiert er Veranstaltungen wie den HackTalk und hält Vorträge bei Konferenzen. Er teilt seine Erfahrung gern mit anderen und freut sich auf spannende Gespräche.

Jenkins

Coding Continuous Delivery – Performance-Optimierung für die Jenkins-Pipeline

Johannes Schnatterer, Triology GmbH

Nachdem in der letzten Ausgabe [1] die Grundbegriffe und eine erste Jenkins-Pipeline beschrieben wurden, zeigt dieser Artikel wie man mit Parallelisierung und Nightly Builds die Laufzeit der Pipelines verkürzen und damit schnelleres Feedback erhalten kann.

Im Folgenden werden die Pipeline-Beispiele aus der letzten Ausgabe sukzessive erweitert, um die Features der Pipeline zu zeigen. Dabei sind die Änderungen jeweils in „declarative"- und in „scripted"-Syntax realisiert. Der aktuelle Stand jeder Erweiterung lässt sich bei GitHub *[2]* nachverfolgen und ausprobieren. Hier gibt es für jeden Abschnitt unter der in der Überschrift genannten Nummer jeweils für „declarative" und „scripted" einen Branch, der das vollständige Beispiel umfasst. Das Ergebnis der Builds jedes Branch lässt sich außerdem direkt auf unserer Jenkins-Instanz *[3]* einsehen.

Wie im ersten Teil sind auch hier die Features des Jenkins-Pipeline-Plug-ins anhand eines typischen Java-Projekts gezeigt. Als Beispiel dient damit auch diesmal der „kitchensink"-Quickstart von WildFly. In der letzten Ausgabe wurden mit simpler Pipeline, eigenen Steps, Stages, Fehlerbehandlung und Properties/Archivierung fünf Beispiele gezeigt.

Parallelisierung

Dank des „parallel"-Steps ist es sehr einfach, Steps oder Stages nebenläufig auszuführen und so die Gesamtlaufzeit der Pipeline zu verkürzen. Am einfachsten ist es, dabei die Schritte parallel auf einem Node auszuführen. Dabei muss jedoch bedacht werden, dass die nebenläufigen Ausführungen im selben Verzeichnis laufen und sie sich dadurch unerwartet gegenseitig beeinflussen könnten. Beispielweise löschen zwei nebenläufig ausgeführte Maven Builds mit „clean"-Phase dasselbe „target"-Verzeichnis, wodurch ein Build wahrscheinlich scheitert.

Alternativ kann man auch mit mehreren Nodes parallelisieren, also mehrere Jenkins Build Executors belegen. Dies verursacht jedoch deutlich mehr Overhead. Es muss dann auf jedem Node der Workspace neu angelegt werden – also Git Clone, gegebenenfalls Maven Dependencies geladen werden etc. Daher ist die Parallelisierung innerhalb eines Node in den meisten Fällen die erste Wahl.

An dieser Stelle eine kleine Begebenheit aus der Praxis zur Motivation für die Parallelisierung: In einem kleineren Projekt (etwa zehn KLOC) mit hoher Testabdeckung (rund 80 Prozent) konnte die Build-Dauer von elf auf acht Minuten verkürzt werden, indem die Unit- und Integrationstest-Stages parallel ausgeführt wurden. Darüber hinaus ließ sich das ursprünglich sequenzielle Deployment von zwei Artefakten durch Parallelisierung von vier auf zweieinhalb Minuten verkürzen. Es lohnt sich also, den „parallel"-Step im Hinterkopf zu behalten, um mit wenig Aufwand schnelleres Feedback zu bekommen. Das Beispiel in *Listing 1* zeigt in „scripted"-Syntax, wie man die Unit- und Integrationstest-Stages gleichzeitig ausführt.

Dem „parallel"-Step wird eine Map übergeben, in der man die verschiedenen Ausführungszweige mit Namen versehen kann und in einem Closure definiert. Anhand des Namens können dann bei der Ausführung die Ausgaben im Log den jeweiligen Zweigen zugeordnet werden. Hier spielt das schon im ersten Teil erwähnte Blue-Ocean-Plug-in seine Stärken aus: Statt wie in der klassischen Ansicht die Pipeline in einer Reihe anzuzeigen , werden gleichzeitig ausgeführte Zweige untereinander dargestellt *(siehe Abbildung 1)*. Außerdem kann man sich die Konsolen-Ausgabe der jeweiligen Branches isoliert ansehen und muss nicht wie in der klassischen Ansicht anhand des in der Pipeline vergebenen Namens unterscheiden *(siehe Listing 2)*.

Der „parallel"-Step kann auch in der „declarative"-Syntax verwendet werden. *Listing 3* zeigt, wie sich dies über geschachtelte Stages abbil-

den lässt. Für komplexere Szenarien sind zur Synchronisierung mehrerer gleichzeitiger Builds Locks *[4]* und/oder das Milestone-Plug-in *[5]* möglich. Unter *[6]* steht ein Beispiel, das beides verwendet.

Nightly Builds

Viele Teams lassen ihren CI-Server lang laufende oder regelmäßig aufzuführende Aufgaben einmal am Tag erledigen, typischerweise über Nacht. Ein Beispiel dafür ist die Überprüfung ihrer Dependencies auf bekannte Sicherheitslücken *[7]*. Diese Nightly Builds sind auch in einer Jenkins-Pipeline möglich. Einen Build regelmäßig auszuführen, ist dabei sehr einfach. *Listing 4* zeigt, wie dies in der „scripted"-Syntax über die im ersten Artikel beschriebenen Properties angegeben wird.

Bei der Ausführung des Jenkins-Files werden dann die angegebenen Ausführungen von Jenkins eingeplant. In *Listing 4* ist zu sehen, dass – wie in einer an Cron-Jobs angelehnten Syntax – ein Schedule angegeben werden kann:

- Minute (0-59)
- Stunde (0-23)
- Tag des Monats (1-31)
- Monat (1-12)
- Tag der Woche (0-7), 0 und 7 sind Sonntag

Der Asterisk (*) steht für jeden validen Wert; in *Listing 4* bedeutet das beispielsweise „jeden Tag in jedem Monat". Das „H" steht für den Hash des Job-Namens. Dabei wird ein Zahlenwert auf Basis des Hashwerts des Job-Namens generiert. Dies führt dazu, dass nicht alle Jobs mit gleichem Schedule zu Lastspitzen führen. So würde „0 0 * * 1-5" dazu führen, dass an jedem Werktag um Punkt 0 Uhr alle Jobs gleichzeitig starten. „H H(0-3) * * 1-5" hingegen verteilt diese Last auf die Zeit zwischen 0 und 3 Uhr. Diese Best Practice sollte so oft wie möglich zum Einsatz kommen.

Wem das zu kompliziert ist, dem stehen die Aliase „@yearly", „@annually", „@monthly", „@weekly", „@daily", „@midnight" und „@hourly" zur Verfügung. Auch diese nutzen zur Lastverteilung das oben beschriebene Hash-System. Beispielsweise bedeutet „@midnight" konkret die Zeit zwischen 0 und 2:59 Uhr. Auf die gleiche Weise spezifiziert man den Schedule auch in der „declarative"-Syntax *(siehe Listing 5)*, allerdings wird diese hier in einer eigenen „triggers"-Directive angegeben.

Die größere Herausforderung liegt allerdings darin zu entscheiden, wo man die nächtlich auszuführende Logik beschreibt. Dafür bieten sich zwei Möglichkeiten an: Man legt ein weiteres Jenkins-File im Repository an, beispielsweise „Jenkinsfile-nightly". Außerdem kreiert man einen neuen Pipeline- oder Multibranch-Pipeline-Job in

```
parallel(
  unitTest: {
     stage('Unit Test') {
        mvn 'test'
     }
  },
  integrationTest: {
        stage('Integration Test') {
           mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
        }
     }
)
```

Listing 1

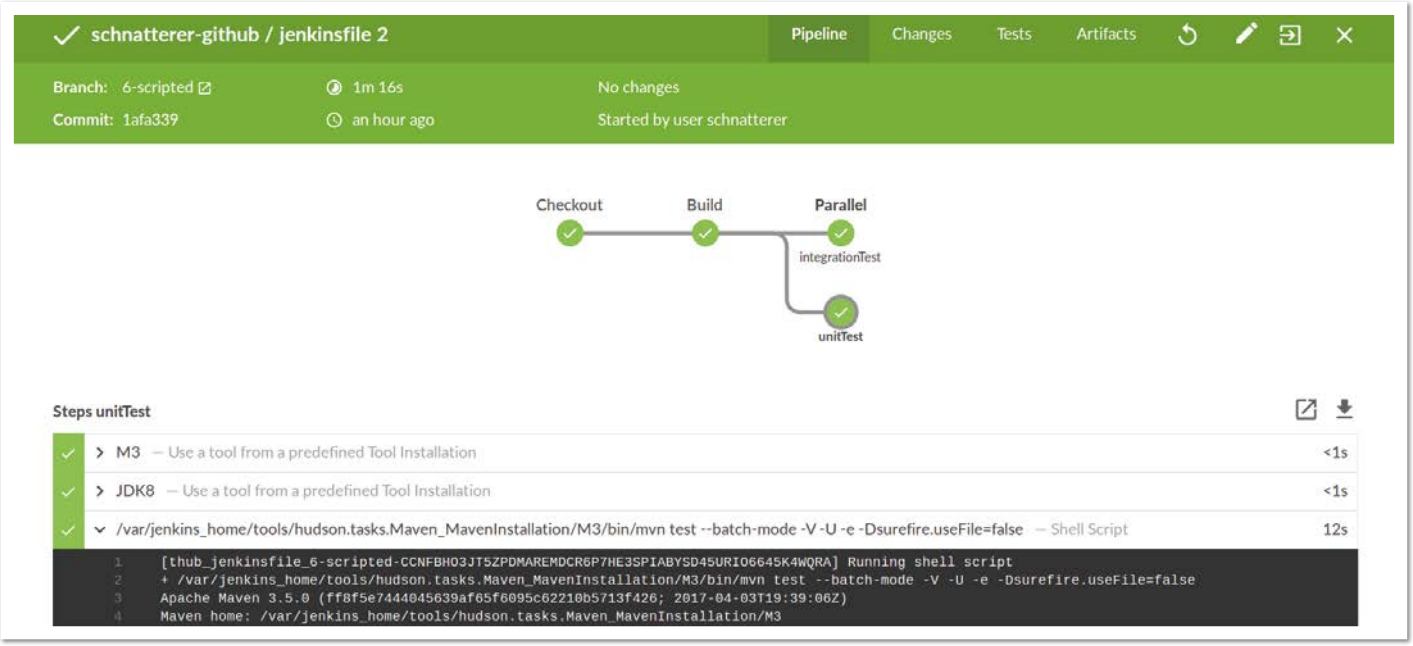


Abbildung 1: Parallelisierung und Log im Blue-Ocean-Theme

```
[integrationTest] [thub_jenkinsfile_6-scripted-CCNFBHO3JT5ZPDMAREMDCR6P7HE3SPIABYSD45URIO6645K4WQRA] Running shell
script
[integrationTest] + /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/M3/bin/mvn verify -DskipUnitTests
-Parq-wildfly-swarm --batch-mode -V -U -e -Dsurefire.useFile=false
[unitTest] [thub_jenkinsfile_6-scripted- CCNFBHO3JT5ZPDMAREMDCR6P7HE3SPIABYSD45URIO6645K4WQRA] Running shell script
[unitTest] + /var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/M3/bin/mvn test --batch-mode -V -U -e
-Dsurefire.useFile=false
```

Listing 2

```
stage('Tests') {
  parallel {
     stage('Unit Test') {
        steps {
           mvn 'test'
        }
     }
     stage('Integration Test') {
        steps {
           mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
        }
     }
  }
}
```

Listing 3

```
properties([
  pipelineTriggers([cron('H H(0-3) * * 1-5')])
])
```

Listing 4

```
pipeline {
  agent any
  triggers {
     cron('H H(0-3) * * 1-5')
  }
}
```

Listing 5

```
boolean isTimeTriggered = isTimeTriggeredBuild()
node {
  // ...
  stage('Integration Test') {
     if (isTimeTriggered) {
        mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
     }
  }
  //...
}
boolean isTimeTriggeredBuild() {
  for (Object currentBuildCause : script.currentBuild.rawBuild.getCauses()) {
     return currentBuildCause.class.getName().contains(‚TimerTriggerCause‘)
  }
  return false
}
```

Listing 6

Jenkins und gibt dort erneut das Repository sowie den Namen des neuen Jenkins-Files an, das gelesen werden soll. Der Vorteil ist, dass dies einfach aufzusetzen ist und eine gewisse Separation of Concerns bietet.

Statt eines monolithischen Jenkins-Files mit sehr vielen Stages, die abhängig vom Trigger ausgeführt werden, hat man zwei Jenkins-Files, bei denen immer alle Stages durchlaufen werden. Dies steht allerdings im Gegensatz zum Pipeline-Gedanken, bei dem jeder Build immer die gleichen Stages durchläuft. Zudem lässt sich bei den beiden Pipelines eine gewisse Redundanz nicht vermeiden. Beispielsweise benötigt man meist in beiden die Build-Stage. Dieser Redundanz kann man zwar mit Shared Libraries oder dem „load"-Step begegnen (siehe unten), trotzdem erhöht dies den Aufwand und die Komplexität.

Außerdem ist ein zusätzlicher Job schwerer zu verwalten, insbesondere wenn man Multibranch-Pipeline-Jobs oder gar eine GitHub-Organization betreibt. Dort werden pro Repository und Branch dynamisch neue Jobs angelegt *(siehe [1])*. Man hat dann mehrere Multibranch-Build-Jobs, bei GitHub-Organizations sogar einen weiteren Multibranch-Build-Job pro Repository, das in den Nightly Builds enthalten ist.

Alternativ pflegt man alle Stages in einem Jenkins-File und unterscheidet, welche Stages immer und welche nur nächtlich ausgeführt werden. Man kann zudem festlegen, welche Branches nächtlich gebaut werden sollen. Dieser Ansatz entspricht dem Pipeline-Gedanken. Hier hat man dann die vollen Vorteile von Multibranch-Pipeline-Jobs, da jeder dynamisch generierte Branch, falls gewünscht, auch nächtlich gebaut wird. Im Moment hat dieser Ansatz noch den großen Nachteil, dass das Abfragen des Build-Triggers umständlich ist.

Aufgrund der beschriebenen Nachteile eines weiteren Jobs liegt hier die Unterscheidung innerhalb des Jobs näher. Ähnlich wie bei den klassischen Freestyle-Jobs kann man in der Pipeline die Auslöser des Builds („Build Causes") abfragen. Im Moment geht dies jedoch nur über das „currentBuild.rawBuild"-Objekt. Den Zugriff darauf muss man sich von einem Jenkins-Administrator freigeben lassen („Script Approval"). Dies ist per Web auf *„https://JENKINSURL/scriptApproval/"* oder direkt im Dateisystem unter „JENKINS_HOME/script-approval.xml" möglich.

Jenkins empfiehlt, den Zugriff auf das Objekt aus Sicherheitsgründen nicht freizugeben. Trotzdem ist dies derzeit die einzige Methode, an die Build Causes zu kommen *[8]*. Die Lösung dafür ist allerdings schon unterwegs: Perspektivisch wird man die Build Causes direkt über das „currentBuild"-Objekt abfragen können *[9]*, wofür kein Script Approval notwendig sein wird. Dies vorausgesetzt, kann man beispielsweise die Ausführung der Integrationstests auf den Nightly Build beschränken *(siehe Listing 6)*.

Auch hier ist die Logik für das Abfragen des Build Cause in einen eigenen Step ausgelagert. Auffällig ist, dass dieser Step außerhalb des Node aufgerufen wird. Man könnte diesen auch direkt in der „Integration Test"-Stage abrufen. Allerdings würde man dann nicht die Aufforderung zum Script Approval bekommen, da sie vom Build Executor (Node) nicht zurück zum Master übertragen wird. Um dieses Beispiel auszuführen, sind die beiden Einträge aus *Listing 7* in der „script-approval.xml" erforderlich.

Wie beschrieben, kann man die Methoden auch per Web freigeben. Allerdings müssen die Methoden einzeln nacheinander zugelassen werden:

- Build ausführen
- Fehlschlag
- „getRawBuild" zulassen, Build ausführen
- Fehlschlag
- „getCauses" zulassen, Build ausführen
- Erfolg

Wem das zu kompliziert ist, der kann sich auch mit dem Workaround aus *Listing 8* behelfen. Hier wird einfach anhand der Uhrzeit entschieden, ob der Build in der Nacht läuft. Dies hat den Nachteil, dass auch anders angestoßene Builds (beispielsweise durch SCM oder manuell gestartet) als Nightly Build betrachtet werden. Außerdem ist hier darauf zu achten, dass sich die Zeiten auf die Zeitzone des Jenkins-Servers beziehen. Unabhängig vom Build Cause hat man immer die Möglichkeit, nur bestimmte Branches in der Nacht zu bauen. *Listing 9* zeigt, wie das in „scripted"-Syntax aussieht.

In der „declarative"-Syntax lässt sich dasselbe erreichen, ist aber anders auszudrücken. Hier gibt es die „when"-Directive, um über die Ausführung einer Stage zu entscheiden. In dieser kann man jedoch nur die von Jenkins oder einer Shared Library (siehe unten) bereitgestellten Steps verwenden. Innerhalb der Pipeline definierte Steps (wie „isTimeTriggeredBuild()" oder „isNightly()") kann man nicht aufrufen. Allerdings lässt sich der Code der Methoden dort direkt verwenden *(siehe Listing 10)*.

Ein Vorteil der „declarative"-Syntax ist die bessere Integration in Blue Ocean. Wird eine Stage aufgrund des negativen Ergebnisses

```
  <approvedSignatures>
     <string>method hudson.model.Run getCauses</string>
     <string>method org.jenkinsci.plugins.workflow.
support.steps.build.RunWrapper getRawBuild</string>
  </approvedSignatures>
```

Listing 7

```
boolean isNightly() {
  return Calendar.instance.get(Calendar.HOUR_OF_DAY) in 0..3
}
```

Listing 8

```
node {
  properties([
     pipelineTriggers(createPipelineTriggers())
  ])
  // ...
}
def createPipelineTriggers() {
  if (env.BRANCH_NAME == 'master') {
     return [cron('H H(0-3) * * 1-5')]
  }
  return []
}
```

Listing 9

```
stage('Integration Test') {
  when { expression { return Calendar.instance.get(Calendar.HOUR_OF_DAY) in 0..3 } }
  steps {
     mvn 'verify -DskipUnitTests -Parq-wildfly-swarm '
  }
}
```

Listing 10

Abbildung 2: Mittels „declarative"-Syntax übersprungene Stage in Blue Ocean

der „when"-Directive übersprungen, ist dies in Blue Ocean entsprechend visualisiert *(siehe Abbildung 2)*. Dies ist mit „scripted"-Syntax nicht möglich.

Es ist nicht intuitiv, mit der „declarative"-Syntax nur bestimmte Branches in der Nacht zu bauen, denn die gezeigte „triggers"-Directive kann nicht konditional ausgeführt werden. Als Notlösung bieten sich der „script"-Step oder das Aufrufen selbst definierter Steps an – dort lässt sich „scripted"-Syntax ausführen. Damit kann man sich die Trigger in den „scripted"-Properties festlegen. Nachteil ist, dass dies nur innerhalb einer Stage ausgeführt werden kann *(siehe Listing 11)*.

Fazit und Ausblick

Dieser Artikel zeigt, wie man die Ausführungszeit der Pipeline verkürzen kann. Mit Parallelisierung ist dies sehr einfach möglich. Eine weitere Möglichkeit ist die Auslagerung lang laufender Stages in den Nightly Build. Sie ist derzeit aber noch aufwendig.

Die Konsolidierung der Pipeline ist ein Vorgeschmack auf den Artikel in der nächsten Ausgabe, in dem mit Shared Libraries und Docker weitere nützliche Werkzeuge vorgestellt werden. Diese vereinfachen den Umgang mit Pipelines durch Wiederverwendung über verschiedene Jobs hinweg, Unit-Testing des Pipeline-Codes und den Einsatz von Containern.

```
Pipeline {
// ...
  stages {
     stage('Build') {
        steps {
           // ...
           createPipelineTriggers()
        }
     }
  }
  // ...
}
void createPipelineTriggers() {
  script {
     def triggers = []
     if (env.BRANCH_NAME == 'master') {
        triggers = [cron('H H(0-3) * * 1-5')]
     }
     properties([
        pipelineTriggers(triggers)
     ])
  }
}
```

Listing 11

Weitere Informationen

[1] J.Schnatterer, D.Behrwind, Coding Continuous Delivery – Grundlagen des Jenkins Pipeline Plug-ins", Java aktuell 01/2018
[2] Jenkinsfile Repository GitHub: *https://github.com/triologygmbh/jenkinsfile*
[3] Triology Open Source Jenkins: *https://opensource.triology.de/jenkins/job/triologygmbh-github/job/jenkinsfile/*
[4] Locks: *https://jenkins.io/doc/pipeline/steps/lockable-resources/#lock-lock-shared-resource*
[5] Milestone Plug-in: *https://plugins.jenkins.io/pipeline-milestone-step*
[6] Locks und Milestone Beispiel: *https://github.com/jenkinsci/workflow-aggregator-plugin/blob/691c8b0c16690f12c8d425d5b3a6aa9019d6bcfb/demo/repo/Jenkinsfile*
[7] J.Schnatterer, Automatisierte Überprüfung von Sicherheitslücken in Abhängigkeiten von Java-Projekten, Java aktuell 01/2017
[8] Pipeline Examples - Build Cause: *https://jenkins.io/doc/pipeline/examples/#get-build-cause*
[9] Jenkins Issue 41272: *https://issues.jenkins-ci.org/browse/JENKINS-41272*
[10] Groovy Scripts vs. Classes: *http://docs.groovy-lang.org/latest/html/documentation/index.html#_scripts_versus_classes*
[11] Groovy Call Operator: *http://docs.groovy-lang.org/latest/html/documentation/#_call_operator*
[12] cps-global-lib-plugin Pull Request 37: *https://github.com/jenkinsci/workflow-cps-global-lib-plugin/pull/37*

Johannes Schnatterer
johannes.schnatterer@triology.de

Johannes Schnatterer ist Solution Architect bei der TRIOLOGY GmbH in Braunschweig. Technologisch ist er dort in den Bereichen „Java EE" und „Web" tätig und versucht, mit besonderem Fokus auf Qualität, Open-Source-Enthusiasmus, einem Hauch von Pedantismus und der Pfadfinderregel die IT-Welt jeden Tag ein bisschen besser zu machen. Derzeit arbeitet er am Cloudogu Ecosystem.

„Es wird zurzeit alles getan, um Java in die richtige Richtung zu lenken …"

Simon Ritter ist Deputy CTO bei Azul Systems und großer Fan des neuen Release-Plans für Java Standard Edition und Java Enterprise Edition, da man damit beiden Anwendergruppen gerecht werde. „Die Übergabe von Java EE an die Eclipse Foundation sorgt für mehr Stabilität und eine bessere Einbeziehung der Community", so Ritter.

Im DOAG.tv-Interview *(siehe „https://www.doag.org/de/home/news/es-wird-zurzeit-alles-getan-um-java-in-die-richtige-richtung-zu-lenken/detail")* mit Andrea Badelt, stellv. Leiter der DOAG Java Community, spricht Ritter außerdem über das Java-Ökosystem, JDK, Support Features und mehr.

Feature Branches und Continuous Integration sind kein Widerspruch

Sebastian Damm, Orientation in Objects GmbH

Continuous Integration und Feature Branches sind zwei allgegenwärtige Themen in der heutigen Software-Entwicklung. Sie repräsentieren zwei Lager, deren Anhänger teilweise keine besonders hohe Meinung über den jeweils anderen haben. Für viele Puristen sind die beiden Themengebiete derart widersprüchlich, dass eine Kombination der beiden Ansätze nicht nur unpraktikabel, sondern sogar undenkbar ist. Dieser Artikel zeigt, warum und wie eine solche Kombination – unter Zuhilfenahme einiger Kompromisse – dennoch möglich ist.

Bevor wir uns den Differenzen der beiden Lager widmen, werden die beiden Ansätze kurz vorgestellt. Leser, denen die Grundprinzipien bereits ausreichend bekannt sind, können die beiden folgenden Kapitel überspringen.

Continuous Integration

Der Kerngedanke von Continuous Integration (CI) ist die mindestens tägliche Integration der Arbeit aller Projekt-Mitglieder. Mit jedem Commit in das Repository wird über einen CI-Server (beispielsweise Jenkins oder Bamboo) der CI-Prozess *(siehe Abbildung 1)* ausgelöst, der die Anwendung kompiliert, paketiert und in der Regel automatisiert testet.

Die stetige und kontinuierliche Integration der Arbeit aller Entwickler soll im Verbund mit dem automatisierten Bauen und Testen der Anwendung gewährleisten, das Projekt möglichst fehlerfrei zu halten

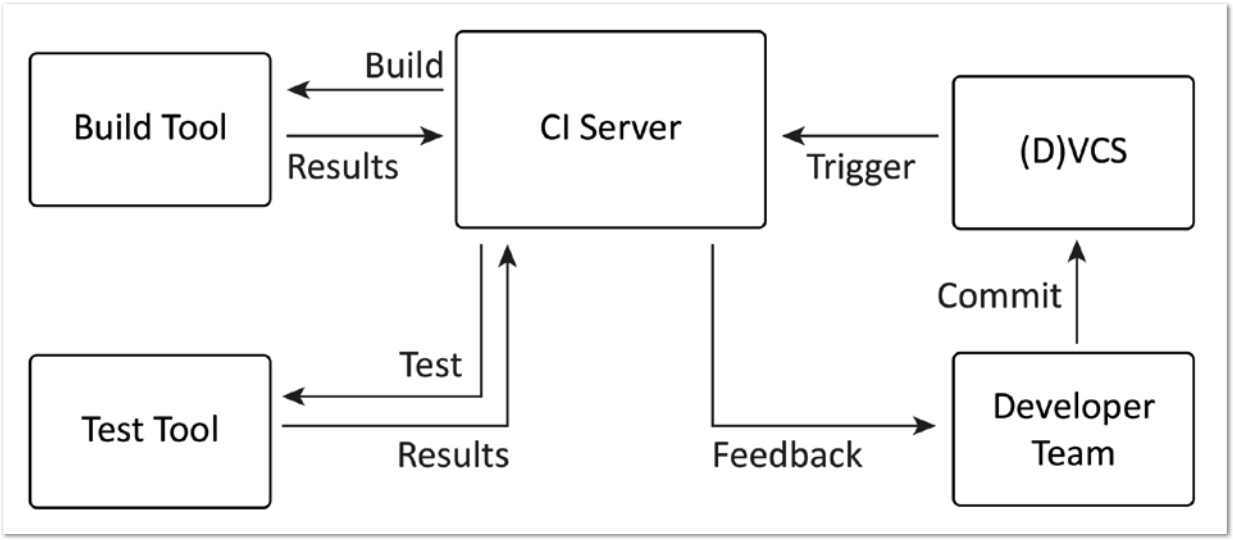


Abbildung 1: Schematische Darstellung des CI

beziehungsweise beim Auftreten von Fehlern diese zumindest unverzüglich zu erkennen. „Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove" *[1]*.

Der Begriff „Continuous Integration" wurde bereits im Jahr 2000 von Martin Fowler mit einem gleichnamigen Artikel *[1]* eingeführt. In diesem nannte er einige explizite Key Practices, die seiner Meinung nach erfüllt werden müssen, um effektiv CI betreiben zu können. Die beiden subjektiv wichtigsten Key Practices sind:

- *Key Practice Nr. 2: Automate the Build*
  Insbesondere in größeren Software-Projekten ist der Vorgang, das Projekt zu kompilieren und in ein ausführbares oder deployabres Format zu überführen, sehr komplex und somit fehlerträchtig, falls er manuell ausgeführt wird. Dementsprechend sollte dieser Prozess komplett automatisiert sein. Durch Build- und Dependency-Management-Tools wie Maven oder Gradle – die ihrerseits sehr einfach in CI-Server zu integrieren sind – ist dieser Punkt aus heutiger Sicht leicht realisierbar.
- *Key Practice Nr. 4: Everyone Commits To The Mainline Every Day*
  Jeder Entwickler sollte mindestens (!) einmal pro Tag seine Code-Änderungen in die Mainline (die Haupt-Entwicklungslinie innerhalb eines Repository, oft auch als „Master" bezeichnet) committen. Auf diese Weise sollen die Anzahl und die Komplexität von Konflikten zwischen mehreren Entwicklern möglichst gering gehalten werden.

Feature Branches

Feature Branches (FBs), beziehungsweise allgemein „Branching", sorgen für die Isolation von Programmcode, sodass mehrere Bestandteile eines Systems parallel bearbeitet werden können, ohne sich gegenseitig – in einem temporär instabilen Zustand – zu beeinflussen. Technisch werden dazu alle Objekte eines Zweigs in einen neuen Zweig kopiert, wodurch fortan Entwickler in beiden Zweigen unterschiedliche Versionen des gleichen Objekts oder der gleichen Datei bearbeiten können.

Der Leitgedanke beim Branching ist, dass Code-Änderungen erst dann, wenn sie sich in einem stabilen und getesteten Zustand befinden, in die Mainline integriert werden. *Abbildung 2* zeigt allerdings auch direkt das größte Problem des Branching: Sobald man einen neuen Branch erstellt, muss dieser irgendwann wieder integriert beziehungsweise gemergt werden, ein unter Umständen sehr fehleranfälliger Vorgang.

Doch nicht nur die CI-Fraktion kann in Form von Martin Fowler mit prominenten Fürsprechern punkten, auch Feature Branches haben mit Linus Torvalds namenhafte Prominenz auf ihrer Seite. Dieser argumentiert, dass der Linux-Kernel mithilfe von Branching entwickelt wird und dass sich viele Entwickler zu viele Sorgen über etwaige Merge-Konflikte machen: „The linux-next tree has something like 8.600 commits, and I've merged about 5.600 in the last few days. [...] Only 17 of the merges had any conflicts. And only a couple of those were annoying. The rest is all git doing all the work." *[2]*.

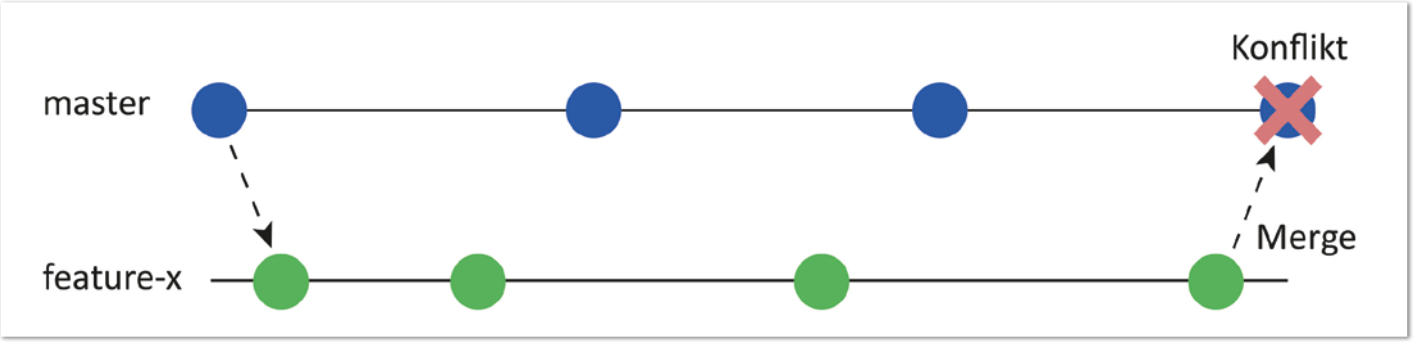


Abbildung 2: Darstellung von Branching und Merges/Merge-Konflikten

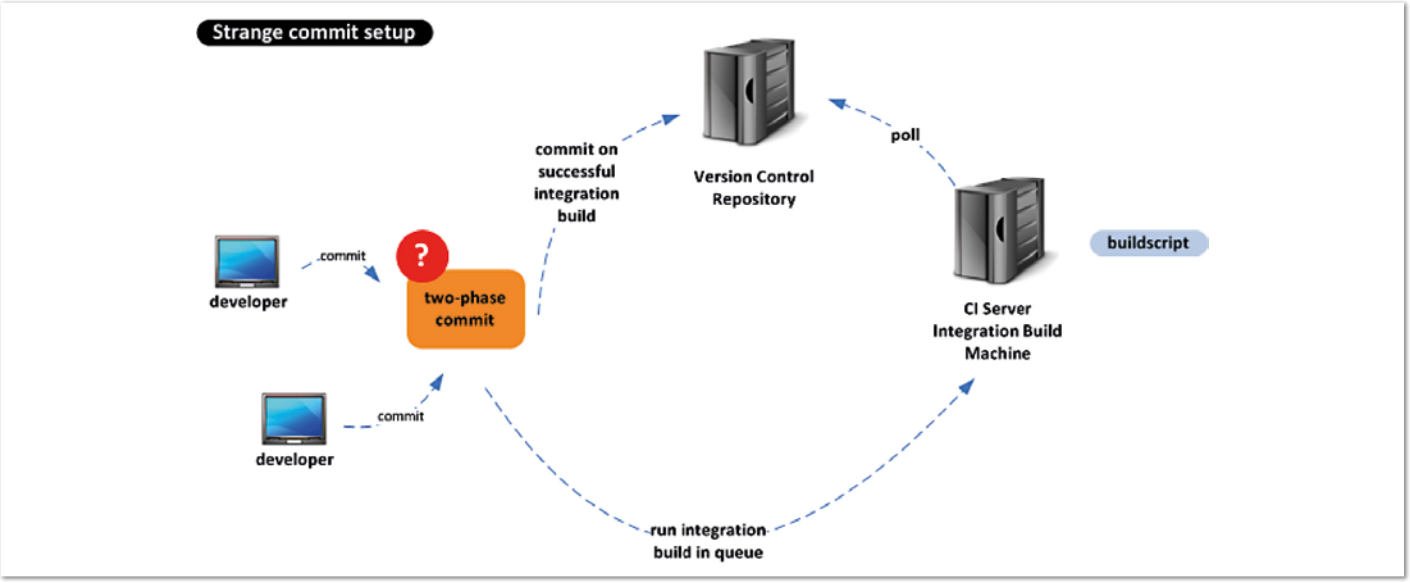


Abbildung 4: Two-Phase-Commit als „Future of CI" (Quellen: Beschreibung aus [8], grafische Darstellung aus [9])

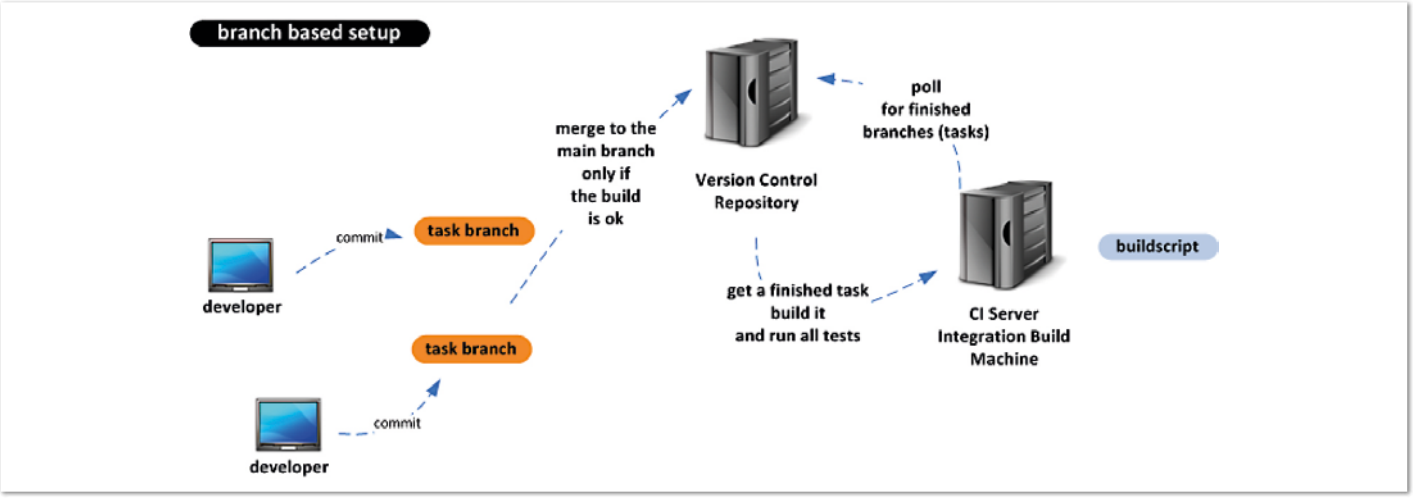


Abbildung 5: Branching Workflow als Antwort zum Two-Phase-Commit (Quelle: [9])

Während es im CI-Umfeld eine ganze Reihe von Artikeln mit Best Practices gibt, findet man im Bereich von Feature Branches in der Mehrzahl Artikel mit sogenannten „Anti-Pattern", also Praktiken, die man vermeiden sollte, falls man Branching erfolgreich einsetzen möchte. In *[3]* nennt der Autor eine ganze Reihe solcher Anti-Patterns, von denen wir auszugsweise zwei betrachten wollen:

- *Big Bang Merge*
  Merging has been deferred to the very end of the development effort and an attempt is made to merge all branches simultaneously.
- *Never Ending Merge*
  Merge activity never seems to end; there's always more to merge.

Let's get ready to rumble

Nachdem die beiden Kontrahenten vorgestellt sind, wollen wir uns nun der Frage widmen, in welchen Argumenten der Streit zwischen den beiden Parteien begründet liegt. Bereits in dem initialen CI-Artikel aus dem Jahr 2000 äußert sich Martin Fowler eher kritisch in Bezug auf FBs: „Keep your use of branches to a minimum".

Man könnte argumentieren, dass man diese Aussage auch im Kontext der Zeit sehen muss und Branching mit damaligen Versionsverwaltungs-Systemen wie CVS oder SVN weder sonderlich komfortabel noch performant war.

Allerdings bekräftigte Fowler im Jahr 2009 seinen Standpunkt zu FBs in einem dedizierten Artikel *[4]*, in dem er sich der Frage widmet, wie CI und FBs zusammenpassen. Er nimmt insbesondere Stellung zu der Tatsache, dass viele Entwickler aus dem FB-Lager behaupteten, dass sie CI und FBs kombinieren, indem sie für jeden Branch mit jedem Commit per CI-Server das Projekt automatisch bauen und testen lassen: „So unless feature branches only last less than a day, running a feature branch is a different animal to CI. I've heard people say they are doing CI because they are running builds, perhaps using a CI server, on every branch with every commit. That's continuous building, and a Good Thing, but there's no integration, so it's not CI." Weitere Gegenargumente von Fowler sind Semantic Conflicts und die Beeinträchtigung von Opportunistic Refactoring durch Branching *[4]*.

Fowler und das CI-Lager kritisieren allerdings nicht nur den (über-

triebenen) Einsatz von Branches, sie liefern auch Alternativen, um sich noch in der Entwicklung befindlichen Code vom Rest des Codes zu isolieren. Eine Alternative sind Feature Toggles.

Hier ist der betreffende Code zwar schon Bestandteil des Projekts, über einen globalen Schalter („Feature Toggle") ist die Ausführung des Codes allerdings zunächst deaktiviert. Sind die Code-Änderungen abgeschlossen und stabil, wird der Schalter umgelegt und der Code somit aktiv. Nachteilig ist hierbei, dass der Code so immer in die Produktion gelangt und dass der Toggle im Anschluss wieder entfernt werden muss. Manche Teams verwenden Feature Toggles allerdings auch wieder, anstatt die alten Schalter aus- und neue einzubauen. Dass dieses Vorgehen katastrophale Folgen haben kann, bewies die Knight Capital Group 2012, als sie innerhalb von 45 Minuten 460 Millionen Dollar verlor *[6]*.

Welche Argumente bringt die FBs-Fraktion gegen reines CI vor? Das Hauptproblem von CI ist, dass es eine reaktionäre Praxis ist. Auch wenn jeder Commit einen automatisierten Build- und Test-Prozess auslöst, so signalisiert ein Compilerfehler oder ein fehlgeschlagener Test letztlich nur, dass die komplette Mainline sich bereits in einem fehlerhaften Zustand befindet. Diese „Mainline Instability" wird auch in einem Artikel aus dem Jahr 2008 angeprangert *[7]*. „All the changes directly hit the mainline, so making it unstable is relatively easy."

Ein Fehler auf der Mainline beeinträchtigt dann mitunter das ganze Projekt-Team, da „[a] bug entering the mainline will be spread to all developers in a short time [and] several developers will end up fixing [it]". Diese Probleme sind der CI-Community allerdings auch bewusst. Paul Duvall et. al beschreiben im Epilog „The Future of CI" des Buches „Continuous Integration: Improving Software Quality and Reducing Risk"*[8]* zwei Lösungsansätze, mit denen man diesem Problem begegnen kann:

- *Two-Phase Commit*
  Before the repository accepts the code, it runs an integration build on a separate machine. Only if … successful will it commit the code …
- *Personal Builds*
  … capability for a developer to run an integration build using the integration build machine and his local changes along with any other changes committed to the version control repository

Wie dieser Two-Phase-Commit aussehen soll, ist allerdings nur schematisch und nicht sehr detailliert dargestellt. Vielleicht denken sich einige Leser an dieser Stelle, dass diese Anforderung des Two-Phase-Commit mit einem anderen Ansatz sehr einfach umsetzbar ist. Das dachte sich auch der Autor des Blog-Eintrags „Continuous integration future?" *[9]* und skizziert zunächst den Two-Phase-Commit, wie er in *[8]* von Duvall et. al beschrieben wurde *(siehe Abbildung 3)*. Er führt dann aus, dass das Problem, das das CI-Lager mit einem nicht näher spezifizierten Two-Phase-Commit lösen will, mit Branching und einem „Branch per Feature"-Workflow leicht zu erledigen ist *(siehe Abbildung 4)*.

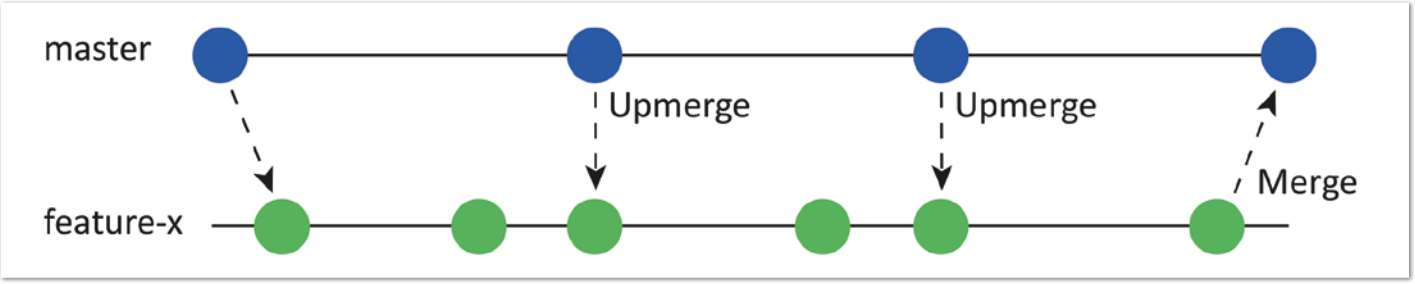


Abbildung 6: Vermeidung von Merge-Konflikten durch sofortige Integration von Änderungen auf der Mainline

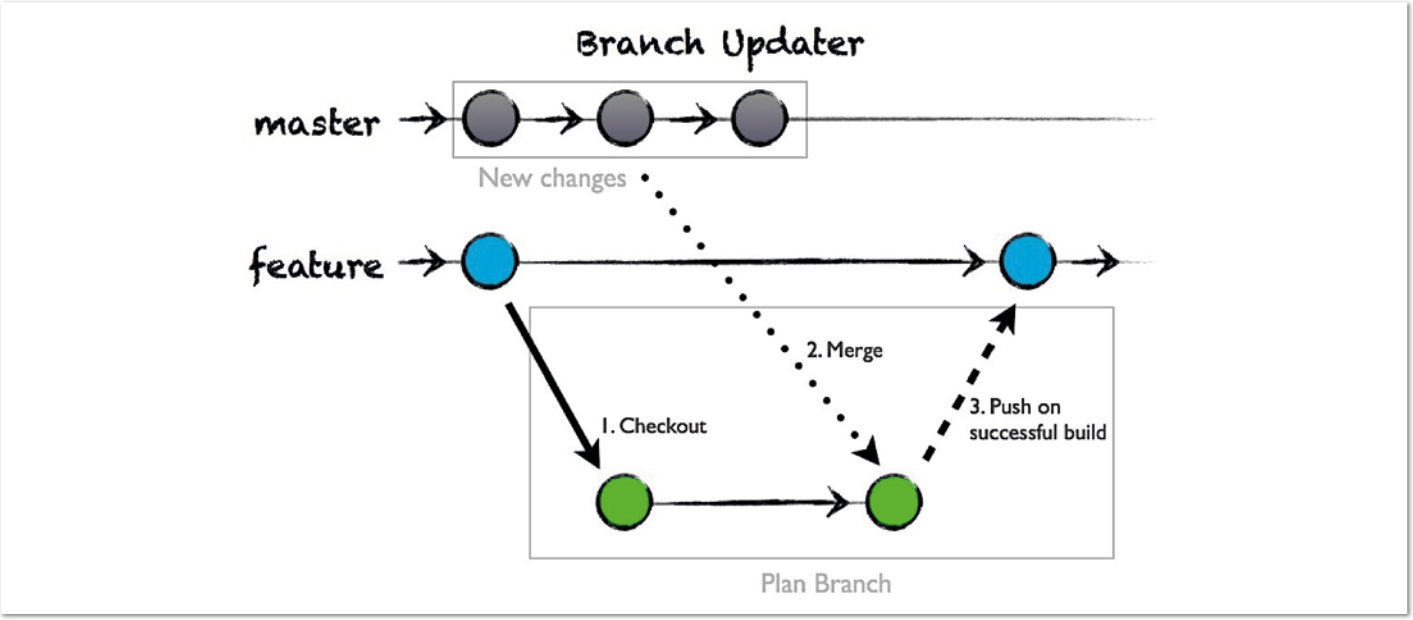


Abbildung 7: Automatischer Branch Updater von Bamboo (Quelle: [12])

Jetzt alles zusammen

Continuous Integration und (Feature) Branching sind zwei Ansätze, die sich in Teilen ihrer Definition nicht nur deutlich unterscheiden, sondern schlichtweg widersprechen: „By definition, if you have code sitting on a branch, it isn't integrated" *[10]*. So kann man zu dem Schluss kommen, dass CI und FBs sich gegenseitig ausschließen und man entweder nur einen der beiden Ansätze verfolgt oder gar keinen *[10]*.

Man kann sich der Thematik allerdings auch von der anderen Seite nähern und nicht nur die Unterschiede der beiden Praktiken betrachten, sondern nach Gemeinsamkeiten suchen beziehungsweise nach Gründen, aus denen eine Kombination überhaupt sinnvoll sein könnte. Wenn man dies tut, stellt man fest, dass beide ein gemeinsames Hauptziel haben, nämlich den absoluten Schutz der Mainline beziehungsweise den Wunsch, zu jedem Zeitpunkt einen absolut fehlerfreien, vertrauenswürdigen Codezustand zu haben, den man so ohne größere Bedenken direkt auch in Produktion deployen würde. Da die beiden Ansätze also ein gemeinsames Hauptziel haben (das sie alleinstehend nicht perfekt erreichen) und das Hauptproblem von CI (die „broken builds", siehe *[8]* oder *[9]*) sich durch FBs beheben lässt, kann eine Kombination durchaus Sinn ergeben.

Wie kann eine pragmatische und funktionierende Kombination von CI und FBs in der Praxis aussehen? Jilles van Gurp nennt einige Fakten für diese Kombination und beschreibt Maßnahmen, die in jedem Fall ergriffen werden sollten *[11]*:

- *You can't do decentralized versioning unless you also decentralize your testing and integration.*
  Ganz entscheidend ist also, dass bereits auf Branch-Ebene (teilweise) integriert sowie mit den gleichen CI-Prozessen gebaut und getestet wird, wie dies für die Mainline der Fall ist.
- *… decentraliz[ing] testing and integration solve[s most] problems before change is pushed upstream. … by decentralizing you have many people working on smaller sets of changes that are much easier to deal with: the collaborative effort on testing decreases and when it all comes together you have a much smaller set of problems to deal with.*
  Integriert man alle Änderungen der Mainline, beziehungsweise (indirekt) von anderen Branches, direkt in seinen Branch, so werden die potenziell fehlerträchtigen Merges auf Branch-Ebene durchgeführt und bei Fehlern die Mainline nicht beschädigt. Bei einer erfolgreichen Integration der Mainline in einen Branch ist die anschließende Integration zurück in die Mainline automatisch konfliktfrei.
- *Rebase against main frequently – If your feature branch is way behind, you have done something very wrong.*
  Abgeleitet aus dem vorherigen Punkt ist es zwingend notwendig, dass jegliche Änderung der Mainline von jedem Entwickler sofort in seinen Branch integriert wird *(siehe Abbildung 5)*. Wird dies nicht berücksichtigt, sammeln sich die Integrations- und Merge-Aufwände an, die Gefahr eines Big-Bang-Merge steigt.
- *Push as early as you can – Feature branches are not about hiding change but about isolating change.*
  Was mitunter falsch verstanden wird, ist, dass es bei Branches nicht darum geht, Änderungen zu verstecken, sondern sie lediglich zu isolieren, bis sie in einem stabilen Zustand sind. Anschließend befinden sich die Code-Änderungen auf einem Branch allerdings in einem stabilen und getesteten Zwischenzustand. Dies kann nur gewährleistet werden, wenn der erste Punkt erfüllt ist und für jeden Branch der CI-Prozess zur Verfügung steht; dann sollten diese bereits zwischendurch immer wieder in die Mainline integriert werden.
- *Pull change rather than pushing it*
  Ein weiterer schöner Aspekt in einem Branch-basierten Workflow ist die Tatsache, dass man Änderungen nicht direkt mergen muss, sondern dass man auch mit Pull Requests arbeiten kann. Entwickler können so eine Anfrage initiieren, die dafür sorgt, dass ein anderer Entwickler oder eine Person mit einer spezifischen „Integrator"-Rolle wie der Projektleiter zunächst die vorgenommenen Änderungen überprüft. Erst nach dessen Bestätigung werden die Änderungen integriert. Dieses Vorgehen kann durch entsprechende Rechte-Verwaltung sogar forciert werden, indem Entwicklern ein direkter Merge in die Mainline schlichtweg verweigert wird.

Es wird deutlich, dass viele dieser Vorgaben stark miteinander verzahnt sind beziehungsweise aufeinander basieren. Der Punkt „Rebase against main frequently" beispielsweise kann nur funktionieren, wenn auch die darauffolgende Empfehlung „Push as early as you can" berücksichtigt wird.

Noch mal Schritt für Schritt

Diese Punkte sind für eine erfolgreiche Kombination von CI und FBs zu berücksichtigen:

- Feature Branch erzeugen
- Continuous Integration für Feature Branch bereitstellen
- Im Feature Branch arbeiten
- Master-Änderungen sofort (!) in Feature Branch übernehmen
- Bei stabilen Zwischenzuständen Merge oder Pull Request vom Branch in die Mainline (nach vorheriger erfolgreicher Integration der Mainline in den Branch)
- Nach Fertigstellung abschließender „Up Merge" in den Master

Was sich sehr aufwendig anhört, ist dank heutiger Tool-Unterstützung nur mit wenig manuellem Aufwand verbunden. Alle gängigen CI-Server unterstützen ein Feature namens „Branch detection". Das Anlegen eines neuen Branch wird erkannt, die definierten CI-Prozesse werden automatisiert für den Branch kopiert, wodurch der zweite Punkt gewährleistet ist.

Einige CI-Server (wie Atlassian Bamboo mit dem „Branch Updater") bieten die Möglichkeit, eine Abhängigkeit zwischen mehreren Branches zu definieren. Ist ein Branch A von einem Branch B abhängig, so wird bei jedem Commit auf A geprüft, ob in der Zwischenzeit (also seit dem letzten Build von A) Änderungen auf dem abhängigen Branch B vorgenommen worden sind. Ist dies der Fall, werden die Änderungen von B automatisch in A integriert und der Build- und Testprozess angestoßen. Falls dieser erfolgreich ist, werden die integrierten Änderungen direkt auf A gepusht *(siehe Abbildung 6)*.

Selbst für die Erstellung und Verwaltung von Branches oder Pull Requests bieten heutige Issue-Tracker und Repository-Management-Tools ausgezeichnete Unterstützung. Der größte Aufwand in einem nach diesen Vorgaben aufgesetzten Projekt liegt damit in der notwendigen Entwickler-Disziplin, stabile Zwischenzustände ihrer Branches zwischendurch immer wieder in die Mainline zu integrieren. Nur so kann der beschriebene und dank Tool-Unterstützung

weitestgehend automatisierte Workflow funktionieren.

Fazit

Auch wenn nicht geleugnet werden kann, dass die beiden Ansätze teilweise zueinander widersprüchliche Definitionen besitzen, ergibt eine Kombination aufgrund eines gemeinsamen Hauptziels dennoch Sinn. Die Maßnahmen für eine erfolgreiche Kombination sind zahlreich, zudem ist eine große Portion Disziplin im gesamten Entwicklerteam notwendig.

Gelingt es allerdings, diese Vorgaben (mithilfe von Tool-Unterstützung) erfolgreich umzusetzen, so wird das gemeinsame Ziel, also der absolute Schutz der Mainline, besser erreicht, als es mit einem der beiden Ansätze alleinstehend möglich ist.

Quellen und weitere Informationen

[1] *https://martinfowler.com/articles/continuousIntegration.html*
[2] *https://plus.google.com/+LinusTorvalds/posts/fDENcrCqHmD*
[3] *https://blog.codinghorror.com/software-branching-and-parallel-universes*
[4] *https://martinfowler.com/bliki/FeatureBranch.html*
[5] *http://martinfowler.com/bliki/FeatureToggle.html*
[6] *https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale*
[7] *http://www.drdobbs.com/architecture-and-design/scm-continuous-vs-controlled-integration/205917960*
[8] Paul M. Duvall, Steve Matyas, Andrew Glover: Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley Signature Series, Pearson Education, 2007
[9] *http://blog.plasticscm.com/2008/03/continuous-integration-future.html*
[10] *https://arialdomartini.wordpress.com/2011/11/02/help-me-because-i-think-martin-fowler-has-a-merge-paranoia*
[11] *http://www.jillesvangurp.com/2011/07/16/using-git-and-feature-branches-effectively*
[12] *https://confluence.atlassian.com/bamboo/using-plan-branches-289276872.html*

Sebastian Damm
sebastian.damm@oio.de

Sebastian Damm arbeitet bei der OIO – Orientation in Objects GmbH in Mannheim als Entwickler, Trainer und Berater. Daneben ist er an der DHBW Mannheim als Informatik-Dozent tätig. Seine Schwerpunkte liegen in den Bereichen Web-Anwendungen mit Java EE beziehungsweise GWT und Spring sowie in der Automatisierung funktionaler Systemtests.

Sie wollen uns persönlich kennenlernen?
Dann treffen Sie uns auf der JavaLand in Brühl!

Bist Du auch IT-Spezialist?

TimoCom ist ein mittelständischer, inhabergeführter IT-Spezialist und Anbieter von Europas größter Online-Plattform für die Transport- und Logistikbranche. In unserer Inhouse-Entwicklung bist Du Teil eines agilen Teams und arbeitest in einem modernen Entwicklungsumfeld (Java, Oracle, CI, DI, JSF). Klingt interessant?

Daria und David - User Experience & Quality
Marc und Jann - IT Development

TimoCom Jetzt bewerben und weiterentwickeln: www.timocom.de/jobs

Java-Architekturen dauerhaft sichern mit ArchUnit

Peter Gafert, TNG Technology Consulting GmbH

Wenn man den Ist-Zustand gewachsener Software-Systeme betrachtet und sich von den Teams oder verantwortlichen Architekten deren Soll-Architektur erklären lässt, wird man häufig gravierende Abweichungen feststellen. Die Gründe sind verschiedener Natur; manchmal konnten die Konzepte nicht leisten, was benötigt wurde, manchmal wurden Features unter Zeitdruck entwickelt und manchmal waren neue Entwickler mit den Konzepten noch nicht ausreichend vertraut.

Bei einem großen System lassen sich Verletzungen der Architektur nicht mal eben nebenher reparieren. Der Verbleib der Verletzungen im Code verleitet Entwickler wiederum dazu, diese schlechten Muster an anderer Stelle zu kopieren, wodurch sich Architektur-Verletzungen auf Code-Basis schnell ausbreiten und eine Reparatur im Laufe der Zeit immer teurer machen.

Zudem ist der Verbleib der Verletzungen auf Dauer nicht kostenfrei. Der Preis, den man dafür zahlen muss, besteht aus mangelhafter Anpassbarkeit und Wartbarkeit. Abhängigkeiten, die unnötiger- und unerwarteterweise auftreten, lassen die tatsächlich benötigte Entwicklungszeit im Vergleich zu Schätzungen explodieren und führen zu neuen Bugs.

Eine Verletzung von Hierarchien und Mustern oder Neueinführung unnötiger Muster erschwert das Verständnis für jeden einzelnen Entwickler erheblich, was die Entwicklungsgeschwindigkeit weiter drückt. Außerdem entsteht ein Kommunikations-Overhead, wenn Entwickler versuchen, die Gründe für verletzte Muster und Hierarchien nachzuvollziehen.

Die Feature-Entwicklung hat über die letzten Jahrzehnte große Fortschritte gemacht und die Industrie hat begriffen, dass wir ohne ein Gerüst aus automatisierten Tests die Funktionalität der Software nicht kontinuierlich und effizient sicherstellen können. Die These dieses Artikels lautet, dass wir diese Erkenntnis, wenn möglich, auch auf die Architektur der Software anwenden sollten – also unsere Architektur-Regeln in einer programmatisch verständlichen Art festhalten und das System kontinuierlich dagegen validieren. Um solche Tests umzusetzen, sollte die Infrastruktur einige Features mitbringen:

- Sie muss möglichst viel Information über die Code-Basis des Projekts zur Verfügung stellen
- Regeln zu spezifizieren, muss einfach für Standardfälle und frei erweiterbar für Spezialfälle sein

- Die Installation und Integration in lokale und zentrale Testumgebungen muss so einfach wie möglich sein
- Die Funktionsweise muss klar, verständlich und wartbar sein, um eine flache Lernkurve zu gewährleisten

Der Artikel erläutert die Open-Source-Bibliothek ArchUnit und wie diese die genannten Kriterien erfüllt.

Aufbau von ArchUnit

Die Struktur von ArchUnit besteht aus mehreren Schichten, von denen ein Teil erläutert wird, beginnend mit der Core-Schicht. Bezüglich der automatisierten Sicherstellung der Architektur haben wir bei Java Glück, da sich ein Großteil der Struktur im statisch kompilierten Bytecode wiederfindet. Einige Eigenschaften lassen sich mit dem bekannten Java-Reflection-API abfragen und damit auch sicherstellen *(siehe Listing 1)*.

```
Method method = SomeController.class.
getDeclaredMethod("call");
assertThat(method.getAnnotation(UiAccess.class))
  .as("Controller dürfen keinen UI-Zugriff deklarieren")
  .isNull();
```

Listing 1

Sobald diese Eigenschaften allerdings nicht mehr lokal sind, wie zum Beispiel Abhängigkeiten zwischen zwei Klassen, bietet uns das Reflection-API diese Information nicht mehr an. Dennoch findet sich diese im Bytecode wieder; *Listing 2* dient als Beispiel. Wenn wir den Bytecode analysieren, lässt sich der Aufruf von „CallingClass" nach „OtherClass" klar nachvollziehen *(siehe Listing 3)*. Die Basis von ArchUnit orientiert sich nun an dem Java-Reflection-API, erweitert es jedoch um die fehlende Information. Das hat den Vorteil, dass dieses Core-API den meisten Entwicklern bereits in ihren Grundzügen vertraut ist.

```
class CallingClass {
  OtherClass other;

  void execute() {
     other.call();
  }
}
class OtherClass {
  void call() {
  }
}
```

Listing 2

Core-Objekte sind nach ihrem Reflection-Pendant mit einem zusätzlichen Java-Präfix benannt. So existieren zum Beispiel „JavaClass", „JavaMethod" und „JavaField", aber auch neue Konzepte wie „JavaMethodCall", die Methodenaufrufe repräsentieren. Grundsätzlich verhalten sich diese Objekte wie ein erweitertes Reflection-API. So bietet „JavaClass" zum Beispiel die Methode „getSimpleName()", die sich wie erwartet verhält. Auf der anderen Seite ist es auch mög-

```
javap -v CallingClass.class
...
  #2 = Fieldref   // CallingClass.other:LOtherClass;
  #3 = Methodref  // OtherClass.call:()V
...
void execute();
  descriptor: ()V
  flags:
  Code:
     stack=1, locals=1, args_size=1
        0: aload_0
        1: getfield      #2 // Field other:LOtherClass;
        4: invokevirtual #3  // Method OtherClass.call:()V
        7: return
     LineNumberTable:
        line 7: 0
        line 8: 7
...
```

Listing 3

```
// Der ClassFileImporter kann zahlreiche Quellen
// von Classpath bis File URL importieren
JavaClasses classes = new ClassFileImporter().importJar(new JarFile("/some/archive.jar"));

Set<JavaClass> services = new HashSet<>();
for (JavaClass clazz : classes) {
  // Wir filtern die Klassen, deren vollqualifizierter Name
  // den Paketinfix '.service.' enthält
  if (clazz.getName().contains(".service.")) {
     services.add(clazz);
  }
}

for (JavaClass service : services) {
  // Wir iterieren über alle Zugriffe, die von der Klasse ausgehen
  for (JavaAccess<?> access : service.getAccessesFromSelf()) {
     String targetName = access.getTargetOwner().getName();

     // Fehlschlag, falls Zugriff auf ein Ziel mit ".controller." im Namen
     if (targetName.contains(".controller.")) {
        String message = String.format(
           "Service %s accesses Controller %s in line %d",
           service.getName(), targetName, access.getLineNumber());
        Assert.fail(message);
     }
  }
}
```

Listing 4

```
ArchRule rule =
  classes().that().resideInAPackage("..service..")
     .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");
```

Listing 5

```
JavaClasses classes = new ClassFileImporter().importPackage("com.myapp");
ArchRule rule = // siehe oben
rule.check(classes);
```

Listing 6

```
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
Rule 'classes that reside in a package ,..service..' should only be accessed by any package ['..controller..',
'..service..']' was violated:
Method <com.myapp.dao.EvilDao.callService()> calls method <com.myapp.service.SomeService.doSomething()> in (EvilDao.
java:14)
```

Listing 7

```
ArchRule layeredArchitecture = layeredArchitecture()
  .layer("Controllers").definedBy("..controller..")
  .layer("Services").definedBy("..service..")
  .layer("Persistence").definedBy("..persistence..")
  .whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
  .whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
  .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Services");
layeredArchitecture.check(classes);
```

Listing 8

```
ArchRule servicesShouldBeFreeOfCycles =
  slices().matching("com.myapp.(**).service..").should().beFreeOfCycles();
servicesShouldBeFreeOfCycles.check(classes);
```

Listing 9

lich, Dinge wie „javaClass.getAccessesToSelf()" abzufragen, was einem wiederum Zugriffe im analysierten Bytecode auf die jeweilige Klasse liefert.

Der Core-Layer beinhaltet ebenfalls den „ClassFileImporter", um Bytecode in eine programmatisch verständliche Form zu bringen. *Listing 4* zeigt ein Beispiel für einen möglichen Test.

Lang-Schicht

Eine reine Verwendung der Core-Klassen für Architektur-Tests ist zwar möglich, allerdings fehlt hier die Ausdrucksstärke. Um diese zu gewährleisten, bietet ArchUnit eine höhere Abstraktionsschicht, die ein Fluent-API und damit IDE-Unterstützung liefert. Das API baut auf den im letzten Absatz beschriebenen Core-Klassen auf und führt einige höhere Konzepte ein:

- *ArchRule*
  Architekturelle Konzepte sind als Regeln festgehalten
- *DescribedPredicate*
  Ein Prädikat zur Auswahl relevanter Klassen
- *ArchCondition*
  Eine Bedingung, die ausgewählte Klassen erfüllen müssen

Durch eine Kombination dieser Elemente lassen sich statische Architektur-Constraints nun beschreiben und automatisch sicherstellen, denn prinzipiell sind die meisten Regeln folgendermaßen aufgebaut: „classes that $PREDICATE should $CONDITION". Im Code könnte sich diese Idee zum Beispiel in folgender Form wiederfinden *(siehe Listing 5)*.

In diesem Beispiel lautet das Prädikat „resideInAPackage(„..service..")", das die zu untersuchenden Klassen auf solche einschränkt, die ein Paket „service" im vollqualifizierten Klassennamen haben. Die Syntax mit den „.." orientiert sich hier an „AspectJ". Die Bedingung an diese Klassen lautet wiederum „onlyBeAccessed().byAnyPackage(„..controller..", „..service..")", die ausgewählten Klassen dürfen also nur von Klassen aufgerufen werden, die im vollqualifizierten Klassennamen ein Paket „controller" oder „service" enthalten. Hat man eine Regel in der obigen Form definiert, lässt sie sich einfach gegen importierte „JavaClasses" auswerten *(siehe Listing 6)*. Bei einem Fehlschlag erhält man eine detaillierte Fehlermeldung inklusive Zeilennummer *(siehe Listing 7)*.

Library-Schicht

Oberhalb von Lang liegt die Library-Schicht. Prinzipiell ist hier eine wachsende Sammlung von typischen Standardfällen angesiedelt. Ein Beispiel ist eine vorgefertigte Sicherstellung einer Schichten-Architektur, die sich in bekannter Weise einbinden lässt *(siehe Listing 8)*. Eine weiterer typischer Anwendungsfall ist die Prüfung auf Zyklen zwischen gewissen fachlichen oder technischen Schnitten *(siehe Listing 9)*.

Hier würden Klassen nach den geklammerten Paketteilen in Schnitte gruppiert – „(**)" steht für beliebig viele Pakete – und diese dann auf Zyklenfreiheit geprüft. Beispielsweise würde die Klasse „com.myapp.order.service.OrderService" im Schnitt „order" landen, wohingegen die Klasse „com.myapp.customer.relations.service.SomeThing" dem Schnitt „customer.relations" zugeordnet werden würde. Greift nun eine beliebige Klasse aus dem Schnitt „order" auf eine Klasse aus dem Schnitt „customer.relations" zu und zudem eine beliebige Klasse aus dem Schnitt „customer.relations" auf den Schnitt „order", so schlägt die Regel fehl und meldet die Details des unerwünschten Zyklus.

JUnit

Momentan bietet ArchUnit optimierte Unterstützung für JUnit 4. Hier existiert ein spezieller Runner, der die importierten Klassen nach URLs cacht, sodass Klassen für mehrere Tests nicht jedes Mal erneut importiert werden müssen. Auch das Auswerten der Regeln muss man in diesem Fall nicht selbst durchführen. Es reicht, „ArchRule"-Felder zu deklarieren *(siehe Listing 10)*.

Eigene Konzepte mit ArchUnit definieren

ArchUnit bringt viele vorgefertigte Regeln mit, beispielsweise das vorgestellte Fluent-API, das IDE-Unterstützung für typische Anwendungsfälle wie Zugriffe von gewissen Klassen auf gewisse Klassen ermöglicht. Grundsätzlich sind diese Regeln immer in der Form „Klassen [Einschränkung, z.B. im Paket service'] sollen [Bedingung]" definiert. Das Fluent-API nutzt dabei im Zentrum ein generisches API der Form „classes.that(predicate).should(condition)". Dabei ist „predicate" vom Typ „DescribedPredicate", also ein Prädikat, das für die Regel relevante Klassen auswählt und eine Beschreibung für die Erstellung des Regeltexts mitliefert. Das Argument „condition" ist vom Typ „ArchCondition" und erlaubt die

```
@RunWith(ArchUnitRunner.class)
@AnalyzeClasses(packages = "com.myapp")
public class MyArchitectureTest {

  @ArchTest
  public static final ArchRule firstRule =
     classes().that().areAnnotatedWith(PayLoad.class)
        .should().beAssignableTo(Serializable.class);

  @ArchTest
  public static final ArchRule secondRule = // ...

}
```

Listing 10

```
DescribedPredicate<JavaClass> resideInAPackageService
= // define predicate...
ArchCondition<JavaClass> onlyBeAccessedByAnyPackage-
ControllerOrService = // define condition...

classes()
  .that(resideInAPackageService)
  .should(onlyBeAccessedByAnyPackageControllerOrServi
ce);
```

Listing 11

```
ClassesTransformer<Slice> slices = // specify how to
transform classes to slices
ArchCondition<Slice> beFreeOfCycles = // check for
cycles between slices
ArchRule slicesRule = all(slices).
should(beFreeOfCycles);
```

Listing 12

```
@ArchTest
public static final ArchRule customConcepts =
  all(businessModules()).that(dealWithOrders()).should(beIndependentOfPayment());

static ClassesTransformer<BusinessModule> businessModules() {
  return new AbstractClassesTransformer<BusinessModule>("business modules") {
     @Override
     public Iterable<BusinessModule> doTransform(JavaClasses classes) {
        // how we map classes to business modules
     }
  };
}

static DescribedPredicate<BusinessModule> dealWithOrders() {
  return new DescribedPredicate<BusinessModule>("deal with orders") {
     @Override
     public boolean apply(BusinessModule module) {
        // return true, if a business module deals with orders
     }
  };
}

static ArchCondition<BusinessModule> beIndependentOfPayment() {
  return new ArchCondition<BusinessModule>("be independent of payment") {
     @Override
     public void check(BusinessModule module, ConditionEvents events) {
        // check if the actual business module is independent of payment
     }
  };
}
```

Listing 13

Implementierung einer Bedingung an Klassen. Die vorgestellte Regel für Paketzugriffe ist lediglich eine spezielle Verwendung dieses API *(siehe Listing 11)*.

Auf diese Art und Weise lassen sich beliebige eigene Predicates und Conditions erstellen, um spezifische Architektur-Konzepte des bestehenden Projekts festzuhalten. Tatsächlich geht ArchUnit aber noch einen Schritt weiter, denn die oben vorgestellte Regel, die sich auf Zyklenfreiheit von „Slices" bezieht, basiert ebenfalls auf einem generischen API, das man für seine Architektur-Konzepte selbst verwenden kann. Die grundsätzliche Regelform bleibt auch hier erhalten, jedoch sind die Objekte, über die man sprechen will, eventuell keine Klassen. Grundsätzlich könnte man die obige Regel für Zyklen auch in folgender Form schreiben *(siehe Listing 12)*. Man kann daher genau an das eigene Projekt angepasste Konzepte einführen *(siehe Listing 13)*.

Der Regeltext wird, sofern nicht explizit überschrieben, jeweils dynamisch aus den spezifizierten Teilbeschreibungen gebildet, in diesem Beispiel wäre der berechnete Text der ausgeführten Regel also bereits „business modules that deal with orders should be independent of payment".

ArchUnit im Projektalltag

Beginnt man, Regeln mit einem Tool wie ArchUnit zu schreiben, fallen einem ganz natürlich immer weitere Anwendungsfälle auf. Beispielsweise lässt sich die fehlerhafte Nutzung von 3rd-Party-Libraries einfach für die Zukunft vermeiden, man möchte Fehler ja nur einmal machen, oder gewisse Code-Smells lassen sich aus dem Projekt verbannen, indem man gezielt gegen diese Muster testet.

Der größte Wert entsteht aber durch das Lebendighalten der Architektur. Während sich der Code-Stand unter normalen Umständen stillschweigend von der gewünschten Soll-Architektur und den gewünschten Konzepten wegentwickelt, schlagen beim Einsatz von ArchUnit tatsächlich explizit Tests im CI-Server an, beziehungsweise bereits beim lokalen Testlauf vor dem Commit. Durch diese Testfehlschläge muss man sich wirklich mit der Abweichung auseinandersetzen. Hier kann es entweder sein, dass man die bestehenden Konzepte aus Unaufmerksamkeit verletzt hat. Es ist aber durchaus auch möglich, dass die gewünschten Konzepte für den Anwendungsfall nicht mehr ausreichen. In einem solchen Fall möchte man sich allerdings explizit überlegen, wie die bestehenden Konzepte erweitert werden sollen, um derartige Anwendungsfälle abdecken zu können. Ansonsten entstehen willkürliche Muster, die dann an anderer Stelle ohne hinreichenden Grund einfach übernommen werden und sich so im System ausbreiten.

Der Erhalt einer konsistenten und sauberen Struktur des Systems ist kein Selbstzweck, sondern eine Grundvoraussetzung, um das System ab einer gewissen Größe wartbar und die Entwicklungsgeschwindigkeit konstant zu halten. Die Kosten für den Erhalt dieser Struktur wiederum sind desto geringer, je schneller man Feedback bei Verletzungen bekommt – das macht die kontinuierliche Prüfung so essenziell.

Fazit

ArchUnit *(siehe „http://www.archunit.org")* ist eine Open-Source-Bibliothek, die viele Möglichkeiten mitbringt, um architekturelle Konzepte dauerhaft sicherzustellen und lebendig zu dokumentieren. Sie ist entwicklerfreundlich und eignet sich besonders gut für größere agile Projekte, in denen Architektur-Verantwortung oft verteilt ist und viele Teams parallel an einer sich stetig weiterentwickelnden Code-Basis arbeiten.

ArchUnit benötigt keine weitere Infrastruktur und kann als Library in jedes beliebige Java-Projekt eingebunden werden, um Architektur auf Unit-Test-Ebene sicherzustellen. Zudem bietet ArchUnit ein mächtiges API, mit dem nicht nur auf viele Eigenschaften im Java-Byte-Code reagiert werden kann, sondern auch auf abstrakter Ebene Architektur-Konzepte dokumentiert und automatisiert getestet werden können.

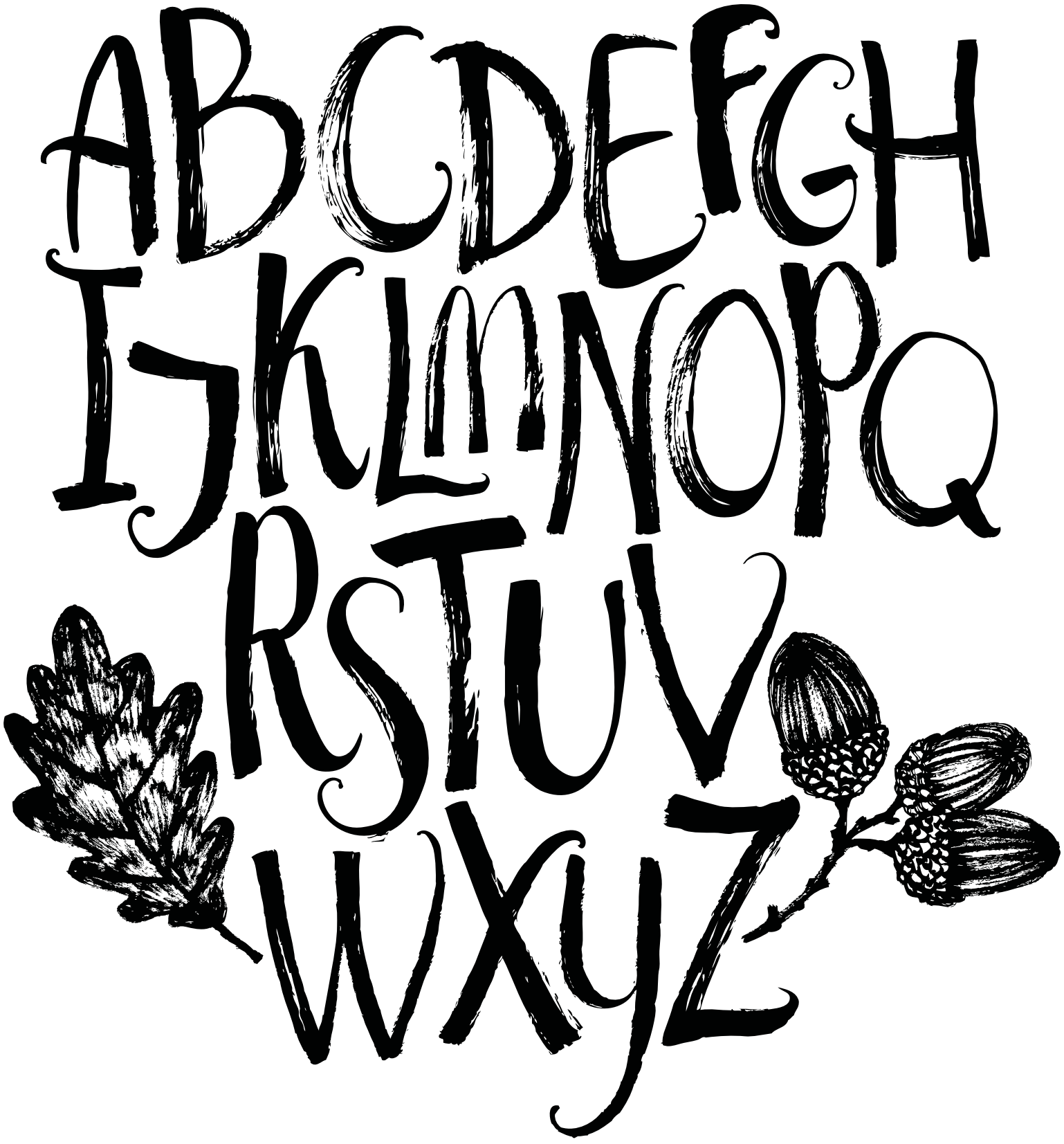
Abschließend sei erwähnt, dass sich automatisierte statische Architekturtests zur Architektur ähnlich wie Test-Coverage zu Testqualität verhalten. So wie 100 Prozent Test-Coverage in keiner Weise garantiert, dass ein System gut getestet ist, so garantiert ein erfülltes System von statischen Architektur-Regeln nicht, dass die Architektur gut ist und Ziele wie etwa geringe Kopplung erfüllt sind. Schließlich kann man beispielsweise sämtliche Aufrufe durch Nutzung des Reflection-API ersetzen, und man wird alle statischen Regeln bezüglich Abhängigkeiten erfüllen. Allerdings gilt im Umkehrschluss entsprechend der Tatsache, dass 0 Prozent Test-Coverage ein Garant für ein schlecht getestetes System ist, dass eine Verletzung der Architektur auf statischer Ebene mit Sicherheit auf ein System hinweist, das die gewollten Architektur-Ziele verfehlt. Insofern kann man die statische Analyse als grundsätzlichen Sanity-Check verstehen, der auf dieser Ebene einen großen Wert mit sich bringt – insbesondere für große Systeme und inhomogene Entwicklungsteams.

Peter Gafert
peter.gafert@tngtech.com

Peter Gafert ist Senior Consultant bei der TNG Technology Consulting GmbH und beschäftigt sich im Projektalltag viel mit Software-Architektur. Um den täglichen Umgang mit Architektur zu verbessern, entwickelt er nebenher die Open-Source-Library ArchUnit.

Von Java Swing zu JavaFX – ein Textsystem macht sich selbstständig

Falk Tengler, dvhaus Software & Solutions GmbH und Vasily Smeltsov, Intechcore GmbH



Bislang existierte in der UI-Technologie Swing mit dem RTFEditorKit eine Komponente, um formatierten Text darzustellen und zu verwalten. In der Nachfolge-Technologie JavaFX steht diese Funktionalität nicht zur Verfügung. In der öffentlichen Verwaltung gibt es demgegenüber konkreten Bedarf nach einer Plattform-unabhängigen, SOA-unterstützenden Textverarbeitung.

Lösung ist eine Java-basierte Textverarbeitung, die auf Microsoft-Open-XML-Standards aufsetzt, alle Features dieser Standards unterstützt und im Headless-Mode automatisiert Textverarbeitung durchführen kann. Der Artikel zeigt, welche Architektur zur Verwaltung von DOM, Dispatcher, ParserWriter etc. notwendig ist, um eine solche Herausforderung zu meistern.

Hintergrund

Der Auftraggeber – eine große Organisation in der öffentlichen Verwaltung – benötigt für mehr als 20.000 seiner Mitarbeiter eine Anwendung für Textverarbeitung, die von Standard-Produkten anderer Hersteller unabhängig bleibt und gleichzeitig die umfangreichen und komplexen Anforderungen abbildet. So soll es neben den bekannten Funktionen wie Formatierung, Bilder, Tabellen etc. spezifische fachliche Funktionen geben. Darüber hinaus soll es möglich sein, Hintergrundverarbeitung (auf SOA-Servern) durchzuführen, Reinschriften zu produzieren, XML-Dokumente in DOCX, PDF, PDF/A und andere Formate zu konvertieren, Dokumente zu signieren sowie Versandpakete mit Fach-Informationen zu erstellen. Das Ergebnis: ein ergonomisches und barrierefrei bedienbares Textsystem, das sowohl auf Windows als auch auf Mac-OS-Clientsystemen ablauffähig ist und sich mit Office-Produkten, die auf dem Markt sind, messen kann.

Mit der Aufgabe, ein komfortables Textsystem zu programmieren, stellte sich den Software-Entwicklern eine Vielzahl von Fragen, deren Antworten elementar für die erfolgreiche Umsetzung waren. Dieser Fragenkatalog macht die Vorgehensweise der Entwickler Schritt für Schritt nachvollziehbar und soll gleichzeitig strukturgebend für dieses Dokument sein.

Vorrangige Bedeutung hat die Festlegung der Definition dessen, was ein Dokument charakterisiert. Für die Umsetzung der konkreten Aufgabenstellung wurde festgelegt, dass ein Dokument in erster Linie aus Text besteht und auf technischer Ebene aus folgenden Elementen:

- Symbole, Bilder, Barcodes
- Absätze
- Tabellen
- Kopf- und Fußzeilen

Der Editor

Kernstück des Textverarbeitungssystems ist der Editor, Kernstück des Editors wiederum das dafür entwickelte Steuerelement FXEditorControl. Menü- und Ribbon-Leiste, Statuszeile etc. werden in der vorliegenden Betrachtung außer Acht gelassen. Die Architektur des Editors basiert auf dem Model-View-Controller-Entwurfsmuster (MVC): Controller („FXEditorController"), Model („FXEditorModel"), View („RichTextAreaSkin") und RichTextArea – enthalten im Controller *(siehe Abbildung 1)*. Die View „RichTextAreaSkin" verantwortet die Darstellung auf „javafx.scene.canvas.Canvas".

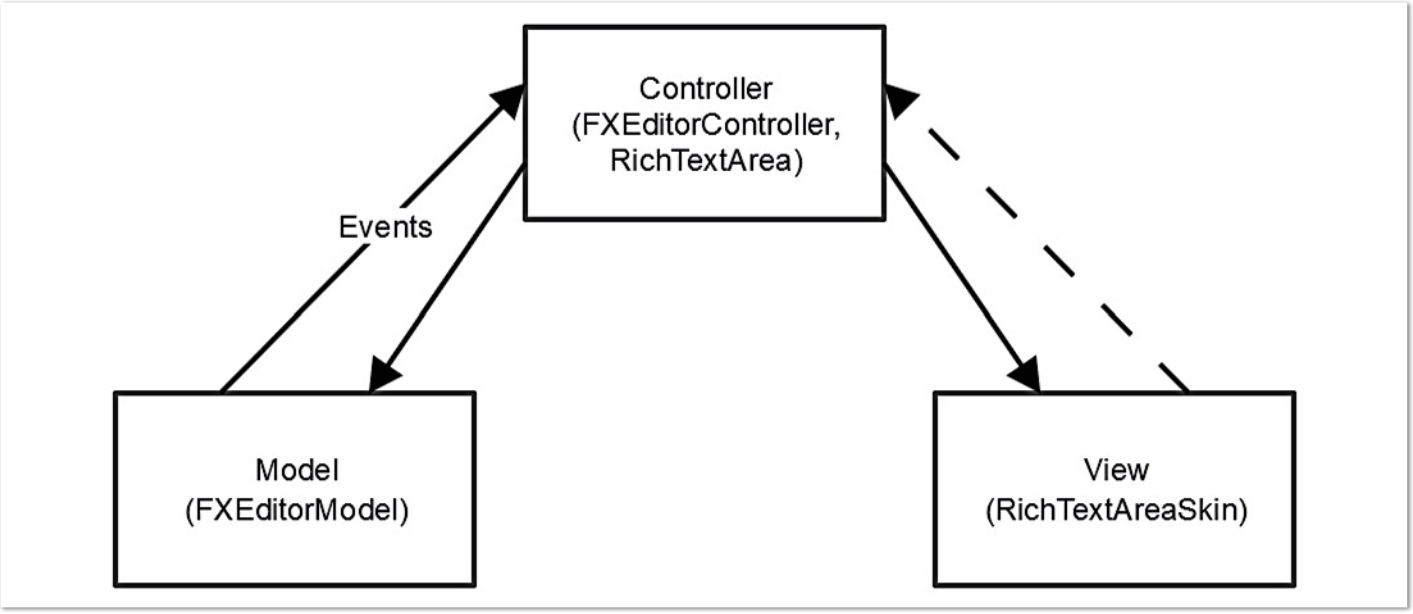


Abbildung 1: FXEditorControl – MVC-Entwurfsmuster mit konkreten Repräsentanten

Die Komponenten

Mit welchen JavaFX-Komponenten kann im Editor gearbeitet werden, um beispielsweise Text, Seiten oder Scroll-Balken anzuzeigen, und wie wirken diese Komponenten zusammen *(siehe Abbildung 2)*? „FXEditorControl" besteht aus den untergeordneten Steuerelementen „Ruler", „RichTextArea" sowie den horizontalen und vertikalen Scroll-Balken. Die Klassen „Scrollbar" und „Canvas" aus dem Package „javafx.scene.canvas" werden unverändert verwendet. „Ruler" und „RichTextArea", abgeleitet von „javafx.scene.Node", stellen Spezifikationen der entsprechenden JavaFX-Standardklassen dar.

Das Zusammenwirken der verwendeten Komponenten gewährleistet, dass bei der Erstellung eines „FXEditorControl" (inklusive „Ruler"), der „Ruler" auch von „RichTextArea" angesprochen werden kann. Das Code-Fragment aus „createRichTextArea()" verdeutlicht dies *(siehe Listing 1)*. Der Ruler ist in „RichTextArea" enthalten und kennt die „RichTextArea", zu der er gehört.

Die gespeicherten Daten

Das dem Editor zugrunde liegende Datenmodell („FXEditorModel") ist ein proprietäres, XML-basiertes und an OpenXML angelehntes Modell. Dokumente können als DOCX (OpenXML) oder RTF exportiert werden. Dokument-Elemente, Nummerierung, Formatvorlagen, Kopf- und Fußzeilen werden – vergleichbar mit OpenXML – separat in sogenannten Storages gespeichert *(siehe Abbildungen 3 und 4)*.

„FXEditorModel" beinhaltet alle für das Laden, Anzeigen und Speichern eines Dokuments erforderlichen Daten eines Dokuments. Auf Canvas tatsächlich dargestellt werden nur bestimmte Elemente, wie zum Beispiel Seiten mit oder ohne Kopf- und Fußzeilen, Absätze, Tabellen, Formatierungen oder auch Bilder. Charakteristisch für das Modell sind die von der Klasse „Element" abgeleiteten Klassen „Paragraph", „Table" und „ActiveTable". Alle Elemente liegen in einem „elementStorage", das intern als doppelt verkettete Liste mit Verweis auf benachbarte Elemente (double linked list) realisiert ist *(siehe Abbildung 5)*. Aus Performance-Gründen ist diese Liste in einer „Java.util.HashMap" abgelegt.

Informationen zu den jeweiligen Elementen auf einer Seite innerhalb des Dokuments sind im „PageStorage" gesichert. Bei jeder wesentlichen Änderung des Dokuments werden diese Informationen neu berechnet und sowohl für die Dokument-Größe als auch für die Berechnung des Scroller verwendet. Ändert der Anwender den Scroller (Stellrad am Scroll-Balken), wird die Information so verarbeitet, dass die jeweilige Seite mit den entsprechenden Elementen angezeigt werden kann.

Das Event-Handling

Alle Ereignisse oder Events, ausgelöst durch Tastatur, Maus oder auch Sprachbefehl, werden in der Hauptsteuerungsklasse „RichTextArea" verarbeitet. Sie enthält unter anderem eine Variable mit Informationen über die aktuelle Cursor-Position. Die neue Position wird auf Grundlage von Informationen aus der View bestimmt. Anhand des Modells lassen sich zu jeder Zeit die Cursor-Position und das an der Position liegende Steuerelement bestimmen.

Eine Vielzahl weiterer Ereignisse wird beispielsweise durch „Scroller

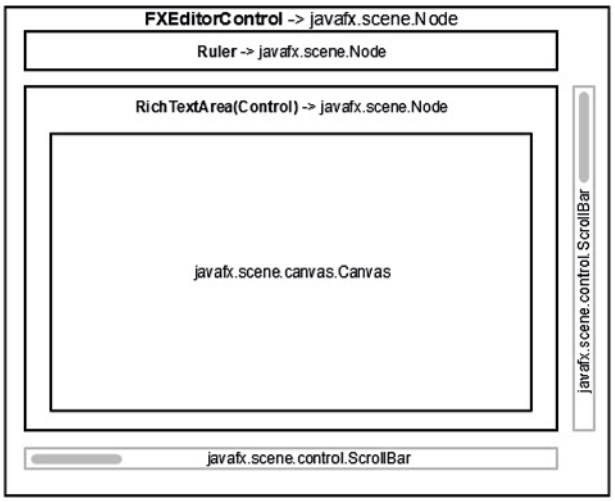


Abbildung 2: Editor-Bereiche mit zugehörigen Komponenten

```
public void createRichTextArea() {
  model.setDrawingFlag(false);
  richTextArea = new RichTextArea(model, ruler);
  ruler.setRichTextArea(richTextArea);
```

Listing 1

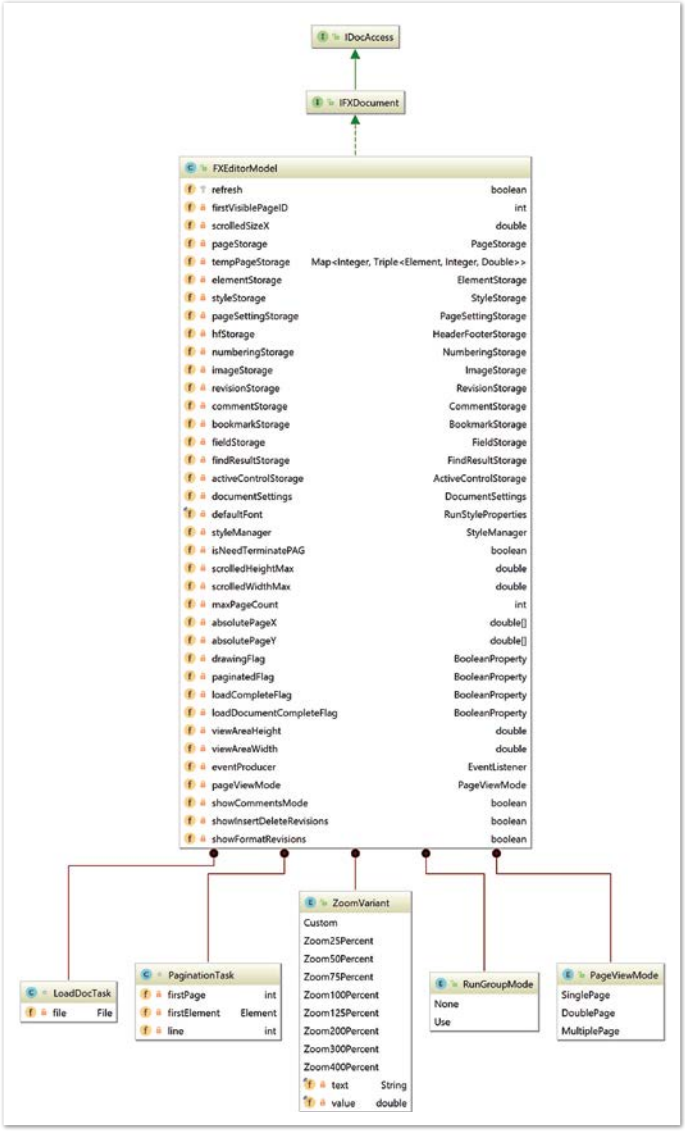


Abbildung 3: Ausschnitt aus dem Editor-Datenmodell

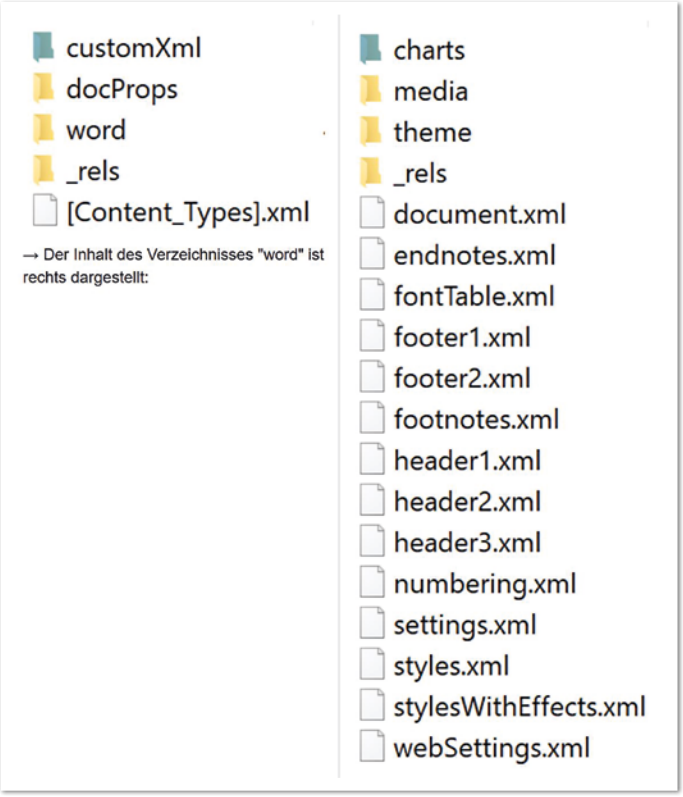


Abbildung 4: Vergleich Word-Verzeichnis und extrahierte Struktur eines DOCX-Dokuments

bewegt" oder „Ruler verändert" ausgelöst. Alle möglichen Ereignisse sind im Modell definiert. Die Ereignisbehandlung erfolgt im Controller, der die Verarbeitung gegebenenfalls an ein untergeordnetes Steuerelement delegiert – zum Beispiel an den Ruler.

Der Text wird gerendert

Texte, Formatierungen etc. werden mithilfe der Klasse „RichString" bearbeitet und für die Darstellung auf Canvas gerendert. Die Klasse „RichString" repräsentiert eine formatierte Zeichenfolge, die im Allgemeinen einen Satz unterschiedlich formatierter Textteile enthält (jeweils Text und Formatierung). Substrings mit gleicher Formatierung werden als „Run" bezeichnet.

Bei mehrzeiligen Text-Fragmenten stehen dem Entwickler Informationen zu formatierten Textteilen zeilenweise zur Verfügung. Sub-Strings werden zeilenweise bearbeitet. Die Hilfsklasse „RichStringBuilder" erstellt, ändert und löscht Zeichenketten. Dimensionen von Zeichenketten werden mithilfe der Klassen „JFXFontMetricsFactory" und „DefaultFontMetrics" bestimmt (Spezialisierungen von „com.sun.Javafx.tk.FontMetrics"), Höhe und Breite (in pt) des Textes in den Methoden „computeStringWidthPt()" und „getLineHeigthPt()" der Klasse „DefaultFontMetrics" berechnet. Da das Laden von Fonts unter Java rechenintensiv ist, wurde hierfür ein Cache implementiert.

Die Textdarstellung

Kurz noch mal rekapituliert: Der Editor wird vom Steuerelement „FXEditorControl" repräsentiert, das als MVC-Pattern realisiert ist und „RichTextAreaSkin" als View enthält. Diese View korrespondiert mit dem im „FXEditorControl" eingebetteten Steuerelement „RichTextAreaControl", das, ebenfalls intern als MVC-Pattern realisiert, die Ausgabe auf die Zeichenebene („Canvas") steuert: Der „ContentController" (Controller) vermittelt zwischen „CanvasModel" (Modell) und „javafx.scene.canfas.Canvas" (View). Es ist letztlich diese View, die die grafische Repräsentation des Modells innerhalb des Editors verkörpert *(siehe Abbildung 6)*.

„CanvasModel" berechnet in einem eigenen Thread (also asynchron) fortlaufend die Seiteneinteilung des Dokuments. Alle Dokument-Elemente (Absatztexte, Bilder, Tabellen etc.) werden fortlaufend analysiert und im Rahmen eines mathematischen Modells hinsichtlich ihrer Höhe und Breite bewertet. Auf diese Weise ist parallel zur Bearbeitung des Dokuments die Seiteneinteilung aktuell gehalten. Die Berechnung umfasst dabei nicht nur denjenigen Dokument-Ausschnitt, der aktuell im Canvas angezeigt wird (inklusive einige Zeilen davor und danach), sondern auch Dokument-Teile, die weitab vom aktuellen Fokus liegen (Caching): Wird zum Beispiel Seite 3 angezeigt, werden auch die Seiten 4 bis 38

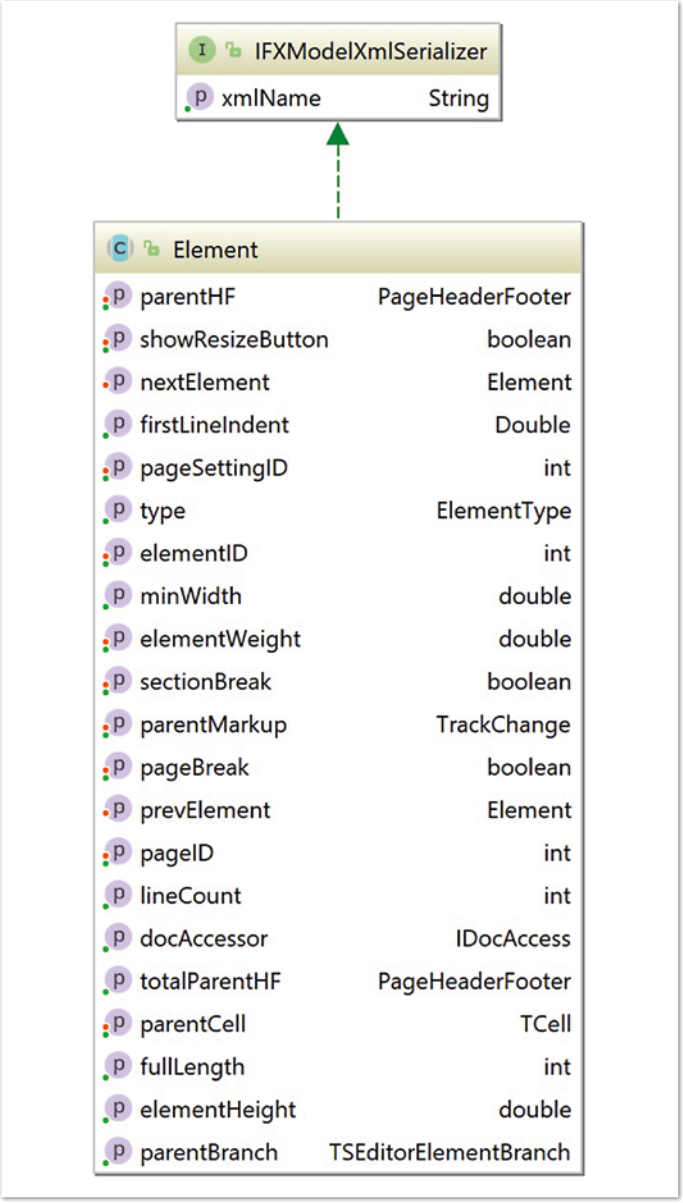


Abbildung 5: Klasse „Element" mit doppelt verketteter Liste („prevElement", „nextElement")

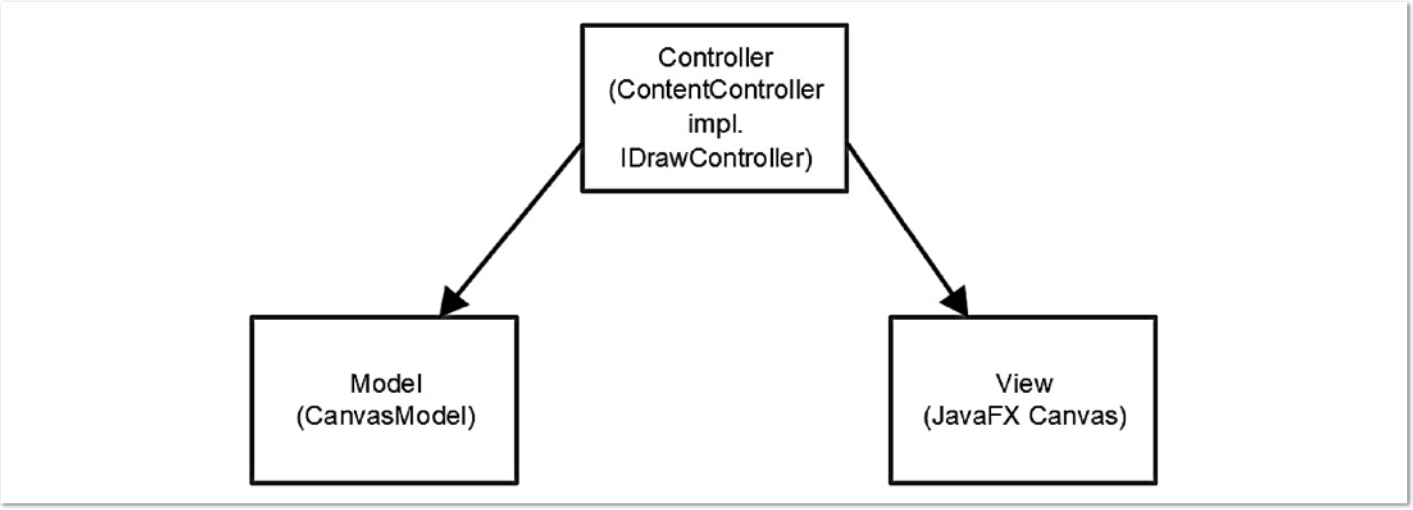


Abbildung 6: Aufbau „RichTextAreaSkin" – View „RichTextAreaControl" und internal MVC-Pattern

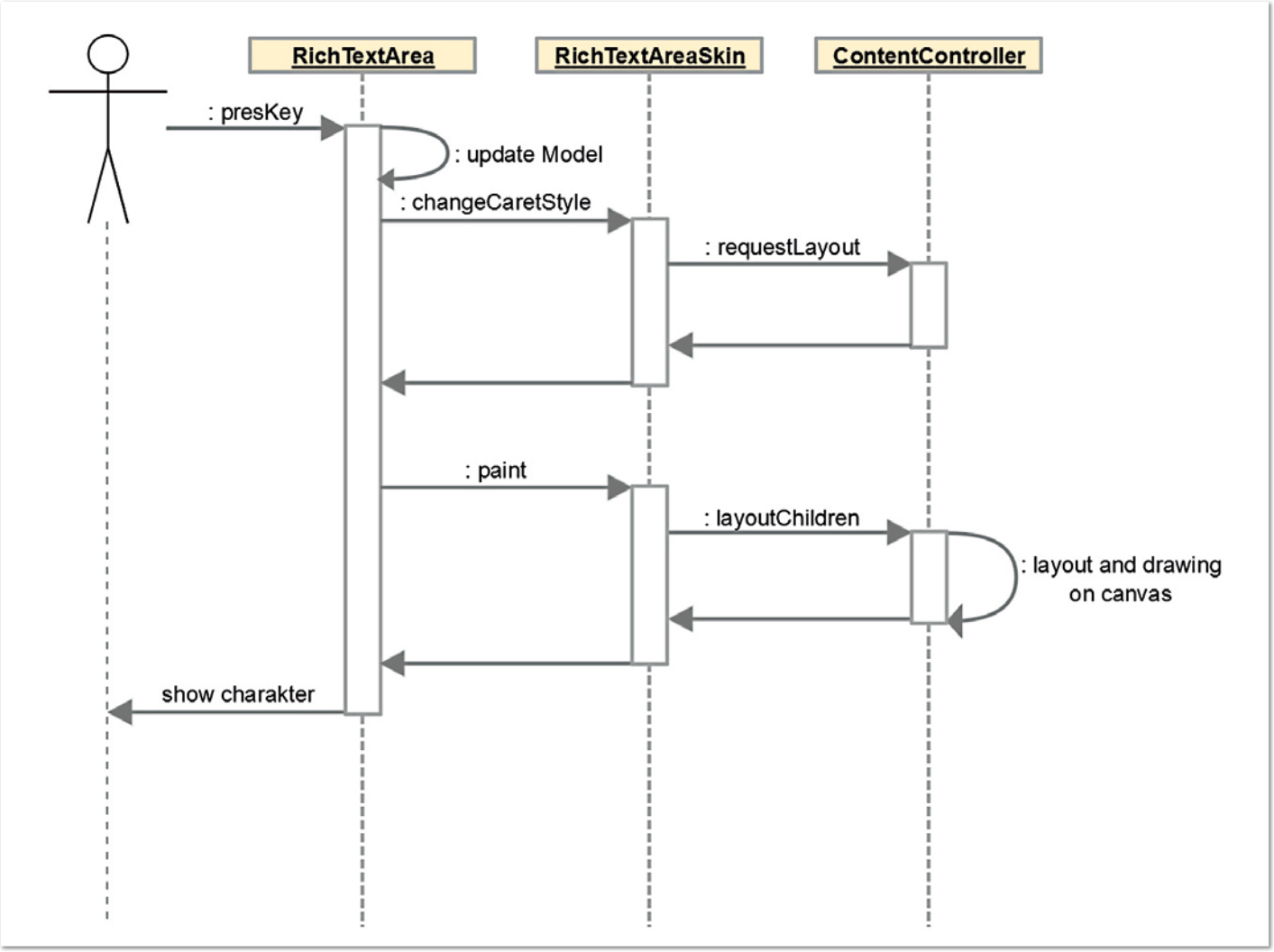


Abbildung 7: Schematische Darstellung der Komponenten-Interaktion

berechnet. Das Caching ermöglicht also eine verzögerungsfreie Anzeige des Textes und schnelles Blättern auch in großen Dokumenten. Die View basiert auf „javafx.scene.canvas.Canvas". Die Darstellung selbst erfolgt mit „javafx.scene.canvas.GraphicsContext". Die Berechnung und Darstellung von Tabellen, Nummerierungen sowie Kopf- und Fußzeilen erfolgt analog. Entscheidend ist lediglich, an welcher Stelle des Dokuments das jeweilige Ereignis ausgelöst wird. Nummerierte Absätze beziehungsweise Aufzählungen werden grundsätzlich – genau wie Text – mit „RichStringBuilder" ausgewertet. Nur die Verwaltung der Nummerierungs- beziehungsweise Aufzählungsebenen und die Formatierung der Nummern sind in das Package Numbering ausgelagert.

Das Zusammenspiel der Komponenten bei Änderungen

Die Komponenten-Interaktion verdeutlicht, welche Prozesse initiiert werden, wenn beispielsweise Buchstaben oder Text einfügt oder gelöscht oder auch wenn Events wie „Absatz erstellen", „Textbereich auswählen", „Schriftart ändern" oder Vergleichbares

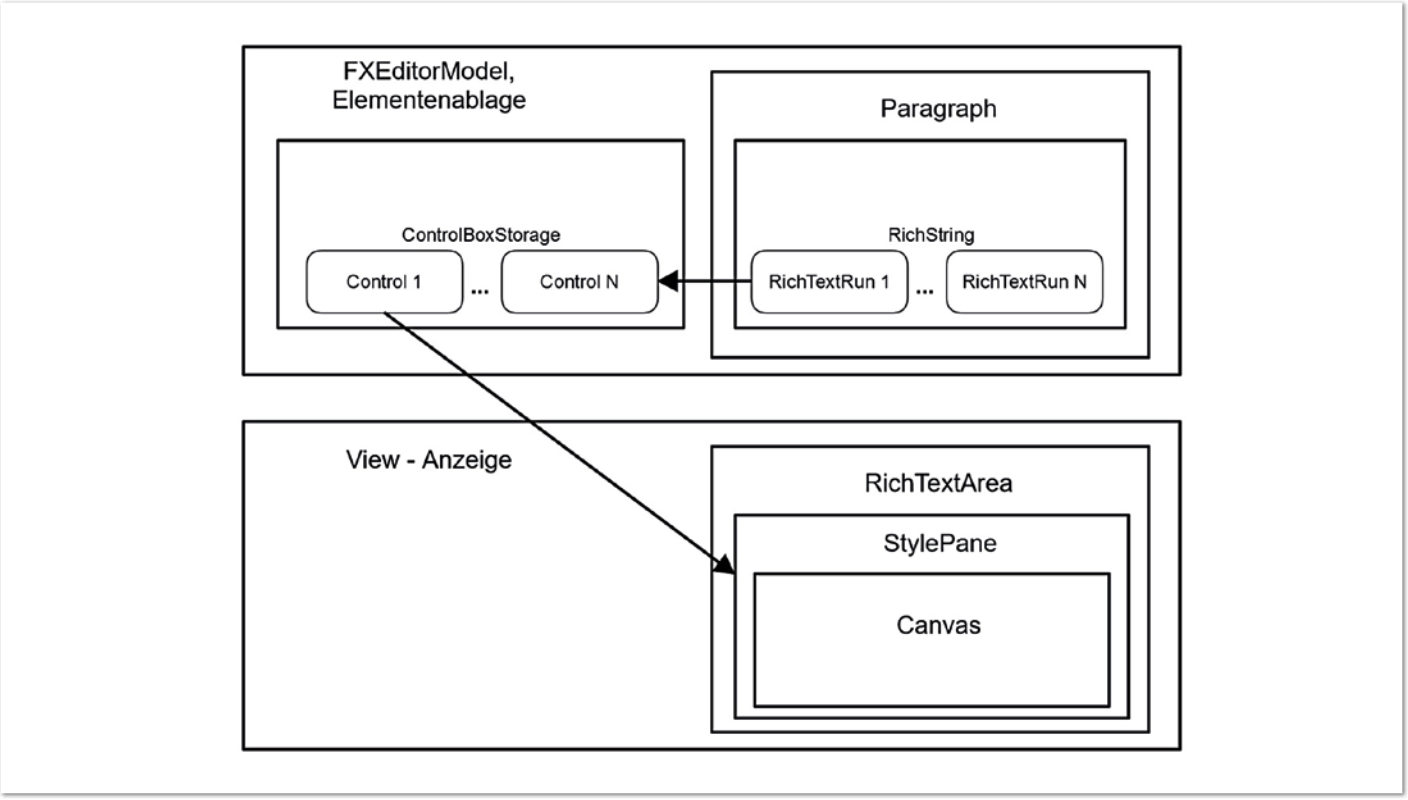


Abbildung 8: Allgemeines Arbeitsschema mit einfachen „ActiveControl"

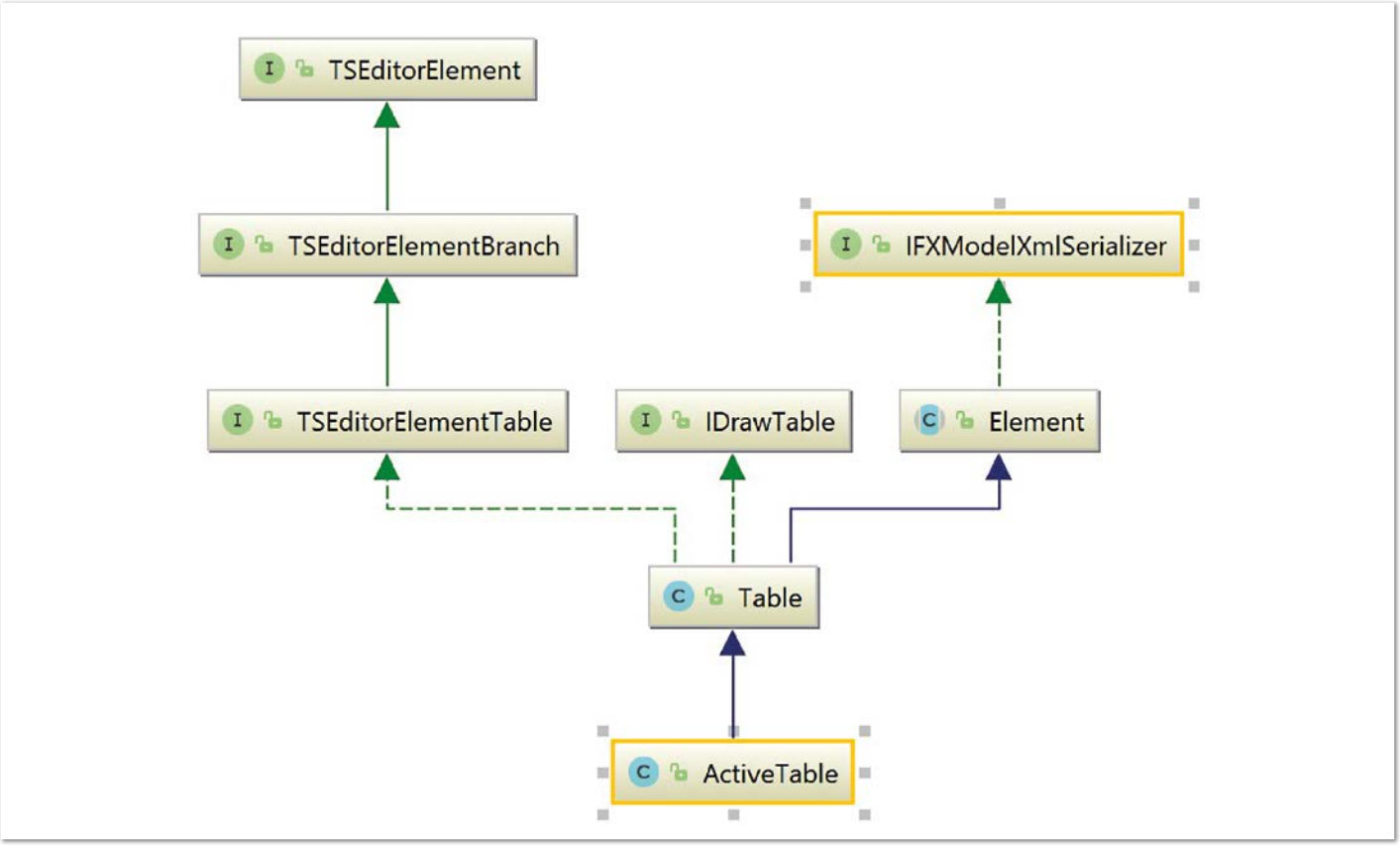


Abbildung 9: Darstellung „ActiveTable"-Steuerelement

ausgelöst werden *(siehe Abbildung 7)*. In Abhängigkeit von der Position, des jeweiligen Event-Beginns, kommen unterschiedliche Prozessoren in Spiel. So kommt beispielsweise für die Behandlung der Maus ein „MouseEvent" zum Einsatz: Ein „MouseEvent" in der Tabelle kann „Cursor" für Tabellengrößen-Änderungen beeinflussen. Dabei kann auch die Anzeige des „Ruler" an die Änderung angepasst werden.

Die Darstellung mehrseitiger Dokumente

Grundsätzlich kann Text sowohl in mehreren Absätzen auf mehreren Seiten als auch in einem Absatz über mehrere Seiten oder in Tabellen (Zeilen oder Spalten) laufen. Für die Darstellung jeder möglichen Variante wurde der Pagination-Mechanismus entwickelt. Dieser verarbeitet die entsprechenden Informationen aus dem Modell und prüft, ob der jeweilige Text auf die entsprechende Seite passt oder eine neue Seite erstellt werden muss. Da diese Operation für größere Dokumente sehr aufwendig sein kann, wird die Pagination nur für die sichtbaren Seiten (+/- 2 Seiten) des Dokuments berechnet. Für die Pagination-Berechnungen darüber hinaus startet ein Background-Prozess. Ergeben die Berechnungen, dass eine neue Seite notwendig ist, werden Formatierung der Absätze und sonstige Seiteneigenschaften kopiert, Schriftarten berechnet und auf die neue Seite übertragen.

Die Umsetzung aktiver Komponenten

Grundsätzlich wird zwischen drei Typen aktiver Steuer-Elemente unterschieden *(siehe Abbildung 8)*:

- *Umschaltbar*
  - „CheckBox" – Kontrollkästchen/Häkchen
  - „RadioButton" – Schalter
- *Editierbar*
  - „ComboBox" – Dropdown-Liste Auswahl Wert/e mit/ohne Dateneingabe
  - „DatePicker" – Auswahl aus Dropdown-Kalender mit freier Dateneingabe
  - „TimePicker" – Eingabe der Zeit mit „+/-„-Auswahl oder freie Eingabe
- *Tabelle*
  - „ActiveTableRowControl" – Eingabe Zeichenfolge mit „ComboBox"-Logik oder herkömmlichen Spalten, mit „LogicBorderElem"-Zeichenfolge aus Textsystem

Steuer-Elemente (Controls) sind in „controlBoxStorage" gespeichert, wo sie ihre eindeutige „CtrlId" erhalten. „RichString" und entsprechende einzelne Runs werden in Daten-Arrays gespeichert. Da die Daten durch „RichStringBuilder" geändert und verarbeitet werden, wird jeder Run von der Klasse „RichTextRun" beschrieben. Diese kann unterschiedlichen Typs sein (Tabulator, Steuerelement, Bild und Text).

Die Steuer-Elemente werden auf dem Canvas des Editors gerendert. Während der Platzierung des Absatzes auf der gerenderten Seite wird die Position des Steuerelements berechnet. Dargestellt ist, was zu dem Control gehört: etwa ein Rahmen, eine Schaltfläche oder auch Listen-/Datums-Auswahl in Dropdown-Listen.

Zur Behandlung von Ereignissen werden diese Komponenten dem „StylePanel" (JavaFX-Panel) hinzugefügt. Wenn die Position/der Maßstab der Dokument-Seite sich ändert, wird die Position des geöffneten Elements neu berechnet. Mit dem Schließen des Elements wird das Element nicht mehr angezeigt (Anzeige wird gelöscht) und der Absatz wird aktualisiert.

Das „ActiveTable"-Steuerelement ist zur Eingabe von Listen typisierter Daten vorgesehen *(siehe Abbildung 9)*. Die Spalten der Tabelle geben den Anzeigetyp an (plain text/Filter-Drop-down-Liste). Jedes Control hat einen „EventHandler", der ein Ereignis an das „FXEditorModel" weiterleitet. Für die tabellarischen Controls ist „ActiveControlStorage" im Modell des Editors vorgesehen. Jede Zeile in „ActiveTable" verweist auf „ActiveControlStorage".

In einem Prototyp werden die Umsetzungsideen visualisiert sowie konzipierte Workflows und Barrierefreiheit getestet. Im Zuge einer hohen Anwenderakzeptanz wurde der Prototyp auch künftigen Nutzern der Anwendung vorgestellt. Nun steigt der Termindruck: Die Nutzer wollen die Anwendung am liebsten heute in die Praxis einführen.

Falk Tengler
ft@dvhaus.de

Falk Tengler, Geschäftsführer der dvhaus Software & Solutions GmbH und Projektleiter, ist Spezialist für die Handhabung großer IT-Projekte bei großen und mittelständischen Unternehmen, aber insbesondere bei Behörden und öffentlichen Institutionen. Seine fachliche Kompetenz in Kombination mit seinem tiefen Verständnis für Kundenanforderungen – und seien sie noch so spezifisch – machen ihn zu einem Dienstleister und Projektpartner auf Augenhöhe, mit dem die Kunden auch über mehrere Jahre sehr gern zusammen arbeiten. Textverarbeitung ist sein besonderes Steckenpferd. Falk Tengler, der seit mehr als zwanzig Jahren in der IT-Branche tätig ist, hat in den letzten Jahren sehr intensiv konzeptionell an der Entwicklung und der technischen Realisation eines Editors mitgewirkt, der sich mit auf dem Markt gängigen Textverarbeitungssystemen messen lassen kann.

iJUG Verbund

Alle Mitglieder auf einen Blick

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

www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, www.ijug.eu) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissenssaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Stefan Kinnen. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
Chefredakteur (ViSdP): Wolfgang Taschner
Kontakt: redaktion@doag.org

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt; Daniel van Ross, FIZ Karlsruhe; André Sept, Freiberufler; Jan Diller, Triestram und Partner

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Fotonachweis:
S. 12: © venimo/123RF
S. 16: © Aleksandra Sabelskaia/123RF
S. 25: © Alexander Tolstykh/123RF
S. 31: © Pichaya Punyakhetpimook/123RF
S. 48: © Jirawat Phueksriphan/123RF
S. 54: © Tommaso Altamura/123RF
S. 59: © nenilkime/123RF

Anzeigen:
Simone Fischer, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

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

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

Alle Rechte vorbehalten. Jegliche Vervielfältigung oder Weiterverbreitung in jedem Medium als Ganzes oder in Teilen bedarf der schriftlichen Zustimmung des Verlags.

Die Informationen und Angaben in dieser Publikation wurden nach bestem Wissen und Gewissen recherchiert. Die Nutzung dieser Informationen und Angaben geschieht allein auf eigene Verantwortung. Eine Haftung für die Richtigkeit der Informationen und Angaben, insbesondere für die Anwendbarkeit im Einzelfall, wird nicht übernommen. Meinungen stellen die Ansichten der jeweiligen Autoren dar und geben nicht notwendigerweise die Ansicht der Herausgeber wieder.

Inserentenverzeichnis

Accenture Technology Solutionss GmbH	S.37
BROCKHAUS AG	S.35
Deutsche Welle	S.41
DOAG e.V.	U2, U3
NovaTec Consulting GmbH	S.29
TimoCom Soft- und Hardware GmbH	S.53
Zertificon Solutions GmbH	U 4