04-2015 | Winter | www.ijug.eu

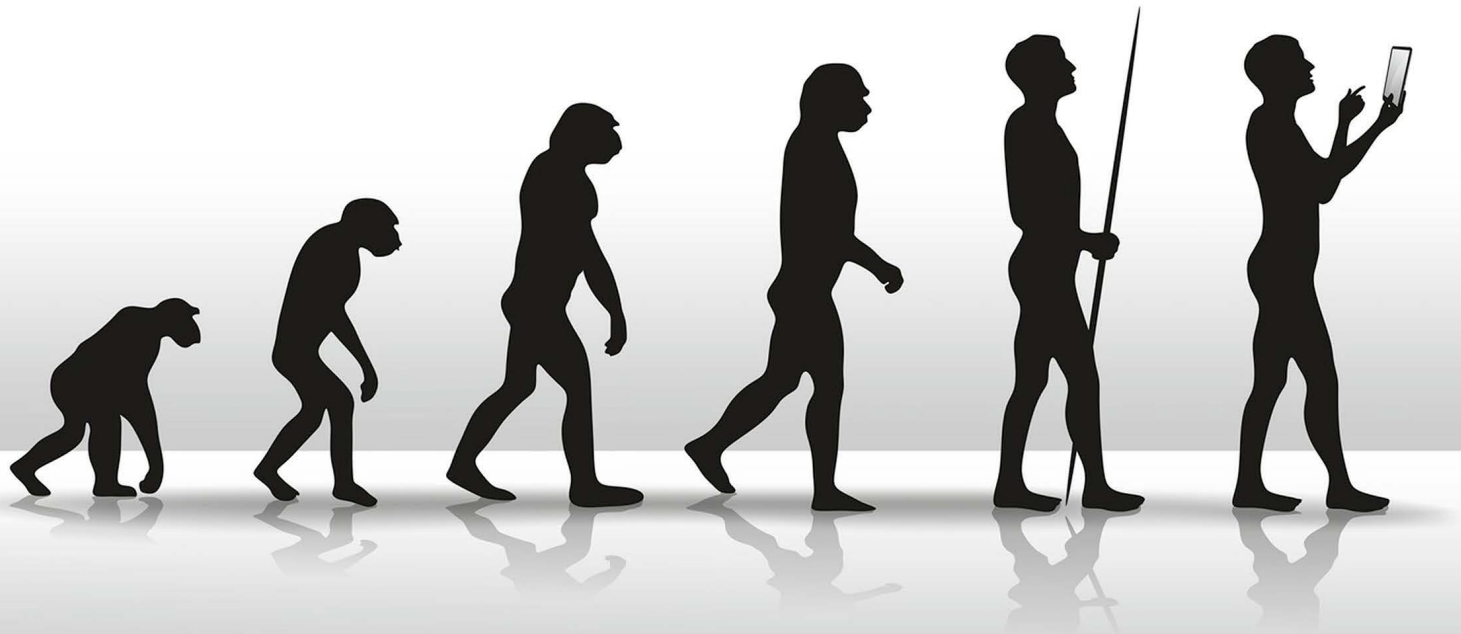
Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Java entwickelt sich weiter



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

Neue Frameworks
AspectJ, Eclipse Scout, Citrus

Raspberry Pi
Projekte mit Java

Java-Performance
Durch Parallelität verbessern

Web-Anwendungen
Hochverfügbar und performant

iJUG Verbund

4 191978 304903 04

Eventpartner: AOUG Austrian Oracle User Group · SOUG Swiss Oracle User Group · ORACLE · iJUG Verbund

2015.doag.org

Wolfgang Taschner
Chefredakteur Java aktuell

„Like" oder „Dislike" für die Java aktuell

Die Java aktuell zählt mittlerweile zu den größten Java-Magazinen im deutschsprachigen Raum. Darüber hinaus ist die Zeitschrift eine der wichtigsten Säulen im Interessenverbund der Java User Groups e.V. (iJUG). In dem nach der Sun-Übernahme durch Oracle gegründeten Verein sind bereits mehr als 25 Java User Groups zusammengeschlossen, um die Interessen der Java-Community im deutschsprachigen Raum zu vertreten *(siehe Seite 66)*.

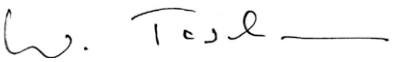
Das fünfjährige Jubiläum der Java aktuell wurde Anfang des Jahres im Anschluss an die JavaLand 2015 gefeiert *(siehe „http://www.doag.org/go/5jahre_javaaktuell")*. Bei Networking und gutem Essen ließen die Mitglieder der Usergroups, Vertreter von Oracle und einige Autoren die letzten fünf Jahre Revue passieren.

Dank des Engagements der Autoren (vielen Dank!) kann ich in jeder Ausgabe aktuelle und hochinteressante Artikel veröffentlichen. Die Leser-Feedbacks zu den einzelnen Artikeln haben geholfen, die Qualität des Inhalts und die Themenauswahl zu verbessern. Jetzt geht es mir um eine generelle Frage zur Java aktuell: „Erfüllt diese Ausgabe der Java aktuell Ihre Bedürfnisse?" Sie können über die URL *„http://ja.ijug.eu/15/4/1"* oder den QR-Code mit „ja" oder „nein" stimmen und natürlich dort auch die Zeitschrift entsprechend kommentieren.

Ich werde öfter gefragt, ob es auch eine digitale Ausgabe der Java aktuell gibt. Hier die Antwort: Die Zeitschrift ist als PDF zum Download für Interessenten der DOAG Deutsche ORACLE-Anwendergruppe e.V. verfügbar. Das heißt, man muss sich zuvor als „Interessent" bei der DOAG *(siehe „http://www.doag.org/?id=289")* registrieren und kann dann über die DOAG-Webseite alle früheren Ausgaben als PDF herunterladen. Lediglich die aktuelle Ausgabe bleibt den Abonnenten und iJUG-Mitgliedern vorbehalten.

Ich freue mich wie immer auf Ihr Feedback, auch an *redaktion@ijug.eu*, und wünsche Ihnen viel Spaß beim Lesen dieser Ausgabe und viel Erfolg bei Ihren Java-Projekten.

Ihr



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

Die Position „Software-Architekt" ist in der Software-Branche etabliert

Java-Champion Kirk Pepperdine gibt Performance-Tipps

Citrus bietet komplexe Integrationstests mit mehreren Schnittstellen

Das Java-Tagebuch

Andreas Badelt, Leiter der DOAG SIG Java

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im zweiten Quartal 2015.

14. April 2015

JSRs zum Anfassen: Java-Security-Session im JavaLand 2015

Der Java-Security-JSR legt jetzt richtig los. Ein Teil der Expert Group war ja im März im JavaLand 2015, um ein paar Ideen zu präsentieren und zu diskutieren, auf der Devoxx France wurden dann letzte Woche die Pläne vorgestellt. Um Ideen unter Beteiligung der Community auszuprobieren, gibt es jetzt einen Playground. Also: Klonen, ausprobieren und Feedback geben.

github.com/javaee-security-spec/javaee-security-proposals

21. April 2015

JBoss EAP 6.4 ist raus – und unterstützt Java 8

JBoss EAP Version 6.4 ist da – und unterstützt jetzt offiziell Java 8 (Oracle und IBM JDK) – rechtzeitig für die Abkündigung des kostenfreien Supports von Java 7 durch Oracle Ende April 2015.

http://www.jboss.org/products/eap/download

5. Mai 2015

Java EE 7 Maintenance Review

Das Maintenance Release für Java EE 7 liegt im „final draft" vor. Die Änderungen beziehen sich, wie zu erwarten, nur auf die Spezifikation selbst, einzelne Formulierungen werden präzisiert. Auswirkungen auf die Referenz-Implementierung haben sie nicht.

https://www.jcp.org/en/jsr/detail?id=336

6. Mai 2015

Red Hat ist strategisches Mitglied der Eclipse Foundation

Red Hat hat schon länger bei der Eclipse Foundation mitgewirkt, jetzt werden sie strategisches Mitglied, mit einem Sitz im Board of Directors. Damit reihen sie sich zwischen den anderen großen Java-Firmen wie IBM, SAP und natürlich Oracle ein. Auf der einen Seite bedeutet das mehr Verpflichtungen, so werden mindestens acht Vollzeit-Entwickler den Eclipse-Projekten als Mitgift zur Verfügung gestellt. Auf der anderen Seite bringt es aber natürlich deutlich mehr Einfluss in einer der wichtigsten Vereinigungen der Java-Welt, die längst über die IDE hinausgewachsen ist.

https://www.eclipse.org/org/press-release/20150505_redhat.php

12. Mai 2015

Mark Reinhold zum Java-9-Fahrplan

Mark Reinhold, Chef-Architekt der Java-Plattform, hat den Zeitplan für Java SE 9 konkretisiert. Der Meilenstein „Feature Complete" soll bis zum 10. Dezember 2015 erreicht werden. In mehreren Schritten soll dann der „Final Release Candidate" bis zum 21. Juli 2016 entstehen. Die offizielle Freigabe ist für den 22. September 2016 geplant. Wer bis dahin nicht warten kann, findet genügend Möglichkeiten von der Preview bis zur aktiven Mitarbeit im Rahmen des Adopt-OpenJDK-Programms.

https://blogs.oracle.com/java/entry/java_9_schedule_is_out

18. Mai 2015

Lebt der GlassFish noch?

Manfred Riem von Oracle, unter anderem Specification Lead für JavaServer Faces und die zukünftige Alternative MVC, hat in seinem Blog betont, dass der GlassFish nicht tot ist, da seine Daily Builds für die laufenden Implementierungen von Mojarra und Ozark genutzt werden (die Referenz-Implementierungen von JSF beziehungsweise MVC). Die Schlussfolgerung scheint zu sein: Er bewegt sich (mit), also kann er nicht tot sein. Für den genannten Zweck ist das wohl auch ausreichend. Zumindest gibt es GlassFish-Images inzwischen auch auf Docker Hub, sie können von dort direkt zum Einsatz mit dem Microservices-Framework genutzt werden. Aber nutzt heute noch jemand GlassFish für ein (neues) Produktivsystem? Berichte darüber gerne an mich oder die Redaktion.

https://www.java.net/blogs/mriem

25. Mai 2015

20 Jahre Java

Herzlichen Glückwunsch, Java, zum 20. Geburtstag! Wer hätte sich vor 20 Jahren diesen überwältigenden Erfolg der Programmiersprache und vor allem der dazugehörigen Plattform, die längst vielen anderen Sprachen dient, vorstellen können? Oracle hat aus diesem Anlass eine eigene Seite eingerichtet.

https://community.oracle.com/community/java/javas-20th-anniversary

26. Mai 2015

Java 9 interaktiv – mit der REPL

Wer viel mit Sprachen wie Lisp oder Python arbeitet, hat vermutlich für Java bislang die direkte Feedback-Schleife „Read Evaluate Print Loop" (REPL) vermisst. Dass man damit nicht nur im Programmier-Kämmerlein, sondern auch auf der Bühne erfolgreich arbeiten kann, haben unter anderem die Jungs von „Overtone" mit ihrer live programmierten Musik gezeigt. Grundsätzlich ist die REPL für die JVM schon verfügbar, beispielsweise für Scala. Was wir mit Java SE 9 jetzt endlich auch für die Sprache Java erwarten können, zeigt Cay Horstmann in einem Blog-Post auf. Momentan muss das Feature noch von Hand gebaut werden, es ist noch nicht Teil der Binary Distribution. Mac-OS- und Linux-Nutzer können aber schnell loslegen. Für Windows ist das Bauen der Preview-Version wohl

noch etwas knifflig, aber als vorübergehenden Ersatz gibt es ja Linux-VMs. Dann können wir uns in Zukunft auch auf Java Live Acts freuen …
weblogs.java.net/blog/cayhorstmann/archive/2015/05/25/trying-out-java-9-repl

28. Mai 2015

AngularBeans: AngularJS-Framework für Java EE

Java EE mit AngularJS? Geht – erfordert aber einiges an Infrastruktur für die Client-Server-Kommunikation oder Boilerplate-Code, wie der Engländer sagt. Bessem Hmidi, JUG Leader aus Tunesien, hat jetzt ein Framework geschrieben, das CDI nutzt, und damit die gesamte Kommunikations-Infrastruktur realisiert, plus clientseitige Bean Validations etc. AngularBeans ist inklusive einer Demo-Applikation auf GitHub verfügbar.
https://github.com/bessemHmidi/AngularBeans

2. Juni 2015

Neue JSF-Erweiterungen

In einem Blog-Post listet der Spec Lead für JavaServer Faces, Ed Burns, ein paar neue JSF-Erweiterungen auf, die er im Zuge seiner Arbeit „entdeckt" hat: BootsFaces (richtig, hat mit Bootstrap zu tun), Material Prime (zum Bauen von WebApps basierend auf Google-„Material Design"-Richtlinien) und Generjee (Generieren von Java EE-/JSF-Applikationen). Futter für alle JSF-Nutzer – und ein Zeichen dafür, dass das JSF-Ökosystem lebt.
https://www.java.net/blogs/edburns

5. Juni 2015

Java EE 8 verzögert sich

John Clingan, Product Manager bei Oracle für Java EE und GlassFish, hat eine Verzögerung des Release 8 der Enterprise Edition angekündigt. Der ursprüngliche Zeitplan sah zwar das offizielle Release für die JavaOne im Herbst 2016 vor – bis dahin ist es ja noch eine Weile. Aber obwohl alle JSRs inzwischen in Fahrt zu sein scheinen, zeichnet sich die Verzögerung wohl schon ab. Vom ersten Halbjahr 2017 ist jetzt die Rede, ein genauerer Termin steht noch nicht fest. Mehr Zeit für die Java-Community, sich an den einzelnen JSRs zu beteiligen, wie Mr. Clingan betont. Aber inzwischen ist auch Adopt-a-JSR gut in der Community verbreitet, daran sollte das Release-Datum also nicht scheitern.
https://blogs.oracle.com/theaquarium/entry/java_ee_8_roadmap_update,

11. Juni 2015

JSON-Merge-Patch soll Teil von JSON-P 1.1 werden

Die Expertengruppe für JSON-P 1.1 will neben JSON Pointer und JSON Patch jetzt auch JSON-Merge-Patch (RFC 7386) zum Standard hinzufügen. Bei Merge-Patch werden im Gegensatz zum anderen Patch nicht die Operationen zur Manipulation einer Ressource im Request geschickt, sondern ein zu „mergendes" Dokument. Einen ersten Entwurf hat die Gruppe bereits verfasst und Näheres zur Spezifikation und zur Referenz-Implementierung dokumentiert. Die Expertengruppe freut sich wie immer über Feedback.
https://jsonp.java.net

15. Juni 2015

Neuerungen in JMS 2.1

Noch mehr Feedback für Java EE 8 erwünscht: Der Specification Lead für JMS hat einige Verbesserungsvorschläge aufgeführt und Lösungsvorschläge für den ersten ausgearbeitet – kurz gesagt, eine einfachere und gleichzeitig flexiblere Deklaration von Message Listeners. Damit soll es unter anderem möglich werden, eine beliebige Listener-Methode zu deklarieren, statt das Listener-Interface zu implementieren. Vielleicht kommt ja auch der zweite Vorschlag noch zum Zuge: JMS-Annotationen, die es ermöglichen, jede beliebige CDI-Bean als Message Listener zu deklarieren.
https://java.net/projects/jms-spec/pages/JMSListener

16. Juni 2015

Java DB ist nicht mehr Teil von JDK 9

Java DB, die bislang mit dem JDK gebündelte Datenbank, weithin als „Apache Derby" bekannt, wird ab Java 9 nicht mehr Teil der Java-Distribution sein. Wer die Datenbank benötigt, muss dann Apache Derby extra hinzufügen.
https://blogs.oracle.com/java-platform-group/entry/deferring_to_derby_in_jdk

18. Juni 2015

Blog über JavaServer Faces 2.3

Nochmal JSF: Wer sich für die Neuerungen in 2.3 beziehungsweise Java EE 8 interessiert, der sollte auch auf dieses Blog von Expert-Group-Mitglied Arjan Tijms schauen. Hier wird über alles Wichtige aus der Insider-Perspektive berichtet – eine gute Ergänzung zur offiziellen Expert-Group-Kommunikation *(siehe „https://javaserverfaces-spec-public.java.net§).*
http://jdevelopment.nl/jsf-23

19. Juni 2015

SIP-Servlet 2.0 unterstützt CDI

Neben dem HTTP-Servlet gibt es schon lange auch das SIP-Servlet, das das Session Initiation Protocol aus der Telekommunikation unterstützt. Es basiert auch auf dem generischen Servlet und wird ebenso über den JCP standardisiert. Mit der Version 2.0 (JSR 359) unterstützt auch das SIP-Servlet nun den „Context and Dependency Injection"-Standard.
https://weblogs.java.net/blog/binod/archive/2015/06/19/sip-servlet-20-and-cdi

30. Juni 2015

Viel Rauch um (fast) nichts

Eine Sekunde, die viel Zeit verschlingt: Heute gibt es mal wieder eine Schaltsekunde, die manchen Administrator zeitsensibler Anwendungen nervös macht. Extra für diesen Fall sieht die „java.util.Date"-Klasse Sekunden im Bereich von „0" bis „61" vor. Da die Schaltsekunden nicht langfristig festgelegt sind, kann mit dem Tool TZUpdater immer die Information im JRE aktualisiert werden (wenn nicht sowieso das JRE oder JDK an sich aktualisiert wird).
http://www.wired.com/2015/01/torvalds_leapsecond

Andreas Badelt
Leiter der DOAG SIG Java

Andreas Badelt ist Senior Technology Architect bei Infosys Limited. Daneben organisiert er seit 2001 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V. Daneben war er von 2001 bis 2015 ehrenamtlich in der Development Community und ist seit 2015 in der neugegründeten Java Community der DOAG Deutsche ORACLE-Anwendergruppe e.V. aktiv.

Wo steht CDI 2.0?

Thorben Jannsen und Anatole Tresch, Credit Suisse

Die Arbeiten an Java EE 8 sind in vollem Gange und einige Expert Groups haben mit dem Early Draft Review (EDR) bereits einen ersten offiziellen Zwischenstand veröffentlicht. Der Artikel zeigt den aktuellen Stand der Spezifikation für den Context and Dependency Inject for Java (CDI) 2.0.

Das Early Draft Review (EDR) soll der Entwicklergemeinschaft die Möglichkeit geben, einen Teil der geplanten Änderungen auszuprobieren und mit ihrem Feedback bereits früh die weitere Entwicklung der Spezifikation zu beeinflussen. Zum Zeitpunkt der Erstellung dieses Artikels steht auch die Expert Group für CDI 2.0 *(siehe „http://cdi-spec.org")* kurz vor der Veröffentlichung des EDR.

Die Liste der geplanten Änderungen für die Version 2.0 ist lang und in folgende Themen unterteilt:

- Verbesserung des Event-Systems
- Verwendung im Java-SE-Umfeld
- Unterstützung von Java-8-Sprachfeatures
- Anpassung der Interceptoren und Dekoratoren zur Verbesserung der aspektorientierten Programmierung
- Modularisierung und Anpassungen der SPI
- Kontexte

Das EDR wird allerdings nur die Änderungen beinhalten, die aus Sicht der Expert Group bereits einen ausreichenden Reifegrad erreicht haben, um von der Community ausprobiert und diskutiert zu werden. Das dabei erhaltene Feedback fließt anschließend in die Spezifikation mit ein und kann somit zu weiteren Änderungen führen. Die Hauptthemen des EDRs sind:

- Sortierung von Events
- Asynchrone Verarbeitung von Events
- Unterteilung der Spezifikation in einen Java-EE- und einen Java-SE-Teil
- Ein Bootstrap-Mechanismus für Java-SE-Umgebungen

Sortierung von Events

Die Erweiterung des Event-Systems war eine der von der Community am häufigsten gewünschten Änderungen für CDI 2.0. Dass dies von der Expert Group entsprechend ernst genommen wird, zeigt sich an den im EDR enthaltenen Änderungen. Die Sortierung von Event-Observern war eines der ersten Features, für die ein konkreter Vorschlag erarbeitet und die in das EDR aufgenommen wurden.

Bisher wurden die Observer-Methoden vom Container in unbestimmter Reihenfolge aufgerufen, sodass eine gesteuerte Verkettung mehrerer, aufeinander aufbauender Observer nicht möglich war. Ab Version 2.0 soll der CDI-Container die Event-Observer basierend auf ihrer Priorität sortieren und anschließend in aufsteigender Reihenfolge aufrufen. Die Priorität kann für jede Oberserver-Methode mithilfe der „@Priority"- Annotation definiert werden *(siehe Listing 1)*.

Observer ohne deklarierte Priorität erhalten die Default-Priorität „APPLICATION + 500". Wenn mehrere Observer-Methoden über denselben Prioritätswert verfügen, ist die Aufruf-Reihenfolge nicht definiert und entspricht somit dem bisherigen Verhalten. Die beschriebene Funktionalität wurde bereits im Release 3.0.0.Alpha1 der CDI-Referenz-Implementierung „Weld" prototypisch implementiert und kann dort ausprobiert werden.

Asynchrone Verarbeitung von Events

Ein weiteres, von der Community häufig gewünschtes Feature ist die asynchrone Verarbeitung von Events. Bisher wurden die Event-Observer vom CDI-Container synchron innerhalb des Thread des Event-Producers ausgeführt. Wenn ein Event ausgelöst wurde, stoppte daher die Verarbeitung des Event-Producers, bis alle Observer das Event verarbeitet hatten oder dieses durch eine Exception abgebrochen wurde.

Ab Version 2.0 soll auch eine asynchrone Verarbeitung der Events möglich sein. Dazu verwendet der CDI-Container ein oder mehrere zusätzliche Threads, sodass die Verarbeitung im Event-Producer direkt fortgesetzt werden kann. Wie die asynchrone Verarbeitung im Detail erfolgen soll, wurde lange innerhalb der Expert Group diskutiert und ist zum aktuellen Zeitpunkt noch nicht in allen Details abschließend entschieden worden. Vor allem die Bereitstellung eines intuitiven und einfachen API bei gleichzeitiger Wahrung der Kompatibilität zu CDI 1.1 und die Zugehörigkeit zu den jeweiligen Kontexten stellen hier eine Herausforderung dar.

Im aktuell diskutierten Vorschlag müssen zwei Bedingungen erfüllt sein, damit ein Event asynchron verarbeitet werden kann. Zum einen muss das Event über die zum Event-Interface hinzugefügte „fireAsync"-Methode ausgelöst werden und zum an-

```
void processEvent(@Observes @Priority(100) MyEvent event) {
// do something
}
```

Listing 1

deren der Event-Observer mit der „@ObservesAsync"-Annotation annotiert sein. Diese doppelte Aktivierung der asynchronen Verarbeitung stellt sicher, dass CDI-1.1-Event-Observer durch die Änderung eines Event-Producers nicht plötzlich asynchron ausgeführt werden.

Ebenso können bestehende Event-Producer, bei deren Erstellung eine synchrone Verarbeitung der Events angenommen wurde, keine asynchrone Verarbeitung der Events auslösen. Dies ist wichtig, da das Event-Objekt veränderbar sein kann und in vielen Anwendungen für den Datenaustausch zwischen Event-Producer und -Observer verwendet wird.

Wenn die Verarbeitung des Events in Zukunft asynchron erfolgen soll, muss der Entwickler sich um Nebenläufigkeitsprobleme kümmern und zusätzlich den Event-Producer gegebenenfalls auf die Event-Observer warten lassen. Somit ist die bestehende Code-Basis genau zu analysieren und bei Bedarf anzupassen, bevor eine asynchrone Eventverarbeitung möglich ist. *Listing 2* zeigt, wie asynchrone Events genutzt werden können.

Unterteilung in einen Java-EE- und einen Java-SE-Teil

Die Modularisierung der CDI-Spezifikation bietet im Gegensatz zu den vorher betrachteten Änderungen des Event-Systems nur wenige Vorteile für die Anwendungsentwicklung. Stattdessen unterstützt es die Entwickler anderer Spezifikationen und Frameworks bei der gezielten Verwendung einzelner Teile der Spezifikation.

Der anfängliche Plan war, die CDI-2.0-Spezifikation in drei Teile zu unterteilen: „CDI full mit Java EE", „CDI full" und „CDI light". „CDI full mit Java EE" entspricht dabei dem vollen Funktionsumfang, wie er aus CDI 1.1 bekannt ist, einschließlich der Abhängigkeiten zu anderen Java-EE-Spezifikationen sowie aller im Rahmen von CDI 2.0 neu hinzugefügten Features. „CDI full" und „CDI light" sollen hingegen auch im Java-SE-Umfeld einsetzbar sein und daher keine Abhängigkeiten auf Java-EE-Spezifikationen aufweisen. „CDI light" soll zusätzlich keine aspektorientierte Programmierung (AOP) und keine Kontexte enthalten, womit es unter Umständen sogar Java-ME-kompatibel werden könnte.

Für das EDR-Release ist allerdings nur ein Teil dieser Änderungen umgesetzt. Vorerst wurde die Spezifikation ausschließlich in „CDI full mit Java EE" und „CDI full" unterteilt. Ob die noch fehlende Definition von „CDI light" zu einem späteren Zeitpunkt erfolgen und welchen Umfang diese dann genau haben wird, wird nach Fertigstellung des EDR-Release diskutiert.

Bootstrapping für Java-SE-Umgebungen

In Java-SE-Umgebungen muss das Starten des CDI-Containers durch den Anwendungsentwickler erfolgen. Dies war bisher nicht Bestandteil der Spezifikation, wurde allerdings durch die Implementierungen „Weld" und „Open Web Beans" mithilfe eigener APIs ermöglicht, für die das Apache-Delta-Spike-Projekt eine containerunabhängige Abstraktion anbietet. Ab CDI 2.0 wird der Bootstrapping-Mechanismus durch die Spezifikation definiert *(siehe Listing 3)*.

Zuerst wird durch die statische „getCDIProvider()"-Methode der Utilityklasse „CDI" ein „CDIProvider" geholt. Im Standardfall wird dazu intern ein „ServiceLoader" verwendet. Anschließend startet die Methode „initialize()" den CDI-Container, der dann für die weitere Verwendung zur Verfügung steht. Wird der Container nicht mehr benötigt, kann ihn die „shutdown()"-Methode wieder beenden. Da die CDI-Klasse das „Autocloseable"-Interface implementiert, kann das Beenden des Containers auch mithilfe eines „try with Resource"-Blocks erfolgen.

Auch wenn bereits ein Vorschlag für das neue API ausgearbeitet wurde, sind noch nicht alle Details zur Verwendung von CDI im Java-SE-Umfeld abschließend diskutiert. Ein noch offenes Problem stellt zum Beispiel die Unterstützung impliziter Bean-Archive (ohne „beans.xml"-Datei) dar. Im Java-EE-Umfeld werden dazu per Default alle vorhandenen Archive gescannt. Im Java-SE-Umfeld kann das Durchsuchen aller Archive jedoch zu aufwändig sein. Die aktuell bevorzugte Lösung sieht daher vor, dass die Unterstützung für implizite Bean-Archive durch einen Konfigurations-Parameter aktiviert sein muss. Somit kann der Entwickler entscheiden, ob diese Funktionalität benötigt wird und ob die benötigte Zeit zum Scannen aller Archive aufgewendet werden soll.

Weitere noch zu diskutierende Fragestellungen betreffen unter anderem die Möglichkeit, mehrere Container in einer Anwendung zu starten, sowie die Unterstützung von Kontexten. Hierzu werden über das EDR-Release hinaus noch Lösungen erarbeitet werden müssen.

Weitere Änderungen

Neben den in das EDR eingeflossenen Änderungen wurden in den bisherigen Releases der CDI-Referenz-Implementierung „Weld"

```
// Producer
@Inject
Event<MyEvent> event;
[…]
MyEvent myEvt = new MyEvent();
CompletionStage<MyEvent> completion = event.fireAsynch(myEvt);

// Observer
public void consumeEvent(@ObservesAsynch MyEvent evt){
  // handle the event
}
```

Listing 2

```
// start container
CDIProvider cdiProvider = CDI.getCDIProvider();
CDI<Object> cdi = cdiProvider.initialize();

// get a bean and do something
MyCdiBean bean = cdi.select(MyCdiBean.class).get();
bean.doSomething();

// shutdown container
cdi.shutdown();
```

Listing 3

(siehe „http://weld.cdi-spec.org“) bereits einige weitere Änderungen prototypisch implementiert, die zwar diskutiert, aber noch nicht spezifiziert sind. Ob diese Änderungen in derselben Form in die finale Spezifikation aufgenommen werden, ist zum aktuellen Zeitpunkt nicht abzuschätzen. Allerdings vermitteln die prototypischen Implementierungen bereits einen guten Eindruck über die noch zu erwartenden Erweiterungen der Spezifikation.

So wird für Qualifier und Interceptoren das mit Java 8 eingeführte „repeating annotation“-Feature verwendet, mit dem Methoden und Eigenschaften mehrfach mit derselben Annotation annotiert werden können. Damit besteht beispielsweise die Möglichkeit, eine Producer-Methode mehrfach mit demselben Qualifier und unterschiedlichen Eigenschaften zu annotieren *(siehe Listing 4)*.

Darüber hinaus wurde das „ProcessObserverMethod“-SPI dahingehend erweitert, dass CDI-Extensions nun einzelne Observer-Methoden überschreiben oder sie mithilfe der „veto()“-Methode des Event-Interface deaktivieren können. Durch diese Änderungen erhält der Entwickler, wie von der Community häufig gewünscht, mehr Kontrolle über die Verarbeitung von Events.

```
@Produces
@PaymentMethod("CreditCard")
@PaymentMethod("Paypal")
public PaymentProvider createPaymentProvider() {
     // ...
}
```

Listing 4

Was noch zu erwarten ist

Wie bereits zu Beginn des Artikels gesagt, enthält das EDR nur die Änderungen der CDI-Spezifikation, für die die Expert Group bereits konkrete Vorschläge erarbeitet hat und die über eine ausreichende Reife verfügen, sodass sie von der Community ausprobiert und diskutiert werden können. Daher gibt es auch noch viele Themen, für die eine Lösung erst noch erarbeitet werden muss. Einige davon wurden bereits in den vorhergehenden Abschnitten genannt und einige weitere werden wir nun kurz betrachten. Die Diskussionen zu diesen Änderungen befinden sich in der Regel noch in einem sehr frühen Stadium oder wurden noch gar nicht aufgenommen. Daher ist zum aktuellen Zeitpunkt noch nicht abzusehen, welche dieser Änderungen Bestandteil des finalen Release werden und wie diese genau gestaltet werden.

Wie alle anderen Java-EE-8-Spezifikationen auch soll CDI 2.0 die Verwendung der mit Java 8 neu eingeführten Sprach-Features unterstützen. Bisher wurden noch keine der daraus resultierenden Änderungen innerhalb der Expert Group diskutiert und nur kleinere Änderungen in den Alpha-Releases der Referenz-Implementierung „Weld“ umgesetzt. Der vollständige Umfang der noch zu erwartenden Änderungen ist zurzeit noch nicht abzusehen. Alle neu hinzugefügten Features werden die neuen Java-8-Features verwenden.

Ein weiteres Thema, das bisher nur wenig bearbeitet wurde, ist die Anpassung des SPI. Dadurch soll unter anderem eine bessere Kontrolle der verwendeten Kontexte, die Erzeugung von Beans zur Laufzeit und der Zugriff auf die von CDI erzeugten Metadaten ermöglicht werden.

Die Anpassungen der Interceptoren und Dekoratoren wurden bisher noch nicht spezifiziert. Allerdings wird für die Dekoratoren in der Referenz-Implementierung „Weld“, wie bereits gesagt, das mit Java 8 eingeführte „repeating annotation“-Feature verwendet. Andere geplante Änderungen, wie Verbesserungen des „interceptor chaining“ und der Aufruf von Interceptoren bei internen Methoden-Aufrufen, wurden bisher noch nicht eingehend diskutiert.

Auch die Möglichkeit des Zugriffs auf das durch einen Proxy „gewrappte“ Objekt würde Entwicklern von Extensions das Leben in vielen Fällen vereinfachen. In eine ähnliche Richtung zielt auch der Vorschlag, alle (nicht nur die CDI-relevanten) Annotationen einer „gewrappten“ Klasse auch in der Proxy-Instanz verfügbar zu machen. Interessant ist auch die Idee, bestehende Interceptoren mit „@Alternative“ übersteuern sowie die Reichweite von Events mittels einer Annotation bestimmen zu können. Damit kann aktiv beeinflusst werden, ob ein Event an alle aktuell verfügbaren Observer weitergereicht wird oder zum Beispiel nur an die, die in der gleichen Deployment-Unit („=jar“) enthalten sind.

In welchem Umfang all diese Vorschläge in der finalen Spezifikation enthalten sein werden, ist aber derzeit noch nicht absehbar: stay tuned!

Anatole Tresch
atsticks@gmail.com

Anatole Tresch war nach dem Wirtschaftsinformatik-Studium an der Universität Zürich mehrere Jahre lang als Managing-Partner und -Berater tätig. Er sammelte weitreichende Erfahrungen in allen Bereichen des Java-Ökosystems vom Kleinunternehmen bis zu komplexen Enterprise-Systemen. Schwerpunkte sind verteilte Systeme sowie die effiziente Entwicklung und der Betrieb von Java in der Cloud. Aktuell arbeitet Anatole Tresch als technischer Architekt bei der Credit-Suisse, ab September wird er als Principal Consultant bei Trivadis tätig sein. Anatole ist Star Specification Lead des JSR 354 (Java Money & Currency) und Gründer des Apache-Tamaya-Projekts.

Thorben Jannsen
thjanssen123@gmail.com

Thorben Janssen arbeitet als Senior-Entwickler und Architekt für die Qualitype GmbH und entwickelt seit mehr als zehn Jahren Anwendungen auf Basis von Java EE. Er ist Mitglied der JSR 365 Expert Group und bloggt über Themen rund um Java Enterprise auf *„http://www.thoughts-on-java.org“*.

Der Software-Architekt in der heutigen Software-Entwicklung

Tobias Biermann, Booxware

Dieser Artikel stellt die Rolle „Software-Architekt" innerhalb von Software-Projekten vor. Er klärt die Aufgaben eines Software-Architekten sowie die Beziehungen zwischen dem Architekten und den anderen Projekt-Teilnehmern.

Die Position „Software-Architekt" ist in der Software-Branche etabliert. Ungefähr acht Prozent aller ausgeschriebenen Software-Entwicklerstellen sind Software-Architektenstellen. Der Software-Architekt stellt zudem oft eine Karrierestufe dar. Außer dem Software- gibt es noch den Unternehmens-Architekten. Während der Software-Architekt technisch für ein Projekt verantwortlich ist, ist der Unternehmungs-Architekt für die Software-Entwicklung im ganzen Unternehmen zuständig. Außerhalb des Studiums gibt es für diese Rolle verschiedene Weiterbildungen durch unterschiedliche Institutionen:

- *SEI (Software Engineering Institute)*
  Weiterbildungen zu den Themen „SOA", „Architektur", „Architektur-Bewertung" (ATAM)
- *Oracle*
  Certified Software Architect Java EE 6
- *iSAQB e.V.*
  Certified Professional for Software Architecture

Der Autor hat das Zertifikat „Certified Professional for Software Architecture" des Vereins iSAQB erworben. Da der Schulungsinhalt zu empfehlen ist, orientiert sich der Artikel an der Sichtweise dieses Vereins – Software-Architektur und auch die Rolle sind nicht einheitlich definiert.

Die Hauptaufgabe des Software-Architekten ist es, eine Software-Architektur zu entwerfen. Da stellt sich die Frage: Was ist überhaupt eine Software-Architektur? Eine Software-Architektur umfasst:

- Zerlegung des Problems in kleine, verständliche und beherrschbare Komponenten
- Die Beziehungen der Komponenten zueinander und zur Umgebung
- Grundsätze, die Design und Entwicklung des Systems bestimmen

Eine Komponente hat folgende Funktionen:

- Ist Baustein eines Systems
- Kapselt ihr inneres Verhalten
- Kommuniziert mit anderen Komponenten über definierte Schnittstellen
- Ist wiederverwendbar und austauschbar

Diese Definitionen veranschaulicht ein Beispiel. *Abbildung 1* zeigt ein Kontext-Diagramm für eine Software-Architektur. Es geht darum, verschiedenen Anwendern (Anlagenplanern, Vertriebsmitarbeitern, Projektverwaltern, Einkäufern) Systeme zur Verfügung zu stellen, mit denen sie ihre Aufgaben bezüglich der Ressource „Artikel" einfacher bewerkstelligen können. Dafür wurden drei Systeme (die man auch als Komponenten bezeichnen kann) eingekauft (Design-, Verkaufs- und ERP-Sys-

tem). Zudem wurde die Komponente „Artikel Provider" neu entwickelt.

Die Architektur hält die Beziehungen der Komponenten fest und außerdem, welche Informationsobjekte (Artikel, Stückliste) über ihre Schnittstellen ausgetauscht werden. Das Innere der Komponenten ist in dieser Beschreibung gekapselt, es zählt nur der Austausch der Objekte. Im nächsten Schritt wird dann die zu entwickelnde Komponente „Artikel Provider" in sinnvolle kleinere Komponenten zerlegt.

Die neue Komponente läuft zudem innerhalb einer Integrations-Plattform. Das bedeutet, dass sie sich an für diese Plattform definierte Rahmenbedingungen halten muss. In diesem Fall ist die Komponente ein RESTful-Webservice und kommuniziert über das http-Protokoll. Dies ist ein Beispiel für die Grundsätze, die ein Design einhalten muss.

Da jedes System eine Architektur hat, gibt es auch in jedem Projekt Aufgaben eines Software-Architekten. Ob der Umfang der Aufgaben es erfordert, dass eine Person sie in einem Projektteam in Vollzeit übernimmt, hängt von dem Projekt ab, insbesondere von seiner Organisation und Größe. Gerade bei kleineren Projekten nehmen die Aufgaben nur einen Teil der Arbeitszeit in Anspruch und in dem Rest der Zeit kann der Architekt die Software mitentwickeln oder andere Aufgaben übernehmen. Bei großen Projekten wiederum kann ein Team von Architekten sinnvoll sein. Es ist auch möglich, die Aufgaben eines Architekten auf mehrere Teammitglieder zu verteilen. Es gibt zwei wichtige Regeln, die sich in der Erfahrung bewährt haben:

- Komplizierte Dinge nicht allein entwerfen beziehungsweise umsetzen (und Architekturen sind in aller Regel komplex)
- Wissen, wovon man redet. Also am besten teilweise mitentwickeln. Das sorgt für schnelles und authentisches Feedback und besseres Verständnis der Mitarbeiter

Zurück zur der Hauptaufgabe des Software-Architekten, eine für die Software passende Architektur zu entwerfen. Die Software-Architektur ist das Bindeglied zwischen der Analyse der Anforderungen und deren Umsetzung. Der Software-Architekt ist wiederum der Ansprechpartner für alle Stakeholder (maßgeblich beteiligte Personen) des Projekts für die technische Umsetzung.

Der Software-Architekt klärt die Anforderungen an die Software. Dazu gehören sowohl funktionale wie „der Anwender sieht alle verfügbaren Artikel" als auch nicht funktionale wie „alle Artikel werden innerhalb von drei Sekunden angezeigt". Hinzu kommen Randbedingungen wie das Budget, das Team, der Fertigstellungstermin und die Maßgabe, dass die Software beispielsweise auf einem Application-Server installiert ist.

In der Regel sind die Anforderungen bereits vom Kunden und einem Requirements Engineer erfasst. Wichtig ist, in Kommunikation mit den Stakeholdern die Anfor-

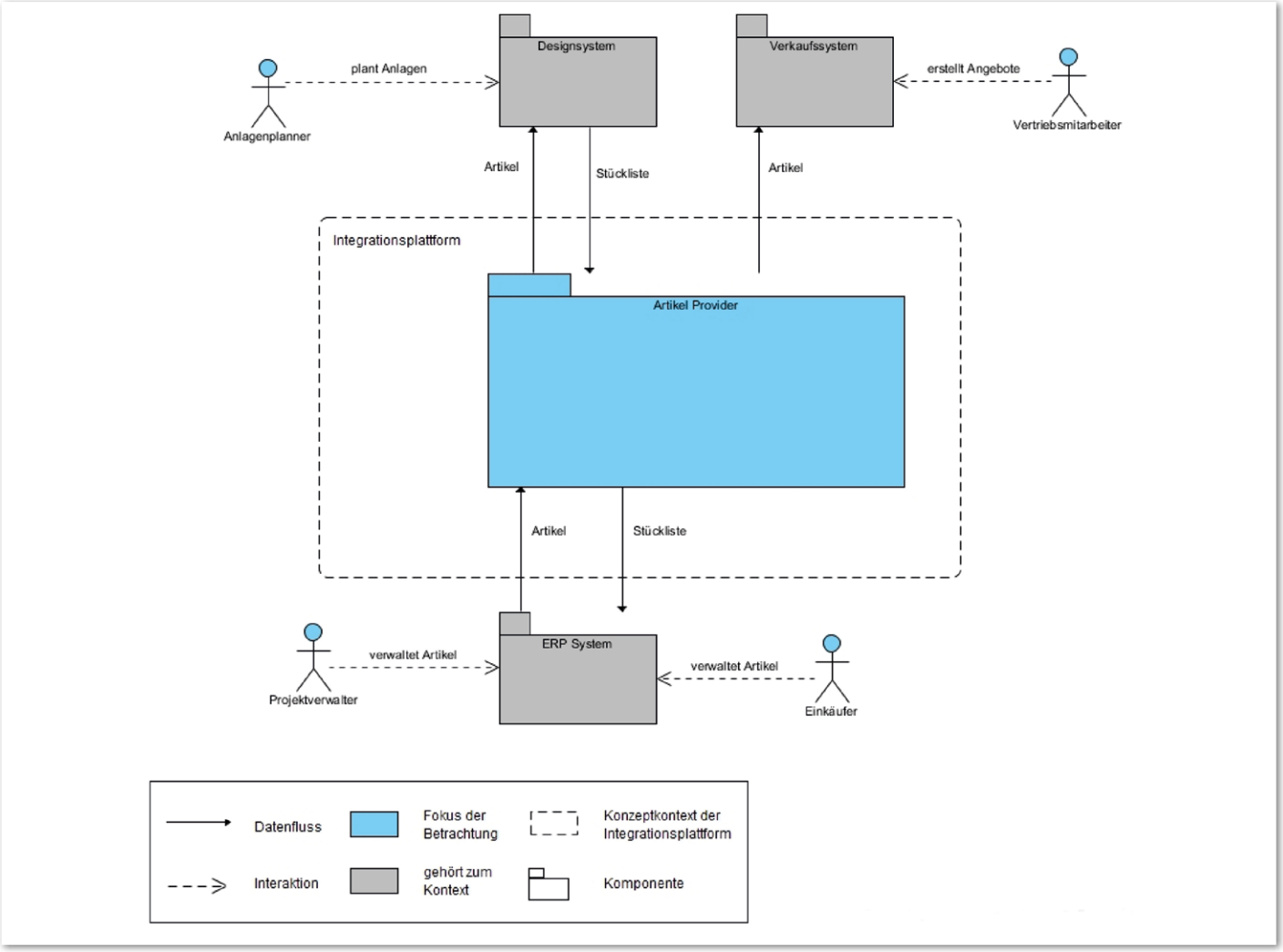


Abbildung 1: Kontext-Diagramm für die Architektur eines Systems

derungen soweit zu klären, dass sie allen Beteiligten klar sind und sich (innerhalb der Rahmenbedingungen) technisch umsetzen lassen.

Der Architekt entwirft dann den Bauplan des Systems. Dafür zerlegt er das System in kleine, verständliche und beherrschbare Bausteine. Er definiert die Beziehungen der Bausteine untereinander und ihre Schnittstellen. Zu den Schnittstellen gehören insbesondere die Objekte, die über sie ausgetauscht werden. Wichtig ist, neben dem Kontext, in den die neuen Komponenten eingebunden sind, zu definieren, mit welchen bestehenden Systemen die Komponenten interagieren und welche Objekte sie austauschen werden. Genauso wichtig ist es festzuhalten, wer (oder welche Rollen) mit dem neuen System und den bestehenden Systemen des Kontextes interagieren werden. Dies kann man gut mit einem Kontextdiagramm veranschaulichen *(siehe Abbildung 1)*.

Zudem erarbeitet und beschließt der Architekt übergreifende technische Konzepte, etwa welche Technologien eingesetzt werden, welche Architektur-Pattern (Microservices, Mehrschicht-Architektur etc.), welche Frameworks eingesetzt werden und vieles mehr. Zu seinen Aufgaben gehört auch die Dokumentation der Architektur. Es ist sinnvoll, hierfür ein Template wie „arc42" zu verwenden. Die Dokumentation ist auch ein Kommunikationsmittel und sollte zielgruppengerecht sein. Es ist wichtig zu beachten, wer die Dokumentation lesen wird. Wird sie von Software-Entwicklern gelesen, so kann sie ruhig sehr technisch detailliert sein. Ist die Zielgruppe das Management, so sollte die Technik nur aus der Vogelperspektive gezeigt und die Auswirkungen der Entscheidungen auf das Geschäft (Projekt-Laufzeit, Projekt-Risiken, Kosten etc.) in den Vordergrund gerückt werden. Oft ist es deshalb sinnvoll, für jede Zielgruppe ein eigenes Dokument zu schreiben.

Die Anforderungen des Projekts werden von dem Auftraggeber oder einem Requirements Engineer ermittelt und dokumentiert. Die Architektur-Dokumentation sollte einfach darauf verweisen. Es ist sinnvoll, viele Dokumente statt eines großen zu schreiben. Nützlich ist dann noch ein Verweisdokument, das alle Dokumente und ihre Orte auflistet.

Kommunikation zählt zu den Hauptaufgaben des Architekten. Er kommuniziert mit vielen verschiedenen Stakeholdern. *Abbildung 2* zeigt den Architekten und die wichtigsten Stakeholder, mit denen er zusammenarbeitet.

Der Auftraggeber

Der Architekt präsentiert dem Auftraggeber die System-Vision und erreicht dessen Akzeptanz. Eventuell vorhandene Wünsche und Einschränkungen werden aufgenommen und berücksichtigt. Auch Projekt-Risiken und die daraus resultierenden Maßnahmen werden besprochen. Der Auftraggeber ist ein wichtiger Stakeholder, deshalb ist es wichtig, sich Rückmeldungen einzuholen. Oft ist das nicht einfach, da viele Auftraggeber wenig Zeit haben. Daher ist eine gute Vorbereitung von Besprechungen empfehlenswert.

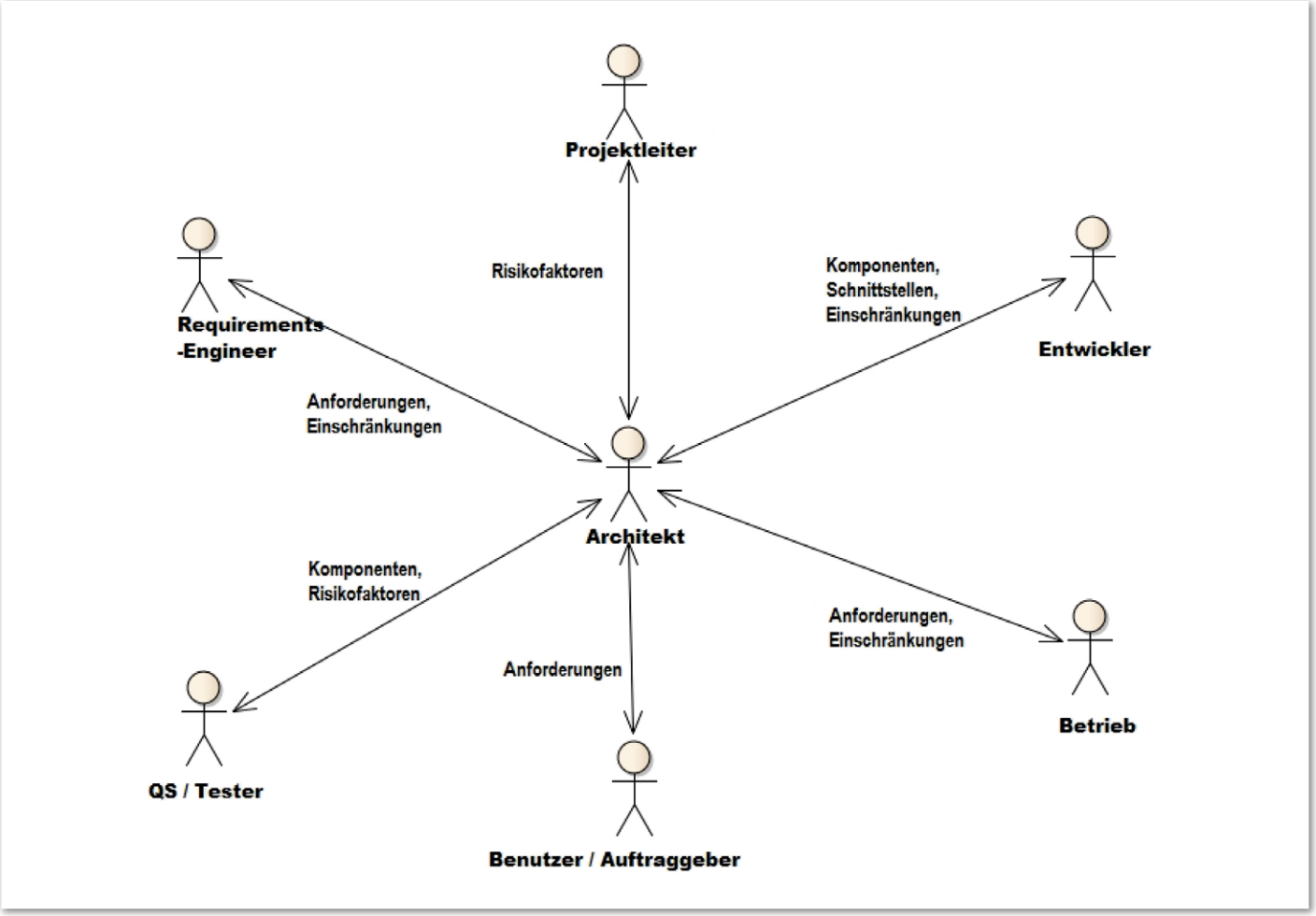


Abbildung 2: Der Architekt und seine Beziehungen zu den anderen Stakeholdern

Der Anwender

Mit den Anwendern der Software hat der Architekt oft wenig Kontakt. Sie sind aber ein wichtiger Stakeholder des Projekts. Es ist sinnvoll, sich auch von den Anwendern in einer frühen Phase des Projekts Rückmeldungen zu holen. So kann man einen Prototyp entwickeln und Pilot- oder Test-Anwender damit arbeiten lassen, um Feedback beispielsweise über die Bedienbarkeit der Software zu gewinnen.

Der Requirements Engineer

Der Requirements Engineer ermittelt die Anforderungen für das Projekt. Der Software-Architekt klärt diese soweit mit ihm, bis sie technisch realisiert werden können. Für unterschiedliche beziehungsweise widersprüchliche Interessen müssen Lösungen erarbeitet werden. Auch Kompromisse für Anforderungen, die nur sehr aufwändig oder kostspielig umgesetzt werden können, müssen gefunden und gegebenenfalls durch einfacher zu realisierende ersetzt werden. Das Gleiche gilt für funktionale Anforderungen, die negative Auswirkungen auf nicht funktionale Anforderungen wie die Performance haben und deshalb so nicht akzeptabel sind.

Der Projektleiter

Im Projekt hat der Architekt die Entscheidungskompetenz für Entwurfsentscheidungen und die technische Umsetzung. Er kommuniziert diese an den Projektleiter. Zusammen mit ihm organisiert er das Team entsprechend dem Architektur-Entwurf. Dabei unterstützt er den Projektleiter bei der Beurteilung der technischen Fähigkeiten von Mitarbeitern und Bewerbern und bei der Auswahl der Team-Mitarbeiter.

Er ist zuständig für die Auswahl der passenden Tools und Technologien für das Projekt. Weitere wichtige Aufgaben sind die Risiko-Überwachung und die Eskalation. Besteht ein Risiko oder steigt die Wahrscheinlichkeit für das Eintreten des Risikos stark an, so informiert der Architekt den Projektleiter und es werden die möglichen Maßnahmen besprochen und eingeleitet.

Die Software-Entwickler

Der Architekt erläutert den Software-Entwicklern die Architektur. Es ist doppelt wertvoll, Entwickler schon beim Entwurf der Architektur zu beteiligen. Die Entwickler haben eine andere Sicht auf die Architektur (oft eine deutlich detailliertere) und sie kennen die Probleme der bestehenden Software, falls es sich nicht um eine komplette Neuentwicklung handelt. Durch ihr Wissen und ihre Erfahrung können sie dazu beitragen, die Architektur zu verbessern. Die Mitwirkung der Entwickler sorgt zudem nebenher für eine gute Akzeptanz der Architektur durch die Entwickler.

Ebenso ist es wichtig zu erläutern, warum welche Entscheidungen getroffen wurden. Insbesondere bei schwierigen und unpopulären Entscheidungen gilt es aufzuzeigen, welche Nachteile hierbei durch den Architekten in Kauf genommen wurden, warum diese Entscheidungen trotzdem so getroffen wurden und welche Alternativen es gab.

Der Architekt moderiert soweit notwendig an den Schnittstellen zu anderen Teammitgliedern – sei es bei Fragestellungen bezüglich des Tests und des Betriebs oder bei Fragen zu den Anforderungen. Oft fehlen hier dem Entwickler die Kenntnisse oder er kann die Sichtweise der anderen Team-Mitglieder nicht leicht verstehen.

Beim Einsatz neuer Technologien und Werkzeuge ist es auch eine Aufgabe des Architekten, die Entwickler in diesen zu schulen. Es ist sinnvoll, sich von den Entwicklern regelmäßig Rückmeldung zu holen. Architektur ist eine komplexe und schwierige Aufgabe, daher ist es sehr wertvoll, von Schwierigkeiten bei der Umsetzung zu erfahren. Auch kann sich erst bei der Entwicklung ergeben, dass es sinnvoll ist, die Architektur zu ändern und damit zu verbessern.

Der Betrieb

Der Architekt stellt in Zusammenarbeit mit dem Betrieb sicher, dass die Software auch betreibbar ist. Dafür legt er Hardware-Anforderungen fest und berücksichtigt dabei Einschränkungen und Anforderungen bezüglich der Hardware. In der Regel ist bereits Hardware vorhanden, sodass die neue Software auf dieser laufen wird. Auch zu berücksichtigen sind die Skills der Mitarbeiter. Gibt es etwa keinen Oracle-Datenbank-Administrator, so hat es keinen Sinn, Oracle als Datenbank-Technologie einzusetzen. Mit dem Betrieb legt der Architekt die Service Level Agreements (SLA) der Lösung fest und entwickelt Verfahren, diese zu erreichen.

QS / Tester

Mit der Qualitätssicherung (QS) oder den Testern legt der Architekt die Test-Infrastruktur fest. Er definiert die Qualitätsmerkmale der Software und sorgt für die Einhaltung einer Mindestqualität. Die wichtigsten Risikofaktoren werden festgehalten und Methoden entwickelt, um diese zu testen. Zudem unterstützt der Architekt die Qualitätssicherung bei White-Box-Tests.

Fazit

Der Architekt hat eine hoch kommunikative Aufgabe. Er redet mit vielen unterschiedlichen Rollen. Die Menschen in diesen Rollen sind sehr unterschiedlich und haben unterschiedliche Sichtweisen auf die Themen. Es ist anspruchsvoll, sich auf diese verschiedenen Menschen einzustellen und die eigenen Punkte auf den Kollegen bezogen gut zu vermitteln. Dadurch ist die Rolle aber auch interessant und spannend. Auch der technische Teil bezüglich des Architektur-Entwurfs ist vielseitig und herausfordernd. Die Position oder Rolle „Software-Architekt" ist also enorm lohnend.

Literaturverzeichnis

[1] Effektive Softwarearchitekturen, Gernot Starke
[2] Arc42, www.arc42.de
[3] Knigge für Softwarearchitekten, Peter Hruschka, Gernot Starke
[4] Software Architectur in Praxis, Len Bass, Paul Clements, Rick Kazman
[5] Pattern Oriented Software Architecture (Volume 4), Frank Buschmann, Kevlin Henney, Douglas C. Schmidt
[6] Pattern of Enterprise Application Architecture, Martin Fowler
[7] SEI, http://www.sei.cmu.edu/training
[8] Oracle Education, http://education.oracle.com
[9] SAQB, http://www.isaqb.org

Tobias Biermann
tobias.biermann@booxware.de

Tobias Biermann studierte Diplom-Informatik und hat mittlerweile 17 Jahre Berufserfahrung in der Software-Entwicklung. Er ist „Certified Professional for Software Architecture" (iSAQB) und arbeitet bei Booxware als Software-Architekt. Seine fachlichen Schwerpunkte sind Software-Architektur, Web-Applikationen, System-Integration und Services (RESTful, SOA etc.).

Hochverfügbare, performante und skalierbare Web-Anwendungen

Daniel Schulz, Capgemini in Deutschland

Cassandra ist eine hochverfügbare und skalierbare Datenbank und wird aufgrund dieser architektonischen Eigenschaften gerne im Big-Data-Umfeld eingesetzt. Zusammen mit einem geeigneten Applikations-Framework lassen sich diese Eigenschaften auch auf die eigenen Anwendungen übertragen. Das Ergebnis sind leicht zu betreibende Deployments, wie sie in der Geschäftswelt des 21. Jahrhunderts immer häufiger vorkommen werden.

Dieses Projekt geht auf eine Forschungs-arbeit zu hochperformanten Shared-Nothing-Software-Architekturen zurück; es wurde später als spezifische Software weiterentwickelt. Der innere, generische Kern dieser Software wurde anschließend als Framework extrahiert; er sorgt sich vor allem um die Persistenz der Daten. Dieses Framework kam in einer Software als Grundlage für ein strategisches Expertensystem für einen Kunden zum Einsatz. Das System soll einen zentralen, derzeit noch manuellen Entscheidungsprozess vor allem inhaltlich deutlich verbessern, aber auch organisatorisch beschleunigen.

Der Name des Frameworks ist „R8". Es ist ein Framework für Kundensysteme, die besondere Anforderungen an Hochverfügbarkeit, Performance und Skalierbarkeit haben. Hochverfügbarkeit (HA) beschreibt die Eigenschaft, dass ein System in jedem Fall antwortet. Dabei können beliebige Knoten im Cluster ausfallen und deren Arbeit wird, nach außen nahtlos, von den verbleibenden Knoten übernommen.

Herausforderung in typischen JEE-Architekturen

In der Regel greifen die Benutzer eines Systems über einen Load Balancer auf einen Cluster von Applikationsservern zu. Diese wiederum sind über einen Cluster von Service-Layer-Knoten mit einer einzelnen oder einer verteilten Datenbank verbunden. In diesem Fall bezeichnen „Cluster" jeweils jeden möglichen Fall – der „triviale Cluster" mit lediglich einem Knoten ist hierbei ebenfalls gemeint. Heutige Anwendungen bestehen damit im Allgemeinen aus vier Schichten: Client, Applikationsserver, optionaler Service- und Persistenz-Layer [1], von oben nach unten in absteigender technischer Wertschöpfungstiefe.

Der optionale Service-Layer kann hierbei auch im Deployment der Applikation enthalten sein. Wenn das nicht der Fall ist – der Service-Layer also auf eigener Hardware läuft – entsteht bei jedem Abruf von Daten eine zusätzliche Netzwerk-Latenz. In Mainframe-Architekturen kann diese jedoch entfallen, da der Service-Layer hierbei etwa bei IBM aus Cobol-Programmen besteht. Diese werden anstelle der reinen Datenbank angesprochen und erledigen dann stellvertretend den Zugriff auf die Datenbank. Da sie auf derselben Hardware laufen, auf der auch die Daten liegen, entsteht keine zusätzliche, reale Netzwerk-Latenz, lediglich der Overhead für die Serialisierung, Deserialisierung und der Zugriff auf das „localhost Loopback"-Interface.

In Java-Programmen kann der Service-Layer auch als Dependency im finalen Deployment enthalten sein. In Maven wären das Module oder Dependencies und in Entwicklungsumgebungen „Projekte" oder „Facetten". Das hat den Vorteil, dass jeder Entwickler einen selbstgewählten Service-Layer-Stand ansprechen kann, ohne seine Kollegen damit zu beeinflussen. Ein eigenständiger Service-Layer erzeugt die genannte zusätzliche Latenz – unabhängig davon, ob er auf derselben oder einer anderen Hardware läuft. Nur das Ausmaß des Overhead richtet sich nach der Host-Hardware: Im ersten Fall ist dieser Overhead allerdings deutlich geringer als bei einer separaten Maschine.

Ein integrierter Service-Layer – ob in der Datenbank oder in der Applikation – erledigt das nur in kaum messbarer Weise. Der Nachteil an einem externen Service-Layer ist neben der erhöhten Latenz auch das Manifestieren eines bestimmten Software-Stands für alle Entwickler. Ältere und neuere Client-seitige Software-Stände als der Service-Layer können im Allgemeinen nicht auf diesen Dienst zugreifen.

Drei-Schichten-Architektur

Die Grob-Architektur in R8 fußt auf einem integrierten Service-Layer: Sie besteht aus einer Drei-Schichten-Architektur. Da der Service-Layer im Deployment enthalten ist, entsteht keine zusätzliche Netzwerk-Latenz und die Entwickler können jeweils auf beliebig neuen und alten Software-Ständen arbeiten. Die Persistenz-Schicht in R8 greift daher direkt auf einen Cassandra-„Keyspace" zu, auch „Schema" oder „Datenbank" genannt.

Die Begriffe „Ring" oder „Cluster" sind im Falle von Cassandra Synonym für die Hardware-seitige Datenbank oder eine Sammlung von Datenbanken. Der Zugriff geschieht über den DataStax-Binärtreiber mittels Cassandra Query Language (CQL, ein SQL-Dialekt für Cassandra). Cassandra und der dazugehörige Binärtreiber sind frei und kommerziell verfügbar. Beide unterstützen das Speichern der Daten in R8. Treiber gibt

es für viele verschiedene Programmiersprachen wie Java, Ruby, C#, C++, Python, Node.js, Apache Spark sowie PHP. Sie sind auf dem GitHub-Account von DataStax veröffentlicht [2].

Der Grund für die Spezialisierung auf Cassandra war unter anderem die besonders hohe Schreibgeschwindigkeit [3]. Cassandra bietet sich – wegen der Fähigkeit, sehr schnell Daten zu schreiben und zu ändern – sogar als externer Cache an. Die kommerzielle Version von Cassandra, DSE Edition (DataStax Enterprise), ist hierfür besonders gut geeignet, da sie auch reine In-Memory-Tabellen unterstützt [4, 5]. Für offene Cassandra-Distributionen ist die Nutzung als externer Cache allerdings ein Antipattern [6, 7], da die Daten zwangsweise früher oder später auf der Festplatte persistiert werden müssen.

Offene Versionen von Cassandra sind beispielsweise Apache Cassandra, auch Vanilla-Cassandra genannt, oder DSC Edition (DataStax Community) [8]. Die besondere Fähigkeit, schneller schreiben als lesen zu können, kommt allen Anwendungsfällen mit hoher Schreiblast entgegen. Zudem kann Cassandra vor allem durch Tabellen-weite Compaction-Strategies, Consistency-Levels je Query und diverse Caches auch für das Lesen optimiert werden.

Die Fähigkeit, besonders schnell zu schreiben, kommt einerseits aus den bereits genannten Features. Darüber hinaus nutzt Cassandra – wie Hadoop HDFS auch – das Append-Only-Pattern. Dabei werden Daten beim Schreiben nicht verändert, da sich hierbei auch die folgenden Daten verschieben würden. Sie werden beim Ändern als gelöscht markiert und erneut geschrieben. Die als gelöscht markierten Instanzen werden nicht in „ResultSets“ angezeigt und bei regelmäßigen beziehungsweise den gelegentlichen Cleaning-Routinen „Compactions“ übergangen – also nicht erneut geschrieben.

Die seltene Möglichkeit, Daten automatisch nach dem Schreiben zu verwerfen, macht Cassandra außerdem zu einem präferierten Kandidaten für das Speichern von Web-Sessions. Damit werden „heiße Sessions“ im Cassandra-Ring beispielsweise eine halbe Stunde vorgehalten und, sobald sie durch das Ablaufen der Session „erkalten“, automatisch gelöscht. Zudem ist Cassandra hochverfügbar und skalierbar, sowohl horizontal als auch vertikal. Das macht sie zu einer idealen Daten-Grundlage für ebenso hochverfügbare, skalierbare Anwendungen.

Das Ziel von R8 ist es, diese Eigenschaften von der Cassandra-Datenbank in die Welt der Anwendungen zu bringen. Einzelne Knoten können damit den Ring verlassen sowie beitreten, ohne jeweils den Betrieb der Anwendung zu stören. Bei einem unabsichtlichen oder bewusst provozierten

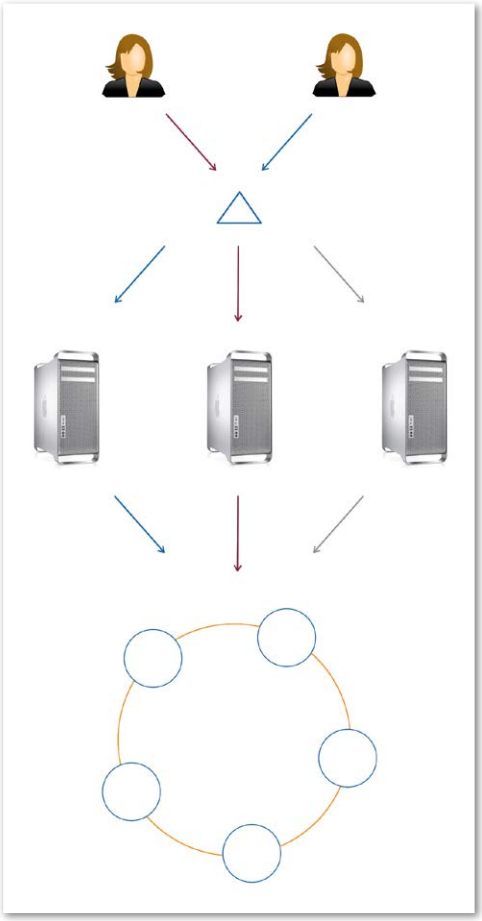


Abbildung 1: Der generelle Aufbau des Softwaresystems besteht aus den drei Schichten Clients, Applikationsserver mit vorgeschaltetem Load Balancer sowie der Datenhaltung in einem Cassandra-Ring. Die roten und blauen Pfeile zeigen den Fluss der Requests der beiden Clients an. Die roten Pfeile zeigen Anfragen resultierend aus Requests des Clients links oben; die blauen Pfeile das Gleiche für den Client rechts oben.

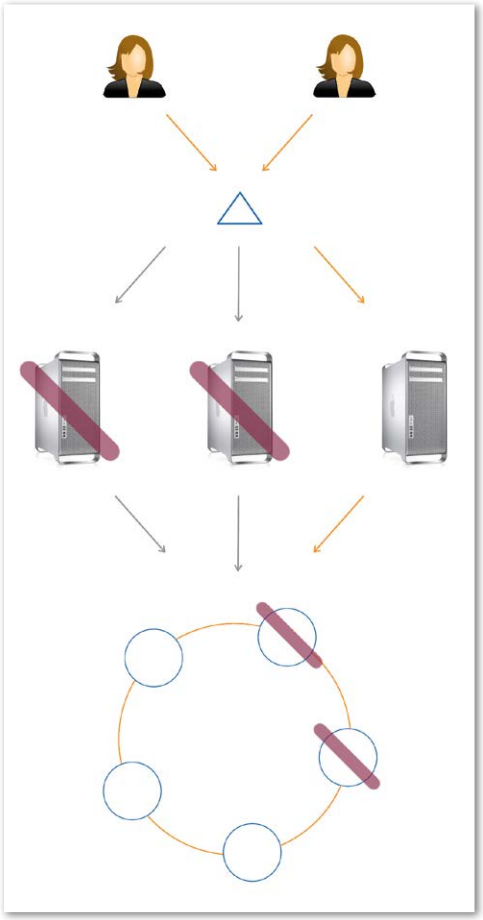


Abbildung 2: Bei Ausfällen einzelner Knoten im Ring werden Requests automatisch an verbleibende Knoten umgeleitet. Das bezieht sich sowohl auf Cassandra-interne als auch auf R8-interne Requests.

Ausfall eines Cassandra-Knotens übernehmen die verbleibenden Knoten im Ring seine Daten und Arbeitslast. Beim Beitritt eines neuen oder vorher ausgefallenen Knotens entlastet dieser seine Pendants.

Hochverfügbare Anwendungen

Die Notwendigkeit von hochverfügbarer Software kommt einerseits aus Anwendungsfällen, in denen eine Anwendung rund um die Uhr für Nutzer auf der ganzen Welt verfügbar sein soll. Ein Beispiel sind Logistik-Portale für weltweit tätige Konzerne mit ihren jeweils lokalen Zulieferern. Darüber hinaus sollten Anwendungen hinsichtlich dem Thema „Continuous Delivery" sicher gegen Ausfälle einzelner Knoten sein. Dabei geht es darum, Software in kleinen Schritten zu verfeinern und so oft wie möglich neu zur Verfügung zu stellen. Diese Iterationsschleifen werden dadurch besonders kurz. Dafür wird die Anwendung sehr oft neu zur Verfügung gestellt. Jedes dieser Deployments kann in einfachen Architekturen der Grund dafür sein, dass die Anwendung offline geht.

Deshalb rollen viele Unternehmen auch lediglich quartalsweise neue Software aus. Auf der anderen Seite können dadurch Fehler nur mit großem, zeitlichem Vorlauf beseitigt werden und Nutzer warten länger auf neue Features. Bei weltweit durchgehend genutzten Systemen gibt es keinen Zeitpunkt, an dem ein neues Deployment ohne Probleme möglich wäre. Daher werden nur sehr selten neue Stände ausgerollt, und zu dem Zeitpunkt, an dem es die wenigsten Anwender stört. Allerdings ist diese Vorgehensweise das Gegenteil von Continuous Delivery. Ausschlaggebend für diese Vorgehensweise ist die Software-Architektur, vor allem das Session-Management – wie für Web-Anwendungen und beispielsweise Datenbank-Sessions.

Bei Continuous Delivery entscheidet das Business über die IT und nicht umgekehrt. Folglich dürfen technische Änderungen den Betrieb und das Testen neuer Stände nicht beeinträchtigen. Damit vergeht für den Anwender deutlich weniger Zeit, bis er neue Funktionalitäten nutzen kann und Fehler beseitigt sind. Die Entwickler der Anwendung können in diesem Szenario Performance-Optimierungen, Parameter-Tuning und den Tausch von Hardware nur auf Test-Systemen vollziehen. Hilfreich wäre es allerdings, das auch im Produktiv-System erledigen zu können. Vor allem Verbesserungen von Laufzeiten und Parametern können einen großen Einfluss auf die System-Performance haben. Diese Änderungen sind jedoch in der Regel nicht im laufenden Betrieb möglich.

Derzeit geschieht das erneute Ausrollen einer neuen Version einer Anwendung normalerweise gleichförmig für alle Applikations-Server. Jeder Server wird zu jeder Zeit mit derselben Version der Software betrieben. Wünschenswert wäre allerdings ein Rolling-Updates-Szenario wie bei Android-Updates: Hier wird eine neue Software-Version zuerst auf einer kleinen Testmenge von Smartphones ausprobiert. Erst wenn sie weitestgehend fehlerfrei läuft und die größten Bugs und Anpassungen für jedes Gerät vollzogen sind, wird die bewährte Version auf der Mehrheit der Geräte installiert *[9]*.

Die Lösung ist eine Architektur, die so hochverfügbar ist wie der Cassandra-Cluster selbst. Damit sind zentrale Anpassungen an einem Knoten möglich, ohne das Gesamtsystem zu stören – selbst wenn dieser Knoten dabei ausfällt. Es muss jeder Knoten unabhängig von anderen ausfallen können, ohne das gesamte System zu beeinträchtigen. Wenn nach außen Ausfälle im Ring verborgen werden, sind auf allen Knoten rollende Updates möglich – wie bei Android. Zudem sind mit dieser Architektur Performance-Optimierungen im Produktiv-System möglich.

Da jeder Knoten unabhängig ausfallen darf, kann seine Java Virtual Machine zum Optimieren der Parameter neu gestartet beziehungsweise auch der Ausfall dieses Teils des Systems riskiert werden. In herkömmlichen Architekturen ist genau das nicht möglich. Netflix optimiert bereits nach dieser Herangehensweise seine Produktivcluster: Sie schießen gezielt Knoten ab („Chaos Monkey") oder verlangsamen gezielt deren Antwort („Latency Monkey") *[10]*.

Data-driven Business

R8 ist ein Framework für ein Daten-getriebenes Geschäft; es erzeugt im Hintergrund automatisch zu jeder Nutzer-Interaktion Daten. Das hilft, die Nutzer des Systems noch besser zu verstehen und das System gezielt zu verbessern. Vor allem sind Metriken interessant, wie lange Service-Level-Agreements noch eingehalten werden können, wo es Performance-Probleme gibt, wo größere Caches sinnvoll wären und auch wo Speicher eingespart werden kann. Da R8 eine Cassandra-Datenbank nutzt, hat die Datenbank keine Probleme mit der Schreiblast und kann bei Bedarf nahezu linear skalieren. Zudem sorgt der Column-basierte Speicher dafür, dass die Daten bestmöglich komprimiert werden.

Darüber hinaus werden gezielt Cassandra-Features genutzt, etwa Batches und asynchrone Requests. Batch-Statements sichern im Fall von R8 die Cache-Statistiken, wie belegter Cache-Anteil, maximale Cache-Größe, Trefferquoten („hit ratios") und „miss penalties". Diese Werte werden zentral gesammelt und alle Statements in einen Batch

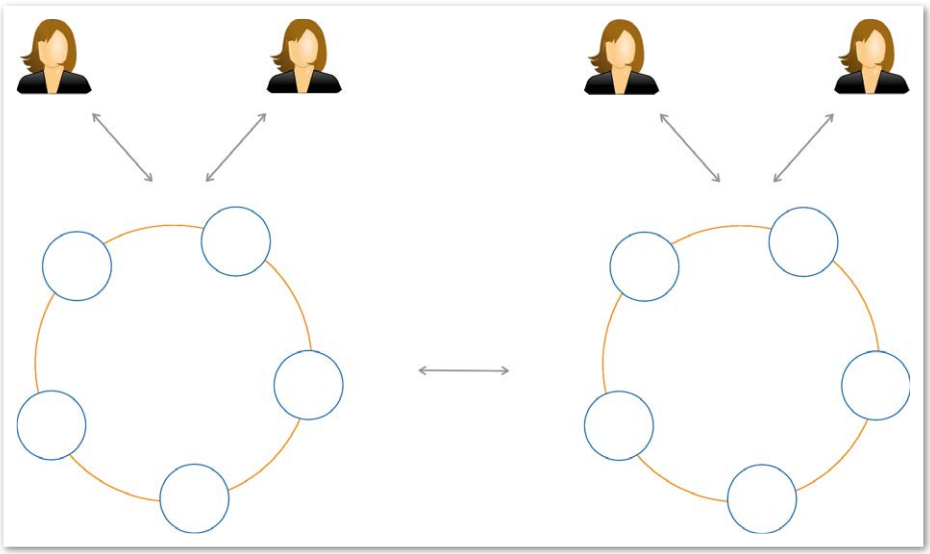


Abbildung 3: Aufbau des Inkonsistenz-Experiments bei Netflix. Es existieren zwei separate Cassandra-Ringe in einem einzigen Keyspace/Datenbank. Beide Ringe bilden jeweils ein Datencenter; diese sind verbunden und werden von Cassandra intern unter Zuhilfenahme des Consistency Levels aus dem Write-Request automatisch synchronisiert.

geschrieben. Das gesammelte Schreiben der Daten ist effizienter und stellt sicher, dass alle Daten den exakt selben Zeitstempel tragen.

Asynchrone Requests wiederum werden für Log-Einträge genutzt. Logs in R8 erfolgen sehr performant nur in die Datenbank und nicht in Logfiles. Per Default werden diese Requests asynchron abgesetzt – also „Fire and Forget". Cassandra hat den Vorteil, dass es durch oben genannte Performance-Features deutlich performanter als das darunterliegende Betriebssystem schreiben kann, wenn dieser Vorgang zeitlich begrenzt stattfindet. Das macht die Applikation nachweislich schneller, da sie nicht auf die Bestätigung von der Datenbank warten muss.

Allein das Schreiben von Logs in die Datenbank durch asynchrone Requests trägt für den Nutzer deutlich spürbar positiv und messbar zur Performance-Optimierung bei. Die so erzielte Beschleunigung ist im Produktiv-System um den Faktor drei besser als synchrone Requests. Per Default erfolgt das Schreiben dieser Logs auf dem Consistency-Level „ANY" *[11]*. Dabei muss der Schreibvorgang nur auf einem aktiven oder passiven („drain") Knoten erfolgen, um erfolgreich zu sein. Ein sogenannter „Hinted Handoff" (Vermerk für das spätere Schreiben auf einem aktiven Knoten) genügt hierbei *[12]*. Änderungen des Default-Verhaltens sind einfach konfigurierbar.

Das Framework kümmert sich darüber hinaus im Hintergrund um die nötigen Data-Definition-Language-Statements (DDL) und legt damit Tabellen mit Spalten, Partition-Keys, Clustering-Columns und Secondary Indizes, aber auch Prepared Statements an. Letztere werden für alle lesenden und schreibenden Datenbank-Zugriffe genutzt. Diese können in besonderen Ausnahmen allerdings auch umgangen werden.

Durch das Nutzen von Prepared Statements allein ist die Laufzeit des Statements ungefähr verdoppelt und SQL-Injections werden deutlich erschwert *[13]*. Das automatische Erzeugen der Datenbank-Strukturen („Keyspace") macht R8 auch für Entwickler interessant, die sich nicht mit Cassandra-Interna auskennen. Sie definieren ein denormalisiertes, abstraktes Schema und R8 legt die optimalen Strukturen inklusive der Datenbank-seitigen Caches, Daten-Komprimierungen und Table-Strategies automatisch an.

Auf der anderen Seite kann R8 auch Full-Table-Scans erzwingen. Diese sind durch die verteilte Natur von Cassandra besonders teuer. Sie können aber eine willkommene Alternative zum Anlegen einer weiteren Tabelle sein, da Cassandra ausschließlich denormalisierte Tabellen unterstützt. Sollte also die Laufzeit für eine Query nicht wichtig sein – wie in einigen analytischen Statements – so kann R8 ein Statement für Cassandra erzeugen, das zu einem Full-Table-Scan führt – mit dem Keyword „allow filtering".

Full-Table-Scans sind in vielen Datenbanken die teuerste Art des lesenden Datenzugriffs. Vor allem verteilte Datenbanken wie Cassandra, MongoDB und HBase zeigen bei Full-Table-Scans durch ihre verteilte Natur deutliche Defizite. Da sie in Cassandra besonders teuer sind, führen sie und R8 diese Statements nur auf ausdrücklichen Wunsch aus.

Besonders in BI-Systemen mit großen Datenmengen sind „JOIN"-Statements zur Laufzeit besonders unperformant. Der negative Effekt verstärkt sich mit steigender Datenmenge, Komplexität der JOINs sowie der Schachtelungstiefe der SQLs. Die beiden letzteren Probleme können in einer Cassandra systembedingt nicht auftreten, da weder JOINs noch „nested statements" unterstützt werden. Durch ihren starken Fokus auf Big-Data-Systeme ist das eine durchaus sinnvolle Restriktion.

In anderen Datenbanken bedarf es in der Regel keiner speziellen Parameter für das Erzwingen dieser Statements. Das hat den Nachteil, dass Full-Table-Scans teilweise ständig im Hintergrund laufen und das gesamte System ausbremsen *[14]*. Um das zu verhindern, bieten sich regelmäßige Performance-Reviews in allen Datenbank-Systemen an.

Architektonische Verbesserungen

Für das gesamte System ist der Einsatz von Cassandra auch vorteilhaft, da die Architekten keine eigene Scale-Out-Strategie festlegen müssen. Eine Cassandra lässt sich in jedem Fall fast linear erweitern. Zudem trägt sie entscheidend zu dem Ziel der Zero-Downtime bei. Durch sinkende Kosten für Rechenzeit in der Cloud und das Ausbreiten von Bot-Netzen gehören DoS- und DDoS-Attacken (Denial of Service) nicht nur für große Unternehmen zum Alltag.

Studien zufolge wird jedes zweite Unternehmen monatlich angegriffen. Diese Unternehmen zahlen jedes Jahr zwischen sechs und zwölf Millionen Euro *[15, 16, 17, 18, 19]* allein für das Beseitigen der direkten Probleme aus solchen Attacken. Investitionen zur Prävention fallen zusätzlich an. Die Kosten für hochverfügbare Software amortisieren sich unter diesem Aspekt kurz nach Umstellung der Systeme *[20]*.

(Verteilte) DoS-Attacken sind allerdings auch auf dem Vormarsch in kleinen und mittelständischen Unternehmen. Da Zulieferer häufig weniger gut abgesichert sind, bieten sich Angriffe hier besonders an *[21]*. Für Banken und Informations-zentrierte Unternehmen können (verteilte) DoS-Attacken auch deutlich teurer als einstellige Millionenbeträge ausfallen. Einige der Hauptprobleme sind der Personaleinsatz, die Imageschäden und möglicherweise das Erliegen des operativen Geschäfts *[22]*. Mit einer hochverfügbaren Architektur ist es für Angreifer deutlich schwieriger, einzelne Knoten oder das gesamte System zu schädigen.

Wenn beliebte Web-Dienste wie soziale Netzwerke ausfallen, macht diese Neuigkeit auf den verbleibenden Kanälen sofort die Runde. Denn es wird erwartet, dass das Internet rund um die Uhr verfügbar ist *[23]*. Die Imageschäden *[24]* durch Berichte in der Presse und in sozialen Netzen führen dazu, dass Hochverfügbarkeit in vielen Unternehmen zu einem strategisch wichtigen Ziel wird. R8 bietet hierfür eine solide Grundlage, um Enterprise-Anwendungen, Webseiten und Service-Layer zu erstellen. Das gesamte System ist dadurch deutlich robuster gegen erhöhte Last und Angriffe. Die genutzte Web-Engine in R8 ist Apache Wicket. Wenn Entwickler lieber JSP/JSF oder ein anderes UI-Framework nutzen, kann R8 auch als reines Backend fungieren.

Verhaltens-basierte Firewall

R8 sichert sich gegen (verteilte) DoS-Angriffe zudem mithilfe einer eigenen Firewall ab. Diese wird „Fort Knox" genannt *[25]*, ist Teil des Deployments und erweitert statische Hardware-seitige Firewalls und andere wie etwa Apache ModSecurity. Fort Knox ist eine verhaltensbasierte Firewall, die Nutzer zeitweise blockiert, wenn diese bösartige Nutzungsmuster zeigen. Diese Muster werden aufgrund der Zugriffe in typischerweise sekundenweisen Intervallen mit historischen Nutzungen abgeglichen.

Die Intervalle für einen solche Abgleich sind einfach konfigurierbar. Dabei werden kurzzeitig höhere Lasten, wie solche durch noch leere Caches beziehungsweise gerade

invalidierte Caches, berücksichtigt. In beiden Fällen sieht Fort Knox den sprunghaften Anstieg der Requests als ungefährlich an. Zudem wird eine nur kurzzeitig anhaltende, aber hohe Datenlast verursachende Nutzung als ungefährlich angesehen – die Firewall toleriert diesen Nutzer. Dieses Verhalten kann beispielsweise durch das Öffnen mehrerer Tabs oder Fenster aus einer Sammlung von Links auftreten. Werden hingegen Angriffe gestartet, die nicht kurzer Natur sind, so blockiert die Firewall unverzüglich das Verarbeiten weiterer Requests von diesem Computer, um die Applikation gegen Angriffe zu schützen.

Session-Management

Es existieren im Allgemeinen drei wichtige Arten von Session-Management: einfache Sessions „nach dem Lehrbuch", Sticky Sessions und Session-Replikation. Sie haben entscheidenden Einfluss auf die Hochverfügbarkeit des Systems. Die einfachste Möglichkeit – nach dem Lehrbuch – ist nicht verteilt und damit auch nicht hochverfügbar, da jeder Webserver seine eigene Session aufbaut. Die Sessions werden im Speicher des jeweiligen Servers gehalten und im Zweifel auf die lokale Festplatte ausgelagert.

Sticky Sessions werden oftmals in Verbindung mit verteilten Systemen genutzt. Hier arbeitet jeder Nutzer immer nur auf einem einzelnen Server. Der Load-Balancer erkennt anhand der Session-ID, welcher das ist, und routet alle Requests immer auf diesen Server. Einen Server-Ausfall verzeiht dieses System allerdings nicht, da die Nutzer ihre Session verlieren, wenn deren Server offline geht. Das Problem ist zwar eingegrenzt, aber das System noch immer nicht hochverfügbar.

Sticky-Sessions sind eine gute Möglichkeit, die Applikation horizontal zu skalieren. Sie sind aus Sicht der Performance-Optimierung ein guter Lösungsansatz, da sie die Server ungefähr gleichverteilt belasten. Sticky Sessions sind allerdings weder notwendig, noch hinreichend für den Bau einer HA-Architektur. Sie sind allerdings eine Best Practice für verteilte Architekturen. Auch hier werden die Sessions im Speicher des jeweiligen Servers gehalten und im Zweifel auf die lokale Festplatte ausgelagert.

Session-Replikation beschreibt ein Szenario, in dem Server ihre Web-Sessions sowohl im Speicher halten als auch in eine zentrale Datenbank persistieren. Aus dieser Datenbank können alle Web-Server Sessions laden, die sie selbst nicht (mehr) im Speicher haben – beispielsweise bei einem Ausfall des ursprünglichen Servers. Der einspringende Server würde in diesem Fall die bereits erstellte Session anhand der Session-ID aus der zentralen Datenbank laden und im Speicher des neuen Web-Servers ablegen. Hier ist das Auslagern auf die lokale Festplatte lediglich empfehlenswert.

Es empfiehlt sich, auch R8-basierte Applikationen hinter einem Load-Balancer mit Sticky Sessions zu betreiben. Denn bei einem Round-Robin-Wechsel vom Client auf immer neue Server je Request führen Lookups in der Regel zu einem Zugriff auf die Datenbank und machen damit Caches deutlich weniger effizient.

R8 hat eine eingebaute und besonders effiziente Session-Replikation. Dieses Feature kann auch ohne R8 genutzt werden, um eine hochverfügbare Architektur zu bauen. Voraussetzung dafür ist, dass der Web-Server Session-Replikation unterstützt. Das ist jedoch bei modernen Web-Servern der Regelfall:

- Tomcat & Jetty (beide jeweils als „Session Clustering")
- JBoss AS & WildFly (als „Session Failover")
- GlassFish als „Session Persistence" / „HADB" in eine Datenbank oder als „In-Memory Replication" zwischen zwei Instanzen
- WebLogic (als „Session Replication")
- WebSphere als „Distributed Sessions" in eine Datenbank oder als „Memory-To-Memory"-Lösung
- TomEE (als „Session Replication")
- Resin (als „Distributed Caching")

Für Netty ist ein externer Cache wie Infinispan oder Hazelcast erforderlich.

CAP-Theorem

Das CAP- beziehungsweise Brewer-Theorem besagt, dass verteilte Systeme nach ihrer Architektur nur zwei von drei möglichen Eigenschaften mit Sicherheit haben können [26, 27]. Diese Eigenschaften sind Konsistenz („Consistceny") *[28]*, Hochverfügbarkeit („Availability") *[29]* und Partitions-Sicherheit („Partition Tolerance") *[30]*. Sowohl Cassandra als auch R8-Applikationen sind nach dem Brewer-Theorem „AP" – also hochverfügbar und perfekt partitioniert *[31]*. Damit ist R8 nach dem CAP-Theorem mit Domain-Name-Servern vergleichbar und das aus folgenden systematischen Gründen: Hochverfügbarkeit bedeutet, dass ein System stets antwortet und einzelne Knotenausfälle im Hintergrund kaschiert werden.

Partitions-Sicherheit bezeichnet die Verteilung der Daten beispielsweise in einer Datenbank. In Cassandra werden verschiedene Zeilen aus einer Tabelle je nach ihrem Partition-Key (Hash aus vorderem Teil ihres Primärschlüssels oder Hash aus ganzem Primärschlüssel) auf diverse Knoten verteilt. Somit kann bei einem Knotenausfall ermittelt werden, wo die jetzt verlorenen Daten in Kopie vorliegen und wo eine weitere Kopie erstellt wird.

Die Eigenschaft, die sowohl Cassandra als auch R8 fehlt, ist die strenge Konsistenz der Daten. Bei Cassandra spricht man von „tunable consistency" *[32]*. Diese Eigenschaft erlaubt es, Geschwindigkeit und Hochverfügbarkeit gegen Konsistenz einzutauschen. Früher sprach man von „eventual consistency". Davon ist man in letzter Zeit abgerückt, da aus dieser Formulierung teilweise der Schluss gezogen wurde, dass man selbst nicht wüsste, ob die Daten nun konsistent seien oder nicht.

Um klarzustellen, dass die Anwender volle Kontrolle über die Konsistenz ihrer Daten haben, spricht man daher von „tunable consistency" *[33]*. Trotz dieses irreführenden Wordings sind Banken ein dankbarer Nutzer von Cassandra – vor allem wegen ihrer Hochverfügbarkeit und der Möglichkeit, nahezu linear zu skalieren. In dieser Branche wird Cassandra in der Regel auf strenge Konsistenz konfiguriert. Für R8 bedeutet Inkonsistenz lediglich, dass verschiedene Versionen einer Anwendung auf den Knoten in einem Cluster betrieben werden können – das eignet sich vor allem für rollende Updates *(siehe Abbildung 3)*.

Inkonsistenz-Experiment mit Netflix

Netflix-Experimente mit zwei großen Cassandra-Ringen haben die tatsächliche Konsistenz auf einem theoretisch inkonsistent konfigurierten Ring untersucht. In zwei verbundene Cluster wurden oft und schnell Daten eingefügt, verändert und im jeweils anderen Cluster wieder gelesen. Die internen Prozesse einer Cassandra sorgen zwar dafür, dass sich die verbundenen Ringe selbst synchronisieren. Allerdings garantiert Cassandra Konsistenz nur, wenn sie als konsistent konfiguriert wird. Erwartet hatten die

Tester, dass durch das Schreiben und Lesen auf dem zweit-geringsten Konsistenzniveau „ONE" in zwei sich oft und schnell ändernden Cassandra-Clustern viele inkonsistente Daten auftreten würden. Gemessen wurde kein einziger inkonsistenter Datensatz *[34]*.

Das heißt, dass Cassandras interne Prozesse so gut arbeiten, dass man kaum einen praktischen Unterschied feststellen kann. Für das Arbeiten mit diesem theoretisch inkonsistenten Stand ergaben sich allerdings ein paar praktische Vorteile: Die Latenz ist um den Faktor drei zurückgegangen und der Ring ist damit auch weniger anfällig gegen Knoten-Ausfälle, da jeweils praktisch nur noch eine Kopie vorhanden sein muss.

Theoretisch müsste ein Quorum, also mindestens die Hälfte aller verfügbaren Kopien, sprich mindestens jeder zweite der Knoten im Ring, verfügbar sein. Die sinkende Latenz ist Ergebnis des Nutzens eines Token-basierten Load Balancer *[35]* im Client und des durch das geringe Konsistenz-Niveau entfallenden Read Repair. Daher betreiben Netflix *[36]* und Spotify *[37]* ihre jeweiligen Cassandra-Cluster auf Consistency-Level „ONE" *[38]*.

Die Eigenschaft, eine Cassandra-Datenbank zu tunen, und das durch die Hochverfügbarkeit auch im Produktiv-System erledigen zu können, ist ein ausschlaggebendes Feature, um Cassandra zu nutzen. Außerdem kann Cassandra im Gegensatz zu vielen anderen Datenbanken, schneller schreiben als lesen. Durch ihre Ring-Architektur ist sie auch besonders einfach zu betreiben und überlebt Ausfälle ganzer Rechenzentren. Der Client-seitige Binärtreiber erlaubt zudem einen sehr effizienten Zugriff und eine gleichverteilte Last auf den Kopien der Daten im Ring. Obendrein kann eine Cassandra eine sehr schnelle und hochverfügbare Analyse-Plattform im Zusammenspiel mit Apache Spark sein. Die oben genannten Features beziehen sich auf die frei einsetzbaren Cassandra-Distributionen.

Die kommerzielle Version DataStax Enterprise Edition (DSE) bietet darüber hinaus technischen Support, zusätzliche Qualitätssicherungsmaßnahmen für viele Plattformen, Management-Tools für Administratoren, eine Cluster-weite Suchfunktion und eine verbesserte Hadoop-Integration. Hinzu kommen auch In-Memory-Tabellen, die Cassandra in einen externen Cache verwandeln, und die Verschlüsselung von gespeicherten, ruhenden Daten – zusätzlich zur Transport-Verschlüsselung in allen Versionen.

Fazit

R8 ist ein sehr spezifisches Framework: Es wurde nicht dazu gebaut, um Backend- oder Frontend-Frameworks abzulösen. Es eignet sich vor allem für Systeme und Anwendungen mit besonders hohen Anforderungen an Hochverfügbarkeit, Performance und Skalierbarkeit. R8 erfordert spezielle Erfahrungen von Entwicklern: Beispielsweise werden kein standardisiertes Persistenz-Framework, eine in der Welt der Anwendungssoftware seltene Datenbank und ein spezielles UI-Framework wie Wicket genutzt. Darüber hinaus stellt es spezifische Anforderungen an das IT-System wie etwa eine Cassandra-Datenbank als Speicher.

Das Framework nimmt dem Nutzer viele Kleinigkeiten ab, zum Beispiel das automatische Anlegen von Daten-Strukturen, das Sammeln von Request-Daten und das Optimieren von Datenbank-Statements. Weiterhin ist R8 out of the box hochverfügbar und seine eigene Firewall sichert es zusätzlich gegen (verteilte) Denial-of-Service-Angriffe ab.

Weitere Informationen

[1] en.wikipedia.org/wiki/Multilayered_architecture#Common_layers
[2] github.com/datastax
[3] planetcassandra.org/nosql-performance-benchmarks
[4] datastax.com/documentation/datastax_enterprise/4.0/datastax_enterprise/inMemory.html
[5] datastax.com/wp-content/uploads/2014/02/WP-DataStax-Enterprise-In-Memory.pdf
[6] Cassandra Design Patterns, von Sanjay Sharma, erschienen bei Packt Publishing, 24. Januar 2014, ISBN 978-1783288809
[7] de.slideshare.net/AxelLiljencrantz/cassandra-summit-2013-how-not-to-use-cassandra
[8] datastax.com/download/dse-vs-dsc
[9] androidauthority.com/how-android-updates-roll-out-force-clear-gcf-318744
[10] techblog.netflix.com/2011/07/netflix-simian-army.html
[11] datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html
[12] wiki.apache.org/cassandra/HintedHandoff
[13] techblog.netflix.com/2013/12/astyanax-update.html
[14] dba-oracle.com/art_dbazine_expert_tuning.htm
[15] infosecurity-magazine.com/news/a-ddos-attack-could-cost-1-million-before
[16] ddosattacks.biz/impact/ddos-can-cost-1m-mitigation-even-starts
[17] securityweek.com/ddos-attacks-cost-40000-hour-incapsula
[18] scmagazine.com/incapsula-found-the-of-ddos-attacks-to-be-substantial/article/383179
[19] incapsula.com/blog/ddos-impact-cost-of-ddos-attack.htmllp.incapsula.com/rs/incapsulainc/images/eBook%20-%20DDoS%20Impact%20Survey.pdf
[20] neustar.biz/resources/whitepapers/ddos-protection/2014-annual-ddos-attacks-and-impact-report.pdf
[21] kaspersky.com/de/about_kaspersky/news/virus/2013/Jahrestrend_2013_Gezielte_Attacken_auf_Unternehmen_Partner_und_Zulieferer
[22] security.radware.com/uploadedFiles/Resources_and_Content/Attack_Tools/CyberSecurityontheOffense.pdf
[23] mercurynews.com/news/ci_27399971/great-instagram-and-facebook-outage-january-2015-social
[24] blog.radware.com/security/2013/05/how-much-can-a-ddos-attack-cost-your-business
[25] Securing Web Applications with a Behaviour-Based Firewall, Lawrie Brown et al., Seminarunterlagen, Belvine University
[26] royans.net/wp/2010/02/14/brewers-cap-theorem-on-distributed-systems
[27] julianbrowne.com/article/viewer/brewers-cap-theorem
[28] en.wikipedia.org/wiki/Consistency_(database_systems)#As_a_CAP_trade-off
[29] en.wikipedia.org/wiki/High_availability#System_design_for_high_availability
[30] en.wikipedia.org/wiki/Network_partition#As_a_CAP_trade-off
[31] radar.oreilly.com/2013/03/returning-transactions-to-distributed-data-stores.html
[32] labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra
[33] youtube.com/watch?v=A6qzx_HE3EU
[34] planetcassandra.org/blog/a-netflix-experiment-eventual-consistency-hopeful-consistency-by-christos-kalantzis
[35] 2014.nosql-matters.org/cgn/wp-content/uploads/2014/04/GoingNativeWithApacheCassandra.pdf
[36] techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html
[37] sics.se/sites/default/files/pub/sics.se/scaling_storage_to_millions_of_users_-_presentation_at_sics_ws.pdf
[38] youtube.com/watch?v=v5stV9Knpw0

Daniel Schulz
daniel.schulz@capgemini.com

Daniel Schulz ist Senior Solution Architect bei Capgemini in Deutschland. Er arbeitet seit vier Jahren im Big-Data-Bereich mit besonderem Fokus auf der Automotive-Branche. Er interessiert sich seit seiner Schulzeit für Statistik, seit dem Studium auch für Machine Learning und dessen Einsatz in der Daten-Analyse. Sein besonderes Interesse gilt Markov-Modellen und der Performance-Optimierung von Software und Datenbanken.

Software-Archäologie mit AspectJ

Oliver Böhm, Java User Group Stuttgart

Neue Projekte auf der grünen Wiese sind selten. Viel öfter wird man mit Altlasten konfrontiert und steht vor der undankbaren Aufgabe, vergangene Architekturen wieder freizulegen und unbekannten Code zu deuten, um das Projekt wieder zu vergangenen Hochkulturen zurückzuführen. Dabei gibt es verschiedene Möglichkeiten, dem Code seine Geheimnisse zu entreißen. Ein vielversprechender Ansatz ist dabei der Einsatz von Aspektorientierung und AspectJ, um unbekannte Code-Stellen zu erschließen und das Programmverhalten zu ergründen.

Historisch gewachsene SW-Systeme können viel erzählen: von Übernahmen durch andere Firmen (erkennbar an verschiedenen Package-Namen), von verschiedenen Programmier-Epochen (Outsourcing) oder vom (Un-)Kenntnisstand der verschiedenen Architekten. Oft enthalten sie aber auch viele Geheimnisse (Warum wurden sie erschaffen? Was machen sie eigentlich?), die die Einarbeitung in solche Relikte vergangener Hochkulturen erschweren, auch weil viele Zeugen des Entwicklungsgeschehens nicht mehr auffindbar sind und das Wissen aus Zweit- oder gar Dritt-Quellen und anderen Überlieferungen abgeleitet werden muss *(siehe Abbildung 1)*.

Die klassische Herangehensweise

Meist ist die Dokumentation die erste Anlaufstelle, um sich mit einer Alt-Anwendung vertraut zu machen. Auch wenn sie oftmals von der Realität abweicht, gibt sie doch einen wertvollen Einblick und wichtige Informationen, die den Einstieg in die Thematik erleichtern.

Im Ideal-Fall ist eine aktuelle Dokumentation vorhanden, die diesen Namen auch verdient und verschiedene Sichten auf die Anwendung erlaubt. Vor allem Übersichts-Dokumente über die Architektur, Infrastruktur und Randbedingen sind hier für die Einarbeitung, aber auch als Nachschlagewerk sehr hilfreich.

Um festzustellen, inwieweit der Code tatsächlich mit der Dokumentation übereinstimmt, sind Testfälle (sofern vorhanden) von eminenter Bedeutung. Sie bilden den Ausgangspunkt, um verschiedene Programmteile zu erkunden und deren Verhalten zu studieren. Die häufigsten Schwierigkeiten bestehen meist darin, die Testfälle zum Laufen zu bringen.

Hat man die Tests am Laufen, kann man sich an Refactoring-Maßnahmen wagen mit dem Ziel, die Business-Logik stärker zum Vorschein zu bringen und die Lesbarkeit und Wartbarkeit zu erhöhen. Vor allem JEE-Anwendungen sind oftmals „over designed", was eine Einarbeitung meist erschwert. Mit den neuen Erkenntnissen sollte nochmals die Dokumentation begutachtet werden: Welche Dokumente müssen überarbeitet, welche können aussortiert oder/und in Form von neuen Testfällen ausgedrückt werden.

Der Normal-Fall

Die Realität sieht oft ganz anders aus: Die Dokumentation ist häufig Teil des Problems und nicht der Lösung. Ehemalige Entwickler sind samt Know-how im Bermuda-Dreieck des Outsourcings verschwunden und als Test-Umgebung dient das Produktiv-System. Das bedeutet dann, im Code zu graben und die Mosaikstücke der ursprünglichen Architektur richtig zu deuten. Alles, was dabei hilft, den Source-Code zu verstehen, hilft auch bei der Rekonstruktion des Gesamt-Systems.

Auch die wenigen Test-Ruinen, die man oft noch findet, sind von unschätzbarem Wert, geben sie doch einen Einblick in die dynamischen Zusammenhänge. Zusammen mit aspektorientierten Code-Injektionen kann man darauf aufbauend neue Erkenntnisse über den Zusammenhalt der Code-Bereiche gewinnen.

Abbildung 1: Unbekannter Code

Die Welt der aspektorientierten Programmierung

Als Einstieg in die Begriffs- und Gedankenwelt von aspektorientierter Programmierung (AOP) dient ein Beispiel aus dem Banken-Bereich *(siehe Listing 1)*.

Dies ist ein sehr einfaches Modell einer Konto-Klasse, bei dem die Business-Logik (Einzahlung, Auszahlung, Überweisung) noch sehr gut erkennbar ist. Eine Warnung vorneweg: Für Real-World-Implementierungen bitte nie „double" als Datentyp verwenden, sonst kann es zu bösen Überraschungen aufgrund von Rundungsfehlern kommen *[1]*.

Schleichende Code-Verschmutzung

„Alle Kontobewegungen müssen protokolliert werden." Diese gesetzliche Anforderung soll jetzt umgesetzt werden. Dazu wird die Konto-Klasse erweitert *(siehe Listing 2)*.

Ganz langsam tritt damit die eigentliche Business-Logik in den Hintergrund. Zudem warten noch jede Menge weitere Anforderungen (oft auch als „Concerns" bezeichnet), die technischer Natur sind und mit der eigentlichen Fachlichkeit nichts zu tun haben: Autorisierung, Sicherheit, GUI, Transaktionen etc. *(siehe Abbildung 2)*.

Während des Informatik-Studiums bekommt man mit Dijkstras „Separations of Concerns" die Empfehlung mit auf den Weg, jeden „Concern" in einem eigenen Modul oder einer eigenen Klasse zu kapseln, aber bereits dieses einfache Logging-Beispiel zeigt, dass das mit den Mitteln der Objekt-Orientierung gar nicht so einfach ist *[2]*.

Separation of Concerns

Ein Ausweg aus diesem Dilemma bietet die Aspektorientierung. Sie verfügt über Konzepte, um solche meist technischen Anforderungen (im AOP-Jargon als „Crosscutting Concerns" bezeichnet) in eigene „Aspekte" kapseln zu können *(siehe Abbildung 3)*. Ein Vertreter aspektorientierter Sprachen ist AspectJ *[3]*, die Java um einige Sprachmittel erweitert und damit das Herauslösen der Logging-Funktionalität aus der Konto-Klasse ermöglicht *(siehe Listing 3)*. Übersetzt bedeutet dieser Code: „Wenn sich der Kontostand ändert, gib eine Log-Meldung aus." Für das Verständnis des Codes sind einige Begrifflichkeiten aus der AOP-Welt von Vorteil, die im nächsten Abschnitt vorgestellt werden.

```
public class Konto {

  private double kontostand = 0.0;

  public double abfragen() {
     return kontostand;
  }

  public void einzahlen(double betrag) {
     kontostand = kontostand + betrag;
  }

  public void abheben(double betrag) {
     kontostand = kontostand - betrag;
  }

  public void ueberweisen(double betrag,
        Konto anderesKonto) {
     abheben(betrag);
     anderesKonto.einzahlen(betrag);
  }

}
```

Listing 1: Eine einfache Konto-Klasse

```
public class Konto {

  private static Logger log = Logger.getLogger(Konto.class);
  private double kontostand = 0.0;

  public double abfragen() {
     return kontostand;
  }

  public void einzahlen(double betrag) {
     kontostand = kontostand + betrag;
     log.info("neuer Kontostand: " + kontostand);
  }

  public void abheben(double betrag) {
     kontostand = kontostand - betrag;
  }

  public void ueberweisen(double betrag,
        Konto anderesKonto) {
     abheben(betrag);
     anderesKonto.einzahlen(betrag);
     log.info("neuer Kontostand: " + kontostand);
  }

}
```

Listing 2: Konto-Klasse mit Logging

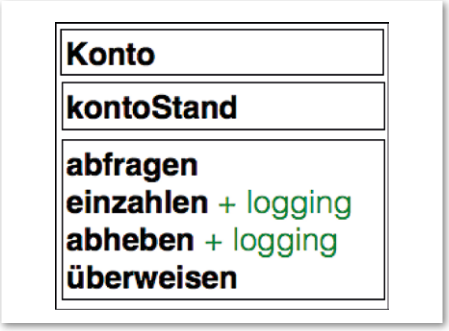


Abbildung 2: Separation of Concerns?

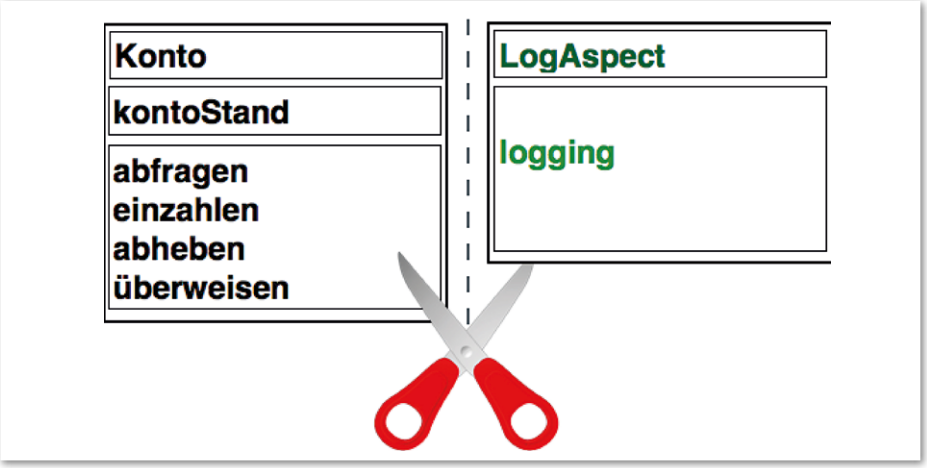


Abbildung 3: Separation of Concerns

Do you speak AOP?

Genauso wie SQL über eigene Begriffe wie „Relationen" und „Normalform" verfügt, grenzt sich die Aspektorientierung mit ihrer eigenen Begriffswelt von anderen Techniken ab. Die wichtigsten sind:

- Aspekt
- Joinpoints
- Pointcuts
- Advice

Der Aspekt ist für AOP das, was für OOP die Klasse bedeutet: der Container für die neuen Sprachmittel *(siehe Listing 4)*.

„Joinpoints" heißen im AOP-Jargon all die Punkte im Programm, auf die Einfluss genommen werden kann. Im Falle von AspectJ (und den meisten anderen AOP-Sprachen) sind dies:

- Aufruf („call") oder Ausführung („execution") einer Methode oder eines Konstruktors
- Initialisierung einer Klasse oder eines Objekts („initialization", „preinitialization", „staticinitialization")
- Zugriff auf eine Instanz-Variable („set", „get")
- Exception-Handling („handler")

Pointcuts adressieren dann diese Joinpoints *(siehe Listing 5)*.

Der ersten Pointcut („setKontostand") ist schon aus dem Beispiel des LogAspect in *Listing 3* bekannt. Er adressiert den schreibenden Zugriff auf das „kontostand"-Attribut. Interessanter ist der zweite Pointcut („callBankMethods"), der alle Methoden der verschiedenen Konto-Klassen anspricht (genauer: jene, die mit „...Konto" im Namen aufhören und im „bank"-Paket liegen) und beliebig viele Argumente hat. Möglich wird dies durch die Unterstützung verschiedener Wildcards („*" und „.."), die der Schlüssel für das Herausziehen der Crosscutting Concerns sind.

Das Ganze lässt sich noch mit booleschen Operatoren verknüpfen und mit weiteren Bedingungen kombinieren, sodass sich damit (fast) alle erdenklichen Szenarien realisieren lassen. Seit AspectJ 5 können auch noch Annotations zur Selektion herangezogen werden, was für die Lesbarkeit und Wartbarkeit enorme Vorteile hat.

„Advices" heißen die Methoden der Aspektorientierung, die Punkte im Programm, die über die Pointcuts adressiert sind, um zusätzliche Funktionalitäten anreichern zu können (oder kurz: zu manipulieren). Dabei besteht die Wahl, wann der Advice gegenüber dem Pointcut ausgeführt wird *(siehe Listing 6)*.

- vor (Before-Advice)
- nach (After-Advice)
- anstatt (Around-Advice)

Hier wird nach dem Pointcut „setKontostand" die entsprechende Log-Meldung ausgegeben. Die Verknüpfung mit dem „args"-Schlüsselwort von AspectJ dient in diesem Beispiel nur dazu, auf das Argument (also das Attribut) des Pointcut zugreifen zu können. Alternativ erlaubt AspectJ auch den Zugriff auf den Kontext des Joinpoint.

Der Webe-Vorgang

Damit die Aspekte in das fertige Programm kommen, gibt es zwei Techniken:

- Compile-Time Weaving (CTW)
- Load-Time Weaving (LTW)

Bei Compile-Time Weaving (CTW) werden die Aspekte nach Java übersetzt und mithilfe des bestehenden Java-Compilers in

```
public aspect LogAspect {

  private static Logger log =
     Logger.getLogger(LogAspect.class);

  pointcut setKontostand() :
     set(double bank.Konto.kontostand);

  after(double neu) : setKontostand() && args(neu) {
     log.info("neuer Kontostand: " + neu);
  }

}
```

Listing 3: LogAspect

```
public aspect LogAspect {
  // ...
}
```

Listing 4: Leerer „LogAspect"

```
pointcut setKontostand() :
  set(double bank.Konto.kontostand);
pointcut callBankMethods() :
  call(* bank.*Konto.*(..));
```

Listing 5: Definition zweier Pointcuts

den bestehenden Java-Code „eingewebt". Dabei muss der Java-Code nicht unbedingt als Source-Code vorliegen, sondern der AspectJ-Compiler verträgt auch Byte-Code beziehungsweise „.jar"-Dateien als Input.

Das Load-Time Weaving (LTW) „webt" die Aspekte während des Ladens der Java-Klassen ein. Diese Technik wird zum Beispiel vom Spring-AOP-Framework verwendet, das intern auf AspectJ aufsetzt. Der Vorteil hierbei ist, dass man mit den Aspekten eine größere Reichweite hat (prinzipiell jede Klasse, die geladen wird). Als Nachteil erkauft man sich Syntax-Fehler, die erst zur Laufzeit bemerkt werden.

AspectJ beherrscht beides und das AJDT-Plug-in für Eclipse unterstützt neben der inkrementellen Kompilierung die Visualisierung von Pointcuts und das Debuggen von Advices.

Grabungstechniken mit AOP

Nach diesem Ausflug in die wunderbare Welt der Aspekte, die uns einen kleinen Einblick in die Denkweise ermöglicht, wenden wir uns wieder unserem Fundstück aus der Steinzeit des Java-Zeitalters zu. Das größte Manko bei Alt-Anwendungen (und nicht nur dort) sind die Testfälle. Meistens fehlen sie oder sind genauso veraltet wie die Dokumentation. Ist die Anwendung noch in Betrieb, kann man versuchen, daraus Testfälle für die weitere Entwicklung abzuleiten. Und genau hier kann AOP helfen, fehlende Log-Informationen zu ergänzen oder die Kommunikation mit der Umgebung aufzuzeichnen.

Die interessanten Stellen sind vor allem die Schnittstellen zur Außenwelt. Bei JEE-Applikationen sind dies meist andere Systeme wie Legacy-Anwendungen oder Datenbanken, die über das Netzwerk angebunden werden. Hier reicht es oft aus, die Anwendung ohne Netzwerkverbindung zu starten und zu beobachten, wo überall Exceptions auftreten *(siehe Listing 7)*.

Aus dieser Exception kann man erkennen, dass die Anwendung auf eine MySQL-Datenbank zugreift, aber kein Connection-Objekt vom „DriverManager" bekommen hat. Mit diesem Wissen kann man alle Connection-Aufrufe genauer unter die Lupe nehmen *(siehe Listing 8)*.

Mit diesem Advice wird am Ende jedes Connection-Aufrufs der Aufruf selbst mitsamt seinen Parametern (wie „SELECT"-Statements) und dem Rückgabe-Wert ausgegeben (über die „getAsString()"-Methode, hier nicht abgebildet). Ähnlich kann man bei Netz- beziehungsweise Socket-Verbindungen verfahren, indem man die Programm-Punkte (Joinpoints) erweitert, über die Daten ausgetauscht werden.

Zugriff von außen

Handelt es sich um eine interne Bibliothek oder ein Framework, die es zu betrachten gilt, sind vor allem die öffentlichen Schnittstellen von Interesse. Hier kann man mit aspektorientierten Sprachmitteln all die Methoden-Aufrufe ermitteln, die tatsächlich von außerhalb aufgerufen werden – denn oft sind nicht alle Methoden, die als „public" deklariert sind, für den Aufruf von außen vorgesehen *(siehe Listing 9)*.

Hier werden alle Methoden eines Bank-Frameworks („bank"-Package) überwacht. Nur wenn der Aufruf nicht von innerhalb kam („!cflowbelow"), wird vor der Ausführung der Methode oder des Konstruktors der Aufrufer anhand des Stack-Trace ermittelt (Methode „getCaller()", hier nicht aufgelistet, *siehe Listing 10)*. Dies ist die Ausgabe, die den Aufruf aus einer JSP zeigt. Lässt man die Anwendung lang genug laufen, erhält man so alle Methoden, die von außerhalb aufgerufen werden.

Daten aufnehmen

Über Serialisierung in Java ist es relativ einfach möglich, Objekte abzuspeichern und wieder einzulesen. Damit lässt sich ein

```
after(double neu) : setKontostand() && args(neu) {
   log.info("neuer Kontostand: " + neu);
}
```

Listing 6: After-Advice

```
java.net.SocketException: java.net.ConnectException: Connection refused
  at com.mysql.jdbc.StandardSocketFactory.connect(StandardSocketFactory.java:156)
  at com.mysql.jdbc.MysqlIO.<init>(MysqlIO.java:283)
  at com.mysql.jdbc.Connection.createNewIO(Connection.java:2541)
  at com.mysql.jdbc.Connection.<init>(Connection.java:1474)
  at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:264)
  at java.sql.DriverManager.getConnection(DriverManager.java:525)
  at java.sql.DriverManager.getConnection(DriverManager.java:193)
  at bank.Archiv.init(Archiv.java:25)
  …
```

Listing 7: Exceptions

```
after() returning(Object ret) :
     call(* java.sql.Connection.*(..)) {
  log.debug(getAsString(thisJoinPoint) + " = " + ret);
}
```

Listing 8: Pointcut und Advice für einen SQL-Aufruf

```
public pointcut executePublic() :
  (execution(public * bank..*.*(..))
     || execution(public bank..*.new(..)))
  && !within(EnvironmentAspect);

public pointcut executeFramework() :
  execution(* bank..*.*(..)) || execution(bank..*.new(..));

public pointcut calledFromOutside() :
  executePublic() && !cflowbelow(executeFramework());

before() : calledFromOutside() {
  Signature sig = thisJoinPoint.getSignature();
  String caller = getCaller(Thread.currentThread()
            .getStackTrace(), sig);
  log.info(caller + " calls " + sig);
}
```

Listing 9: Pointcuts und Advice für Aufrufe von außerhalb

```
...
jsp.index_jsp._jspService(index_jsp.java:54) calls bank.Konto(int)
jsp.index_jsp._jspService(index_jsp.java:58) calls void bank.Konto.einzahlen(double)
jsp.index_jsp._jspService(index_jsp.java:60) calls double bank.Konto.abfragen()
...
```

Listing 10: Stack-Trace

einfacher Objekt-Recorder bauen, um damit die Schnittstellen-Daten aufzunehmen *(siehe Listing 11)*. Hinter „thisJoinPoint" verbirgt sich der Kontext der Aufrufstelle, die der AspectJ-Compiler als Objekt bereitstellt.

Daten einspielen

Wenn man die notwendigen Schnittstellen-Daten gesammelt hat, kann man anhand dieser Daten einen (einfachen) Simulator bauen. Man kennt die Punkte, über die diese Daten hereingekommen sind, man muss jetzt lediglich an diesen Punkten die aufgezeichneten Daten wieder einspielen *(siehe Listing 12)*.

Hat man so die Anwendung von der Außenwelt isoliert und durch Daten aus früheren Läufen simuliert, kann man das dynamische Verhalten der Anwendung genauer untersuchen. Damit lassen sich weitere Aspekte hinzufügen, die automatisch Sequenz-Diagramme erzeugen oder wichtige Programmzweige visualisieren (etwa mithilfe der UMLGraph-Bibliothek *[4]* oder über die Webseite von websequence diagrams *[5]*). So erhält man neben der statischen Sicht durch Klassen-Diagramme, die sich notfalls mittels Reverse-Engineering wiedergewinnen lassen,auch eine Visualisierung des dynamischen Verhaltens.

Code-Änderungen

Nach diesen Vorbereitungen kann mit den ersten Code-Änderungen (Refactorings *[6]*) begonnen werden. Soll der Original-Code (noch) unverändert bleiben, was bei unbekannten Abdeckungen vorhandener Testfälle sinnvoll sein kann, liefert die Aspektorientierung die Möglichkeit, den Code getrennt vom Original abzulegen. Man kann sogar verschiedene Varianten einer Komponente vorhalten und diese während der Laufzeit austauschen. Auf diese Weise kann man experimentell verschiedene Varianten und Code-Manipulationen ausprobieren, um deren Verhalten auf die Gesamtanwendung studieren zu können.

```
after() returning(Object ret) : sqlCall() {
  objLogger.log(thisJoinPoint);
  objLogger.log(ret);
}
```

Listing 11: Aufnahmepunkte für den Daten-Recorder

Fazit

„Don't patch bad code – rewrite it" (aus: „The Elements of Programming" *[7]*). Der Aufwand, sich in unbekannten Code einzuarbeiten, wird oft unterschätzt. Langfristig ist es meist wirtschaftlicher, vorhandenen Code komplett neu zu entwickeln, wenn die Dokumentation veraltet, der Code an vielen Stellen ausgewuchert ist und auch die Testfälle nicht vorhanden oder von zweifelhafter Qualität sind. Oftmals hat man allerdings keine andere Wahl, als auf den bestehenden Code aufzusetzen, weil er die einzig gültige Quelle der Dokumentation darstellt. Und hier bietet die Aspektorientierung eine Vielzahl von Möglichkeiten:

- Zusätzliche Log-Möglichkeiten einbauen
- Sequenz-Diagramme generieren *[8]*
- Schnittstellen überwachen
- Aufrufe und Events aufzeichnen und simulieren
- Exceptions provozieren

Daraus lassen sich weitere Testfälle schreiben und neue Erkenntnisse gewinnen, um Refactoring-Maßnahmen einzuleiten oder sich zu einer Neuentwicklung zu entschließen. Allerdings entbindet auch AOP nicht von der Aufgabe, bestehenden und neuen Code so zu gestalten und umzuformen, dass künftige Generationen damit glücklich werden.

```
Object around() : sqlCall() {
  Object logged = logAnalyzer.
getReturnValue(thisJoinPoint);
  return logged;
}
```

Listing 12: Aufnahme einspielen

Weitere Informationen

[1] http://haupz.blogspot.de/2009/01/ich-bin-reich.html
[2] Edsger Wybe Dijkstra, A Discipline of Programming, Prentice Hall (Juni 1976), ISBN 978-0132158718
[3] Oliver Böhm, Aspektorientierte Programmierung mit AspectJ 5, dpunkt.verlag, 1st edition, 2005. ISBN-10 3-89864-330-15
[4] UMLGraph, http://umlgraph.org
[5] websequence diagrams, https://www.websequencediagrams.com/
[6] Martin Fowler. Refactoring. Addison-Wesley, 7th edition, 2001, ISBN 0-201-48567-2
[7] Kernighan and Plauger, The Elements of Programming, McGraw-Hill, 2nd edition, 1974, 1978. ISBN 0-07-034207-5
[8] Generation of Sequence Diagrams, https://sourceforge.net/p/patterntesting/wiki/Generation%20of%20Sequence%20Diagrams

Oliver Böhm
ob@jugs.org

Oliver Böhm beschäftigt sich mit Java-Entwicklung unter Linux und Aspekt-Orientierter SW-Entwicklung. Neben seiner hauptberuflichen Tätigkeit als JEE-Architekt bei T-Systems ist er Buchautor, Projektleiter bei PatternTesting und Board-Mitglied der Java User Group Stuttgart.

Code-Review mit Gerrit, Git und Jenkins in der Praxis

Andreas Günzel, EXXETA AG

Die Sicherstellung einer konstant hohen Software-Qualität ist ein kritischer Erfolgsfaktor für jedes IT Projekt. Automatisierte und manuelle Code-Reviews stellen eine Möglichkeit dar, dieses Ziel zu erreichen. Dieser Artikel erläutert ein mögliches Vorgehensmodell und zeigt den praktischen Einsatz anhand der Open-Source-Tools Gerrit Code-Review, Git und Jenkins.

Build-Server wie Hudson, Jenkins, Bamboo etc. haben sich in den letzten Jahren am Markt etabliert. Kaum ein Software-Entwicklungsteam möchte noch auf die Vorteile eines automatisierten Builds oder Deployments verzichten. Der Einsatz von Tools wie FindBugs oder PMD zur statischen Code-Analyse hat ebenfalls deutlich zugenommen. Ein grüner – also erfolgreicher – Build kann jedoch teilweise über den tatsächlichen Zustand der Software hinwegtäuschen. Denn oftmals werden die für die Qualität entscheidenden Faktoren nicht berücksichtigt: Erfüllt die Anwendung die Performance-Anforderungen? Sind die Coding-Guidelines eingehalten? Entspricht der Code der vorgegebenen Architektur?

Diese nicht funktionalen Anforderungen sind ebenso wichtig wie die funktionalen. Softwaretester legen ihr Augenmerk jedoch meist eher auf letztere. Damit ist es umso mehr die Aufgabe des Entwicklungsteams, die Qualität der Software konstant hoch zu halten.

Wann done „done" ist

Teams, die nach Scrum arbeiten, sollten sich zu Beginn der Zusammenarbeit auf die sogenannte „Definition of Done" verständigen. Dies ist die Einigung eines agilen Teams darauf, was getan werden muss, damit ein Feature als „fertig" angesehen wird. Ein wichtiger Punkt darin ist die Einhaltung der Architektur- und Coding-Vorgaben. Dabei stellt sich die Umsetzung in der Praxis oft als schwierig heraus.

Ein verbreitetes Mittel, um gegen dieses Problem anzugehen, ist das „Pair Programming". Dabei arbeiten zwei Entwickler zusammen an einem neuen Feature. Zu den positiven Effekten dieser Art von Teamarbeit zählen unter anderem eine höhere Software-Qualität und die Tatsache, dass mindestens zwei Personen das Stück Code kennen. Mit Fertigstellung des Features wird der Code in das gemeinsame Source-Code-Repository des Teams übernommen.

Der Nachteil dieses Vorgehens: Es erfolgt keine weiterführende, technische Validierung. Kann der Source-Code nicht mehr kompiliert werden oder schlagen Tests fehl, führt dies zum Abbruch des Build-Jobs auf dem Build-Server. Da die Änderung bereits in den aktuellen Entwicklungs-Branch übernommen wurde, ist das gesamte Team (einschließlich QS) betroffen.

Einen weiteren Nachteil des Pair Programming stellen die rein rechnerisch hohen Kosten dar, da stets zwei Entwickler zusammenarbeiten. Zudem muss bei der Team-Zusammenstellung darauf geachtet werden, dass sich die fachliche Kompetenz der Partner auf einem ähnlichen Niveau bewegt. Andernfalls besteht das Risiko, dass einem der Partner die Konzentration verloren geht und die gewünschte Kontrollfunktion nicht greift.

Eine Alternative zum Pair Programming sind sogenannte „Code-Reviews". Dabei arbeitet ein Team-Mitglied alleine an der Entwicklung eines neuen Features. Wenn die Implementierung aus seiner Sicht abgeschlossen ist, zeigt und bespricht er seinen Code mit einem oder mehreren anderen Entwicklern. Diese begutachten die Ergebnisse hinsichtlich möglicher Fehler, Vereinfachungen und der Definition of Done. Entspricht die Implementierung allen Vorgaben, wird der Code freigegeben und in das gemeinsame Source-Code-Repository übernommen.

Im Vergleich zum Pair Programming adressiert dieses Vorgehen damit vor allem den Nachteil der hohen Kosten. Vor allem unerfahrene Entwickler erhalten zudem schnell umsetzbares, qualifiziertes Feedback zu ihrer Arbeit. Daraus ergibt sich jedoch auch, dass der Code-Review in der Regel durch erfahrene Entwickler durchgeführt werden sollte. Dies kann schnell zu einem Flaschenhals werden, wenn bereits umgesetzte Tasks zu lange auf den Review warten.

Dieses Problem wird zudem verstärkt, wenn nicht erkennbar ist, welcher Task noch in der Entwicklung ist und welcher auf einen Review wartet. Arbeitet ein Team mit einem Task Board, kann man dieses Problem schnell lösen, indem eine neue Spalte „Review" eingeführt wird *(siehe Abbildung 1)*.

Automatisierte Code-Reviews mit Gerrit

Das im Rahmen des Pair Programming diskutierte Problem der technischen Validierung löst ein Code-Review in der beschriebenen Form ebenfalls nicht. Eine Möglichkeit wäre, diese Überprüfung als Aufgabe des Reviews zu definieren. Dies belastet die knappen Ressourcen der Reviewer jedoch noch weiter. Ein besserer Ansatz ist die Kombination des manuellen Code-Reviews mit einer technischen Bewertung des Source-Codes. Dies ist möglich, indem man jedes neue Feature auf einem eigenen Branch entwickelt. Beim Check-in wird der betroffene Branch dann automatisch auf einem Build-Server ausgeführt. Der Umfang des Build-Jobs richtet sich dabei nach vom Team definierten Kriterien. Das Kompilieren des Source-Codes und das Ausführen der Unit-Tests sind jedoch als Pflicht anzusehen.

Existieren Coding-Guidelines, die zum Beispiel mithilfe von SonarQube validiert werden können, sollte man den Job um eine entsprechende Überprüfung erweitern. Dies gilt ebenso für automatisierte System- oder Oberflächentests. Schlägt einer der definier-

ten Build-Schritte fehl, muss der gesamte Build fehlschlagen. Der Task ist in diesem Fall durch den technischen Review gefallen.

Für die praktische Umsetzung des beschriebenen Review-Prozesses gibt es verschiedene Software-Lösungen am Markt. Der Artikel stellt die Implementierung anhand des Open-Source-Tools „Gerrit Code Review" vor. Es entstand als Werkzeug zur Durchführung eines Peer-Reviews von Source-Code-Anpassungen vor der Übernahme in ein zentrales Repository, die erste Version im Rahmen des Android Open Source Project. Die aktuelle Version 2 ist in Java implementiert und nutzt Git für die Versionsverwaltung.

Die Idee hinter Gerrit ist einfach: Jeder Commit wird als eigenständiger Change angesehen und initiiert den Review-Prozess. Alle Änderungen sind zunächst einem Review zu unterziehen und freizugeben, bevor sie in die Code-Basis des Teams übernommen werden. Hierfür steht eine Web-Oberfläche bereit. Sie ermöglicht ebenfalls das Betrachten der Code-Änderungen oder das Verfassen von Kommentaren.

Gerrit agiert als eine Art „Proxy" vor dem eigentlichen Git-Repository. Das Hochladen von Änderungen in das zentrale Repository (in Git „push" genannt) löst somit automatisch den Review-Workflow aus. Zudem können Build-Server über eine Änderung informiert werden, um einen Build anzustoßen. Das Ergebnis des Builds kann dann ebenfalls von Gerrit verarbeitet und bei der Change-Bewertung berücksichtigt werden. *Abbildung 2* zeigt die Stellung von Gerrit als zentrales Repository.

Voraussetzung für die Nutzung von Gerrit ist das grundsätzliche Verständnis der wenigen, aber wichtigen Konzepte:

- Jede Änderung am Code wird als eigenständiger Change angesehen und muss den Review-Prozess separat durchlaufen. Gerrit erstellt einen Change, sobald ein neuer Commit in das zentrale Repository „gepusht" wird. Enthält ein „push" mehrere Commits, werden diese als mehrere aufeinander aufbauende Changes betrachtet.

- Ein Change lässt sich vor seiner Freigabe beliebig oft verändern. Dabei ist jedoch stets die Zuordnung zum ursprünglichen Commit zu wahren. Um dies zu erreichen, wird für die Anpassung kein neuer Commit erstellt, sondern der bestehende über die „amend"-Operation von Git verändert.

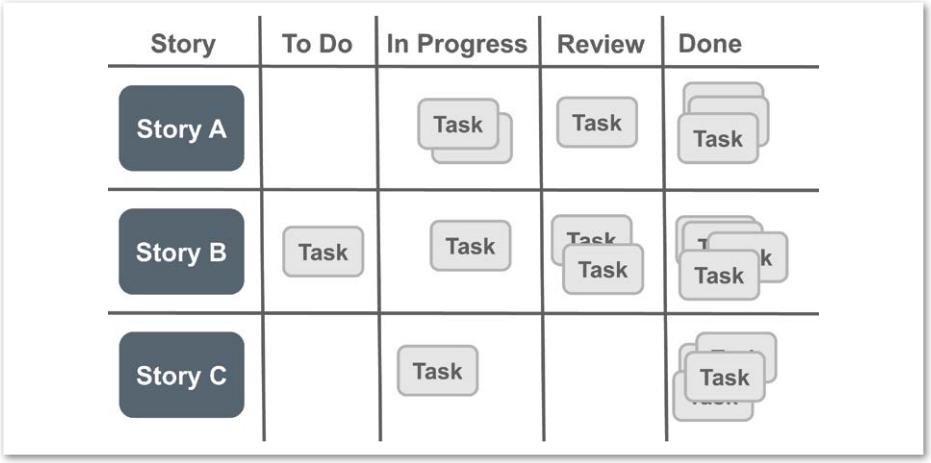


Abbildung 1: Task Board mit Review-Spalte

- Um mehrfache Änderungen innerhalb eines Change vergleichen zu können, erstellt Gerrit bei jedem „push" in das Repository eine neue Version. Diese Versionen werden „Patch-Sets" genannt. Wird ein Change freigegeben, wird nur das letzte Patch-Set in das zentrale Repository übernommen.

- Jeder Change wird durch einen oder mehrere Reviewer bewertet. Diese unterscheiden nach zwei Kategorien. Das Label „Verified" bedeutet, dass der Code kompiliert und Unit Tests bestanden hat. Mit dem Setzen des Labels „Code Review" bestätigt ein Reviewer, dass er den Code gelesen hat und diesen als plausibel und angemessen ansieht.

Insbesondere der letztgenannte Punkt der Change-Bewertung lässt sich an die Bedürfnisse des aktuellen Projekts und Teams anpassen. Die beiden Labels inklusive der Bewertungsbereiche werden initial bei der Installation von Gerrit vorkonfiguriert. In der Praxis hat sich dieser Default gut bewährt: So bietet es sich an, die Kategorie „Verified" durch einen Build-Server bestimmen zu lassen. Das Label „Code Review" und die finale Freigabe (in Gerrit „Submit" genannt) sollte dann durch ein Teammitglied erfolgen.

Die Beurteilung erfolgt nach einem Punktesystem. Der Default für die Bewertung des Labels „Verified" liegt im Bereich zwischen „-1" und „1", für „Code Review"

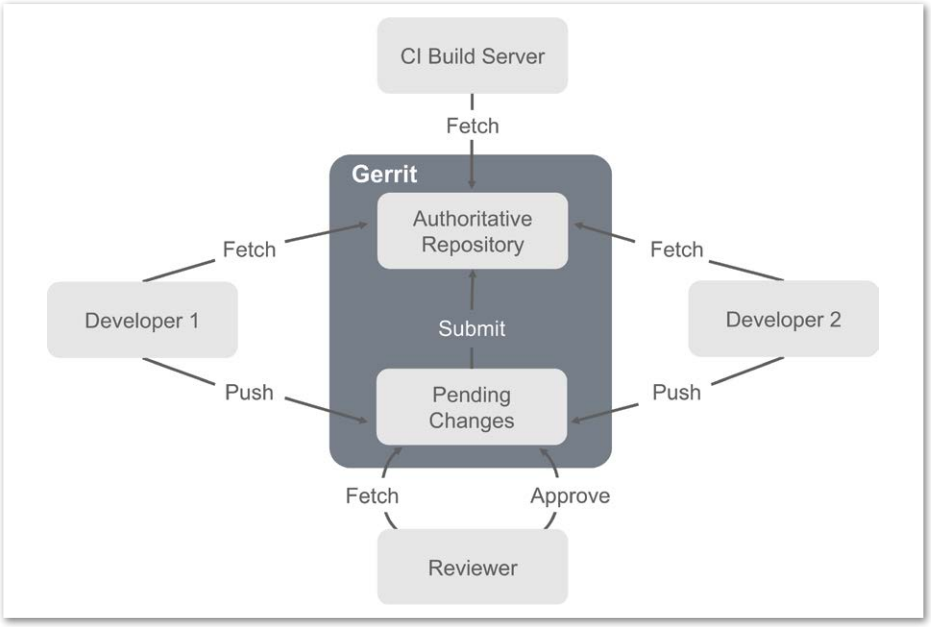


Abbildung 2: Gerrit als zentrales Repository

von „-2" bis „+2". Die einzelnen Werte haben in Gerrit folgende angedachte Bedeutung:

- *+2*
  Änderungen sehen gut aus und können freigegeben werden
- *+1*
  Änderungen sehen zwar gut aus, es sollten jedoch noch Kleinigkeiten verbessert werden
- *0*
  Änderungen wurden noch nicht bewertet
- *-1*
  Änderungen sehen fehlerhaft oder schlecht umgesetzt aus, eine Freigabe sollte erst nach deren Berichtigung erfolgen
- *-2*
  Änderungen sind definitiv falsch, eine Freigabe darf in dieser Form auf keinen Fall erfolgen

Alle Werte zwischen „-1" und „1" werden durch den „push" eines neuen Patch-Sets zurückgesetzt und sind daher als Empfehlung anzusehen. Im Gegensatz dazu stellen die Werte „-2" und „+2" eine Ablehnung beziehungsweise Genehmigung dar. Nur wenn ein Reviewer den Change mit „+2" bewertet, erlaubt Gerrit eine Freigabe. Auch das Punktesystem kann bei Bedarf je nach Projekt angepasst werden.

Gerrit Setup

Die aktuelle Gerrit-Version kann unter *[1]* heruntergeladen werden. Die Anwendung wird als Web Application Archive („.war"-Datei) ausgeliefert und kann somit auf einem beliebigen Webserver betrieben werden. Wer nicht gezwungen ist, einen vorhandenen Webserver zu nutzen (oder sich das Aufsetzen sparen möchte), kann Gerrit ebenso als Daemon betreiben – in diesem Fall auf einem Embedded Jetty Server. Als weitere Voraussetzungen werden lediglich ein installiertes JDK (die Gerrit Version 2.11 verlangt JDK Version 1.7 oder höher) sowie eine relationale Datenbank benötigt.

Gerrit unterstützt mit H2, Oracle, MySQL oder PostgreSQL zwar gängige, jedoch nicht alle Datenbank-Systeme. Alternativ kann auch hier per Default eine Embedded Datenbank (H2) verwendet werden. Für die Authentifikation der Benutzer unterstützt Gerrit ebenfalls eine Reihe verschiedener Möglichkeiten. Diese reichen von einer HTTP Basic Authentication über OpenID bis hin zu einer Anbindung an einen vorhandenen LDAP-Server. Für die Installation wird an dieser Stelle auf die sehr ausführliche Dokumentation unter *[2]* verwiesen.

Nach Abschluss des Setups ist Gerrit bereit für den Einsatz. Der erste Schritt ist der Aufruf der Web-Oberfläche. Die Startseite listet, sofern vorhanden, alle offenen Changes. Bevor nun die ersten Changes „gepusht" werden können, ist zunächst ein Projekt zu erstellen. Alternativ kann auch die Gerrit-REST-Schnittstelle verwendet werden. Zu jedem Projekt legt Gerrit ein gleichnamiges Git-Repository an. Für eine detailliertere Betrachtung sei an dieser Stelle auf die Gerrit-Dokumentation unter *[3]* verwiesen.

Ein erster Test

Um die Handhabung von Gerrit zu demonstrieren, legen wir ein Projekt mit dem Namen „gerrit-demo" an. *Listing 1* zeigt den Befehl zum lokalen Check-Out des Repository. In *Listing 2* erzeugen wir nun eine neue Datei und „pushen" diese Änderung in das zentrale Repository.

Jeder neue Change sollte auch lokal auf einem separaten Branch (in diesem Fall mit dem Name „first_change“) erstellt werden.

Die Befehle „git add“ und „git commit“ erzeugen im lokalen Repository einen neuen Commit mit der leeren Datei „empty_file.txt“. Für Entwickler, die bereits mit Git gearbeitet haben, ist erst der „push“-Befehl interessant, denn entgegen der Erwartung „pushen“ wir die Änderungen nicht gegen den Master („refs/heads/master“). Die Arbeit mit Gerrit erfordert hier einen „push“ gegen „refs/for/<Ziel-Branch>“. Gerrit nutzt „refs/for/*“ als eine Art „magische Referenz“. Im Hintergrund werden ein neuer Change erstellt und der Review-Prozess initiiert.

Die Änderung am Source-Code wird auf dem zentralen Repository in einem neuen, temporären Branch angelegt. Damit haben wir bereits unseren ersten Change erzeugt. Dieser wird nun in der Web-Oberfläche dargestellt und kann dort von einem Team-Mitglied dem Review unterzogen werden.

Jenkins-Integration

Als Grundlage für die Kommunikation zwischen Gerrit und Jenkins dienen sogenannte „Stream-Events“, die mit Gerrit ab Version 2.7 eingeführt wurden. Über eine SSH-Verbindung können sich Clients über verschiedene Ereignisse auf dem Gerrit-Server in Echtzeit informieren lassen. Zu den typischen Ergebnissen zählen das Erstellen, das Update oder die Freigabe eines Change.

Um die Events empfangen zu dürfen, benötigt der Jenkins-Server die entsprechenden Berechtigungen. Hierzu muss über die Gerrit-Web-Oberfläche zunächst ein neues Benutzerprofil für Jenkins samt SSH-Key angelegt und der vordefinierten Benutzergruppe „Non-Interactive Users“ hinzugefügt werden. Anschließend ist dieser Gruppe die Berechtigung zum Lesen, Setzen des Labels „Verified“ und Empfang von Stream Events für das gewünschte Projekt zu erteilen.

Damit ist die Konfiguration auf Seiten des Gerrit-Servers bereits abgeschlossen. Das Setup des Jenkins ist etwas aufwändiger: Zunächst muss das Gerrit-Trigger-Plug-in installiert und im Anschluss ein passender Build-Job aufgesetzt werden. Die angesprochene Erweiterung kann über den Jenkins-Plug-in-Manager installiert oder alternativ unter *[4]* heruntergeladen werden.

Nach Abschluss der Installation ist dem Plug-in zunächst mitzuteilen, wie der Gerrit-Server zu erreichen ist. In der Jenkins-Web-Oberfläche müssen dazu unter „Konfiguration“ auf der Seite „Gerrit Trigger“ URL und Port des Gerrit Servers sowie ein Benutzer und ein SSH-Key hinterlegt werden. Es ist derselbe Benutzer, der zuvor in Gerrit angelegt wurde. Weiter unten im selben Dialog können die durch Jenkins vergebenen Punkte für die Bewertung der Changes konfiguriert werden. Hier ist darauf zu achten, dass die hinterlegten Werte der Konfiguration in Gerrit entsprechen.

Nun ist noch ein Build-Job anzulegen, der jeden neuen Change validiert. Da meistens bereits ein Job zum Bauen des aktuellen Entwicklungs-Branch existiert, bietet es sich an, diesen als Vorlage zu verwenden und entsprechend den weiter oben beschriebenen Build-Schritten anzupassen. Build-Auslöser ist „Gerrit event“. Hier wird eine Reihe von Optionen angeboten, die zum Großteil ohne Anpassungen übernommen werden können. Lediglich unter dem Punkt „Gerrit Projekt“ muss der zu bauende Branch eingetragen werden. Damit ist das Setup des Jenkins ebenfalls abgeschlossen. Jedes Patch-Set, das nun in das Gerrit Repository „gepusht“ wird, führt zu einem automatischen Build.

```
$ git clone ssh://andreas@gerrit-server:29418/gerrit-demo
```

Listing 1

```
$ git checkout -b first_change
$ touch empty_file.txt
$ git add .
$ git commit -m "My first commit"
$ git push origin HEAD:refs/for/master
```

Listing 2

Fazit

Ein Code-Review kann die Software-Qualität einer Anwendung merklich steigern. Durch die tägliche Bewertung des Source-Codes wird ein kontinuierlicher Verbesserungsprozess erreicht, der am Ende die Time-to-Market und die Anzahl der auftretenden Fehler spürbar reduziert. Obwohl die Einführung eines Code-Reviews im Rahmen dieses Artikels am Beispiel des Vorgehensmodell Scrum erläutert wurde, eignet sich dieser Ansatz ebenso für klassische Projektmanagement-Methoden.

Tägliche Code-Reviews erfordern einen Zeitaufwand, der oft höher ist, als zunächst angenommen. Dies zahlt sich in der Regel jedoch aus, da einzelne Änderungen einfacher zu beurteilen sind als der Source-Code im Ganzen. Zudem kann vor allem die technische Validierung eines Change automatisiert werden.

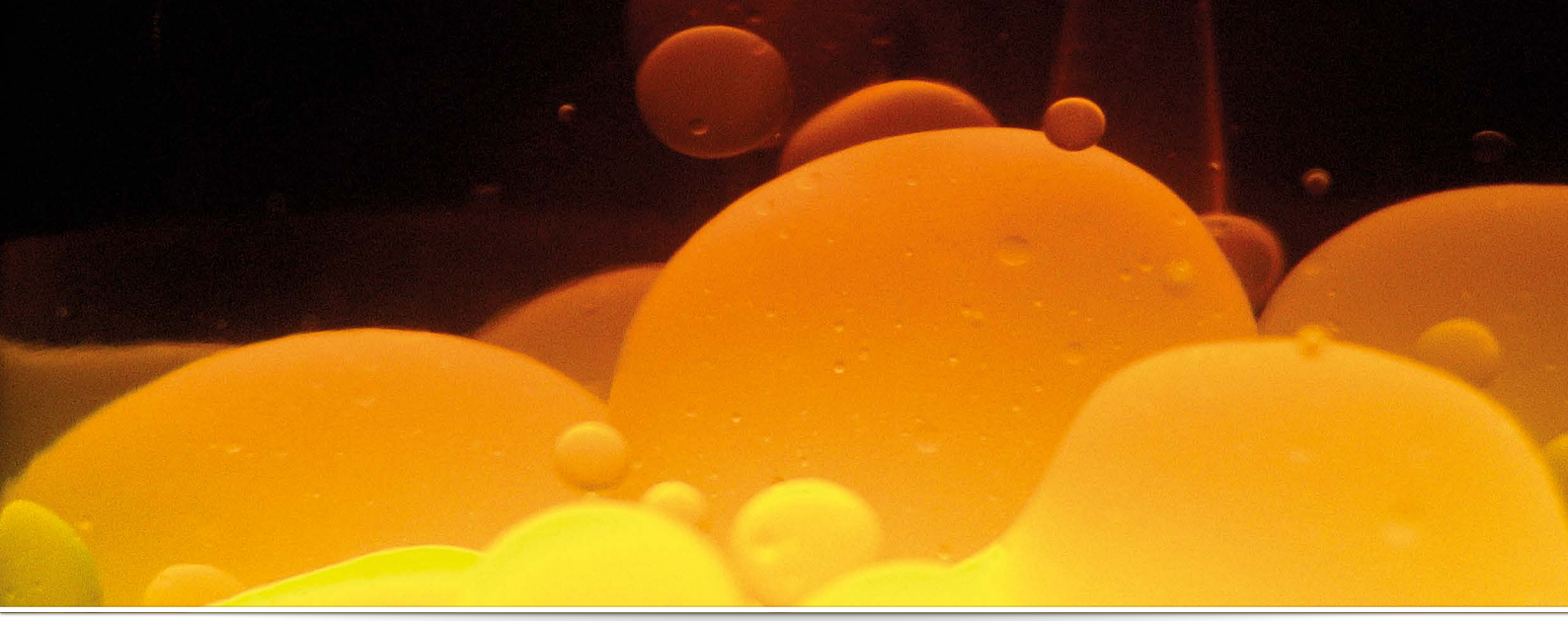
Die Verwendung von Gerrit zur Unterstützung des Review-Prozesses ist für alle Interessierten einen Blick wert. Das Setup des Gerrit-Servers, auch im Zusammenspiel mit Jenkins, ist vergleichsweise einfach und sehr gut dokumentiert. Zudem können Teams, die bereits mit Git arbeiten, ein vorhandenes Repository unkompliziert migrieren. Andere Tools zur Versionsverwaltung werden jedoch nicht unterstützt. Auch sollte nicht verschwiegen werden, dass die von Gerrit stark genutzten Git-Features rund um „commit –amend“ und die Reference Namespaces für viele Entwickler noch Neuland sind.

Weitere Informationen

[1] http://www.gerritcodereview.com
[2] http://gerrit-documentation.storage.googleapis.com/Documentation/2.11.1/install.html
[3] http://gerrit-documentation.storage.googleapis.com/Documentation/2.11.1/access-control.html
[4] http://wiki.jenkins-ci.org/display/JENKINS/Gerrit+Trigger

Andreas Günzel
andreas.guenzel@exxeta.com

Andreas Günzel ist Principal Consultant bei der EXXETA AG in Frankfurt. Seine langjährigen Erfahrungen stammen aus verschiedenen Projekten bei Banken und Mittelstand. Seine fachlichen und technologischen Schwerpunkte liegen im Umfeld von Java Enterprise und agiler Software-Entwicklung.



Kreative Signalgeber für Entwickler

Nicolas Byl, Netpioneer GmbH

Extreme Feedback Devices (XFDs) sorgen dafür, dass wichtige Mitteilungen, etwa zum Projekt-Status oder zu einem Fehler bei der Software-Entwicklung, unmittelbar wahrgenommen werden und nicht in der täglichen E-Mail-Flut untergehen. Der Artikel erläutert, welche unterschiedlichen Formen ein XFD annehmen kann, was Unternehmen bei einer Einführung beachten sollten, welche Herausforderungen sich damit lösen lassen – und welche nicht.

Insbesondere bei größeren Projekten hat sich in der Software-Entwicklung das Prinzip der kontinuierlichen Integration (Continuous Integration) etabliert. Hierbei wird fortlaufend Code aus mehreren Quellen in ein zentrales System eingecheckt und regelmäßig auf Fehler geprüft. Wird ein Problem festgestellt, ist es entscheidend, das Team umgehend darüber zu informieren, damit der Fehler behoben und kein weiterer Code eingecheckt wird. Häufig erfolgen diese Benachrichtigungen noch per E-Mail – bei vielen Dutzend eingehenden Mails pro Tag ein denkbar ungeeigneter Kanal, wenn es um die Übermittlung dringender Nachrichten geht, die eine sofortige Reaktion erfordern. Ist ein Build gescheitert und wird weiter Code eingecheckt, weil keiner das Problem wahrgenommen hat, macht dies die Fehlersuche deutlich schwieriger und komplizierter.

Um diese Situation zu vermeiden, entwickelte der IT-Spezialist Alberto Savoia 2004 die ersten sogenannten „Extreme Feedback Devices" (XFD): Lampen, die für jeden gut sichtbar im Raum aufgestellt wurden und je nach Zustand der getesteten Software in unterschiedlichen Farben leuchteten. So lässt sich sicherstellen, dass eine Fehlermeldung umgehend wahrgenommen wird und nicht in der üblichen E-Mail-Flut untergeht.

Auch wenn ein XFD, der ein optisches Signal übermittelt, recht naheliegend und in vielen Situationen am geeignetsten ist, muss es nicht zwingend eine Lava-Lampe im Stil von Savoia sein. Darüber hinaus können selbstverständlich auch andere Sinnesorgane angesprochen werden, wenn die räumliche Umgebung dies erlaubt, es dem Team nutzt und die Mitarbeiter es akzeptieren.

Für die Augen: Optische Signale

Von unterschiedlichsten Lampen in wechselnden Farben bis hin zur original Verkehrsampel sind Lichtquellen beliebte und häufig genutzte XFDs. Zum einen lassen sie sich relativ günstig herstellen, unkompliziert anbinden und einfach programmieren: Um loszulegen, benötigt man lediglich eine per Software schaltbare Steckdosenleiste sowie ein entsprechendes Leuchtmittel. Zum anderen signalisieren sie den Projektstatus unaufdringlich, aber dennoch für jeden sofort sichtbar. Bei verteilten Teams in unterschiedlichen Locations eignen sich hierfür ersatzweise auch leuchtende USB-Sticks. Dann müssen sich die Teammitglieder jedoch dazu verpflichten, den Stick beispielsweise auch im Homeoffice tatsächlich einzusetzen, damit er seinen Zweck erfüllen kann.

Manch einer mag sich mit profanen Lampen als XFD nicht zufriedengeben und entwickelt kreativere Lösungen. Das Spektrum reicht vom plötzlich an der Wand auftauchenden Batman-Schattenriss *(siehe „http://www.jensjaeger.com/2010/04/extreme-feedback-device-the-batman-lamp")* bis hin zum Zimmerspringbrunnen, der sich immer dann aktiviert, wenn Code eingecheckt wird *(siehe „https://schneide.wordpress.com/2007/03/04/extreme-feedback-device-das-code-flow-o-meter")*.

Weniger originell, dafür jedoch unauffällig in die Büro-Umgebung integrierbar und mit der Möglichkeit, etwas mehr an Informationen zu übermitteln, sind LED-Schriftbänder oder Wall-Displays. Bei Letzteren handelt es sich in der Regel um Monitore, die den Projektstatus anzeigen, gegebenenfalls mit einem farblich unterlegten Ampel-Schema.

Für die Ohren: Akustische Signale

Natürlich kann man das Ereignis „kaputter Build" lautstark per Martinshorn im Büro vermelden. Der Vorteil: Die Mitarbeiter nehmen die Message garantiert wahr. Der Nachteil: Vermutlich früher als später wird ein entnervter Mitarbeiter dieses XFD wieder entfernen. Etwas diskreter ist die Option, zusätzlich zu einem optischen Device dezente Sound-Effekte einzusetzen. Bei Teams, in deren Büros üblicherweise Musik beim Arbeiten läuft, lässt sich zudem mit der Unterbrechung der Playlist durch unpassende oder unbeliebte Musiktitel Aufmerksamkeit erzielen. Wie nachhaltig und wirkungsvoll diese Taktik jedoch ist, bleibt fraglich und hängt individuell vom Team ab.

Für die Nase: Olfaktorische Signale

Rein theoretisch ist auch ein Feedback über den Geruchssinn möglich. Immerhin haben Gerüche großen Einfluss auf das zentrale Nervensystem und können dazu beitragen, Stress zu reduzieren oder die Konzentration zu erhöhen. In der Praxis gestaltet sich dieser Weg jedoch eher problematisch. Zum einen geht es beim Extreme Feedback darum, eine wichtige Nachricht sehr schnell zu kommunizieren. Nutzt man aber dezente Duftsignale, etwa über elektrische Luftbefeuchter mit Aromazusätzen *(siehe https://schneide.wordpress.com/2009/09/08/smell-if-its-well)*, kann es durchaus eine Weile dauern, bis die Botschaft ihre Empfänger erreicht. Für einen schnelleren und eindeutigeren Effekt lassen sich natürlich auch Hilfsmittel finden – aber wer möchte schon in einem Büro arbeiten, in dem regelmäßig Stinkbomben gezündet werden?

Auf olfaktorischen Signalen beruhende XFDs sind eher nicht zu empfehlen. Abgesehen vom Risiko potenzieller Gesundheitsgefährdungen (Allergiker, Schwangere) ist die Wahrnehmung bei Gerüchen oft sehr unterschiedlich – was der eine als angenehm empfindet, ist für den nächsten unerträglicher Gestank. Und möglicherweise ist der Fehler im Build wesentlich schneller behoben, als es dauert, das Geruchssignal vollständig zu neutralisieren.

Für den Körper: Haptische Signale

Wem optische oder akustische Feedback Devices zu dezent sind, kann optional auch zu spürbareren Kommunikationsmitteln greifen. USB-Raketenwerfer in unterschiedlichsten Ausführungen und Größen erfreuen sich hier einiger Beliebtheit. Voraussetzung für den Einsatz dieser etwas brachialeren Kommunikationsmittel sind natürlich entsprechende Räumlichkeiten, die Teammitglieder müssen also zusammen in einem Raum und an fest vergebenen Schreibtischplätzen sitzen. Sonst kann es passieren, dass die Rakete einen völlig Unbeteiligten trifft.

Manche Teams bevorzugen ohnehin eine allgemeiner gehaltene Benachrichtigung, etwa durch Lichtsignale, gegenüber dem individuellen Hervorheben eines „Schuldigen" durch gezielten Beschuss. Wer also über den Einsatz eines solchen XFD nachdenkt, sollte dies auf jeden Fall mit allen Beteiligten absprechen.

Ist sich ein Team darüber einig, dass ein Extreme Feedback Device ein hilfreiches Tool wäre, gilt es, über die am besten geeignete Ausprägung zu entscheiden. Bei der Platzierung sollte speziell bei Lampen und Wall-Displays darauf geachtet werden, dass diese bei allen Lichtverhältnissen jederzeit gut sichtbar beziehungsweise lesbar sind. Hat man schließlich aus USB-Steckdosenleisten, Lampen und farbigen Glühbirnen beispielsweise eine XFD-Ampel gebastelt und erfolgreich implementiert, darf man einen Fehler jedoch nicht machen: Erwarten, dass mit dem neuen Signalgeber alle prozessualen Probleme des Entwickler-Teams gelöst sind.

XFDs sind nützlich, um alle Beteiligten umgehend über einen gebrochenen Build zu informieren und so dafür zu sorgen, dass niemand mehr weiteren Code eincheckt und dadurch die Fehlersuche möglicherweise erschwert. Nicht mehr, aber auch nicht weniger. Dass grundsätzlich die Code-Qualität stimmt, dass ein Team-Mitglied sofort die Verantwortung dafür übernimmt, einen gebrochenen Build zu reparieren, und dass prinzipiell die Motivation vorhanden ist, erfolgreiche Entwicklungsarbeit zu übernehmen – all dies ist und bleibt Aufgabe der beteiligten Mitarbeiter, nicht eines USB-Raketenwerfers oder einer Lava-Lampe.

Nicolas Byl
nicolas.byl@netpioneer.de

Neun Tipps zur Einführung von Extreme Feedback Devices

- Wählen Sie einen gut sichtbaren Ort, der ein einfaches Ablesen der Status-Anzeige gewährleistet. Platzieren Sie das XFD nicht direkt vor einem Fenster oder in einer unzugänglichen Ecke. Gut geeignet sind neben dem Büro des zuständigen Teams auch Plätze in häufig genutzten öffentlichen Räumen, etwa neben der Kaffeemaschine.
- Wenn keine farbigen Lampen genutzt werden, lässt sich der Systemstatus zusätzlich über die Hintergrundfarbe darstellen, beispielsweise bei Wall Displays. So können Mitarbeiter die Statusanzeige auch aus weiterer Entfernung auf einen Blick ablesen.
- Übermitteln Sie so wenig Information wie nötig; eine klare Statusanzeige ist ausreichend. Detailliertere Informationen lassen sich dann dem zugrunde liegenden System entnehmen.
- Versuchen Sie, Bewegungen und Statuswechsel weitestgehend zu minimieren, da jede Aktion Aufmerksamkeit abzieht. Nutzen Sie Ihr XFD daher möglichst nur für Status-Wechsel, um die Aufmerksamkeit zielgerichtet zu lenken.
- Auch wenn es noch so verlockend ist: Ein Extreme Feedback Device ist kein Spielzeug – selbst dann nicht, wenn es in Form eines USB-Raketenwerfers daherkommt. Wird ein XFD häufig grundlos aktiviert, verliert sich die Signalwirkung, wenn es um eine wirklich wichtige Rückmeldung geht.
- Ist ein XFD eingerichtet und funktionsfähig, sollten Sie zunächst testen, ob das Device dem Team hilft, bevor weiter daran gebastelt wird. Der XFD-Einsatz kann auch ein regelmäßiger Punkt im Rahmen der Scrum-Retrospektive sein.
- Devices sind kein Ersatz für Disziplin! Sie helfen lediglich, das Feedback zu beschleunigen, lösen aber keine Probleme. Die angezeigten Fehler zu beseitigen, ist und bleibt Aufgabe des Teams.
- Überlegen Sie schon vor der Implementierung des XFD gemeinsam im Team, wie Sie mit dem Fehlersignal umgehen wollen. Klären Sie, wer wofür die Verantwortung übernimmt. „Ich habe dafür keine Zeit!" ist keine Ausrede. Setzen Sie die Regeln durch, die Sie für den Fall eines Fehlersignals aufgestellt haben.
- Devices sind keine QA-Metrik. Wie oft die rote Lampe aufleuchtet, sagt noch nichts über die Qualität der Software aus. Aber: Wenn der Build sehr häufig kaputt ist, liegen meist grundsätzliche Probleme vor, die in der Regel in einer mangelhaften Zusammenarbeit begründet sind.

Java-Engines für die Labordaten-Konsolidierung

Matthias Faix, IPM Köln

Kliniken der medizinischen Maximalversorgung sowie medizinische Versorgungszentren ermitteln Labordaten in aller Regel dezentral. Die Proben werden an verschiedenen Stellen entnommen und schließlich in ein Labor gebracht. Die Daten müssen dann an die entsendenden Stationen zurückübermittelt werden. Diese Daten und deren Workflow sind aus verschiedenen Gründen sehr sensibel. Der Einsatz von Java im Frontend sowie ein Applikationserver bringen hier Vorteile.

Vor wichtigen Operationen muss der Arzt über den Zustand des Patienten Bescheid wissen. Das Gelingen der Operation und die Planung hängen von Labordaten ab. Ein Wert, der zum Beispiel bei einer Herzkatheter-Untersuchung betrachtet werden muss, ist die Gerinnung. Die Gerinnung wird in Quick – ein veralteter Wert – oder in INR angegeben. Der Wert wird aus einer Blutprobe des Patienten im Labor ermittelt. Er ist an sich nicht stabil, sondern ändert sich unter der Medikamentierung des Patienten, aber auch durch die Lebensumstände. Er hat eine zeitliche Dimension, man verfolgt zum Beispiel beim Absetzen einer Gerinnungshemmung den Verlauf der Gerinnung mit der Zeit.

Probleme bei der Datenübermittlung von Labordaten

Labordaten bestehen aus dem eigentlichen Wert und den Dimensionen. Zu den Dimensionen aus den SI-Einheiten und den abgeleiteten Einheiten (kg/Liter) gesellen sich Dimensionen wie Temperatur, Zeit, Messumstände. Es ist relativ schwierig, Labordaten miteinander zu vergleichen, die aus unterschiedlichen Laboren oder zu unterschiedlichen Zeiten genommen wurden. Medizinische Entscheidungen sind abhängig von den Werten oder von den Zeitreihen der Werte. Gerade die Gerinnung und deren Verlauf sind wichtig dafür, ob ein Patient operationsfähig ist oder nicht.

Die Vergleichbarkeit der Daten ist ein zentrales Problem der Labordatenverwaltung. Bei einigen Messungen empfiehlt es sich auch, die Messmethode respektive das Messkit anzugeben, um Vergleichbarkeit zu erreichen. Ein Wert zieht einen Kometenschweif an vernetzten Dimensionen hinter sich her. Dieser Kometenschweif ist nun alles andere als stabil, er ändert sich fortwährend. Hier zeigt sich ein weiteres Problemfeld bei der Verwaltung von Labordaten, das Ändern der Dimensionen. In der Informatik wird diese Figur auch als „Slowly Moving Dimensions" bezeichnet. Man sieht, Labordatenkonsolidierung führt den Entwickler schnell in die Sphären von OLAP. Die Laborwertkonsolidierung führt im Eigentlichen zum Aufbau eines Datawarehouse, mit der Option der Verwaltung der Daten für den aktuellen Patienten, aber auch weitergehend mit der Option des Daten-Digging für größere Patientengruppen.

Methoden der Datenbank-Verwaltung

Schnell wird hier klar, dass die Daten letztendlich auf eine mittlere Datenbankschicht konsolidiert werden müssen. Aktiv dient im Moment eine relationale Datenbank als eine Art Relais für die Konsolidierung der Labordaten. Die Dimensionen werden mit Zeitstempel in der Datenbankschicht gehalten. Wir haben hier eine klassische Drei-Tier-Schichtenbildung – Dimensionen mit Historisierung. Die Bindung der Laborwerte an die Dimensionen geschieht übrigens über ein Codesystem des DIMDI aus Köln – LOINC. Die Nachrichten zwischen den einzelnen Schichten wiederum werden in einem HL7/XML-Format übermittelt. HL7 2.5 ist eine Sprache und Struktur im Gesundheitswesen, XML stellt die äußere Kommunikationsstruktur dar.

Die relationale Struktur der Verwaltung hat einige Nachteile – speziell in der Organisation der Daten und in der Geschwindigkeit. Eine

Laborwerte

Analyt	Messwert	Referenzbereic	Einheit	Datum	
Kreatinin	0.4	0.2 - 0.8	mg/dl	08.02.2014	16:03
glom. Filtr. Rate					
Thrombozyten	169.0	200 - 460	G/l	08.02.2014	16:03
Quick	111.0		%	08.02.2014	16:28
PTT					
T3 (Trijodthyronin)					
T4 (Thyroxin)					
TSH					
Bilirubin gesamt					
AP					
CRP	13.9	3	mg/l	08.02.2014	16:03
Hb	15.4	10.7 - 13.9	g/dl	08.02.2014	16:03
INR					
AFP					
CA 15-3					
CA 19-9					
CEA					
NSE					
PSA					
Troponin I	0.2	0.05	ng/ml	08.02.2014	16:03
D-Dimere					

Abbildung 1: Das Labordaten-Tupel

Optimierung des Denormalisierungsprozesses kann durch den Einsatz von sogenannten NoSQL-Datenbanken erreicht werden.

NoSQL-Datenbanken

Relationale Datenbanken, wie auch Oracle- oder SQL-Server, haben ihre Stärke in der Konsistenz. Es gibt eben Anwendungen, bei denen zeitgenau gearbeitet werden muss. Bei Hotelbuchungen sind Locking-Mechanismen essenziell. Doppelbuchungen können eben nicht akzeptiert werden. Das Ändern des Datenbestandes ist Regeln unterworfen. Wer jemals mit Excel versucht hat, eine Datenverarbeitung aufzubauen, der weiß letztlich eine relationale Datenbank zu schätzen.

Es gibt allerdings Datenbanken, die arbeiten schneller und geben ultraschnelle Antworten. Projekte wie Evernote wären mit relationalen Mitteln nicht erfolgreich. Der Benutzer kann sein ganzes Leben hier digitalisieren und von überall auf die Daten zugreifen. Evernote zeigt allerdings auch einen Aspekt der Datenhaltung, die später noch bei der Labordatenverwaltung wichtig wird. In einer Übergangszeit sind die Recherche-Ergebnisse variabel, am Ende jedoch sind die Ergebnisse stabil. Dafür gibt es die schöne Formel „Basically Available, Soft state, Eventual consistency" (BASE). „Eventual consistency" bedeutet hier, die Datenbank befindet sich am langen Ende in einem konsistenten Zustand.

Wenn man in der Wasseranalytik oder in der Laboranalytik arbeitet, dann kommen beide Anforderungen vor – „Konsistenz" und „Immer verfügbar". Es gibt im Laborwesen drei Szenarien: Die aktuellen Labordaten müssen am Anfang schnell und immer verfügbar sein. Es muss nichts gelöscht werden, es werden immer neue Werte hinten angehängt. Die Konsistenz ist weniger wichtig als die schnelle Verfügbarkeit der Daten. Die Konsistenz der Dimensionen zu den Daten muss jedoch gewährleistet sein. Falsche Zuordnungen an dieser Stelle können zu erheblichen Fehleinschätzungen der behandelnden Ärzte führen und letztlich Menschenleben gefährden.

Der Weg zu vernünftigen Daten

Vernünftige Daten im Labor, speziell in Kliniken, sind Daten, auf deren Basis der Arzt den Patienten behandeln kann. Der Gang der Dinge ist der, dass dem Patienten auf der Station Proben entnommen werden. Diese Stoffproben gehen ins Labor und werden dort mit Labormethoden untersucht. Die

Abbildung 2: Armband mit E-INK zur Kopplung Fall-/Labordaten

Daten beginnen mit dem Beschriften der Laborröhrchen zu leben. Im Labor werden die Daten entnommen und mit den Dimensionen vernetzt. Ein typisches Messwertbild ist in *Abbildung 1* zu sehen. Die Daten zerfallen in Messwerte (Spalte 2) und Dimensionen. Der Analyt ist auch eine Dimension – zum Beispiel Kreatinin, Referenzbereich, Einheit.

Die Messwerte sowie die Kopplung an Episode und Patient werden in relationalen Formen – hier Oracle Datenbank – gespeichert. Zu den Messwerten kommt ein LOINC–Parameter hinzu. Dieser LOINC-Qualifier ist schlussendlich der Eingangspunkt in eine Dimensionen-Datenbank. Eine Mischung von Messwerten und Dimensionen ergibt das Datenblatt oben. Die relationalen Daten werden aber ebenfalls in eine NoSQL-Datenbank geschoben – so können zeitliche Ableitungen („Zeitreihen") schnell ermittelt werden. Bei kurzfristigen Operationen ist die Zeitreihe „Gerinnung" essenziell, um das Go für eine Operation zu bekommen.

Eine spannendes letztes Kopplungselement sind hier übrigens Versuche mit elektronischen Patientenarmbändern. Mit GS-Barcodes, die auf den Armbändern angebracht sind, kann man einen Einstieg in das Laborsystem bekommen und einen letzten „Bedside Test" machen – bevor eine falsche Blutkonserve appliziert wird.

Als NoSQL-Datenbanken werden derzeit MongoDB und Neo4J getestet. Der Charme dieser Datenbanken ist, dass man Daten komplett auf einem Kleinrechner halten und ohne zentrale Datenverarbeitung Behandlungen ansteuern könnte. Dies ist aber leider noch im Testbetrieb.

Literatur

[1] https://www.dimdi.de/static/de/klassi/loinc
[2] http://loinc.org
[3] http://neo4j.com
[4] http://de.wikipedia.org/wiki/NoSQL
[5] http://de.wikipedia.org/wiki/Apache_Hadoop#HBase

Matthias Faix
matthias.faix@koeln.de

Matthias Faix war nach dem Studium in Biologie mit Nebenfach Informatik ab dem Jahr 1990 bei textware mit Schwerpunkt auf Oracle Forms und Reports. Er ist seit dem Jahr 2003 bei today IPM Köln im Internet-Projekt-Management tätig.

Java-Champion Kirk Pepperdine

Performance durch Parallelität verbessern

Kirk Pepperdine, Java Champion und Performance-Expert

Die Einführung von Lambdas und Streams in Java SE 8 erlaubt es Entwicklern, sehr einfach ihre Applikationen zu parallelisieren. Aber wird das Parallelisieren von Teilen der Applikation wirklich die Performance verbessern? Um diese Frage zu beantworten, sehen wir uns an, wie der in Java SE7 eingeführte „ForkJoinPool" konfiguriert und genutzt wird. Wir werden uns auch damit beschäftigen, wie man dieses Framework instrumentiert, um herauszufinden, wann und wie es konfiguriert werden muss.

In der März-Ausgabe 2005 von Dr. Dobb's Journal prägte Herb Sutter den inzwischen berüchtigten Satz „The free lunch is over". Dieser wurde gefolgt von „Concurrency is the next major revolution in how we write software", wiederum gefolgt von viel Gerede um Amdahl, Moore's Law, Wärme-Abführung, wie Entwickler und Programmiersprachen sich ändern müssten und so weiter. Es gab auch viele Diskussionen um Read-Write-Barrier, „Lock free"- und „Wait free"-Algorithmen sowie um die Nutzung von Verfahren, die auf Devide and Conquer basieren. Eine der „Divide and Conquer"-Techniken, die sehr viel Zuspruch gefunden hat, ist das Fork/Join-Framework.

Das Fork/Join-Framework

Das Framework erlaubt das Zerlegen großer Tasks in eine Reihe kleinerer Sub-Tasks. Diese werden weiter zerlegt, bis man letztendlich atomare Arbeitseinheiten hat, die alle ausgeführt werden. Die Ergebnisse dieser atomaren Einheiten werden dann zusammengeführt („Join"), um das Gesamtergebnis zu erstellen.

In Java SE 7 wurde mit „ForkJoinPool", „ForkJoinTask" und „ForkJoinWorkerThread" eine Top-Level-Unterstützung für das Fork/Join-Framework eingeführt. Unglücklicherweise ist das Programmierkonzept, das im Fork/Join-Framework Verwendung findet, recht komplex und daher wurde diese Technik von Java-Entwicklern, die unter Zeitdruck eine Business-Anwendung nach der anderen implementieren, bisher weitgehend ignoriert.

Das hat sich mit Java SE 8 und der Einführung von Streams grundlegend geändert. Unter der Haube werden parallelisierte Streams in Fork/Join-Tasks umgewandelt. Das macht es für Entwickler einfach, Parallelität zu ihren Anwendungen hinzuzufügen, ohne die darunterliegenden Mechanismen verstehen zu müssen. So sehr Parallelität der Performance zuträglich sein mag, so kann das willkürliche und unbedachte Nutzen des Fork/Join-Frameworks der Performance auch schaden.

Streams

Ein Stream nimmt eine Collection und stellt jedes Element einem Prozess zur Verfügung (ähnlich wie ein Iterator). Streams selbst speichern keine Daten und können sowohl begrenzt als auch unbegrenzt sein. Zudem unterstützen Streams sowohl intermediäre („lazy") als auch abschließende („eager") Operationen. Intermediäre Operationen geben immer einen neuen Stream zurück, während abschließende immer ein Ergebnis liefern. Dies alles geschieht seriell; es sei denn, es wird nach einem parallelen Stream verlangt – vorausgesetzt, die zugrunde liegende Datenquelle unterstützt dies auch. *Listing 1* zeigt Beispiele für sequenzielle und parallele Streams. Anmerkung: Bei CPUs mit Hyperthreading gibt es zwei logische Kerne für jeden physikalischen Kern.

In *Listing 1* arbeitet „stream()" die Datenquelle, bei der es sich um einen Array von Garbage Collection Logs handelt, sequenziell ab. Der Aufruf von „parallelStream()" auf der Datenquelle führt dazu, dass die Anweisungen als Fork/Join-Operation ausgeführt werden. Standardmäßig verwendet das Fork/Join-Framework einen sogenannten „Common Thread Pool", dessen Default-Größe der Anzahl der logischen Prozessorkerne entspricht.

Antwortzeiten, Threads und Queues

Die Antwortzeit (oder Latenz) einer Anfrage setzt sich zusammen aus der Zeit, in der der Thread seine Arbeit verrichtet, und der Zeit, in der er angehalten ist („Dead Time"). Typischerweise macht die Dead Time einen signifikanten Teil der Antwortzeit aus und ein Großteil der Zeit wird in einer Queue verbracht. Demzufolge sollte man die Latenz signifikant verringern können, wenn man die Zeit reduziert, die der Thread in Queues verbringt.

Was wir im Fall von Fork/Join-Operationen wissen möchten, ist, wie viele Threads sich im Common Thread Pool befinden und wie viele Anfragen in der Queue darauf warten, bearbeitet zu werden. Diese Information kann uns zusammen mit der Auslastung anderer Ressourcen dabei helfen zu entscheiden, ob unser System von einem größeren oder kleineren Common Thread Pool profitieren würde.

Es ist einfacher, über Queues zu sprechen, wenn wir uns erstmal mit der Theorie dahinter beschäftigen, insbesondere mit dem Gesetz von Little. Dieses besagt: Die Anzahl der Aufgaben in einem System (L) ist gleich der durchschnittlichen Zeit, die das System benötigt, um die Aufgabe abzuarbeiten („lambda"), multipliziert mit der durchschnittlichen Rate, mit der neue Aufgaben in das System gelangen („N"). Kurz: „L = lambda * N".

Was das Gesetz von Little so wichtig macht, ist, dass es erlaubt, das ganze System als Black Box zu betrachten. Der Autor hat beispielsweise an einem System gearbeitet, bei dem die Anforderung 1.500.000 Transaktionen pro Stunde vorsah. Die durchschnittliche Latenz betrug 300 Millisekunden.

1.500.000 Transaktionen pro Stunde entsprechen 417 Transaktionen pro Sekunde. Könnte man in das System hineinsehen, würde man zu jedem beliebigen Zeitpunkt erwarten, „0,300 * 417 = 125 Anfragen" vorzufinden. Ist die Größe des Thread Pools auf „8" begrenzt, ergäbe sich daraus, dass acht Threads gerade ihre Arbeit verrichten, während 117 in der Queue warten. Vorausgesetzt, die Hardware unterstützt mehr aktive Threads, wäre es sehr wahrscheinlich, dass das System von einem größeren Thread Pool profitiert.

Überwachung von Fork/Join-Operationen

Um Statistiken für die oben beschriebene Analyse zu erhalten, müssen wir das System genauer überwachen. Nach sorgfältiger Analyse fand der Autor heraus, dass er nur „ForkJoinTask" überwachen muss. Das Monitoring sollte jedoch über „ForkJoinPool" stattfinden.

Definieren wir uns ein paar Test-Szenarien, die den Kontext für die nachfolgenden Betrachtungen bilden. Bei jClarity haben wir sehr viel Zeit damit verbracht, uns Garbage-Collector-Logs (GC) mit Censum anzusehen. Der Autor hat das zum Anlass genommen zu betrachten, wie man Lambdas verwenden könnte, um ein GC-Log zu parsen. Im folgenden Beispiel werden wir uns auf die Einträge „application stopped time" und „application concurrent" konzentrieren. *Listing 2* zeigt eine Regel zum Parsen von „application stopped time" aus einem GC-Log.

Der Code in *Listing 2* nimmt eine Liste von GC-Log-Einträgen, filtert sie nach solchen, die dem Muster entsprechen, und berechnet dann die Statistiken, die das „stoppedtime"-Verhalten beschreiben. Wir können diesen Code ausführen, indem wir ihn direkt dem Fork/Join-Pool übergeben, wie in *Listing 3* gezeigt.

Der Code in *Listing 3* erzeugt einen neuen „ForkJoinTask" und übergibt ihn an den „ForkJoinPool". Wenn der Task beendet ist, erfolgt ein Aufruf an „ForkJoinTask::setCompletion()", anschließend kann das Ergebnis abgerufen werden. Der Parameter für diesen Aufruf ist ein Statuscode, der dem Aufrufenden mitteilt, ob der Task normal beendet wurde oder ob Fehler aufgetreten sind. Jeder Aufruf von „get()", der erfolgt, bevor „setCompletion()" aufgerufen wurde, bewirkt, dass der aufrufende Thread blockiert.

Idealerweise wollen wir die Zeit vom ersten Aufruf von „submit()" bis zum Beenden des Jobs messen. Das lässt sich sehr leicht einrichten, wenn wir die Fork/Join-Operation explizit aufrufen. Jedoch schlägt das in Fall von *Listing 4* fehl, obwohl die Ausdrücke in dem Listing in einen ähnlichen Code wie in *Listing 3* transformiert werden.

Um dies zu umgehen, kann das Monitoring tiefer in den Call Stack eingebettet werden. Dadurch wird den Messwerten ein kleiner Fehler hinzugefügt, dieser ist jedoch weit unter den Schwankungen, die man in der Applikation ohnehin erwartet. Der Code in *Listing 5* stammt aus dem OpenJDK und wurde mit Aufrufen versehen, die ein Signal geben, wenn die Ausführung des Tasks beginnt und wenn sie beendet ist.

Definition der „ForkJoinPoolMonitorMXBean"

Jetzt, wenn wir ein Signal haben, mit dem wir arbeiten können, brauchen wir etwas, um es zu analysieren. Eine vollständige Beschreibung von „MXBeans" sprengt den Rahmen dieses Artikels, es handelt sich dabei um eine Technik, die der Autor in seinen Performance-Tuning-Workshops beschreibt und die in einem Online Tutorial im Oracle Technology Network behandelt wird. Legen wir einige Anforderungen fest und führen wir dann ein Interface für die MXBean ein.

Die „ForkJoinPoolMonitorMXBean" sollte Folgendes ausgeben: Die Raten der ein- und ausgehenden Tasks sowie die Gesamtanzahl der Tasks und die durchschnittliche Zeit, die ein Task im System verbracht hat. Wir erinnern uns, das Gesetz von Little besagt, dass die durchschnittliche Zeit eines Task im System, multipliziert mit der Eingangsrate, die Anzahl der zurzeit im System befindlichen Tasks ist. Mithilfe der durchschnittlichen Länge der Queue, kombiniert mit der Anzahl der Threads und einigen Zeitangaben, kann die Dead Time ausgerechnet werden. Dies könnte zur Anpassung der Größe des Thread Pools führen, vorausgesetzt, die Hardware unterstützt diese Rekonfiguration. *Listing 6* zeigt die Definition der „ForkJoinPoolMonitorMXBean".

Zusätzlich zu diesen Methoden implementiert „ForkJoinPoolMonitor" die Methoden „submitTask(ForkJoinTask)" und „retireTask(ForkJoinTask)". Die Implementation *(siehe Listing 7)* nutzt eine „ConcurrentHashMap", um alle Tasks zu speichern, die sich aktuell im System befinden. Der Code, um die „MXBean" in den „PlatformMBean"-Server einzubinden, steht auch bereit. Damit steht die „MXBean" den Java Management Extensions Clients (JMX) wie jConsole und dem VisualVM-JMX-Plug-in zur Verfügung.

Die Arbeitseinheit

Der Aufruf von „concurrentParallelStream()" in *Listing 8* resultiert in der wiederholten Ausführung von zwei Runnable Lambdas. Die „Submit()"-Methode nimmt das Runnable-Objekt und verpackt es in einen „ForkJoinTask". Innerhalb von „calculateParallel()" befindet sich der Aufruf „Files.lines(path).parallel()". Dies führt zu einem neuen „ForkJoinTask", der die Zeilen im GC-Log parallel abarbeitet.

Die beiden Tasks werden nacheinander und vor dem Aufruf von „get()" übergeben. Das heißt, sie sollten parallel abgearbeitet werden. Die darunter liegenden Lambda-Ausdrücke in den Aufrufen von „calculateParallel()" werden dies stören, da sie einen intrinsischen Aufruf von „get()" machen. Es wird durchaus eine Überschneidung bei der Ausführung der Aufrufe geben; wie die Zahlen zeigen, ist diese jedoch begrenzt.

Die Ausführung

Um alles zum Laufen zu bringen, müssen wir unsere modifizierten Versionen von „ForkJoinPool" und „ForkJoinTask" ausführen, die wir uns vom OpenJDK ausgeliehen haben. Dort finden sich die Original-Implementationen dieser Klassen in „rt.jar"; diese wird vom Bootstrap-Classloader geladen. Glücklicherweise müssen wir „rt.jar" nicht zerpflücken und mit unseren Versionen der Klassen wieder zusammensetzen. Die Java Virtual Machine (JVM) bietet die Möglichkeit, Option-Klassen mit dem Parameter „-Xbootclasspath/p:" dem Boot-Classpath voranzustellen *(siehe Listing 9)*. Hinweis: In Java SE 9 werden sich das Classloading und damit auch die hier gezeigte Methode wahrscheinlich ändern.

Diskussion der Ergebnisse

Die beiden Arbeitseinheiten in diesem Beispiel sind beide groß und komplex genug,

dass sie von einer Parallelisierung profitieren sollten. Die erste Arbeitseinheit liest die benötigten Daten aus dem Speicher, die zweite von der Festplatte. Beide werden seriell mit parallelen Streams und dann beide noch einmal mit Fork/Join-Operationen ausgeführt. Die Ergebnisse sind in den Tabellen 1 und 2 zu finden. *Tabelle 1* zeigt das Parsen der GC-Logs ohne die I/O-Operationen und *Tabelle 2* mit I/O-Operationen.

Bevor wir uns die Ergebnisse weiter ansehen, eine Warnung: Es ist genug Rauschen in den Ergebnissen, um keinen Wert auf die kleinen Unterschiede zu legen. Die Variation zwischen den Durchläufen lag in der Größenordnung von einigen Sekunden, weshalb die Ergebnisse auf wenige Stellen gerundet sind. Der Benchmark ist jedoch gut genug, um die folgende Frage zu beantworten: Können wir die Performance verbessern, wenn wir die Größe des Fork/Join-Pools anpassen? In diesem Fall ist die Antwort gemischt:

- Parallelisierung bringt in den meisten Fällen etwas. Der einzige (und erwartete) Verlust entsteht, wenn die Parallelisierung zusätzliche Last im I/O-Subsystem verursacht.
- CPUs sind zu etwa 20 Prozent inaktiv, wenn der Lambda-Ausdruck implizit Fork/Join-Operationen nutzt. Explizite Verwendung der Fork/Join-Operationen führt zur Ausnutzung dieser 20 Prozent und geht einher mit einer entsprechenden Verringerung der Antwortzeiten.
- Die Verringerung der Antwortzeiten tritt nicht ein, wenn I/O-Operationen Teil der verursachten Last sind. Das ist auch schlüssig, da I/O-Operationen um ein Vielfaches langsamer sind als Memory-Operationen.

Diese Beobachtungen verlangen weitere Experimente. Vermischen wir die CPU- und I/O-lastigen Operationen und erhöhen wir die Parallelität. Wir können die Anzahl der Threads kontrollieren, indem wir den JVM-Parameter „-Djava.util.concurrentForkJoinPool.common.parallelism=N" setzen. *Tabelle 3* zeigt das Ergebnis der vermischten Arbeitsabläufe mit 4, 8 und 16 Instanzen von „ForkJoinWorkerThreads". Die Ergebnisse bestätigen die Vermutung, dass mehr Threads keinen signifikanten Vorteil bringen. Der CPU-Auslastungsgraph in *Abbildung 1* gibt einige Hinweise, warum dies der Fall ist.

Die zwei Aufgaben wurden so übergeben, dass die daraus resultierenden Tasks sich hätten vermischen sollen. Wie auch immer, es gibt einen deutlichen Hinweis in *Abbildung 1:* die großen roten Bereiche aufgrund der Überladung des Kernels mit I/O-Operationen. Der erste Durchlauf mit 16 Kernen wird dominiert von Kernel-Aktivität. Im Gegensatz dazu zeigen die anderen beiden Durchläufe mit 8 und 4 Threads wesentlich weniger Kernel-Aktivität. Alle drei Durchläufe haben die gleiche Datenmenge bearbeitet, der einzige Unterschied ist die Anzahl der Threads im Common Thread Pool.

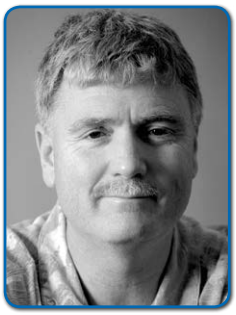
Fazit

Diese Experimente zeigen, dass man scheinbar nicht viel gewinnt, wenn man die Anzahl der Threads erhöht. Tatsächlich demonstrieren diese Ergebnisse, dass die Erhöhung der Thread-Anzahl und damit die Erhöhung der Last auf das bereits ausgelastete I/O-Subsystem ausgesprochen schädlich für die Performance sein kann.

Die Devise „Messen statt raten" sagt uns, dass unsere Optimierungen immer auf soliden Messwerten basieren sollten. Um die benötigten Messwerte zu erhalten, müssen wir oft die Überwachbarkeit der Applikation erhöhen. In diesem Fall waren wir in der Lage, durch das Hinzufügen von Messpunkten besser zu verstehen, wie das Fork/Join-Framework arbeitet. Zu diesem Zweck wurde der Code von „ForkJoinPool" direkt modifiziert. Byte-Code-Manipulation (mithilfe des Agent-Frameworks) wäre ein weiterer Weg gewesen. Bei „Adopt OpenJDK" auf GitHub wird ein entsprechendes Projekt entwickelt, das hoffentlich als offizieller Patch in Java einfließen wird.

Hinweise: Die Listings, Grafiken und Tabellen zum Artikel sind unter *„http://www.oraclejavamagazine-digital.com/javamagazine/march_april_2015#pg28"* verfügbar. Der Artikel ist im Java Magazine im April 2015 erschienen. Aus dem Amerikanischen übersetzt von Daniel van Ross, Java User Group Deutschland e.V.

Kirk Pepperdine
http://kirk.blog-city.com/



Kirk Pepperdine beschäftigte sich viele Jahre mit Hochleistungssystemen und verteilten Anwendungen mit Schwerpunkt auf Performance, Architektur, Entwicklung und Optimierung von Anwendungen für Cray und andere High-Performance-Computing-Plattformen. Er ist heute auf Java fokussiert mit dem Ziel, die Performance in jeder Phase des Projekt-Lebenszyklus zu verbessern. Aktuelles über Werkzeuge und Techniken zum Thema „Java Performance Tuning" vermittelt er in seinem Blog (http://kirk.blog-city.com) und auf der Webseite „Java Performance Tuning" (http://www.javaperformancetuning.com).

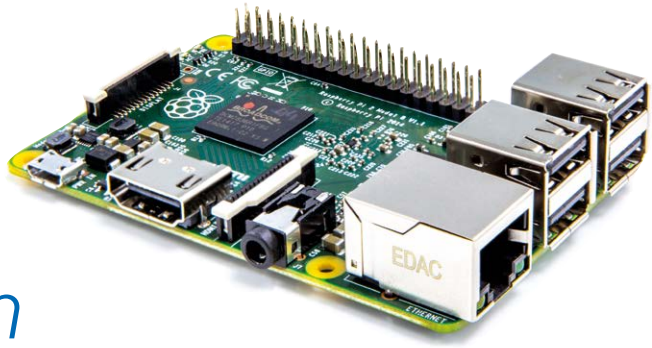
JavaLand 2016: Ticket-Verkauf gestartet

Ab sofort können sich alle Java-Enthusiasten ihre Tickets für JavaLand 2016 zum Frühbuchertarif sichern.

Vom 8. bis 10. März 2016 wird das Phantasialand in Brühl bei Köln wieder zum Zentrum der Java-Community. Denn dann soll die Besiedelung des JavaLandes bereits zum dritten Mal mit zahlreichen Community-Aktivitäten gefeiert werden. Auf die Teilnehmer warten außerdem ein zweitägiges Vortragsprogramm mit nationalen und internationalen Topspeakern, viele Gelegenheiten zum Mitmachen sowie aufregende Fahrgeschäfte. Bei der JavaLand 2015 waren rund 1.000 Java-Begeisterte in Brühl zusammen gekommen.

Tickets für die JavaLand-Konferenz sind ab sofort zum Frühbucher-Preis erhältlich. Wer vor Ort übernachten möchte, kann über die JavaLand-Webseite Hotelzimmer in den angrenzenden Hotels Ling Bao und Matamba buchen. Die Zimmerkontingente sind begrenzt.

Kaffee und Kuchen: Projekte mit Java Embedded 8 auf dem Raspberry Pi



Jens Deter

Wer vielleicht auch wie der Autor in den frühen 1980er Jahren mit dem Home-Computing eingestiegen ist und auch damals bereits gerne mit Elektronik gebastelt und programmiert hat, wird sich bestimmt auch über die kleinen, preiswerten Minicomputer gefreut haben, die seit einiger Zeit auf dem Markt sind.

Ursprünglich war der Raspberry Pi (RPi), der im Jahr 2012 auf den Markt kam, als Lehrplattform für digitale Elektronik auf Basis der Sprache Python konzipiert (daher das „Pi" im Namen, „Python interpreter"). Der Python Interpreter sollte ursprünglich fest eingebaut sein, aber am Ende wurde es dann doch eine sehr offene Architektur und so hat sich um dieses Gerät sehr schnell eine recht große Community gebildet.

Bis heute wurden bereits mehr als 5 Millionen Stück verkauft *[1]*. Dieser hohe Verbreitungsgrad und die damit verbundene sehr breite Unterstützung machen den Einstieg ins digitale Experimentieren sehr leicht und vor allem preiswert: Ein aktueller Raspberry Pi 2 Model B mit Gehäuse, Netzteil, WLAN-Stick und SD-Karte liegt bei 70 bis 80 Euro. Auch Java-Entwickler kommen hier inzwischen voll auf ihre Kosten, denn seit geraumer Zeit bringt Oracle regelmäßig parallel zu den „normalen" JDK 8 Builds jeweils ein „JDK 8 for ARM" *[2]* heraus, das speziell für ARM-basierte Plattformen wie den Raspberry Pi gedacht ist.

Anfangs wurde die Unterstützung von JavaFX für den Einsatz von grafischen Elementen explizit von Oracle beworben. Mit dem aktuellen „Update 33"-Release wurde JavaFX jedoch klamm und heimlich aus den offiziellen Builds ausgeklammert *[3]* und dem OpenJDK-Projekt übergeben. Johan Voss hat glücklicherweise gleich ein Paket zusammengestellt, das vom „javafxports"-Repository heruntergeladen und extrahiert werden kann *[4]*. Die darin verpackten Bibliotheken lassen sich einfach in entsprechende Verzeichnisse eines installierten ARM-JDK von Oracle kopieren. Damit steht JavaFX für Embedded wieder zur Verfügung.

Bisher waren die Gestaltungsmöglichkeiten von JavaFX-Oberflächen auf dem RPi aufgrund eingeschränkter Leistung überschaubar. Vor kurzem ist jedoch der Raspberry Pi 2 mit QuadCore-Prozessor und 1 GB RAM erschienen und damit wird es wieder interessanter, UIs für den RPi zu entwickeln *[5]*. Da JavaFX als Besonderheit direkt im Frame-Buffer läuft, ist kein laufender X-Server erforderlich und es können deutlich Ressourcen gespart werden. Dieser Artikel zeigt exemplarisch, wie sich Eingangs- und Ausgangs-Zustände eines Raspberry Pi über eine JavaFX-basierte Touch-Oberfläche manipulieren und visualisieren lassen.

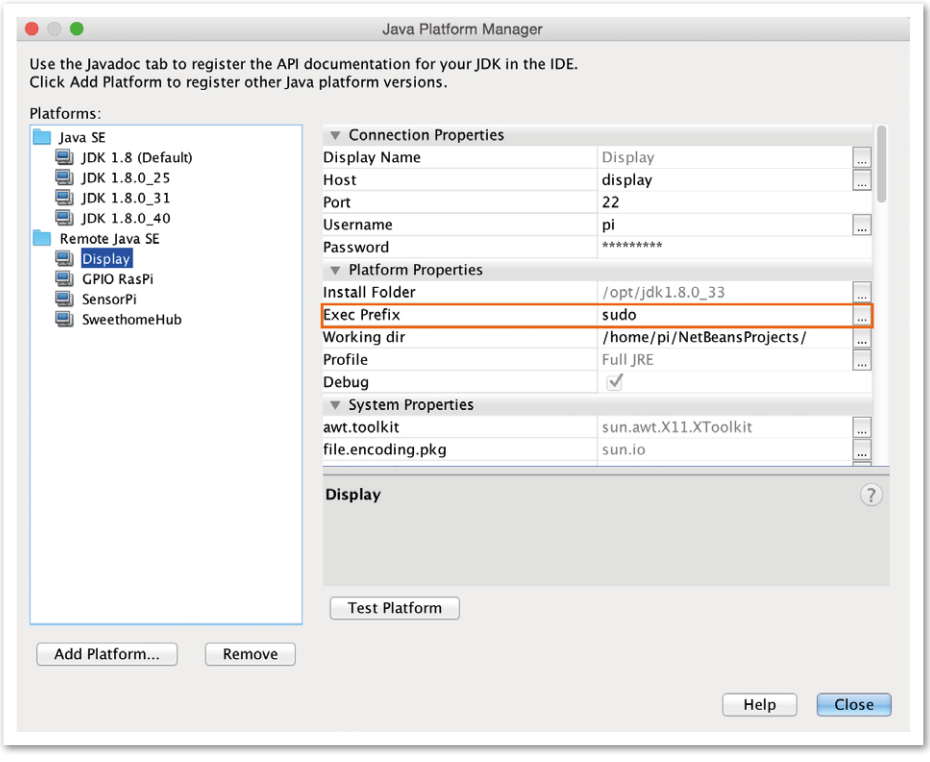


Abbildung 1: Remote Plattform in NetBeans 8.0.2

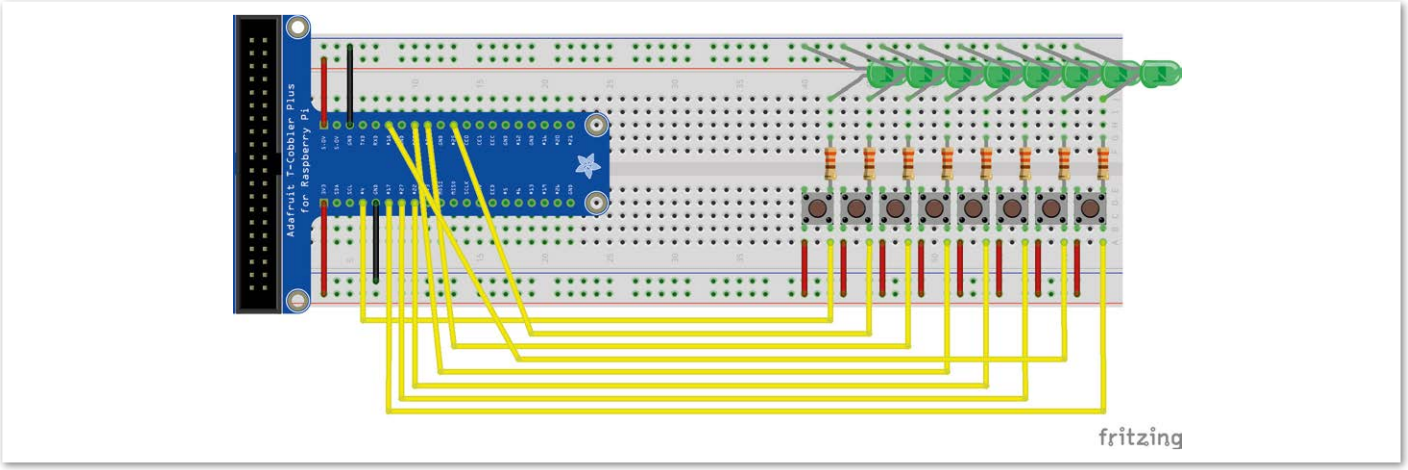


Abbildung 2: Schaltungsaufbau der Acht-Kanal-I/O-Steckplatine

NetBeans 8

NetBeans 8 *[6]* hält für Internet-of-Things-Java-Entwickler ein sehr beachtenswertes Feature bereit, das bereits in der Standard-Installation voll integriert zur Verfügung steht: die Möglichkeit, direkt aus der IDE ein Projekt auf ein Embedded Device wie einen Raspberry Pi einzurichten und dort „remote" auszuführen. Dabei kann man das Projekt auch im Debug-Modus ausführen lassen oder mit dem Profiler zur Laufzeit überwachen.

Vorausgesetzt, auf dem entfernten Gerät läuft ein SSH-Dienst, kann eine neue Plattform vom Typ „Remote Java Standard Edition" erstellt werden. Hier sind dann im folgenden Dialog Informationen wie Adresse des Geräts, Login Credentials und Pfad zum benutzenden JRE hinterlegt. Danach kann diese in den Projekteinstellungen dann als Ziel-Plattform ausgewählt und das Projekt laufen gelassen werden *(siehe Abbildung 1)*.

Dabei wird das Programm mit allen Abhängigkeiten lokal gebaut, per SCP übertragen und „remote" via „SSHExec" ausgeführt. Die Ausgaben von „stdout" und „stderr" werden schließlich in die Konsole von NetBeans umgeleitet. Das Remote-Deployment dient nicht nur zur entfernten Ausführung, sondern kann eben auch als Auslieferung verstanden werden, denn die Applikation wird anschließend nicht vom RPi gelöscht.

Besonders betont sei hier noch das Setzen von „sudo" als „Exec Prefix"-Property. Diese Einstellung sorgt dafür, dass das Java-Programm mit „root"-Rechten auf dem entfernten Gerät ausgeführt wird. Wenn keine anderweitigen Gruppen- und Rechte-Umstellungen auf dem Raspberry Pi eingerichtet werden sollen, ist dies unbedingt nötig, um auf die GPIO-Schnittstelle zugreifen zu dürfen.

Da all diese Schritte ANT-basiert sind, kann hier leider nicht auf Maven-Repositories zugegriffen werden. Es ist also erforderlich, Bibliotheken und Abhängigkeiten manuell mit NetBeans-Bordmitteln zu verwalten.

GPIO und Pi4J

Zunächst ein Blick auf die General-Purpose-Input/Output-Schnittstelle (GPIO) des Raspberry Pi *[7]*. Sie ist das IO-Interface des RPi, deren Funktionen sich sehr leicht mit „wiringPi" *[8]*, einem auf C basierenden API von Gordon Henderson („@drogon"), nutzen lassen. Praktischerweise gibt es mit Pi4J ein Java-API, das genau auf „wiringPi" aufsetzt und dieses gleich passend automatisch mitbringt.

Der GPIO-Header steht im Mittelpunkt der ausgehenden und eingehenden Kommunikation. Beim Modell B können insgesamt 17 Pins zum I/O angefordert werden: acht GPIO-Pins, zwei Pins vom I2C-Interface, fünf Pins Serial Peripheral Interface, zwei Pins Serial UART plus vier weitere via P5-Connector (nur Rev. 2.0). Das neuere Modell B+ hält noch weitere neun GPIOs bereit. Pi4J unterstützt alle RPi-Modelle, vom einfachen I/O über PWM und SPI bis I2C bleiben keine Wünsche offen.

Pi4J als Bibliothek einrichten

Zunächst lädt und entpackt man ein aktuelles Build, z.B. den Pi4J-1.0-Release-Candidate von der Pi4J-Site *[9]*. Anschließend

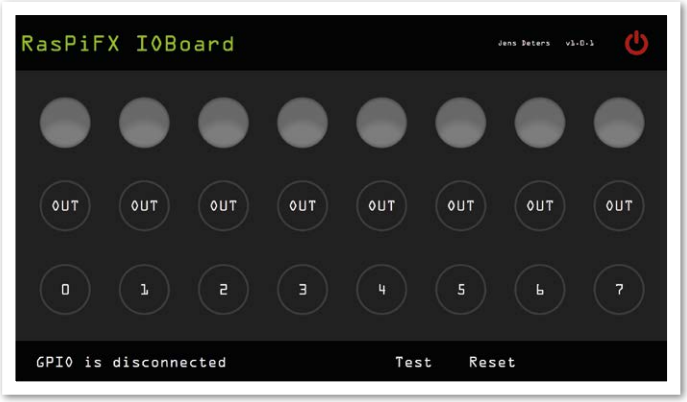


Abbildung 3: Das JavaFX-UI zum Acht-Kanal-I/O

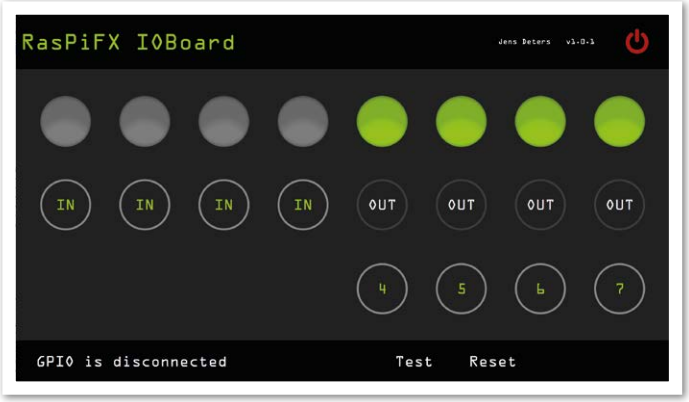


Abbildung 4: Über das UI kann zum Beispiel auch der I/O-Modus pro Kanal (IN/OUT) gewählt werden

Abbildung 5: Ein Kanal der Schaltung wird in der UI durch eine Spalteneinheit repräsentiert

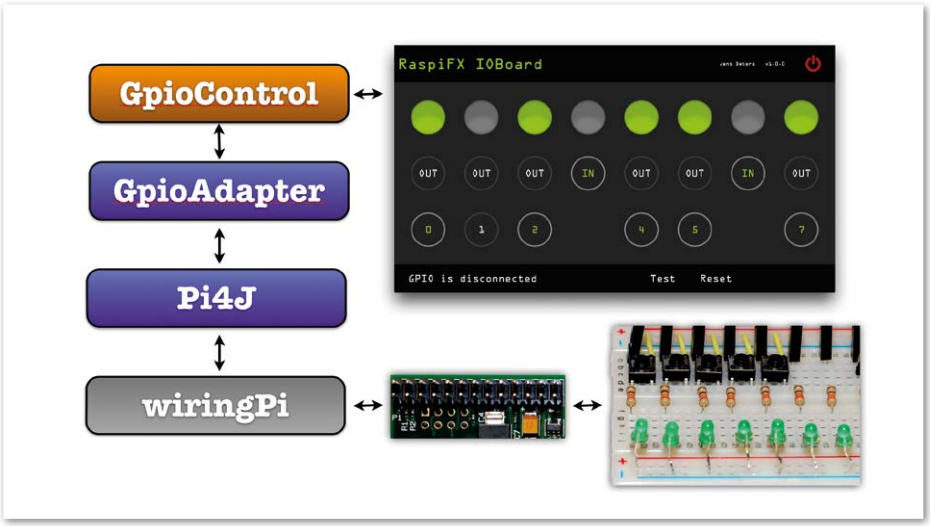


Abbildung 6: Die Architektur des Datenflusses vom Taster zur UI und zurück

kann dann in NetBeans eine Bibliothek eingerichtet werden. Für das Projekt „Acht-Kanal-I/O-Interface mit JavaFX-basierter Touch-Oberfläche" sind folgende Bauteile notwendig *(siehe Abbildung 2)*:

- Raspberry Pi (B oder B+)
- Breakout-Kit
- Zwei Breadboards
- Steckverbinder
- 7" Touch-Display von Chalk-Elec [10]
- Acht LEDs
- Acht 330 Ohm Vorwiderstände
- Acht Taster

Die Zustände für die Ausgänge werden über Toggle-Buttons der grafischen Oberfläche manipuliert, die Eingänge über Taster getriggert und die Zustände dann in beiden Fällen an dem UI angezeigt. Der I/O-Modus (Eingang/Ausgang) kann jeweils pro Kanal zur Laufzeit umgeschaltet werden *(siehe Abbildungen 3 bis 5)*.

Zum besseren Testen der Oberfläche kann zudem der GPIO-Controller wahlweise aktiviert oder deaktiviert sein. Außerdem darf ein „Exit"-Button nicht vergessen werden: JavaFX kann direkt aus der Konsole den Frame-Buffer übernehmen und die Anwendung kann dann nicht ohne Weiteres beendet werden (CTRL-D ist wirkungslos, weil JavaFX Keyboard-Events abfängt).

Für die UI soll die eigentliche GPIO-Kommunikation transparent sein. Daher werden alle Zustände über einen JavaFX-Properties-Adapter gekapselt. Diese Zwischenschicht bildet die Verbindung von UI und einer Einheit, die über Pi4J das GPIO-Interface anspricht *(siehe Abbildung 6)*.

Der Code des Projekts (RaspiGPIO-ControllerFX) kann via BitBucket bezogen werden *[11]*. Zudem gibt es noch drei YouTube-Videos des Autors zum Thema „NetBeans Remote Deployment", die auch für den Oracle Virtual Developer Day 2014 verwendet wurden *[12, 13, 14]*.

Weiterführende Links

[1] http://www.heise.de/open/meldung/Raspberry-Pi-Mehr-als-fuenf-Millionen-verkaufte-Kleinstcomputer-2553159.html
[2] JDK 8 for ARM Downloads: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-arm-downloads-2187472.html
[3] JDK 8 for ARM Update 33 Release Notes: http://www.oracle.com/technetwork/java/javase/jdk-8u33-arm-relnotes-2406696.html
[4] JavaFX for ARM: https://bitbucket.org/javafx-ports/arm/downloads
[5] RubikFX (JavaFX 3D) on Raspberry Pi 2: https://youtu.be/vkpyfpv849s
[6] NetBeans 8 Download: https://netbeans.org/downloads
[7] GPIO Pinbelegung: http://raspberrypiguide.de
[8] wiringPI: http://wiringpi.com
[9] Pi4J: http://pi4j.com/download.html
[10] Chalk-Elec 7" TouchDisplay: http://www.chalk-elec.com/?page_id=1280#!/7-open-frame-universal-HDMI-LCD-with-capacitive-multi-touch/p/21750207/category=3094861
[11] Bitbucket: RaspiGPIOControllerFX: https://bitbucket.org/Jerady/raspigpiocontrollerfx
[12] NetBeans 8 - Shortcut to Remote Platforms, Part 1, headless: https://youtu.be/BIGgUZeTI4w
[13] NetBeans 8 - Shortcut to Remote Platforms, Part 2, JavaFX on Pi: https://youtu.be/rgrHzq83BCQ
[14] NetBeans 8 - Shortcut to Remote Platforms, Part 3, Remote Platform Debugging: https://youtu.be/22xDsjMEMHE
[15] James Gosling, Robots, the Raspberry Pi, and Small Devices [UGF8907] (NetBeans Day), James Gosling, Jose Pereda, Shai Almog, Johannes Weigend, Jens Deters: https://oracleus.activeevents.com/2014/connect/sessionDetail.ww?SESSION_ID=8907
[16] Debugging and Profiling Robots with James Gosling [CON6699], James Gosling, Mark Heckler, Jose Pereda, Geertjan Wielenga, Jens Deters): https://oracleus.activeevents.com/2014/connect/sessionDetail.ww?SESSION_ID=6699

Jens Deters
mail@jensd.de

Etwa 25 Jahre ist es her, dass Jens Deters mit dem Home-Computing begonnen hat, damals zogen ihn Computer in ihren Bann und die letzten 15 Jahren hatte er verschiedenste Rollen im IT- und Telekommunikations-Umfeld inne (Software-Entwickler, Trainer, Berater, Projekt- und Produkt-Manager). Zurzeit arbeitet er im Bereich ziviler und militärischer Luftfahrtbehörden mit Blick auf Java EE, Java FX, Swing und Web-UI. Er schreibt regelmäßig über seine Projekte unter „www.jensd.de" sowie „www.mqttfx.org" und trägt zur JavaFX- und IoT-Community bei. Jens Deter ist Mitglied des NetBeans Dream Team.

MySQL und Java für die Regelung von Asynchronmaschinen

Eric Aristhide Nyamsi

Die Energietechnik-Informatik ist das neue Tool zur Programmierung der Schnittstellen von objektorientierter Programmierung und Energie-Effizienz. Es findet Begeisterung in der Industrie und in der öffentlichen Forschung. Mithilfe der objektorientierten Programmierung, etwa mit Java, werden Anwendungen für die Regelung des Asynchronmotors entwickelt.

Die Energietechnik-Informatik beruht auf Anbindungen von Datenbanken wie MySQL mithilfe der objektorientierten Programmierung mit Java am Projekt. Der Artikel zeigt am Beispiel einer Energiemanagement-Anwendung für die Regelung der Asynchronmaschine, wie die Umsetzung vom Datenmodell bis zur grafischen Oberfläche mit dem Java-XDEV-Framework ablaufen kann. Die Regelung des Asynchronmotors fokussiert auf die Entwicklung von Energiemanagement-Anwendungen.

Der gezeigte Fall bezieht sich auf die Erfassung der Messdaten für die Rotorfrequenz-Regelung. Ziel ist es, MySQL als Werkzeug für Datenbanken darzustellen. Mithilfe von Java und der MySQL-Datenbank werden IT-Lösungen in Bezug auf mechanische Drehzahl, elektromagnetische Winkelgeschwindigkeit, Rotorkreisfrequenz, Ständerkreisfrequenz und Kennlinienspannung entwickelt.

Schnittstelle zwischen IT und Regelung

Die Energietechnik-Informatik hat in Deutschland durch die Initiative der Entwickler, Energietechniker, Web-Designer, Freiberufler, Forscher, Zeitschriften und Verlage massiv an Bedeutung gewonnen. Sie basiert

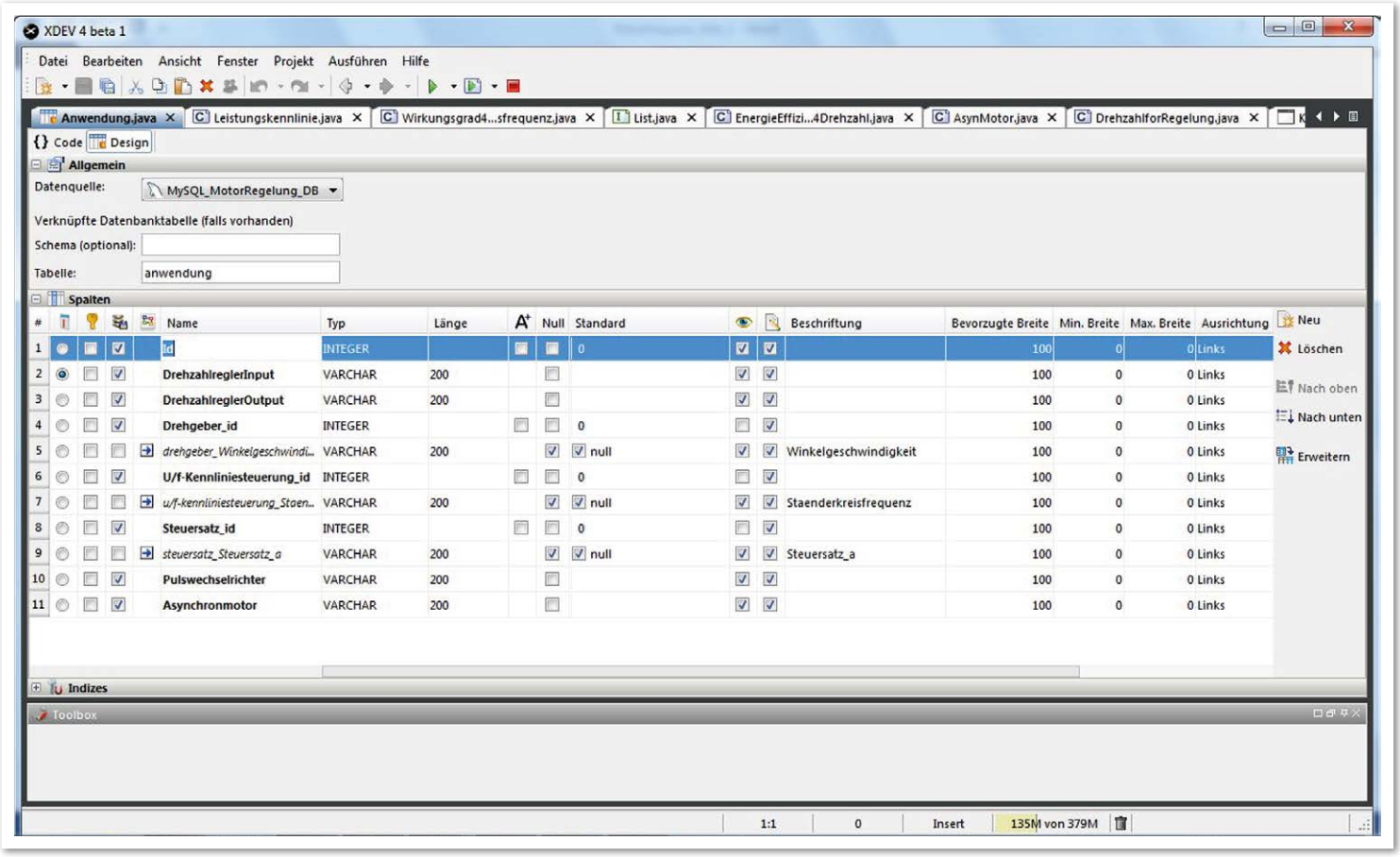


Abbildung 1: Datenbank-Anwendung für die Rotorfrequenz-Regelung mit U/f -Kennliniensteuerung

auf der Anbindung von Datenbanken mit objektorientierten Programmierungssprachen an die Entwicklungsprojekte. Ziel ist es, sowohl Daten mithilfe der objektorientierten Programmierung zu analysieren und zu interpretieren als auch die technische Realisierung über Schnittstellen zu modellieren und zu simulieren. Die Informationsverarbeitungsmöglichkeiten energietechnischer Systeme begeistern Entwickler der Energiemanagement-Anwendungen. Der Artikel gibt einen Überblick über die Grafikprogrammierung von Asynchronmotor-Anwendungen bezüglich der Rotorfrequenzregelung.

Entwicklung der Anwendung

Das XDEV-Application-Framework stellt eine Basis-Architektur und -Infrastruktur für Datenbank-Anwendungen mit grafischen Benutzeroberflächen zur Verfügung, die auf dem Client-Server-Modell basieren *[1]*. MySQL ist ein weit verbreitetes, relationales Datenbank-Management-System (RDBMS) und Xampp ein freies Software-Bundle, bestehend aus Web- und FTP-Server, MySQL, Apache sowie weiteren Komponenten. MySQL besteht einerseits aus dem Datenbank-Verwaltungssystem und andererseits aus der eigentlichen Datenbank mit den Daten *[2]*. *Abbildung 1* zeigt die Erzeugung der Datenbank-Anwendung über MySQL-Xampp.

Die Rotorfrequenz-Regelung basiert auf U/f-Kennliniensteuerung. Der Motor wird in die elektromechanische Winkelgeschwindigkeit umgerechnet. *Abbildung 2* und die *Listings 1 und 2* zeigen die Rotorfrequenz-Regelung mit U/f-Kennliniensteuerung.

Entwicklung von GUI-Anwendungen

Mit einem Grafik-Programm des XDEV4-Frameworks lassen sich Java-Oberflächen erstellen und die Oberflächen-Komponenten aus einem Palettenfenster per Drag & Drop in das Arbeitsfenster einfügen *[1]*. *Abbildung 3* und *Listing 3 bis 5* geben einen Überblick über die Konstruktion der grafischen Oberfläche „RotorManagement"

Die Abbildung stellt die Aktivierung des Layouts mithilfe der Option „BorderLayout" dar. Das Hauptfenster „RotorManagement" lässt sich durch „XdevWindowContainer" modularisieren. In diesem Fenster wird dann für jedes Feature ein „XdevWindowContainer" als Platzhalter eingefügt. Ein Fenster-Aufruf findet zur

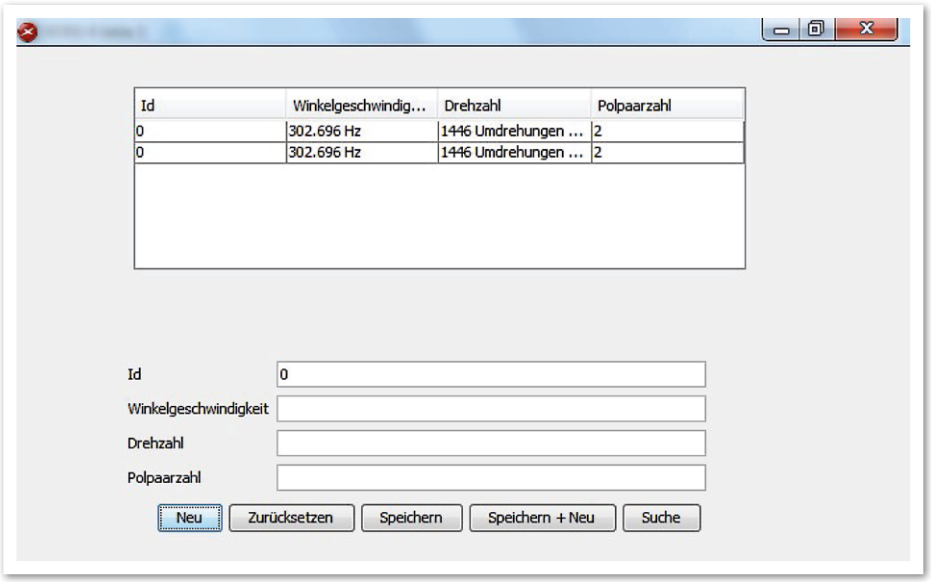


Abbildung 2: Grafik der Rotorfrequenz-Regelung mit U/f-Kennliniensteuerung

```
public class Kennliniensteuerungen
extends XdevWindow
{

  @EventHandlerDelegate void this_windowClosing(WindowEvent event)
  {
     close();
  }

  @EventHandlerDelegate void cmdNew_actionPerformed(ActionEvent event)
  {
     formular.reset(VirtuelleTabellen.UFKennliniesteuerung.VT);
  }

  @EventHandlerDelegate void cmdReset_actionPerformed(ActionEvent event)
  {
     formular.reset();
  }

  @EventHandlerDelegate void cmdSave_actionPerformed(ActionEvent event)
  {
     if(formular.verifyFormularComponents())
     {
        try
        {
           formular.save();
        }
        catch(Exception e)
        {
           e.printStackTrace();
        }
     }
  }
```

Listing 1: Implementierung des Interface „XdevWindow" in der Klasse „Kennliniensteuerungen"

```
@EventHandlerDelegate void button_actionPerformed(ActionEvent event)
  {
     XDEV.OpenWindow(new OpenWindow()
     {
        @Override
        public void init()
        {
           setXdevWindow(new Kennliniesteuerungen());
           setContainerType(ContainerType.DIALOG);
           setModal(true);
        }
     });
  }
```

Listing 2: Grafik-Programmierung in der Klasse „Kennliniensteuerungen"

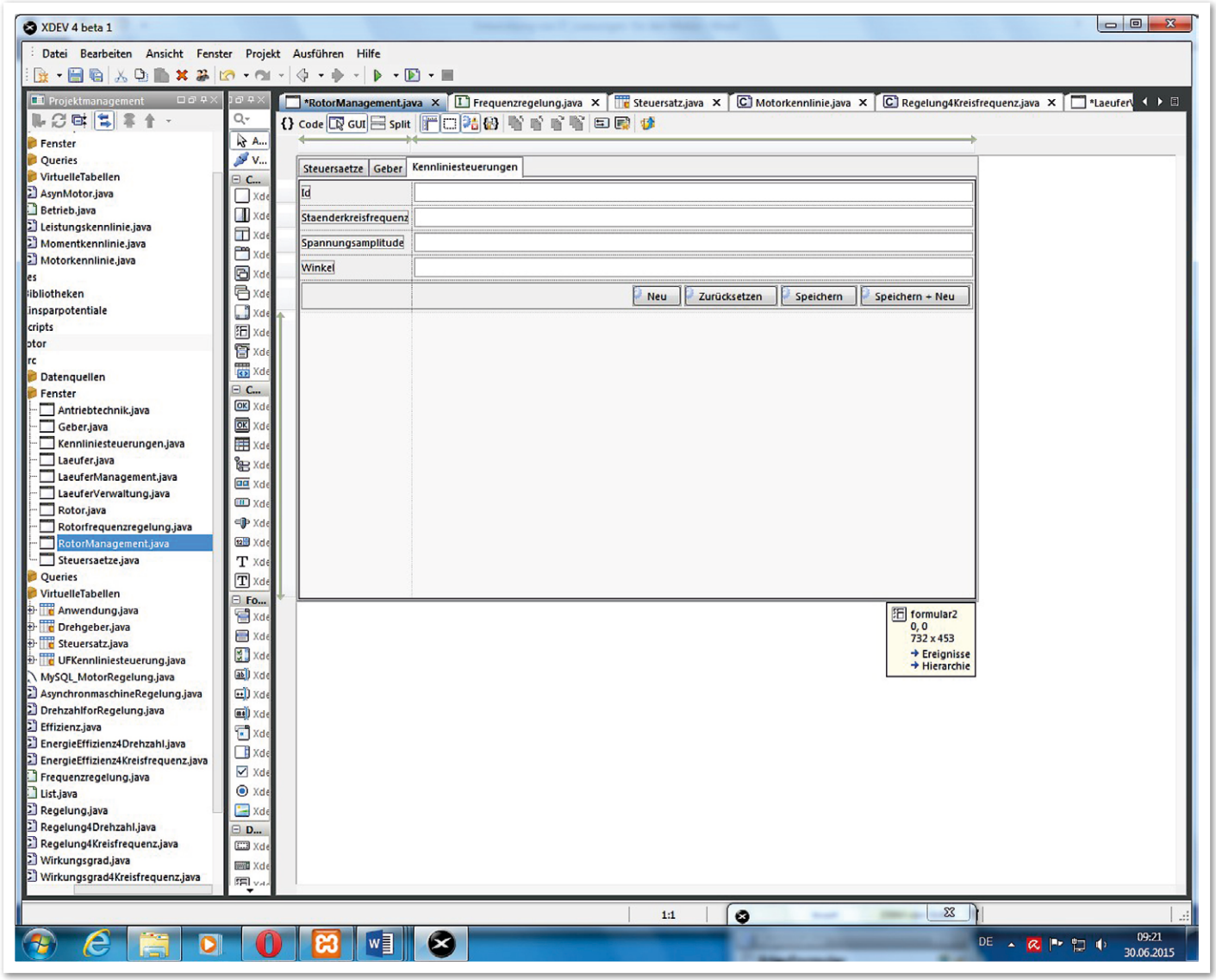


Abbildung 3 : Konstruktion der grafischen Oberfläche „RotorManagement"

```
@EventHandlerDelegate void cmdSaveAndNew2_
actionPerformed(ActionEvent event)
  {
     if(formular2.verifyFormularComponents())
     {
        try
        {
           formular2.save();
           formular2.reset(VirtuelleTabellen.UFKenn-
liniesteuerung.VT);
        }
        catch(Exception e)
        {
           e.printStackTrace();
        }
     }
  }
```

Listing 3: Überblick über die Speicherung von Datensätzen für die Tabelle „UFKennliniesteuerung"

```
@EventHandlerDelegate void cmdSaveAndNew1_
actionPerformed(ActionEvent event)
  {
     if(formular1.verifyFormularComponents())
     {
        try
        {
           Formular1.save();
           Formular1.reset(VirtuelleTabellen.Drehge-
ber.VT);
        }
        catch(Exception e)
        {
           e.printStackTrace();
        }
     }
  }
```

Listing 4: Überblick über die Speicherung von Datensätzen für die Tabelle „Drehgeber"

Laufzeit automatisch statt. Die *Abbildungen 4 bis 6* zeigen die Anwendungen der grafischen Oberflächen bezüglich der virtuellen Tabellen „Drehgeber" beziehungsweise „Steuersatz" und „UFKennliniesteuerung".

Fazit

Die Energietechnik-Informatik ohne Datenbanken ist sinnlos, weil das wesentliche Ziel

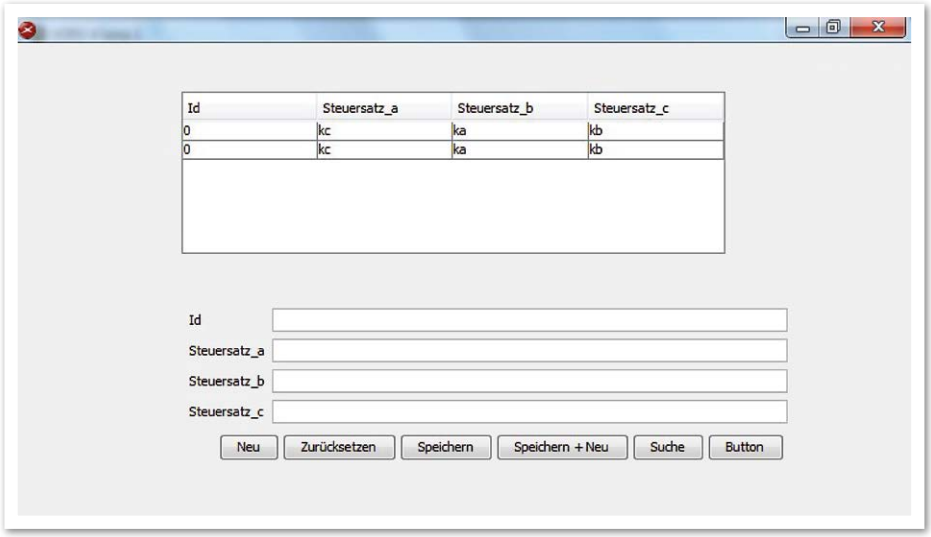


Abbildung 4 : Grafische Oberfläche der Tabelle „Drehgeber"

```
@EventHandlerDele-
gate void cmdSaveAndNew3_
actionPerformed(ActionEvent event)
  {
     if(formular3.verifyFormular-
Components())
     {
        try
        {
           Formular3.save();
           Formular3.
reset(VirtuelleTabellen.Steuer-
satz.VT);
        }
        catch(Exception e)
        {
           e.printStackTrace();
        }
     }
  }
```

Listing 5: Überblick über die Speicherung von Datensätzen für die Tabelle „Steuersatz"

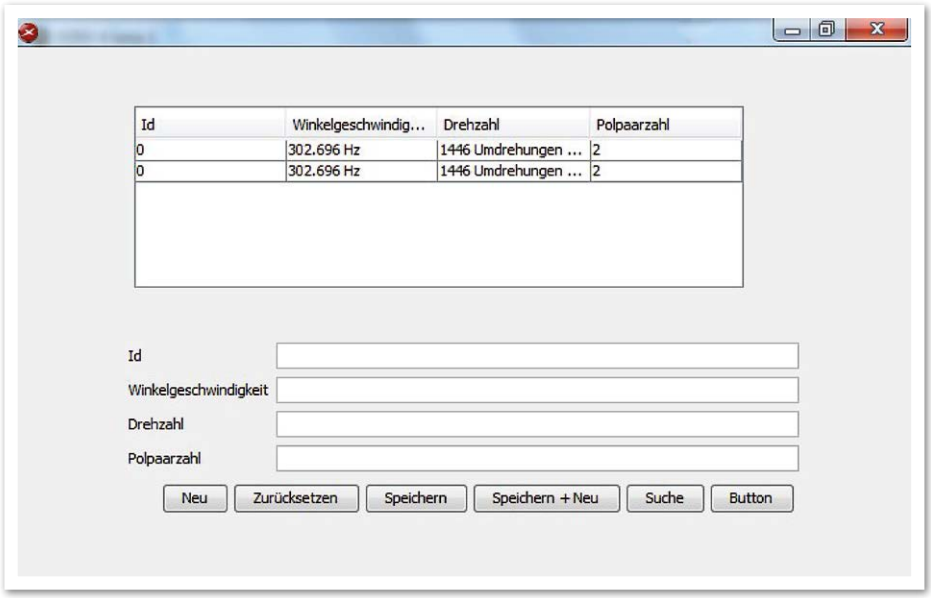


Abbildung 5: Grafische Oberfläche der Tabelle „Steuersatz"

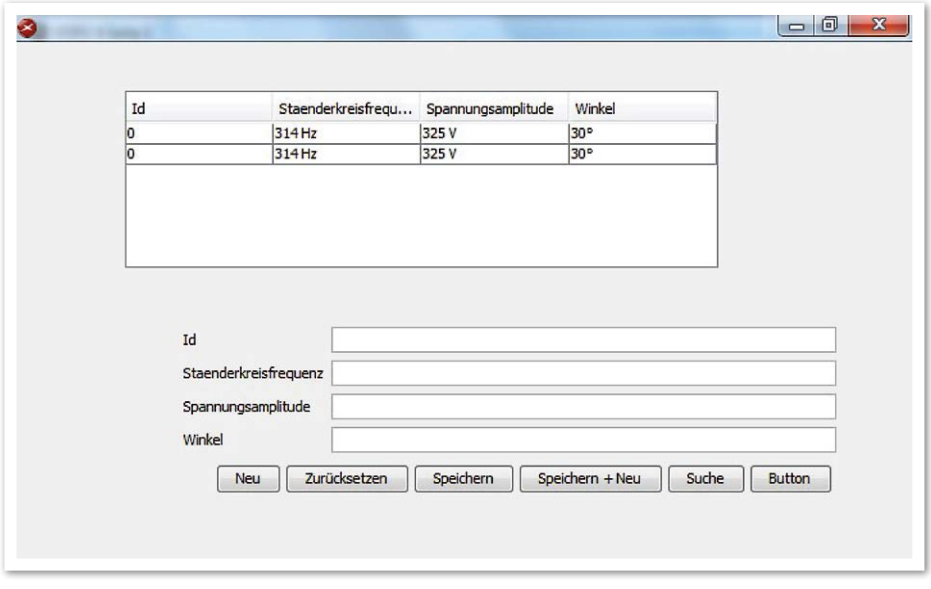


Abbildung 6: Grafische Oberfläche der Tabelle „UFKennliniesteuerung"

der Anwendungen der IT in der Energietechnik in Anbindungen von Datenbanken liegt. Der Datenbank-Einsatz zur Regelung der Asynchronmaschine ist selbstverständlich ein Kernelement der Anwendungen der IT in der Energietechnik. Die Anbindungen der Datenbanken wie MySQL mit Java sind wichtig, um automatisierte Datenbank-Abfragen zu ermöglichen.

Quellen

[1] http://cms.xdev-software.de/xdevdoku/HTML
[2] Johannes Teigelköter, Energieeffiziente Antriebe, Vieweg+Teubner, Springer Fachmedien, 2013
[3] Florence Maurice, PHP 5.4 & MySQL 5.5, Programmierung dynamischer Websites, Addison-Wesley Verlag, 2012, S. 283-320

Eric Aristhide Nyamsi
eric.aristhide@gmx.de

Eric Aristhide Nyamsi, M.Sc., ist Freelancer für Software-Entwicklung in Karlsruhe und entwickelt Rapid Application Development und Software-Architektur. Sein Fokus liegt sowohl auf Software-Design als auch auf der Schnittstelle von IT und Engineering. Er entwickelt Anwendungen auf Basis von Java für die Schnittstelle von IT und Energietechnik. Sein Aufgabenfeld umfasst die Entwicklung von Administration-Frontends auf Basis von Java.

Bean Testing mit CDI: Schnelles und produktives Testen von komplexen Java-EE-Anwendungen

Carlos Barragan, NovaTec GmbH

Enterprise-Software-Systeme stellen in der Entwicklung hohe Ansprüche an Umfang und Tiefe der Tests. Ohne ausreichende Test-Abdeckung besteht die Gefahr, dass komplexe Software-Systeme bei fachlichen oder technischen Änderungen rasch degenerieren und unwartbar werden.

Während beim Unit Testing von Java-EE-Anwendungen geringer Testumfang und hoher Aufwand vor allem beim Mocking störend in Erscheinung treten können, wirkt bei Integrationstests häufig langsame Feedback-Geschwindigkeit als limitierender Faktor. Bean Testing mit Java CDI ist ein Mittelweg, der die Vorteile beider Verfahren vereint: Anwendungen können mit ihren Abhängigkeiten ohne hohen Simulationsaufwand mit ebenso schnellem Feedback wie beim Unit-Test einzelner Methoden getestet werden. Bean Testing ist dabei kein Ersatz für Unit- oder Integration-Testing, sondern ein zusätzliches schnelles, schlankes und mächtiges Test-Instrument, das sich in der Praxis bei der Entwicklung großer Software-Projekte bereits vorteilhaft bewährt hat.

In der Geschäftslogik einer Unternehmenssoftware arbeiten in der Regel mehrere Services zusammen. Testet man einzelne Units, etwa mit einem Framework wie Mockito, erhebt sich zwangsläufig die Frage, welche Services in das Mocking einbezogen werden sollen und welche nicht. In jedem Fall entsteht durch das Mocking zusätzlicher Aufwand. Zudem ist jede Simulation eine potenzielle Fehlerquelle und wirkt sich damit möglicherweise negativ auf die Testqualität aus. Das notwendige Deployment kostet Zeit, unter Umständen schon einmal mehrere Minuten, dadurch sinkt die Zahl der in der verfügbaren Zeit möglichen Testvorgänge. Treten dazu noch Probleme mit dem Application-Server auf, kann zusätzlich hoher Zeitaufwand entstehen.

Setzt man hingegen ein Werkzeug für Integrationstests wie Arquillian von JBoss ein, automatisiert dieses Tool zwar das Deployment auf einen Applikationsserver, sodass eine Anwendung ohne allzu großen Aufwand innerhalb eines Containers getestet werden kann. Allerdings ist dieses Verfahren für nicht triviale Anwendungen, die im Enterprise-Bereich gang und gäbe sind, nicht einfach zu konfigurieren. Kommen neue Abhängigkeiten hinzu, muss gegebenenfalls manuell neu- oder umprogrammiert werden. Das verleitet dazu, Integrationen und Schnittstellen tendenziell eher wegzulassen – wiederum zulasten der Testqualität. Für Unit-Tests ist das Verfahren auch deswegen weniger geeignet, weil das Deployment die Feedback-Geschwindigkeit wieder reduziert.

CDI als Werkzeug für das Bean Testing in Java-EE-Anwendungen

Java CDI wurde als Spezifikation für Context and Dependency Injection bereits unter Java EE 6 eingeführt, steht also auf allen Application-Servern ab Java EE 6 „out of the box" zur Verfügung, unter Java EE 7 in der erweiterten Version CDI 1.1. Mit einer Vielzahl von Features und Erweiterungen ist Java CDI vielseitig einsetzbar und inzwischen weithin bekannt.

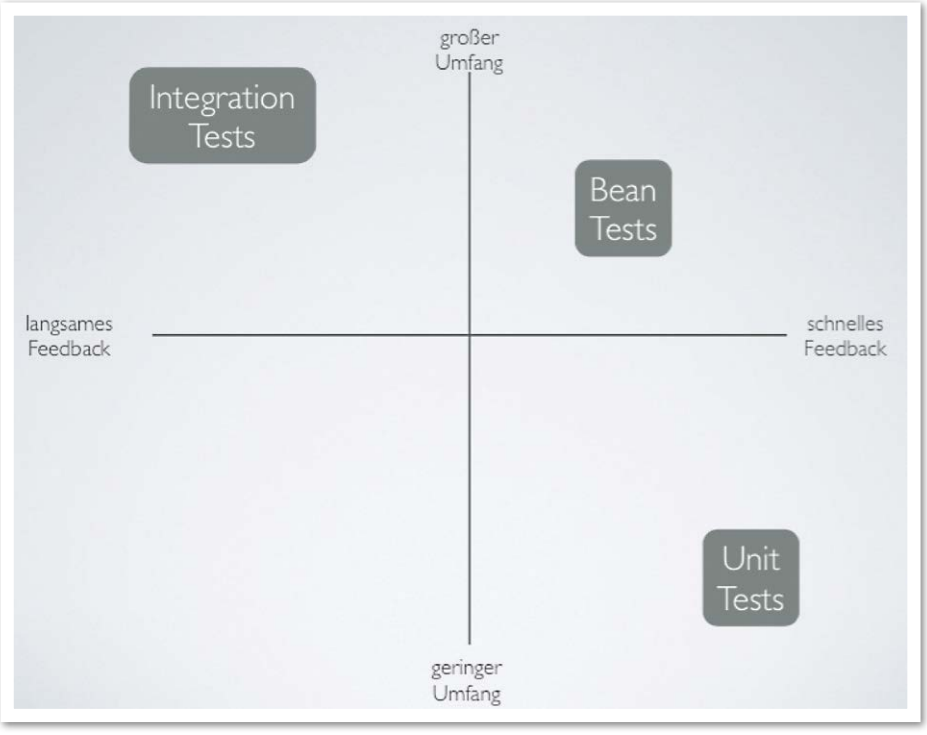


Abbildung 1: CDI-Bean-Test im Vergleich zu Unit- und Integrationstest

Im Rahmen eines Projekts zur Entwicklung einer Java-EE-Anwendung für Bewerbungsprozesse entstand der Gedanke, CDI als Test-Instrument auf Enterprise-Anwendungen zu übertragen, um Unit-Tests zu erstellen und durchzuführen. Damit konnte die Anforderung erfüllt werden, schnelles Feedback ähnlich zu dem eines normalen JUnit-Tests zu erreichen und Abhängigkeiten automatisch zu injizieren, als ob die Anwendung auf dem Applikationsserver laufen würde, um so das Mocking von Abhängigkeiten zu vermeiden.

Die Ablaufgeschwindigkeit von Tests mit CDI ist dabei deutlich höher als bei Integrationstests mit Arquillian. Da kein Applikationsserver gestartet werden muss, laufen die Tests in Millisekunden ab; auch die für nicht triviale Java-EE-Anwendungen aufwändige Konfiguration eines Embedded Application Servers entfällt *(siehe Abbildung 1)*.

Mit einfachen Modifikationen der Metadaten werden Enterprise JavaBeans in CDI-Beans konvertiert, die der CDI-Container erkennt. Grundsätzlich wird jede EJB in eine CDI-Bean konvertiert. Diese Konvertierung findet statt, ohne dass der Quellcode verändert werden muss.

Damit kann der Entwickler die EJBs faktisch unverändert testen. Abhängigkeiten werden nicht simuliert, sondern von CDI injiziert. Zum einen wird so die Zeit für die Erstellung der Mocks eingespart; zum anderen werden mit den Bean-Tests die tatsächlichen Abhängigkeiten und der Quellcode selbst getestet *(siehe Listing 1)*.

Für das Bean Testing mit CDI entsteht ein einmaliger Aufwand, um die Umgebung zur Verfügung zu stellen. Ist die Test-Umgebung aufgesetzt, kann man die EJBs direkt ohne Mocking testen. Der Test selbst wird vom JUnit-Framework ausgeführt, entspricht also faktisch einem typischen JUnit-Test.

Das Bean-Test-Framework übernimmt die Deployment Injection (DI) des EntityManager und der CDI-Container kümmert sich um die DI jeder Bean. Allerdings haben wir keine CDI-Beans, sondern EJBs. Damit der CDI-Container die EJBs erkennen beziehungsweise injizieren kann, kommt eines der mächtigsten Features von CDI zum Einsatz. Die CDI-Test-Extension überprüft die Markierungen der Beans, fügt über Änderungen am Metamodell Annotationen hinzu, ohne den Bytecode zu modifizieren, und stellt die modifizierte Bean anstelle der ursprünglichen zur Verfügung. Bei der Initialisierung des CDI-Containers wird der ganze Classpath nach möglichen CDI-Beans, Interceptors, Events etc. gescannt. Auf diese Weise erkennt der CDI-Container, wo die Dependencies zu finden sind.

Bei diesem Verfahren kommen ausschließlich standardisierte Mittel zum Einsatz, um EJBs zu testen. Die EJBs müssen nicht eingerichtet sein und teilweise sind keine Mocks erforderlich. Im Falle externer Abhängigkeiten müssen Mocks zur Verfügung stehen, das wird aber an einer zentralen Stelle über „CDI-Producers" umgesetzt. Dependencies werden also tatsächlich aufgelöst und Code wird so aufgerufen, wie er auch im Container ablaufen würde. Trotzdem wird der Test auf einem normalen Entwicklerrechner in ein paar Sekunden ausgeführt.

Eine solche Feedback-Geschwindigkeit kann man durchaus als „Unit Testing" betrachten. „Bean Testing" heißt das Verfahren, weil es sich nicht um einen klassischen Unit-Test handelt, auch wenn der eigentliche Test davon kaum zu unterscheiden ist und die Feedbackgeschwindigkeit dem auch sehr nahekommt *(siehe Abbildung 2)*.

Vorteile und Möglichkeiten des Bean Testing mit CDI

Die Vorteile dieses Verfahrens liegen auf der Hand. Weil CDI getrennt vom Applikationsserver läuft, ist ein JEE-Applikationsserver oder auch ein Embedded Server nicht er-

```
public class BeanTestExtension implements Extension {

  public <X> void processInjectionTarget(@Observes ProcessAnnotatedType<X> pat) {
     if (pat.getAnnotatedType().isAnnotationPresent(Stateless.class) || pat.getAnnotatedType().isAnnotationPresent(
MessageDriven.class)) {
        modifyAnnotatedTypeMetaData(pat);
     } else if (pat.getAnnotatedType().isAnnotationPresent(Interceptor.class)) {
        processInterceptorDependencies(pat);
     }
  }

  /**
  * Adds {@link Transactional} and {@link RequestScoped} to the given
  * annotated type and converts its EJB injection points into CDI injection
  * points (i.e. it adds the {@link Inject})
  *
  * @param <X> the type of the annotated type
  * @param pat the process annotated type.
  */
  private <X> void modifyAnnotatedTypeMetaData(ProcessAnnotatedType<X> pat) {
        Transactional transactionalAnnotation = AnnotationInstanceProvider.of(Transactional.class);
        RequestScoped requestScopedAnnotation = AnnotationInstanceProvider.of(RequestScoped.class);

        AnnotatedType at = pat.getAnnotatedType();
        AnnotatedTypeBuilder<X> builder = new AnnotatedTypeBuilder<X>().readFromType(at);
  builder.addToClass(transactionalAnnotation).addToClass(requestScopedAnnotation);
        addInjectAnnotation(at, builder);
        //Set the wrapper instead the actual annotated type
        pat.setAnnotatedType(builder.create());
  }

  // Restlicher Code ausgelassen.
 }
```

Listing 1: CDI-Extension zur Konvertierung von EJBs in CDI-Beans

Abbildung 2: Ausführungsgeschwindigkeit des Bean-Tests einschließlich Bereitstellung der Datenbank und Testdaten

forderlich, um Abhängigkeiten aufzulösen. Das Feedback wird dadurch erheblich kürzer und schneller. Der CDI-Container kann ohne Application-Server gestartet werden; Bean Testing stellt dabei eine Reihe wichtiger Funktionen eines Application-Servers wie Transaction Management oder Transaction Propagation zur Verfügung. Anders als bei Arquillian, das eine vergleichbare Testtiefe bietet, sind keine aufwändigen Zusatzkonfigurationen notwendig.

Weil die Iterationen sich dadurch reduzieren und die Feedback-Zyklen schneller werden, kann hinsichtlich Zeit und Umfang mehr getestet werden – ein klarer Vorteil gegenüber reinem Unit Testing. Die hohe Feedback-Geschwindigkeit entspricht dabei der von Unit-Tests. Das bietet die Möglichkeit, verschiedene Konstellationen auszuprobieren, ohne lange auf Ergebnisse warten zu müssen. Mit einer Vielzahl von nicht komplizierten Tests ist so ein hoher Testumfang bei schnellem Feedback möglich. Beispielsweise lässt sich mit Bean Testing im CDI-Container der Transactional Interceptor viel weiter in der Kette testen als mit anderen Verfahren *(siehe Listing 2)*.

```
@Interceptor
@Transactional
public class TransactionalInterceptor {

  /**
  * Exceptions that should not cause the transaction to rollback according
to Java EE Documentation.
  * (http://docs.oracle.com/javaee/6/api/javax/persistence/PersistenceEx-
ception.html)
  */
  private static final Set<Class<?>> NO_ROLLBACK_EXCEPTIONS=new
HashSet<Class<?>>(Arrays.asList(
        NonUniqueResultException.class,
        NoResultException.class,
        QueryTimeoutException.class,
        LockTimeoutException.class));


  @Inject
  @PersistenceContext
  EntityManager em;

  private static final Logger LOGGER = LoggerFactory.getLogger(Transactiona
lInterceptor.class);

  private static int INTERCEPTOR_COUNTER = 0;

  @AroundInvoke
  public Object manageTransaction(InvocationContext ctx) throws Exception {

     EntityTransaction transaction = em.getTransaction();
     if (!transaction.isActive()) {
        transaction.begin();
        LOGGER.debug("Transaction started");
     }

     INTERCEPTOR_COUNTER++;
     Object result = null;
     try {
        result = ctx.proceed();

     } catch (Exception e) {
        if (isFirstInterceptor()) {
           markRollbackTransaction(e);
        }
        throw e;
     } finally {
        processTransaction();
  }

        return result;
  }

  // Restlicher Code ausgelassen.
}
```

Listing 2: Transactional Interceptor

Während beim Unit Testing jeweils eine Methode oder eine Einheit überprüft wird, kann man mit Bean Testing ganze Module und Prozesse auf ihre Funktionalität testen. Da der CDI-Container auch für den Einsatz in Memory-Datenbanken konfigurierbar ist, können Datenbank-Abfragen oder Call/Transaction-Prozesse in der Abrufkette ebenso untersucht werden wie verschiedene Testcases, die Business-Logik und das Development. Security-Tests sind mittels CDI ebenfalls sehr einfach realisierbar: Es können Tests mit unterschiedlichen Benutzern und Rollen durchgeführt werden, um so die korrekte Definition der Sicherheitsregeln zu überprüfen.

Dank simulierter Java-EE-Laufzeitumgebung sind auch Tests der Business-Umgebung möglich und machbar. Fehleranfällige Simulationen können dabei entfallen, außer bei Abhängigkeiten in externen Modulen. Grundsätzlich gilt: Was im jeweiligen Modul enthalten ist, wird auch getestet; was sich außerhalb des Moduls befindet, kommt ins Mocking. Simulationen werden hierbei mit ganz schmalen Mocks durchgeführt, das Verfahren entspricht dem, das auch unter Mockito zur Anwendung kommt *(siehe Listing 3)*.

Weil beim Bean Testing alle Prozesse unter CDI ablaufen, können die Funktionalitäten ohne große Mühe erweitert werden. Dabei kommen die gewohnten Konfigurationswerkzeuge wie „persistence.xml", „beans.xml", „JUnit" etc. zum Einsatz. CDI beachtet das Java-EE-Paradigma „Convention over Configuration": Es wird aktiviert, indem eine leere „beans.xml"-Datei im Classpath zur Verfügung gestellt wird. Darüber hinaus bietet CDI ein typsicheres Verfahren für Deployment Injection an. Mögliche Fehler

infolge nicht existierender bzw. mehrdeutiger Abhängigkeiten werden so bereits bei der Initialisierung der CDI-Implementierung und nicht erst während der Laufzeit sichtbar *(siehe Listing 4)*.

Praxiserfahrung und Verfügbarkeit

Festzuhalten ist, dass Bean Testing mit CDI kein Ersatz für Unit Testing oder Integration Testing ist, sondern ein anderes, zusätzliches Verfahren, mit dem sich das Testen von Java-EE-Anwendungen optimieren lässt. Für essenzielle Business-Logiken sind Unit-Tests nach wie vor unverzichtbar. Und um zu überprüfen, dass alles so läuft, wie es soll, müssen selbstverständlich für jedes Projekt auch weiterhin Integrationstests durchgeführt werden.

Der Vorteil am Bean-Testing-Verfahren mit Java CDI liegt vielmehr darin, dass diese Herangehensweise gewissermaßen „das Beste aus beiden Welten" in sich vereinigt: die Geschwindigkeit von Unit-Tests mit der Abdeckungsbreite von Integrationstests, und das bei minimalem Konfigurationsaufwand und unter Einsatz vertrauter Standard-Frameworks wie „JPA", „CDI", „Mockito" und „JUnit". Weil für diese Tests kein Applikationsserver erforderlich ist, lassen sie sich zudem in den üblichen Testprozess integrieren, sodass schon während des Entwicklungsprozesses laufend Tests mit der Bandbreite von Integrationstests stattfinden können.

In der Praxis hat sich das „Bean-Testing"-Verfahren bereits in realen, großen Projekten bewährt – in internen und in Kundenprojekten. Gegenwärtig setzt der Autor das Verfahren in einem großen Kundenprojekt ein mit dem Ergebnis, dass richtige Iterationstests damit durchgeführt werden, darunter auch „Blackbox"-Tests zur Überprüfung von Anfragen und Reaktionen. Bean Testing ermöglicht die Entwicklung von echten Integrationstests, die sich nicht mit dem Testen des Persistenz-Layers begnügen, sondern die Schnittstellen nach außen testen.

Fazit

Nach mehrjähriger Erfahrung mit dem Einsatz in internen und externen Projekten lässt sich feststellen: Feedback-Geschwindigkeit und Testabdeckung gewinnen durch Bean Testing mit CDI deutlich. Je größer das Projekt, desto größer sind dabei die Vorteile. Gerade bei komplexeren Projekten bereitet der modulare Aufbau bei Deployment und Tests häufig Schwierigkeiten. Zudem sind in der Praxis viele Java-EE-Applikationsserver installiert, die aufwändig zu konfigurieren sind und beim Deployment und Testen von Anwendungen viel Zeit beanspruchen; jeder Testvorgang, bei dem man auf ihren Einsatz verzichten kann, schafft da große Erleichterung.

Derzeit setzt der Autor Bean-Tests in einer Reihe größerer Kundenprojekte ein, bei denen es sich um komplexe Java-EE-Anwendungen mit mehreren Subsystemen handelt, die wiederum aus einer Reihe von Modulen bestehen. Größere Probleme sind dabei nicht aufgetreten, die meisten Komplikationen ließen sich mit CDI-Standardfeatures beheben. Auch bei kleineren und mittleren Java-EE-Projekten lässt sich das Verfahren vorteilhaft einsetzen. Für alle Anwendungsfälle gilt: Einmal aufgesetzt, ist Bean Testing bereit für den Einsatz.

Weitere Informationen

Das Framework für Bean Testing von Java-EE-Applikationen mit CDI ist auf GitHub Open Source verfügbar und dokumentiert *(siehe „https://github.com/NovaTecConsulting/BeanTest")*.

```
public class ExternalServicesMockProducer {

  private static MyExternalService externalServiceMock=Mockito.
mock(MyExternalService.class);

  @Produces
  public static MyExternalService getExternalService() {
     return  externalServiceMock;
  }

}
```

Listing 3: Beispiel für Simulation mit einem schmalen Mock

```
public class TestEJBInjection extends BaseBeanTest {

  @Test
  public void shouldInjectEJBAsCDIBean() {
     MyEJBService myService = getBean(MyEJBService.class);
     //An Entity should be persisted and you should see a message logged in
the console.
     myService.callOtherServiceAndPersistAnEntity();
     //Let's create a reference of another EJB to query the database.
  MyOtherEJBService myOtherService = getBean(MyOtherEJBService.class);

     assertThat(myOtherService.getAllEntities(), hasSize(1));

  }
}
```

Listing 4: Bean-Test

Carlos Barragan
carlos.barragan@novatec-gmbh.de

Carlos Barragan arbeitet als Consultant bei der NovaTec Consulting GmbH und beschäftigt sich seit mehr als zehn Jahren mit Enterprise-Anwendungen. An verschiedenen Projekten war er als Software-Entwickler, Berater und Architekt beteiligt. Seine Schwerpunkte liegen auf Java, Java EE und verteilten Anwendungen.

Vom proprietären Framework zum Open-Source-Projekt: Eclipse Scout

Matthias Zimmermann, BSI Business Systems Integration AG

Der Einsatz von Open-Source-Technologien entspricht heute einem akzeptierten Standard in der Industrie. Selbst entwickelte Libraries und Frameworks unter einer Open-Source-Lizenz herauszugeben, jedoch noch nicht. Der Artikel beschreibt die Erfahrung mit dem Open-Source-Anwendungs-Framework Eclipse Scout und zeigt, dass ein solcher Schritt auch für mittelständische Unternehmen in der Software-Industrie möglich ist und große Vorteile bringen kann.

Zu Beginn des Computerzeitalters war es selbstverständlich, dass Software kostenfrei und uneingeschränkt modifiziert und geteilt werden konnte. Durch die Aufteilung des Geschäfts auf Hard- und Software-Lieferanten hat sich diese Praxis vorübergehend deutlich verändert. Kommerzielle, proprietäre Software wurde zur Norm. In den letzten zehn Jahren hat sich das Blatt jedoch wieder zugunsten des Open-Source-Ansatzes gewendet.

Gründe dafür gibt es viele. Aus Kundensicht erhöht der Einsatz etablierter Standards und bewährter Open-Source-Komponenten die Flexibilität, um besser auf neue Anforderungen reagieren zu können. Gleichzeitig verringern sich bei diesem Ansatz die Abhängigkeiten von einzelnen Lieferanten. Den Software-Herstellern hilft die Entwicklung von Open-Source-Software, innovative Lösungen schneller auf den Markt zu bringen, Kosten zu senken und ihre Attraktivität bei jungen und talentierten Entwicklern zu erhöhen. Als jüngeres Beispiel für diesen Trend soll hier die TODO Group [1] erwähnt werden, die zum Ziel hat, Erfahrungen bei der Pflege von Open-Source-Software zu teilen.

Die Ausgangslage

Das Anwendungs-Framework Scout wurde bei der Business Systems Integration AG (BSI) im Jahr 1999 erstmals eingesetzt. Das Ziel bestand darin, die Entwicklungseffizienz und Code-Qualität bei der Entwicklung des Vorläufers der heutigen BSI-CRM-Lösung zu verbessern. In der ursprünglichen Form war Scout auf eine proprietäre Windows-Umgebung zugeschnitten, bevor im Jahr 2001 mit der Version 2 der erste Technologie-Wechsel auf einen Java- und XML-basierten Ansatz gemacht wurde. Im Jahr 2007 erfolgte mit der Scout-Version 3 der Umstieg auf die Basis von Java- und Eclipse Rich Client Platform (RCP).

Aufgrund der Begeisterung für Open Source und anerkennender Kommentare von Eclipse-Experten zum neuen (aber immer noch internen) Framework und Tooling, entstand um Jahr 2009 bei einer Handvoll BSI-Mitarbeiter der Wunsch, das Scout-Framework [2] bei der Eclipse Foundation [3] als Open-Source-Projekt unter der Eclipse Public Licence (EPL) zu veröffentlichen. Wie viel mehr als dieser initiale Enthusiasmus für diesen Schritt erforderlich war, kam erst in den darauf folgenden Monaten heraus.

Interne Überzeugungsarbeit

Da Scout die Basis fast aller kommerziellen Produkte bei BSI darstellt, war von Anfang an klar, dass für diesen Schritt die Unterstützung der Geschäftsleitung und des Verwaltungsrats erforderlich war. Mit der Naivität

begeisterter Techniker war man überzeugt, schon bald mit der Eclipse Foundation über das Scout Proposal sprechen zu können.

Die ersten Bemerkungen und Fragen zu diesem Unterfangen waren allerdings ausgesprochen kritisch. Es war allen klar, dass das Vorhaben, ein internes Framework unter eine Open-Source-Lizenz zu stellen, zuerst einmal Aufwände verursacht. Die wirklich ernsthaften Fragen drehten sich jedoch um Themen wie Haftung und die zusätzlichen Risiken, denen man durch diesen Schritt ausgesetzt sein würde.

Auf der anderen Seite existierte aber auch das Risiko, dass in der Zukunft ein anderes, ähnliches Framework im Open-Source-Bereich auftauchen und populär werden könnte. Dies hätte BSI in die unangenehme Position gebracht, den Kunden erklären zu müssen, weshalb die Software auf Scout und nicht auf eben diesem neuen Framework basiert. Mit diesem Zwischenstand wurde das Anliegen vom Verwaltungsrat in einer ersten Runde noch nicht gut geheißen.

Die wichtigste Auflage des Verwaltungsrats bestand darin, die juristischen Risiken mithilfe von externen Experten detailliert abklären zu lassen. Die daraus resultierenden Fragen wurden anschließend in mehreren Telefonkonferenzen zwischen der Geschäftsleitung und der Eclipse Foundation diskutiert. An dem Punkt angelangt, an dem die Risiken deutlich kleiner erschienen als die mit Open Source verbundenen Hoffnungen, wurden die Erkenntnisse zusammen mit einem Marketing- und einem Business-Plan ein weiteres Mal dem Verwaltungsrat vorgestellt. Dieser gab darauf grünes Licht unter der Auflage, dass alle bisherigen Produkt-Kunden keine Einwände gegen die Open-Source-Strategie für das Scout-Framework haben dürften.

Nach vielen Kundenbesuchen waren schließlich auch aus Kundensicht alle offenen Fragen und Sorgen adressiert und zufriedenstellend beantwortet. Der Prozess der notwendigen internen und externen Diskussionen von der Idee bis zur Freigabe durch die Geschäftsleitung hat mit acht Monaten deutlich länger gedauert, als man sich ursprünglich vorgestellt hatte.

Vom internen Framework zum Eclipse-Projekt

Nach den BSI-internen Vorbereitungs- und Entscheidungsphasen war man nun soweit, bei der Eclipse Foundation durch die offiziellen Phase „Project Proposal" und „Creation Review" laufen zu können. Das Project-Proposal-Dokument war nach einigen wenigen Iterationen fertig, sodass Scout offiziell in der Proposal-Phase war.

Daraufhin folgten eine öffentliche Diskussion über Scope und dessen Hintergründe sowie die Prüfung der Verfügbarkeit des gewünschten Projekt-Namens. Da bei der Apache Foundation seit längerer Zeit bereits eine Library mit dem Namen „Scout" existierte, gab es eine kurze Phase der Unsicherheit. Glücklicherweise konnten sich die beiden Foundations darauf einigen, für das Anwendungs-Framework den Namen „Eclipse Scout" verwenden zu dürfen. Die Bestätigung des Projektnamens „Eclipse Scout" war gleichzeitig die Freigabe, mit den Code Renamings und dem Erstellen des initialen Marketing-Materials inklusive Logo beginnen zu können.

Nach zwei weiteren Monaten war das Project Creation Review erfolgreich abgeschlossen. Damit verbunden hat die Eclipse Foundation auch die notwendige Infrastruktur wie Repository, Bugtracker, Forum, Newsgroup, Wiki und das Hosting der Projekt-Webseite zur Verfügung gestellt.

Wie bei jedem bereits existierenden Projekt musste auch bei Scout entschieden werden, welche Komponenten man als Open Source bei Eclipse veröffentlichen wollte und welche weiterhin intern bleiben würden. Diese Entscheidung fiel klar zugunsten einer möglichst umfassenden Veröffentlichung. Somit lagen die Gründe für das Zurückbehalten von Komponenten in erster Linie bei Abhängigkeiten zu externen Komponenten mit Lizenzen wie beispielsweise LGPL, die nicht zur Eclipse Public Licence (EPL) kompatibel waren. Diese Aufteilung von Scout in den Open-Source- und den BSI-internen Teil hatte auch einige zusätzliche Refactorings zur Folge. Auch die interne Build- und Test-Infrastruktur war entsprechend anzupassen.

Bevor auch nur eine Zeile Code bei Eclipse eingecheckt werden durfte, musste die gesamte Scout-Codebasis in einem nächsten Schritt durch den umfangreichen Intellectual-Property-Prozess (IP) der Eclipse Foundation laufen. Dieser stellt sicher, dass bei Eclipse gehosteter Code lizenztechnisch einwandfrei ist und bestehende Eigentumsrechte nicht verletzt werden. Konkret wird in diesem IP-Prozess der gesamte Projekt-Code maschinell mit allen bekannte Quellen abgeglichen und verifiziert, dass dieser auch tatsächlich vom Projekt selbst und nicht aus undeklarierten Quellen stammt.

Der IP-Prozess greift konsequenterweise bis auf die Stufe einzelner Code-Zeilen. Da nicht nur der Projekt-Code selbst, sondern auch alle notwendigen Abhängigkeiten gemäß den Richtlinien der Eclipse Foundation „IP Clean" sein müssen, hat dieser Prozess durchaus das Potenzial, größere Refactorings zu erzwingen. Glücklicherweise waren hier nur vereinzelte und kleinere Refactorings erforderlich. Trotzdem haben die Project-Proposal- und Creation-Phasen vom Antrag bis zur Freigabe zum Einchecken bei Eclipse noch einmal vier Monate gedauert. Somit umfasste der gesamte Open-Sourcing-Prozess ein ganzes Jahr.

Der Einfluss der Community auf unsere Entwicklungsprozesse

Die Auswirkungen der Umstellung interner Software auf Open Source hinsichtlich der Entwicklungsprozesse innerhalb der BSI zeigten sich insbesondere durch die Teilnahme am jährlichen Eclipse Release Train. Projekte, die an diesem Release Train teilnehmen, müssen unter anderem durch mehr als sieben Milestones und vier Release-Kandidaten ihre Abhängigkeiten zu anderen teilnehmenden Eclipse-Projekten aktualisieren. Dank dieser Vorgabe und der strikten Milestone-Planung schafft die Eclipse Community jährlich die taggenaue Lieferung von fast siebzig Projekten mit insgesamt mehr als sechzig Millionen Codezeilen.

Die Teilnahme von Scout am Eclipse Release Train hat in der BSI dazu geführt, dass die Scout-Entwickler vermehrt automatisierte Tests geschrieben haben und die externen Abhängigkeiten von Scout zur Eclipse-Plattform und zu neuen Java-Versionen früher und aktiver aktualisiert und getestet wurden. Im Vergleich zur Praxis vor der Umstellung auf Open Source brachte dies zwar einen erhöhten Aufwand im Scout-Team für das regelmäßige Building und Testing von Scout. Als Mehrwert für BSI ergibt sich jedoch unmittelbar eine deutlich höhere Stabilität des Anwendungs-Frameworks. Dies kommt wieder jedem Kunden-Projekt zugute und erhöht auch die Produktivität der Entwickler in den kommerziellen Projekten.

Mit der wachsenden Community stieg das Bewusstsein dafür, wie wichtig eine frühzeitige und transparente Kommunikation bei der Scout Roadmap und bei Framework-Änderungen ist. Für eine Organisation wie BSI, die bis dahin auch substanzielle

Änderungen schnell und unkompliziert umsetzen konnte, entspricht das einem Kulturwechsel, der nicht unterschätzt werden darf. Da bei der wachsenden BSI auch die interne Kommunikation immer wichtiger wird, hilft diese neue Sichtweise auch beim eigenen Wachstum. Das ist ein wichtiger Punkt, denn schließlich sind bei der BSI heute etwa doppelt so viele Entwickler angestellt wie zum Zeitpunkt der Entscheidung, Scout unter der EPL zu veröffentlichen. Der große Anteil dieser Entwickler arbeitet täglich mit Eclipse Scout.

Die Kosten

Die Umstellung auf Open Source von Eclipse Scout hat mit den initialen Abklärungen, Code-Anpassungen und Eclipse-Prozessen ein gutes Personenjahr gekostet. Die jährlichen Aufwände für Marketing, Dokumentation und die Pflege des Forums erfordern im jährlichen Durchschnitt ein weiteres Personenjahr. Der Hauptanteil der Marketing-Aktivitäten entfällt dabei auf das Vorbereiten und Halten von Vorträgen sowie die Betreuung der Standplätze bei Konferenzen. Der Aufwand für die Erweiterung der bestehenden Dokumentation und die Pflege des Forums halten sich in etwa die Waage.

Beim Erweitern und Pflegen der Dokumentation wurde auch gelernt, dass mehr Dokumentation nicht immer besser ist. Damit der Wert der Dokumentation erhalten bleibt, muss diese mit den Änderungen im Framework Schritt halten. Dies bedeutet, dass die Dokumentation so modular und minimal wie möglich sein sollte. Dadurch können Teile der Dokumentation für verschiedene Zwecke verwendet werden und Anpassungen müssen bei Framework-Änderungen häufig nur an einer Stelle gemacht werden. Denn auch für die Dokumentation gelten die bekannten Entwickler-Akronyme „don't repeat yourself" (DRY) und „you ain't gonna need it" (YAGNI).

Auch beim Source Code und bei der Build-Infrastruktur fallen gegenüber einer rein internen Software zusätzliche Aufwände an. Einerseits nimmt die Produkt-Komplexität durch die Aufteilung der Software in öffentliche und interne Komponenten zu, andererseits muss das Framework jetzt auch zusätzlich für Mac- und Linux-Umgebungen, die für bestehende Kunden nicht relevant waren, eingerichtet und getestet werden.

Die Vorteile

Die Umstellung auf Open Source von Eclipse Scout hat BSI bisher handfeste Vorteile gebracht und viele diesbezügliche Erwartungen und Hoffnungen erfüllt. Dieser Schritt hat insbesondere in den letzten beiden Jahren dazu beigetragen, den Umsatz im Consulting, Training und Support zu steigern. Auch die Attraktivität von BSI als Arbeitgeber konnte davon profitieren und hat in Einzelfällen dazu geführt, dass sich qualifizierte Interessenten überhaupt erst gemeldet haben.

In den letzten Jahren es ist auch gelungen, erste Kunden für Scout-Trainings, Consulting und Support zu gewinnen. Als häufigstes Szenario für den kommerziellen Einsatz von Scout außerhalb der BSI hat sich der Einsatz in Modernisierungsprogrammen für Legacy-Applikationen herausgestellt. In diesem Bereich sind einfache Anwendungsentwicklung, gute Wartbarkeit und hohe Langlebigkeit wichtige Anforderungen.

Auch beim Verkauf kommerzieller Produkte wie BSI CRM, einer Customer-Relationship-Management-Lösung, hat sich die Umstellung von Scout auf Open Source als ausgesprochen hilfreich erwiesen. Dank der Tatsache, dass die kommerziellen Produkte direkt auf Eclipse Scout basieren, kann man Kunden mit eigenen Entwicklungsabteilungen beim Verkauf des BSI CRM auch anbieten, dass sie in den Integrationsprojekten aktiv mitwirken. In der Vergangenheit hat man mehr als einmal die Beobachtung gemacht, dass die interne IT beim Kunden diesen Punkt als Stärke des Angebots bewertet hat. Eclipse Scout ist keine Blackbox – jeder Kunde hat jederzeit Einblick in das Fundament der Produkte von BSI.

Scout als Open Source Framework hat auch auf die tägliche Arbeit mit Scout intern positive Konsequenzen. Beispielsweise wird Scout von den Projekt-Teams in Kunden-Projekten als deutlich stabiler als in der Vergangenheit erlebt. Auch Upgrades auf neuere Scout-Versionen fallen deutlich leichter, was in erster Linie einer besseren Disziplin der Scout-Entwickler mit dem Umgang von API-Changes zu verdanken ist. Diese beiden Verbesserungen sind nicht zuletzt der Teilnahme am Eclipse Release Train zu verdanken. Dieser Prozess hält regelmäßig dazu an, Framework-Änderungen zu dokumentieren sowie früher und häufiger zu testen.

Fazit

Seit der Umstellung von Scout auf Open Source konnte BSI von den erwarteten und auch von einigen unerwarteten Vorteilen, etwa beim Verkauf von kommerziellen Produkten, profitieren. Die Vergangenheit hat aber auch gezeigt, dass die für den Aufbau des Consulting- und Supportgeschäftes notwendigen Zeiträume deutlich länger als zuerst gehofft sind.

Dank der Tatsache, dass das BSI-Produktgeschäft hauptsächlich auf dem Scout-Framework basiert, haben Unterhalt und Weiterentwicklung von Scout auch intern eine hohe Priorität. Wäre man beim „Open Sourcen" von Scout von Beginn an zu Umsatzsteigerungen verpflichtet gewesen, hätte man dieses Vorhaben nach den ersten ein bis zwei Jahren wahrscheinlich wieder abgebrochen.

Für das weitere Wachstum der BSI hilft nicht nur die gewonnene Stabilität des Eclipse-Scout-Frameworks; auch die deutlich umfangreichere Dokumentation bringt neue Mitarbeiter dazu, schneller und effizienter produktiv zu werden. Außerdem decken sich die Erwartungen der Scout Community bezüglich einer zeitnahen und transparenten Kommunikation zur Weiterentwicklung des Frameworks gut mit dem zunehmenden internen Kommunikationsbedarf einer wachsenden Firma.

Zusammengefasst lässt sich trotz des nicht unerheblichen Aufwands eine positive Bilanz ziehen. Die Entscheidung, Scout bei der Eclipse Foundation als Open-Source-Projekt zu veröffentlichen, war richtig und hilft dabei, die Zukunft des Unternehmens abzusichern.

Literatur

[1] TODO Group: http://todogroup.org
[2] Eclipse Scout: http://www.eclipse.org/scout
[3] Eclipse Foundation: https://eclipse.org/org/foundation

Matthias Zimmermann
matthias.zimmermann@bsiag.com

Matthias Zimmermann ist als Co-Lead Eclipse Scout für das Open-Source-Engagement bei der BSI Business Systems Integration AG verantwortlich und arbeitet in Kunden-Projekten als Projektleiter.

Sofortkopien – minutenschnell in Selbstbedienung

Karsten Stöhr, Delphix

Ob die Entwicklung eine neue Umgebung benötigt, ob ein QA-Test mit dem Gesamt-Datenbestand ausgeführt werden soll, ob die Revision nochmal den Stand vor sechs Monaten prüfen will – egal für welchen Zweck, alle Projekte müssen auf die Einrichtung einer entsprechenden Umgebung warten.

Während auf breiter Front die Server-Virtualisierung für eine schnelle und platzsparende Bereitstellung der Plattform genutzt wird, verschlingt die Bereitstellung der Daten immer noch viel Zeit. Oder es leidet die Qualität des Projekts, weil zur Vereinfachung eine andere Kopie wiederverwendet oder mit einem kleineren Test-Datenbestand gearbeitet wird. Mithilfe der Delphix Agile Data Platform können jedem Projekt und jedem Team für seine Zwecke passende virtuelle Daten zur Verfügung gestellt werden – in Minutenschnelle, ohne dabei den mehrfachen Platz der Daten zu verbrauchen.

Sisyphos-Arbeit: Daten kopieren

Datenbank-Administratoren sichern nicht nur die Verfügbarkeit der Daten, auf die aktuell von den Unternehmens-Anwendungen zugegriffen wird. Sie müssen auch den künftigen Bedarf im Blick haben sowie Strategien und Lösungen für Datenwachstum, Leistung der Infrastruktur und Einführung neuer Technologien entwickeln. Doch anstatt sich um strategische Aufgaben kümmern zu können, müssen Datenbank-Administratoren in den meisten Organisationen einen großen Teil ihrer Zeit (bis zu 50 Prozent) für Tätigkeiten wie das Kopieren und Bewegen von Daten sowie die Überprüfung ihrer Richtigkeit opfern. Fakt ist, wir kopieren Daten – immer und immer wieder. Viele Datenbank-Administratoren versuchen, die sich ständig wiederholenden, manuellen Tätigkeiten mithilfe selbst geschriebener Skripte zu automatisieren. Trotzdem müssen sie aufgrund der Komplexität von Anwendungsprojekten immer wieder individuelle Anfragen nach Aktualisierung, Wiederherstellung und Integration der Daten manuell bearbeiten.

Ein Ausweg: Daten-Virtualisierung

Die Bereitstellung von Servern war in der Vergangenheit ähnlich zeit- und kostenintensiv wie heute die Bereitstellung von Daten häufig noch ist. Eine deutliche Erleichterung in der Bereitstellung von Servern brachte der Wechsel von physischer Hardware zu virtuellen Maschinen. Ein sogenannter „Hypervisor" lässt einen einzelnen physischen Server durch eine intelligente gemeinsame Nutzung der vorhandenen Hardware-Ressourcen wie CPU und Speicher wie mehrere separate Server aussehen. Das Bereitstellen einer virtuellen Maschine erfordert nur Minuten im Gegensatz zum mehrtägigen Aufwand, um einen neuen physischen Server einzurichten.

Auf ähnliche Weise lässt sich auch die Bereitstellung von Daten beschleunigen. Wenn man eine neue Kopie einer Datenbank er-

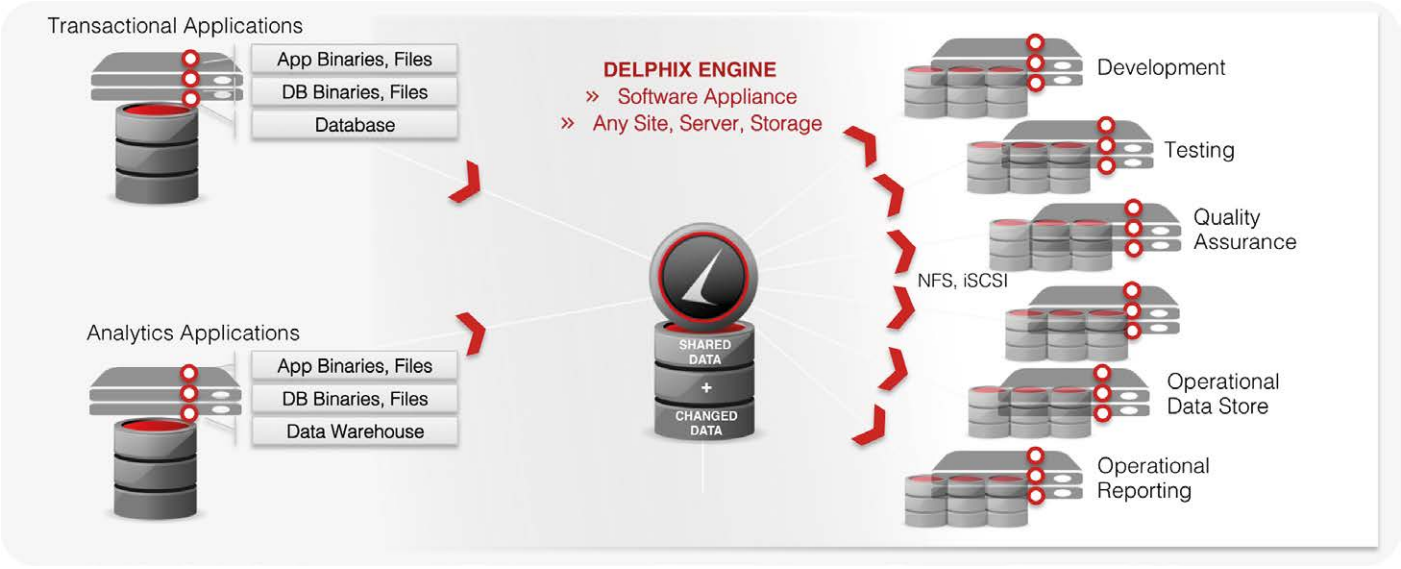


Abbildung 1: Flexible, eigenständige Daten-Umgebungen durch Daten-Virtualisierung

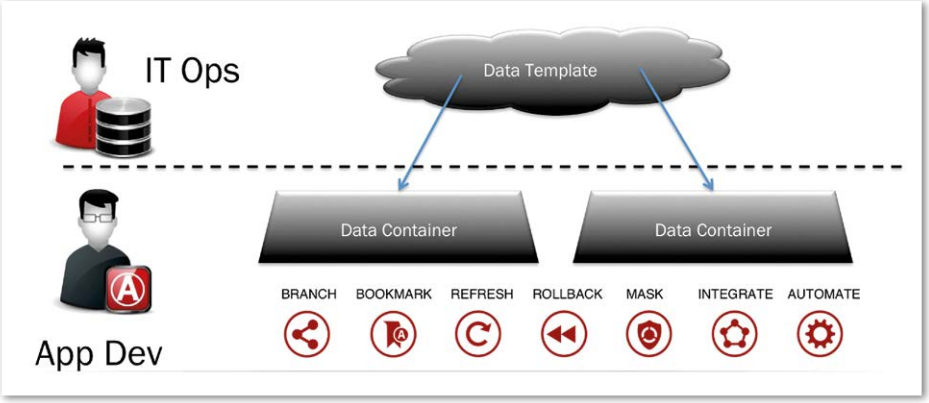


Abbildung 2: Selbstbedienung und Trennung der Aufgaben von IT-Administration und Anwendern

stellt, etwa für Entwicklung oder Test, dann sind zunächst alle Daten der Kopie identisch zur ursprünglichen Datenbank. Erst im Laufe der Bearbeitung werden gewisse Daten geändert oder neue erzeugt. Aber der Großteil der Daten bleibt identisch. Dies ermöglicht – geeignete Technologie vorausgesetzt –, dass dieselben Datenblöcke für mehrere Kopien gemeinsam verwendet werden können (neudeutsch „sharing"), ähnlich wie ein Hypervisor die Hardware-Ressourcen für mehrere virtuelle Server verwendet.

Nur die von der jeweiligen Instanz individuell geänderten Datenblöcke müssen getrennt gespeichert werden („copy-on-write"). Die Kombination aus geteilten und individuell geänderten Datenblöcken lässt mehrere Kopien derselben Datenbank jeweils als eine vollständige separate Instanz erscheinen. Dabei verschlingt jede weitere Instanz nur einen Bruchteil des ursprünglichen Plattenplatzes, nämlich die Größe der individuell geänderten Daten – die bei der Instanziierung einer neuen Kopie ja noch gleich Null ist. Auf diese Weise dauert die Erstellung einer neuen Instanz selbst bei sehr großen Datenbanken nur noch Minuten statt diverser Stunden bis Tage.

Zu den Applikationsdaten gehören nicht nur in Datenbanken gespeicherte Daten, sondern auch diverse Dateien. Eine geeignete Plattform zur Daten-Virtualisierung muss daher auch die entsprechenden Dateien auf ähnliche Weise bereitstellen können.

Nächster Schritt: Selbstbedienung

Anwender, Entwickler und Tester stellen ihre Anfragen nach Daten-Umgebungen an die Datenmanagement-Abteilung. Dort werden die Anfragen geprüft, priorisiert und ausgeführt. Mittels der geschilderten Daten-Virtualisierung können Administratoren diese Anfragen schneller und zuverlässiger beantworten, doch sind Datenbank-Administratoren häufig stark belastet und Anfragen mit höherer Priorität schieben andere Anfragen nach hinten.

Angenommen, ein Entwickler möchte eine neue Funktion mit großem Datenbestand testen, um die Performanz zu messen und eventuell zu verbessern. Dazu muss er denselben Test mehrere Male mit verschiedenen Parametern ausführen; vor jedem neuen Testlauf müssen die Daten wieder auf den originären Stand zurückgesetzt werden. Der Testlauf selbst dauert vielleicht weniger als eine Stunde, aber für das Zurücksetzen der Daten muss der Anwender jedes Mal wieder eine Anfrage an die Administration stellen und auf deren Ausführung warten.

Wenn für jede Datenmanagement-Operation ein Anwender sich erst an die IT-Administration wenden muss, ist das ineffektiv und kostet zu viel Zeit. Es wäre wünschenswert, wenn einfache Operationen auf vordefinierten Umgebungen, wie das Zurückset-

Abbildung 3: Grafische Oberfläche für die Administratoren zur automatisierten Bereitstellung virtueller Daten-Umgebungen

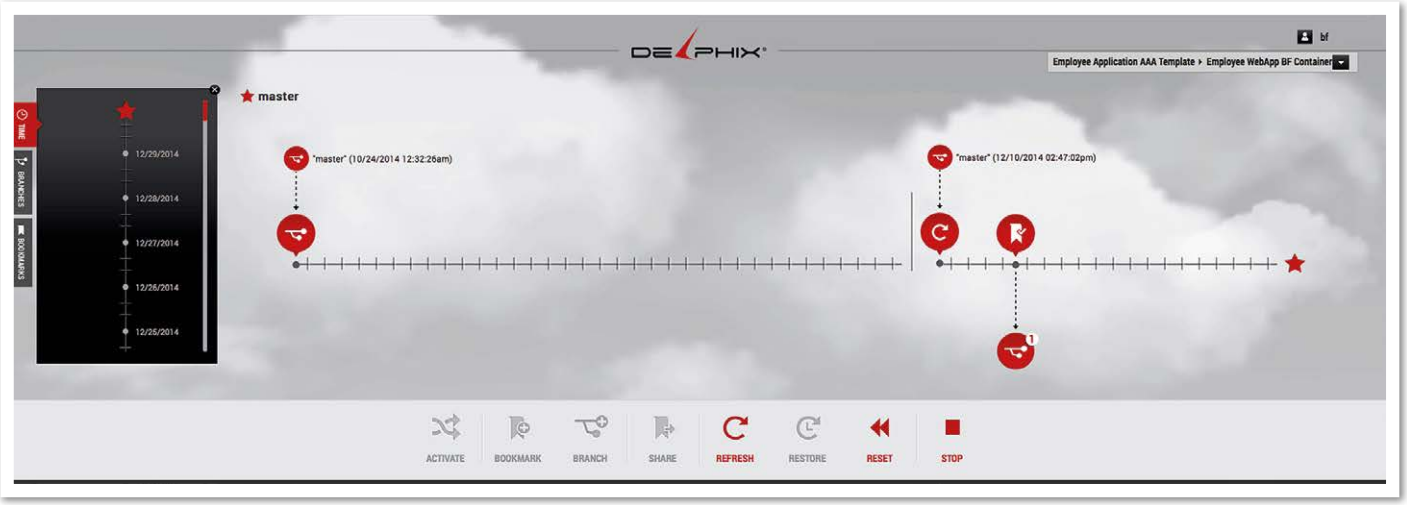


Abbildung 4: Grafische Oberfläche zur Selbstbedienung wiederkehrender Aufgaben durch Anwender

zen („Reset") oder Aktualisieren („Refresh") der Daten, vom Anwender selbst ausgelöst werden könnten. Dies führt zu einer Aufgabenteilung zwischen der IT-Infrastruktur und dem Management der Daten aus Anwendersicht. Die IT-Administration behält volle Kontrolle über die Ressourcen und deren Verteilung und kann eine definierte Datenumgebung (ein Template) einem Anwender zuordnen. Die Anwender haben dann die Möglichkeit, in Selbstbedienung die zugewiesenen Umgebungen zu nutzen, zu aktualisieren, zurückzusetzen oder mit anderen Anwendern zu teilen – ohne die physischen Ressourcen zu kennen, die ihre Daten bereitstellen *(siehe Abbildung 2)*.

Delphix Agile Data Platform

Die Agile Data Platform von Delphix ist eine Plattform zur Daten-Virtualisierung, die alle zuvor geschilderten Anforderungen erfüllt. Dahinter steht eine Applikation mit eigenständiger operativer Umgebung, die als virtuelle Appliance geliefert wird. Delphix verwendet Standard-APIs, um sich mit physischen Quell-Datenbanken zu verbinden, ihre Datenblöcke zu kopieren und eine „goldene Kopie" zu erzeugen, die sich zwanzig und mehr virtuelle Umgebungen teilen können. Dabei nutzt die Delphix Engine intelligente Filterung und Komprimierung, um die Kopie einer Quell-Datenbank auf bis zu 50 Prozent ihrer originalen Größe zu reduzieren. Die Kopie der Quell-Datenbank wird zusammen mit allen inkrementellen Updates gespeichert als sogenannte „dSource".

Nach dem initialen Laden hält die Delphix Engine die Synchronisation mit der Quelle aufrecht, basierend auf konfigurierbaren Strategien, zum Beispiel einmal täglich oder innerhalb von Sekunden nach der jeweils letzten Transaktion. Einmal verbunden, wird ein zeitlicher Ablauf-Timeflow der Quell-Datenbank geführt. Diese fortlaufende Aufzeichnung der Datei- und Log-Änderungen wird über ebenfalls konfigurierbare Regeln aufbewahrt, etwa „bewahre die letzten zwei Monate auf".

Von jedem beliebigen Zeitpunkt innerhalb des Aufbewahrungsfensters kann augenblicklich eine virtuelle Datenbank bereitgestellt werden. Diese wird vom selben Speicherplatz des „dSource"-Timeflow beliefert („shared storage"); sie verbraucht also keinen zusätzlichen Plattenplatz *(siehe Abbildung 3)*.

Virtuelle Daten

Mehrere VDBs können von beliebigen verschiedenen Zeitpunkten innerhalb des Timeflow sekundengenau erzeugt werden. Eine VDB kann also ein „Live"-Abbild des jetzt aktuellen Stands der Daten enthalten, eine andere dagegen einen historischen Stand von Bedeutung, etwa den Stand einen Tag vor Entdeckung eines Fehlers, eine Woche vor einem System-Upgrade oder die Daten von vor einem Jahr auf Anfrage eines Compliance-Verfahrens.

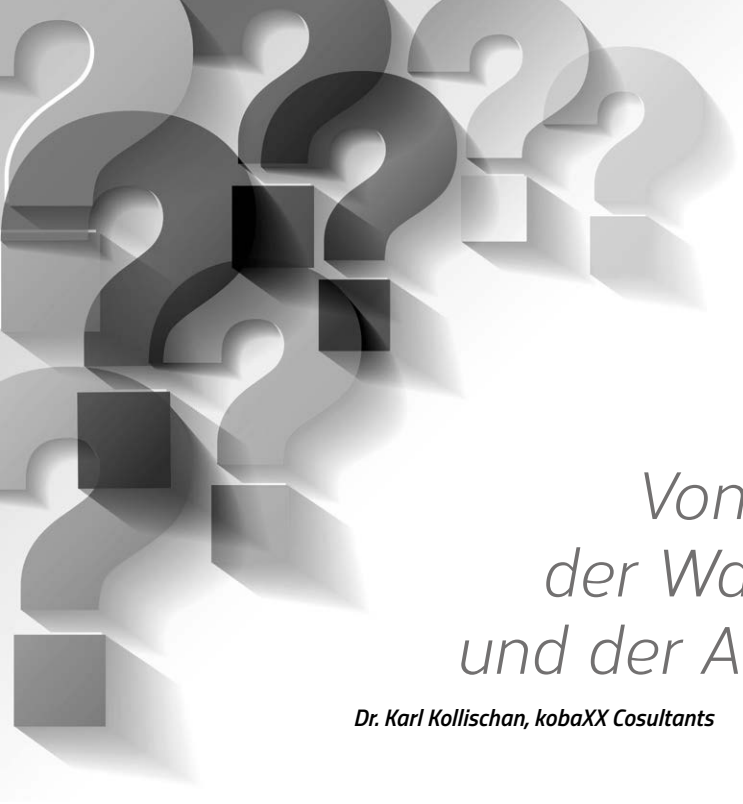
Einmal erzeugt, ist jede VDB eine eigenständige und unabhängige Datenbank mit vollständigen Lese- und Schreib-Zugriffen. Änderungen an der VDB durch Benutzer oder Applikationen werden in neue, komprimierte Blöcke geschrieben. Von einer VDB können wiederum neue VDBs erzeugt werden; die Daten innerhalb einer VDB können von der dSource oder der übergeordneten VDB neu geladen werden („Refresh").

Minutenschnell in Selbstbedienung

Delphix bietet eine einfach zu bedienende Web-Oberfläche, die es allen Anwendern aus Entwicklung, Qualitätskontrolle, Analyse etc. ermöglicht, aus den ihnen zugewiesenen Ressourcen ihre eigenen virtuellen Datenumgebungen zu erzeugen und aus Datensicht zu managen – ohne immer erst eine Anfrage an die IT-Administration stellen zu müssen *(siehe Abbildung 4)*.

Karsten Stöhr
karsten.stoehr@delphix.com

Karsten Stöhr hat mehr als 25 Jahre Erfahrung als Solutions Consultant bei Software-Unternehmen wie Oracle, BEA, Sun Microsystems und Tibco. Die letzten acht Jahre hat er sich auf Daten-Integration, -Replikation und Hochverfügbarkeit spezialisiert. Die Evaluierung der Delphix Agile Data Plattform hat ihn so sehr überzeugt, dass er im Sommer 2014 dem Unternehmen Delphix beigetreten ist, um Anwendern im zentraleuropäischen Raum den Nutzen virtueller Daten zu zeigen und bei deren Einführung zu unterstützen.



Alles klar? Von wegen!

Von kleinen Zahlen, der Wahrnehmung von Risiken und der Angst vor Verlusten

Dr. Karl Kollischan, kobaXX Cosultants

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

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

Das Gesetz der kleinen Zahlen

Eine Studie über die Häufigkeit von Nierenkrebs in 3.141 Countys der Vereinigten Staaten brachte folgendes Ergebnis: „Die Landkreise mit der niedrigsten Nierenkrebshäufigkeit liegen überwiegend in ländlichen, dünn besiedelten und traditionell republikanischen Bundesstaaten im Mittleren Westen, Süden und Westen."

Wenn wir diesen Satz lesen, steigt die Aktivität in unserem Gehirn deutlich an, was sich zum Beispiel daran zeigt, dass sich die Pupillen weiten. System 2 (das bewusste Denken) formuliert Hypothesen, um dieses Ergebnis zu erklären. Und natürlich ist auch unserer System 1 (Denkprozesse, die automatisch und mühelos ablaufen) nicht untätig – die Operationen von System 2 basieren auf den Inhalten, die aus dem assoziativen Gedächtnis abgerufen werden. Vermutlich wurde die Hypothese, dass dieses Ergebnis der Politik der Republikaner zu verdanken sei, sofort wieder verworfen.

Naheliegend und verlockend ist allerdings diese Schlussfolgerung: Die Schadstoffbelastung in ländlichen Gebieten ist geringer, es gibt keine so starke Luft- und Wasserverschmutzung wie in Großstädten, die Menschen haben Zugang zu unbelasteten und frischen Lebensmitteln. Klingt sehr plausibel, oder? Die gleiche Studie förderte aber auch folgendes Ergebnis zutage: „Die Landkreise mit der höchsten Nierenkrebshäufigkeit liegen überwiegend in ländlichen, dünn besiedelten und traditionell republikanischen Bundesstaaten im Mittleren Westen, Süden und Westen." Würde man nur das lesen, könnte man leicht folgern, dass dieses Ergebnis auf Defizite des ländlichen Lebensstils zurückzuführen sei, wie schlechtere medizinische Versorgung, fettreiche Ernährung, mehr Alkohol- und Tabakkonsum.

Was ist die Erklärung für diese beiden scheinbar widersprüchlichen Ergebnisse? Richtig – der entscheidende Faktor ist hier nicht, dass die Landkreise in ländlich geprägten oder überwiegend republikanischen Regionen liegen. Entscheidend ist allein die Tatsache, dass ländliche Landkreise eine geringere Bevölkerungszahl haben. Die Auflösung des scheinbaren Widerspruchs haut vermutlich nicht vom Hocker – doch wie schon bei Steve, dem vermeintlichen Bibliothekar (siehe letzte Ausgabe), zeigt sich hier wieder die schwierige Beziehung zwischen unserem Intellekt und der Statistik.

Große Stichproben liefern verlässlichere Ergebnisse als kleine Stichproben. Umgekehrt führen kleine Stichproben häufiger zu extremen Ergebnissen als große. Die starken Schwankungen der Nierenkrebshäufigkeit in dünn besiedelten Gegenden sind schlicht und ergreifend auf Zufallsereignisse zurückzuführen. Dies ist das Gesetz der gro-

ßen Zahlen – und auch wenn sie dieses Gesetz nicht explizit kennen, ist dessen Inhalt vielen Menschen durchaus bewusst.

Trotzdem kommen sie hier zu falschen Schlussfolgerungen – selbst Statistik-Experten sind vor dieser Denkfalle nicht gefeit und vertrauen bei der Wahl von Stichprobengrößen eher ihrer Intuition als ihrem Wissen. System 1 versteht sich hervorragend auf eine bestimmte Form des Denkens, es erkennt automatisch und mühelos kausale Verknüpfungen zwischen Ereignissen – manchmal sogar dann, wenn diese Beziehung gar nicht vorhanden ist. Beim Lesen über die Countys mit den niedrigen Krebsraten nimmt man sofort an, dass es irgendeinen Grund gibt, der diese Countys von den übrigen unterscheidet. Das Merkmal „dünn besiedelt" war in obigem Beispiel nicht sofort als relevant zu erkennen. System 1 ist recht unbeholfen, wenn es mit bloßen statistischen Fakten konfrontiert ist, die die Wahrscheinlichkeit von Ergebnissen verändern, diese aber nicht verursachen.

Kahnemann und Tversky bezeichneten dies augenzwinkernd als den Glauben an das „Gesetz der kleinen Zahlen", das besagt, dass das Gesetz der großen Zahlen auch für kleine Zahlen gilt.

Die Verfügbarkeits-Heuristik

Das Jahr 2014 war ein schwarzes Jahr für die Luftfahrt, insgesamt hat es mehr Todesfälle durch Unfälle gegeben als in den Jahren zuvor. Auch Meldungen über Terroranschläge sind in den Medien allgegenwärtig. Vielen Menschen erscheint das Risiko, durch einen Flugzeugabsturz oder einen Terroranschlag ums Leben zu kommen, als besonders hoch, wenn über ein entsprechendes Ereignis berichtet wird. Und selbst, wenn man Statistiken kennt und weiß, dass etwa in Deutschland die Zahl von Todesopfern im Straßenverkehr wesentlich höher ist als die von Terroropfern, verspürt man nach solchen Ereignissen eher Angst auf großen öffentlichen Plätzen, in Flughäfen oder in Bahnhöfen als im eigenen Auto auf dem Weg zur Arbeit. Dies führt teilweise zu sehr kontraproduktiven Verhaltensweisen. So haben etwa nach den Terroranschlägen am 11. September 2001 viele Menschen in den USA aus Angst vor dem Fliegen lieber das eigene Auto benutzt, was einen signifikanten Anstieg der Verkehrstoten in dem Jahr danach zur Folge hatte *(siehe „http://www.mpg.de/6344003/Terrorismus_Verkehrstote")*.

Unsere Risiko-Wahrnehmung ist verzerrt. Wie in anderen Urteils-Heuristiken beantwortet System 1 eine einfachere Fragestellung *(siehe Teil 2)*. Die Frage: „Wie groß ist eine Kategorie oder die Häufigkeit eines Ereignisses?" wird ersetzt durch die Frage: „Wie leicht lassen sich Beispielfälle aus dem Gedächtnis abrufen?"

Wissenschaftler führten zahlreiche Befragungen über die öffentliche Wahrnehmung von Risiken durch. Den Teilnehmern wurden jeweils Paare von Todesursachen gezeigt; zu jedem Paar sollten sie angeben, was ihrer Ansicht nach die häufigere Todesursache sei, und das Verhältnis der Häufigkeiten schätzen. Die Antworten wurden mit aktuellen Gesundheitsstatistiken verglichen. Hier ein paar Ergebnisse:

- Schlaganfälle verursachen fast doppelt so viele Todesfälle wie alle Unfälle zusammengenommen. 80 Prozent der Teilnehmer schätzen einen Unfalltod als wahrscheinlicher ein.
- Asthma fordert zwanzigmal mehr Todesopfer als Tornados, nach Einschätzung der Probanden gingen mehr Todesfälle auf das Konto von Tornados.
- Tod durch Blitzschlag ist 52-mal häufiger als durch Lebensmittelvergiftung, wurde aber trotzdem als weniger wahrscheinlich eingeschätzt.
- Diabetes führt etwa viermal so häufig zum Tod wie Unfälle, nach Ansicht der Probanden lag das Verhältnis bei „1:300".

Diese Ergebnisse zeigen sehr deutlich, wie stark die Medienpräsenz von Ereignissen die Einschätzung ihrer Häufigkeit beeinflusst. Ein hervorstechendes Ereignis, das die Aufmerksamkeit auf sich zieht, lässt sich leicht aus dem Gedächtnis abrufen. Kleiner Test: Was fällt Ihnen als Erstes ein, wenn Sie an Bill Clinton denken?

Aber auch persönliche Erfahrungen, Bilder und anschauliche Beispiele lassen sich leichter abrufen als abstrakte Sachverhalte. Das ist der Grund, aus dem man heute in der Software-Entwicklung Anforderungen in Form von User Stories beschreibt, ohne zum Beispiel Personas zu verwenden. Man erreicht damit eine bessere Anschaulichkeit und durch die Leichtigkeit des Abrufs eine höhere Präsenz der Anforderung.

In einer der bekanntesten Studien zur Verfügbarkeit wurden Eheleute (getrennt voneinander) befragt, wie groß ihr persönlicher Beitrag beim Aufräumen ihrer gemeinsamen Wohnung sei. Wie zu erwarten, lag der Gesamtbeitrag einiges über 100 Prozent. Der eigene Anteil ist als persönliche Erfahrung besser verfügbar und wird höher eingeschätzt als der Fremdanteil. Sich dieser kognitiven Verzerrung bewusst zu sein, ist der Harmonie in Beziehungen durchaus zuträglich. Auch für die Arbeit in Teams kann das Wissen um diesen Effekt eine konstruktive und harmonische Zusammenarbeit fördern, denn die gleiche kognitive Verzerrung trägt auch zu der häufigen Beobachtung bei, dass Team-Mitglieder den Eindruck haben, einen höheren Beitrag geleistet zu haben als andere oder dass ihr Anteil nicht entsprechend gewürdigt wird.

Verlust-Aversion

Unsere Aversion gegen Verluste wird mit folgendem Gedankenexperiment gut dargestellt. Fall1: Angenommen, es gäbe die Wahl zwischen den beiden Optionen, 900 Euro sicher zu erhalten oder mit einer 90-prozen-

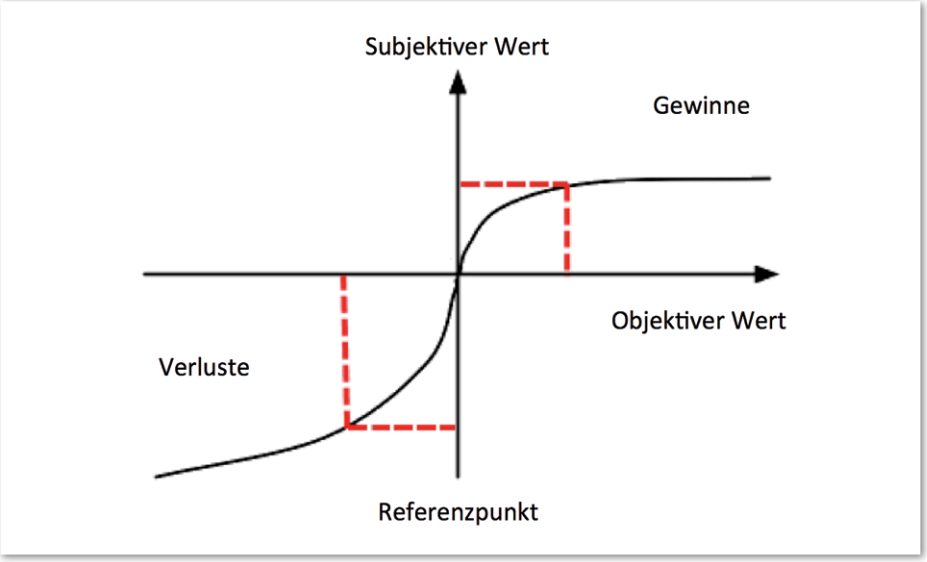


Abbildung 1: Verlust-Aversion

tigen Chance 1000 Euro zu gewinnen. Die meisten Menschen wären wohl risikoscheu und würden sich für die sichere Variante entscheiden. Der subjektive Wert eines Gewinns von 900 Euro ist ohne Zweifel größer als die 90-prozentige Chance auf einen Gewinn von 1000 Euro.

Fall 2: Hier gibt es die Wahl zwischen den beiden Optionen, 900 Euro sicher zu verlieren oder mit einer 90-prozentigen Chance 1000 Euro zu verlieren. Hier würden sich wohl die meisten Menschen für das Risiko entscheiden. Die Erklärung für die risikoreiche Wahl hier ist spiegelbildlich zur Erklärung der Risikovermeidung im ersten Fall: Der (negative) Wert eines Verlusts von 900 Euro ist größer als der (negative) Wert der 90-prozentigen Chance, 1000 Euro zu verlieren. Die Verlust-Aversion besagt, dass Verluste subjektiv schwerer wiegen als Gewinne.

Abbildung 1 zeigt, wie sich der psychologische Wert von Gewinnen und Verlusten zum objektiven Wert verhält. Rechts vom neutralen Referenzpunkt, der den Status quo repräsentiert, werden objektive Gewinne dargestellt, links davon die Verluste. Viele Optionen, die uns begegnen, sind gemischt, es gibt sowohl ein Verlust-Risiko als auch eine Gewinn-Chance. Wir müssen entscheiden, ob wir das Risiko eingehen oder nicht.

Zum Beispiel folgende Wette auf einen Münzwurf: Wenn die Münze Zahl zeigt, verliert man 100 Euro. Wenn die Münze Kopf zeigt, gewinnt man 150 Euro. Obwohl der Erwartungswert des Nutzens positiv ist (- 0,5 * 100 + 0,5 * 150), würden die meisten Menschen diese Wette wohl ablehnen. Die rationale Entscheidung, diese Wette abzulehnen, wird von System 2 getroffen. Doch wesentlich für diese Entscheidung sind die emotionalen Reaktionen, die von System 1 geliefert werden. Natürlich ist die Verlust-Aversion abhängig vom zu erwartenden Gewinn. Würde der Gewinn in obigem Beispiel 1000 Euro betragen, gingen die meisten Menschen wohl auf die Wette ein.

Zahlreiche Experimente haben ergeben, dass die Verlust-Aversionsrate in etwa zwischen „1,5" und „2,5" liegt. Dies ist selbstverständlich ein Mittelwert, manche Menschen sind wesentlich risikofreudiger als andere. Die Verlust-Aversionsrate ist darüber hinaus abhängig vom möglichen Verlust. Tendenziell steigt die Rate, je höher der Einsatz ist, und geht gar gegen unendlich, wenn der mögliche Verlust die Existenz bedroht. Insgesamt gilt jedoch, dass die Angst, etwas zu verlieren, Menschen stärker motiviert als die Aussicht, den gleichen Wert zu gewinnen.

Die Verlust-Aversion hat Einfluss auf Entscheidungen, ob ein aus dem Ruder gelaufenes Projekt fortgeführt oder beendet werden soll. Wer kennt nicht den Schmerz, den man dabei empfindet, wenn man den Aufwand betrachtet, der schon betrieben worden ist, und nun alles für die Katz gewesen sein soll. Rational finden wir viele Argumente dafür, das Projekt fortzuführen – und oft wird gerade mit dem schon geleisteten Aufwand argumentiert. Rational betrachtet sollte einzig und allein ausschlagegebend sein, wie die Mittel ab jetzt am nutzenbringendsten eingesetzt werden sollen.

Woher kommt diese Verlust-Aversion? Eine mögliche Erklärung liefert die Evolutionsbiologie. Zahlreiche Versuche und Aufnahmen im Hirnscanner haben gezeigt, dass schlechte oder bedrohliche Nachrichten schneller wahrgenommen werden als gute und heitere. Bilder von wütenden Gesichtern, die ja eine potenzielle Bedrohung darstellen, lösen eine stärkere Reaktion aus und werden schneller verarbeitet als Bilder von zufriedenen Gesichtern. Selbst wenn das Bild den Probanden nur so kurz gezeigt wurde, dass sie es nicht bewusst wahrgenommen haben, wurden diese Reaktionen ausgelöst.

Es gibt einen superschnellen Verarbeitungskanal direkt ins Gefühlszentrum, die Amygdala, der die langsamere, bewusste Verarbeitung via Sehrinde umgeht. Unser Gehirn, und auch das Gehirn anderer Säugetiere, scheint darauf ausgelegt zu sein, schlechten Nachrichten den Vorrang zu geben. Indem die Zeitspanne zur Wahrnehmung eines Fressfeindes um ein paar hundertstel Sekunden verkürzt wird, werden die Überlebenschancen erhöht.

Dieser Mechanismus führt dazu, dass bei drohenden Verlusten mehr Kräfte und Ressourcen aktiviert werden als bei erhofften Gewinnen. Dies zeigt sich zum Beispiel bei den Revierkämpfen von territorialen Tieren: Der Revierhalter gewinnt fast immer den Kampf. Der Erhalt des Status quo wird mit aller Kraft verteidigt. Auch in unserer Lebenswirklichkeit zeigt sich die Verlust-Aversion in zahlreichen Situationen. So werden bei Verhandlungen Zugeständnisse als schwerwiegender empfunden als Gewinne. Die Verlust-Aversion erzeugt hier eine Asymmetrie, die eine Einigung behindert.

Auch bei Veränderungsprozessen in Organisationen ist die Verlust-Aversion ein erschwerender Faktor. Zum einen führt die Verlust-Aversion zu einer Präferenz für den Erhalt des Status quo beziehungsweise zu einer Abneigung, die Komfortzone zu verlassen. Aber auch wenn die angestrebten Veränderungen für viele Betroffene positiv sind und von einer Mehrheit unterstützt werden, gibt es in der Regel auch eine Minderheit, die negativ betroffen ist. Und diese Minderheit setzt sich häufig überproportional stark durch, weil sie aktiver wird und sich mit aller Macht zur Wehr setzt. Das angestrebte Ergebnis wird dadurch verzerrt und verwässert. Man denke nur mal an die Steuer- oder Gesundheitsreform.

Die Prä-Mortem-Analyse

Gerade bei neuen Projekten führen neben anderen kognitiven Verzerrungen die Verlust-Aversion (das Ergebnis des Projekts wird bereits als Referenzpunkt wahrgenommen) und die Verfügbarkeitsheuristik zu einer verzerrten Risiko-Wahrnehmung. Auch wenn wir um diese Verzerrungen wissen, basiert unser bewusstes Entscheidungsvermögen immer auf den Eingangsdaten, die uns System 1 liefert. Von daher können wir diese Verzerrungen nicht vermeiden. Jedoch lässt sich ihre Wirkung in manchen Kontexten abschwächen.

Eine dieser Methoden bei der Risikoanalyse von Projekten ist die sogenannte „Prä-Mortem-Analyse". Wenn ein Projekt gescheitert ist, findet normalerweise danach – also „Post Mortem" – eine Analyse der Ursachen für das Scheitern statt. Bei der Prä-Mortem-Analyse stellt man diese Situation spielerisch am Anfang her. Dazu stellt man einer Gruppe von Personen, die alle mit dem Projekt vertraut sind, folgende Aufgabe: „Stellen Sie sich vor, wir befinden uns ein Jahr in der Zukunft und wir haben das Projekt komplett gegen die Wand gefahren. Schreiben Sie jeder für sich die Geschichte des Scheiterns (ca. 5 bis 10 Minuten)."

In den Geschichten zeigen sich die tatsächlich vorhandenen Risiken, die zu einer Katastrophe führen könnten. Die Prä-Mortem-Analyse hat neben einer besseren Risiko-Wahrnehmung den Vorteil, dass sie es erlaubt, Zweifel und Bedenken zu äußern. Dies ist oft zu Beginn eines Projekts – in der „euphorischen Phase" – gar nicht so einfach, da man nicht als Bedenkenträger dastehen möchte. Gerade wenn man Teil des Teams ist oder eine wichtige Rolle wie Projektleiter, Product Owner oder Architekt einnimmt, möchte man nicht demotivierend wirken. Manchmal bestehen auch Ängste – berechtigt oder nicht –, dass Zweifel als mangelnde Loyalität gegenüber dem Team, dem Projekt oder dem Unternehmen ausgelegt werden.

Die Prä-Mortem-Analyse schafft ein spielerisches Umfeld, in dem es einfach ist, Zweifel zu äußern. Sie führt rasch zu Ergebnissen und

macht auch noch Spaß. Prinzipiell kann sie bei allen wichtigen anstehenden Entscheidungen angewendet werden. Die Prä-Mortem-Analyse ist natürlich kein Allheilmittel und schützt nicht vollständig vor unliebsamen Überraschungen, doch hilft sie, den schädlichen Einfluss von kognitiven Verzerrungen zu minimieren.

Fazit

Die in dieser Artikelserie vorgestellte Liste der kognitiven Verzerrungen ist bei Weitem nicht vollständig. An ausgewählten Beispielen wurde gezeigt, in welchem Maße die unbewussten Operationen von System 1 unser Denken und unsere Entscheidungen beeinflussen und dass wir nicht die rationalen Individuen sind, für die wir uns oft halten.
Die Frage ist, ob uns dieses Wissen dabei hilft, zu einem besseren Denken und zu Entscheidungen zu gelangen, die einer analytischen Überprüfung standhalten würden. Die Antwort lautet eher „nein", Daniel Kahnemann formuliert es in seinem Buch sehr treffend: „Ich habe dieses Buch geschrieben und denke immer noch genauso wie vorher".

Man kann im Alltag nicht alles rational durchdenken, und selbst wenn System 1 abgeschaltet werden könnte und wir zu einer Art Super Sheldon Cooper mutieren würden – die Denk- und Entscheidungsprozesse wären viel zu anstrengend und langwierig; wahrscheinlich würden wir wie Sheldon unserer Umwelt ziemlich auf die Nerven gehen.

Auch wenn es an der einen oder anderen Stelle erschreckend sein mag – nehmen wir es lieber gelassen. Unser Denkapparat ist wie das Auge, das zwar manchmal irrt, im Großen und Ganzen aber hervorragend funktioniert.

Dr. Karl Kollischan
karl.kollischan@kobaxx.com

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

APM – Agiles Projektmanagement

gelesen von Daniel Grycman

Der Autor und seine Co-Autoren versprechen auf dem Buchtitel die erfolgreiche Steuerung anspruchsvoller Software-Projekte beim Einsatz der APM-Methodik.

Bereits im Jahr 2007 ist das erste Buch zu agilem Projektmanagement (APM) erschienen; es etablierte sich bald zu einem Standardwerk im Bereich „agiler Methoden" im deutschsprachigen Raum. In diesem Buch hatte Uwe Vigenschow bereits als Co-Autor mitgewirkt und nun ist unter seiner Federführung eine neue Ausgabe entstanden. Es ist hierbei zu beachten, dass das Buch komplett neu geschrieben wurde und dies unter Berücksichtigung der gewonnenen Erfahrung im Umgang mit agilen Prinzipen geschehen ist. Ebenso ist anzumerken, dass auch die Co-Autoren Andrea Grass, Alexandra Augstin und Michael Hoffmann tief in der Materie verwurzelt sind.

Der Autor führt den interessierten Leser über fünf Teile mit insgesamt 23 Kapiteln in die agile Methodik ein. Im ersten Teil gibt Vigenschow eine Einführung in die Konzepte von Agilität und des Projektmanagement-Frameworks APM. Der zweite Teil beschäftigt sich mit dem Aufsetzen eines agilen Projekts. Im nächsten Teil widmet sich der Autor dem Einfluss der Agilität auf die Software-Architektur. Mit der iterativen Herangehensweise von agiler Software-Entwicklung beschäftigt sich der vierte Teil. Ebenso legt Vigenschow Wert darauf, dass sich der Leser mit dem Rollenverständnis der Beteiligten auseinandersetzt. Der letzte Teil stellt die Anwendung der Methodik im Rahmen von Großprojekten dar und beschreibt zusätzlich die Einführung in Unternehmen selbst.

Man bemerkt beim Lesen des Buchs, dass Uwe Vigenschow über eine langjährige Erfahrung als Trainer verfügt. Dies spiegelt sich sowohl in dem didaktischen Aufbau der einzelnen Teile und Kapitel als auch in der sprachlichen Ausgestaltung des gesamten Werks wider. Durch den sehr detaillierten Index eignet sich das Buch auch als Nachschlagewerk. Das beiliegende Poster veranschaulicht dem Leser die Grundlagen.

Fazit:

Das Buch „APM – Agiles Projektmanagement" ist jedem zu empfehlen, der sich in die Methodiken der agilen Softwareentwicklung einarbeiten möchte. Zudem bietet sich dieses Werk auch allen Praktikern als gutes Nachschlagewerk an.

Titel:	APM – Agiles Projektmanagement
Autor:	Uwe Vigenschow, unter Mitarbeit von Andrea Grass, Alexandra Augstin und Michael Hoffmann
Auflage:	1. Auflage 2015
Verlag:	dpunkt.verlag, Heidelberg
Umfang:	451 Seiten
Preis:	44,90 Euro, eBook (downloadbar) nicht im Preis enthalten
ISBN:	978-3-86490-211-6

Daniel Grycman
daniel.grycman@bilsteingroup.com

Automatisierte Integrationstests mit Citrus

Christoph Deppisch, ConSol Software GmbH*

Software-Schnittstellen sind fester Bestandteil eines Enterprise-Java-Projekts. Dies kann ein einfacher REST-Service zwischen Front- und Backend sein oder eine komplexe Schnittstellen-Landschaft in einem Enterprise-Service-Bus.

Die moderne Software-Entwicklung arbeitet mit entkoppelten Komponenten, die Nachrichten synchron oder asynchron über diverse Kommunikationswege miteinander austauschen. Eine frühzeitige Integration dieser Schnittstellen ist sehr wichtig für die spätere Zusammenschaltung der einzelnen Software-Komponenten im Produktivbetrieb. Die hohe Komplexität bei der Simulation des Nachrichtenaustauschs erschwert jedoch den automatisierten Test. Der Artikel zeigt eine schnelle und einfache Lösung eines voll automatisierten Integrationstests mit dem Open-Source-Testframework Citrus.

In einer Testumgebung steht die Software der Schnittstellen-Partner in der Regel nicht zur Verfügung und der Tester ist gezwungen das Verhalten entsprechend zu simulieren. Dieses Vorhaben wird deutlich erschwert, wenn die zu testende Software mit mehreren Schnittstellen (synchron, asynchron, http, TCP/IP, FTP, JMS etc.) und unterschiedlichen Nachrichtenformaten (SOAP, XML, JSON, CSV etc.) gleichzeitig arbeitet. Aufgrund fehlender Tool-Unterstützung wird das Zusammenspiel der Schnittstellen und deren Integration erst sehr spät im Software-Entwicklungsprozess in einer End-to-End-Testumgebung manuell durchgeführt. Dabei wäre eine entwicklungsbegleitende Integration aller verwendeten Schnittstellen für eine frühzeitige Problemerkennung und einen reibungslosen Übergang in den Produktivbetrieb sehr wertvoll.

Im Unit-Test werden die Services ausgiebig getestet, jedoch liegt der Fokus auf der Business-Logik, die isoliert von externen Einflüssen durchlaufen wird. Die Schnittstellen-Aufrufe an externe Partner werden über ein Mocking-Framework komplett ersetzt und der eigentliche Aufruf über den Transportweg entfällt. Damit sind die spezifischen Schnittstellen-Einstellungen, die ebenfalls einen großen Einfluss auf das Verhalten einer Software haben, im Unit-Test nicht geprüft.

Nehmen wir das Beispiel Java Message Service (JMS). In diesem Zusammenhang sind Consumer Einstellungen, Connection-Pools, Timeouts und Transaction Handling am Message Broker für das Verhalten der gesamten Software prägend. Es ist zwingend notwendig, diese Einstellungen über einen realen Nachrichtenaustausch in einem Integrationstest automatisiert zu testen. Zusätzlich muss das Einhalten des Schnittstellen-Vertrags und das passende Design der Schnittstelle entwicklungsbegleitend in einem automatisierten Test abgeprüft werden.

Beispiel eines Test-Szenarios

Abbildung 1 zeigt eine Incident-Manager-Anwendung sowie vier unterschiedliche Schnittstellen zu externen Systemen. In einem typischen Anwendungsfall werden mehrere dieser Schnittstellen in einer logischen Abfolge verwendet. Sagen wir, der Kunde öffnet ein neues Ticket über das Kundenportal (1). Das beschriebene Problem des Kunden wird zunächst an eine Netzwerkanalyse-Software weitergeleitet (2). Je nach Analyse-Ergebnis auf der Netzwerk-Leitung muss nun eventuell ein Techniker beauftragt werden, um vor Ort beim Kunden eine Problembehebung durchzuführen (3). Während des gesamten Vorgangs wird der Kunde per SMS Gateway fortlaufend über den Fortschritt und den endgültigen Abschluss des Tickets informiert (5).

Dieses einfache Beispiel zeigt nun schon vier verschiedene Software-Schnittstellen, die in einem Anwendungsfall zusammenspielen. Dieser Anwendungsfall soll nun automatisiert getestet und alle angeschlossenen Schnittstellen-Partner (Kundenportal, Netzwerk, FieldForce und SMS Gateway) simuliert werden.

Die Test-Umgebung

Um diesen Anwendungsfall voll automatisiert mit allen verwendeten Schnittstellen zu testen, muss zunächst ein Software-Build-Lifecycle aufgebaut werden. Der automatisierte Aufbau der Test-Umgebung ist zwar nicht zentraler Gegenstand dieses Artikels, jedoch bildet er die Basis für einen erfolgreichen automatisierten Integrationstest. Ziel des Lifecycles ist es, den aktuellen Entwicklungsstand der Software in einer Test-Umgebung zur Verfügung zu stellen. Hier bieten sich Tools wie Maven oder Gradle an, um den Java Code automatisiert zu kompilieren, alle Unit-Tests auszuführen und die letztendlichen Software-Pakete und Artefakte („.jar", „.war") zu erstellen.

Die Artefakte müssen im nächsten Schritt automatisiert in einem Java Application Server eingerichtet werden. In einer Test-Umgebung für Continuous Integration kann das auch ein Embedded Application Server (Jetty, Tomcat, Glassfish) sein, der sehr leichtgewichtig und schnell zu starten ist. In Zusammenarbeit mit einem Continuous-Build-Tool wie Jenkins lässt sich damit ein zentraler Software-Build-Lifecycle aufbauen, der mit jeder Code-Änderung automatisiert abläuft und den aktuellen Software-Stand in die Test-Umgebung bringt. *Abbildung 2* zeigt einen typischen

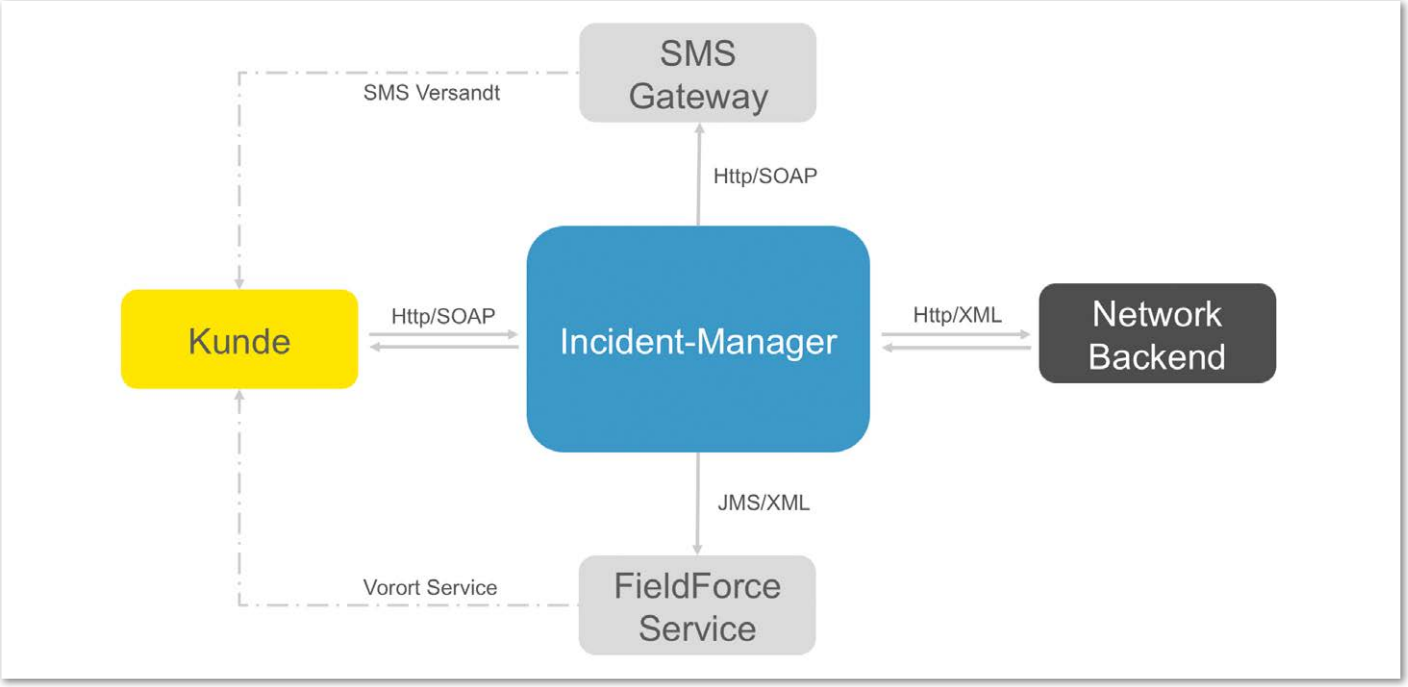


Abbildung 1: Beispiel-Anwendung Incident Manager

Build-Lifecylce bis zu den Citrus-Integrationstests.

Automatisierte Integrationstests

Sobald der aktuelle Entwicklungsstand einer Software in der Test-Umgebung zur Verfügung steht ist es Zeit, die Schnittstellen automatisiert zu testen. Das Open-Source-Framework Citrus ist in der Lage. als Client oder Server mit unterschiedlichsten Schnittstellen zu interagieren. *Abbildung 3* zeigt, wie man alle beteiligten Schnittstellen-Partner in einem Test simuliert.

Citrus nimmt also die Rollen der angebundenen Schnittstellen-Partner ein und bietet einsatzbereite Komponenten für das Senden und Empfangen von Nachrichten sowie fertige Server-Komponenten, zum Beispiel für das Starten eines SOAP-WebService-Endpoints. So kann das Framework alle benötigten Schnittstellen aus dem Beispiel client- und serverseitig simulieren.

Im Citrus-Test werden die Schnittstellen über den realen Transportweg aufgerufen und Nachrichten im finalen Datenformat (wie SOAP, JSON) ausgetauscht. Das bedeutet, dass die Daten tatsächlich über die Leitung gehen, zum Beispiel bei JMS über eine reale TCP/IP-Verbindung zum JMS Message Broker. Bei SOAP-WebServices arbeitet Citrus über eine reale Client-Server-http-Kommunikation. Dabei kommen alle Einstellungen am Transportweg zum Einsatz und werden dementsprechend auch getestet.

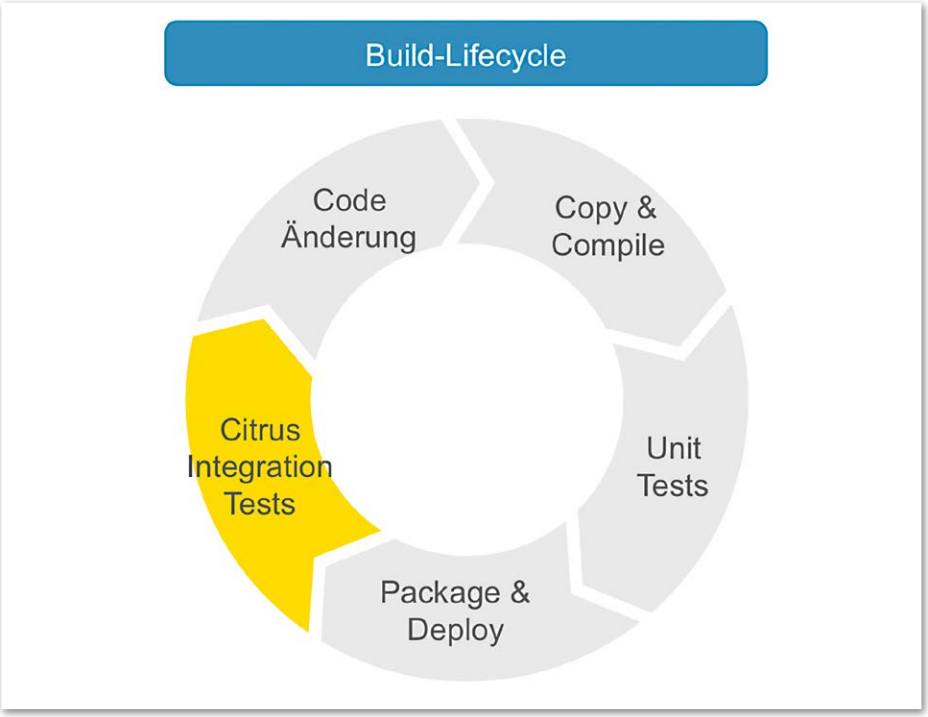


Abbildung 2: Software-Build-Lifecycle

Konfiguration der Citrus-Komponenten

Citrus arbeitet eng mit dem Spring-Framework zusammen. Die Konfiguration der Komponenten geschieht daher auch über Spring-Beans in einem Spring Application Context. *Listing 1* zeigt beispielhaft einige Citrus-Komponenten aus dem Beispiel, die über eine XML-Konfiguration im Spring-Framework definiert sind. Im Detail ist dies eine SOAP-Client-Komponente zum Versenden von SOAP-Request-Nachrichten oder eine Http-Server-Komponente, die Request-Nachrichten als Http-Server entgegen nimmt.

Die beispielhaft dargestellten JMS-Komponenten tauschen die Nachrichten über eine JMS-Queue aus. Dazu ist noch eine JMS-ConnectionFactory erforderlich, die als normale Spring-Bean in den Application Context hinzugefügt wird.

Die Citrus-Komponenten sind direkt als Spring-Beans im Application Context konfiguriert und beinhalten alle notwendigen Einstellungen, wie Server-Ports, JMS-Queue-Namen, Timeouts und Request-URIs. Damit sind alle erforderlichen Komponenten definiert, um die Schnittstellen aus dem Beispiel aufrufen zu können.

Citrus wird im Test automatisch einen SOAP-Client für den Aufruf über das Kundenportal erstellen sowie einen voll einsatzfähigen Http-Server starten, der als Netzwerk-Analysesoftware Request-Nachrichten entgegennimmt. Über JMS wird Citrus die FieldForce-Schnittstelle im Beispiel simulieren und Nachrichten über den JMS-Message-Broker konsumieren sowie entsprechende Antworten verschicken.

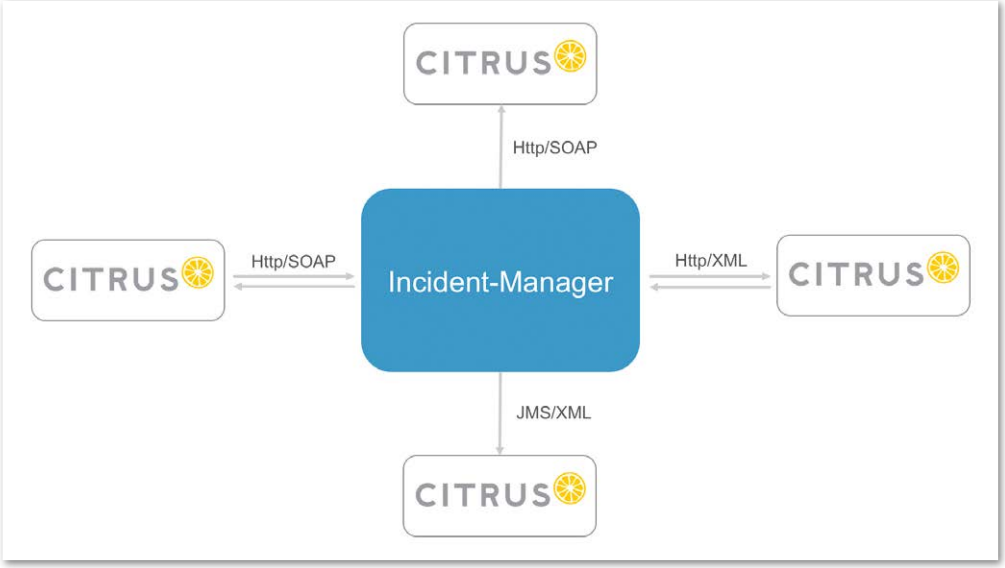


Abbildung 3: Die Beispiel-Anwendung im Test

Citrus-Tests schreiben

Der Citrus-Test ist eine gewöhnliche JUnit- oder TestNG-Testklasse, wie sie aus dem Unit-Test bekannt ist. Dadurch lassen sich die Citrus-Tests sehr einfach mit Maven oder Gradle ausführen. Die Test-Ergebnisse sind identisch zu den Unit-Tests standardisiert und können einfach in einer Java-IDE wie Eclipse oder einem Build-Tool wie Jenkins angezeigt und ausgewertet werden.

Der Testfall in *Listing 2* leitet von der Citrus-Testklasse „TestNGCitrusTestBuilder" ab. Dadurch werden der Spring Application Context automatisch beim Start des Tests geladen und alle zuvor definierten Einstellungen und Komponenten erzeugt. Die Testklasse nutzt eine normale „TestNG @Test"-Annotation. Dadurch ist der Test als normaler Unit-Test ausführbar.

In der Testmethode selbst findet nun die Logik des Tests ihren Platz. Es wird eine Abfolge von „send()"- und „receive()"-Testaktionen verwendet, so wie es durch den aktuell getesteten Anwendungsfall vorgeschrieben ist. Eine Java-DSL baut mit vorgefertigten Methoden die Logik eines Integrationstests zusammen. Der Umfang der bereitgestellten Citrus-Testaktionen ist beachtlich. Der Tester ist unter anderem in der Lage, Datenbank-Abfragen auszuführen, XSL-Transformationen durchzuführen und ANT-Skripte aufzurufen.

Im Beispiel kommen ausschließlich die „send()"- und „receive()"-Methoden für den Nachrichtenaustausch vor. Jede dieser Aktionen verschickt beziehungsweise empfängt eine Nachricht über den jeweiligen Transportweg. Die zuvor definierten Citrus-Komponenten sind per Spring Dependency Injection direkt im Test verfügbar.

Die Komponenten sind für das Senden und Empfangen der Nachrichten zuständig. Sie wissen in welches Nachrichtenformat (etwa SOAP) eine Nachricht umgewandelt werden muss und wie die Nachricht verschickt werden soll (etwa Http oder JMS). Der Nachrichten-Inhalt wird in diesem Test als externe Datei eingelesen. Die Arbeit mit Zeichenketten oder Java-Objekten (wie JAXB-Objekte) ist möglich.

Spezifische Header-Werte wie die SOAP Action werden auch direkt in der Testaktion gesetzt. Jede „receive()"-Aktion im Test erwartet eine Nachricht. Citrus wartet und verzögert die Testausführung hier so lange, bis eine Nachricht über den angegebenen Transportweg angekommen ist. Für jede eingehende Nachricht wird dann ein spezieller Validierungsmechanismus gestartet. Sollte eine Nachricht ausbleiben, schlägt der Test mit einer entsprechenden Timeout-Meldung fehl.

Nachrichten-Inhalte validieren

Sobald Citrus eine Nachricht empfängt, erfolgt automatisch eine Schema-Validierung. Dafür definiert der Tester das XSD-Schema

```
<!-- Incident Manager SOAP Web Service Client -->
<citrus-ws:client id="incidentManagerClient"
          request-url="http://localhost:18001/incident/v1"/>

<!-- Network backend Http server -->
<citrus-http:server id="networkBackendServer"
          auto-start="true"
          port="18002"
          timeout="10000"/>

<!-- FieldForce JMS endpoints -->
<citrus-jms:endpoint id="fieldForceOrderEndpoint"
      destination-name="JMS.Citrus.v1.FieldForceOrder"/>

<citrus-jms:endpoint id="fieldForceNotificationEndpoint"
      destination-name="JMS.Citrus.v1.FieldForceNotification"/>

<!-- JMS connection factory -->
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
   <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
```

Listing 1: Citrus-Komponenten

```
@Test
public class IncidentManager_Test extends TestNGCitrusTestBuilder {

   @Autowired
   @Qualifier("incidentManagerClient")
   private WebServiceClient incidentManagerClient;

   @Autowired
   @Qualifier("networkBackendServer")
   private HttpServer networkBackendServer;

   private Resource incidentRequest = new ClassPathResource("templates/IncidentRequest.xml");
   private Resource analyseRequest = new ClassPathResource("templates/AnalyseRequest.xml");
   private Resource analyseResponse = new ClassPathResource("templates/AnalyseResponse.xml");
   private Resource incidentResponse = new ClassPathResource("templates/IncidentResponse.xml");

   @CitrusTest(name = "IncidentManager_Ok_Test")
   public void testIncidentManagerOk () {
       variable("ticketId", UUID.randomUUID().toString());

       send(incidentManagerClient)
           .fork(true)
           .payload(incidentRequest)
           .header(SoapMessageHeaders.SOAP_ACTION, "/IncidentManager/openIncident");

       receive(networkBackendServer)
           .payload(analyseRequest);

       send(networkBackendServer)
           .payload(analyseResponse)
           .header("Content-Type", "application/xml");

       receive(incidentManagerClient)
           .payload(incidentResponse);
   }
}
```

Listing 2: Citrus-Testfall

```
<net:AnalyseIncident xmlns:net="http://www.citrusframework.org/sche-
ma/samples/NetworkService/v1">
   <net:incident>
       <net:ticketId>${ticketId}</net:ticketId>
       <net:description>Something went wrong!</net:description></
net:incident>
   <net:network>
       <net:lineId>@isNumber()@</net:lineId>
       <net:type>SOFTWARE</net:type>
       <net:connection>@ignore@</net:connection>
   </net:network>
</net:AnalyseIncident>
```

Listing 3: Nachrichten-Inhalt mit Citrus-Variablen und -Funktionen

oder die WSDL in der Citrus-Konfiguration. Zusätzlich kann der Tester ein erwartetes Nachrichten-Template angeben, um die Semantik der Nachricht zu überprüfen. Dies kann wie im Beispiel der Inhalt einer externen Datei sein. Dabei hat Citrus sehr mächtige Validatoren für XML- und JSON-Datenformate.

Jedes Nachrichten-Element wird Schritt für Schritt mit dem erwarteten Template verglichen. XML-Namespaces, Attribute, CDATA-Sections, JSON-Arrays und JSON-Objekte sind dabei kein Problem. Nicht vorhersehbare Elemente (wie Zeitstempel) können bei der Validierung einfach deklarativ ignoriert werden. Andere Teile der Nachricht (wie generierte IDs) lassen sich per XPath für spätere Test-Aktionen im Test speichern. Sollte ein Element nicht vorhanden sein oder nicht der Erwartung des Testers entsprechen, wird der Testfall mit einem Validierungsfehler abgebrochen.

Variablen, Funktionen, Validation Matcher

Die Testdaten im Citrus-Test lassen sich mit dynamischen Werten und mächtigen Validatoren robust gestalten. Der Citrus-Test soll wiederholbar sein. Dafür sind teilweise dynamische IDs und fachliche Werte in den Testdaten erforderlich. Dafür gibt es in Citrus Variablen, Funktionen und Validation Matcher, die man überall im Test einbauen und auch direkt in die Nachrichteninhalte einfügen kann. *Listing 3* zeigt das Beispiel einer Nachricht mit den fachlichen Schlüsseln und Werten für den Test. Dabei werden einige Werte mit einem Citrus-Variablen-Ausdruck besetzt („${ticketId}“) und über „ValidationMatcher“ dynamisch überprüft („@isNumber()@“).

Darüber hinaus können einzelne Elemente in der Validierung ignoriert werden („@ignore@“). Die Test-Variablen werden zu Beginn eines Testfalls definiert und sind in allen Testaktionen mehrmals referenzierbar. Das macht die Wartung des Tests einfacher und ermöglicht uns auch die Parametrisierung, zum Beispiel über einen Data Provider.

Fazit

Im Beispiel wurden die Schnittstellen einer Software automatisiert getestet. Citrus hat als Client und Server Nachrichten mit der zu testenden Anwendung ausgetauscht. Der Anwendungsfall läuft automatisch ab und ist vor allem wiederholbar. In weiteren Schritten können nun komplexere Test-Szenarien mit allen vier Schnittstellen aus dem Beispiel in einem zusammenhängenden Test behandelt werden. Dabei sind auch Fehlerfälle mit automatischem Retry und expliziter Fehlerbehandlung wichtige Testszenarien. Citrus kann hier zum Beispiel sehr einfach mit SOAP-Fault-Nachrichten umgehen.

Als normale Unit-Tests lassen sich die Citrus Tests einfach in den Build Lifecylce integrieren. Das Framework bietet ein breites Spektrum an Komponenten für den Nachrichtenaustausch mit unterschiedlichen Formaten und Transportwegen. Bei Bedarf lässt sich das Open Source Framework auch einfach durch eigene Adapter, Funktionen und Testaktionen beliebig erweitern, so dass man alle denkbaren Schnittstellen mit einem einzigen Testtool abdecken kann.

Damit sind komplexe Integrationstests mit mehreren Schnittstellen entwicklungsbegleitend möglich. Das Entwicklungsteam erhält stetiges Feedback zur Schnittstellen-Konformität der kompletten Anwendung. Mit jeder Code-Änderung werden alle Citrus-Tests automatisch ausgeführt und das Team kann sicher sein, dass festgestellte Probleme mit einer Schnittstelle behoben sind und auch in Zukunft bleiben.

Weiterführende Links

1. Citrus: *http://www.citrusframework.org*
2. Beispiel-Code: *https://github.com/christophd/citrus-samples/tree/master/incident*
3. Spring Framework: *http://spring.io*

Christoph Deppisch
christoph.deppisch@consol.de

Dipl. Inf. (FH) Christoph Deppisch arbeitet als Senior Software Consultant bei der ConSol* Software GmbH in München. Dort entwickelt er seit vielen Jahren Software-Systeme im Bereich "Java EE" und ist Spezialist für Middleware-Integration. Sein besonderes Interesse gilt der automatisierten Qualitätssicherung in großen Enterprise-Projekten. Christoph Deppisch hat maßgeblichen Anteil an der Entwicklung des Integration-Test-Frameworks Citrus und betreut das daraus entstandene Open-Source-Projekt bis heute.

„Kommunikation ist der wichtigste Faktor …“

Usergroups bieten vielfältige Möglichkeiten zum Erfahrungsaustausch und zur Wissensvermittlung unter den Java-Entwicklern. Sie sind aber auch ein wichtiges Sprachrohr in der Community und gegenüber Oracle. Wolfgang Taschner, Chefredakteur der Java aktuell, sprach darüber mit Uwe Sauerbrei und Daniel Galán y Martins von der Java User Group Hamburg (JUGHH).

Wie ist die Java User Group Hamburg organisiert?
Uwe: Um die Organisation kümmere ich mich mit Daniel Galán y Martins und Oliver Hock zu dritt. In erster Linie werden alle Planungen und Veranstaltungen mit Meetup durchgeführt. Großer Vorteil ist hier die Warteliste. Inzwischen haben uns mehr als fünfzehn Firmen und Organisationen „Kost & Logis“ angeboten und wir können, entsprechend dem Thema, zwischen verschiedenen großen Veranstaltungsräumen wählen.

Welche Ziele hat die Java User Group Hamburg?
Uwe: Wir wollen als Community weiter wachsen und verstärkt Referenten aus dem Großraum Hamburg die Möglichkeit zu Vorträgen geben. Natürlich spielt auch die Vernetzung eine große Rolle. Hier ergeben sich Möglichkeiten über die JUG hinaus, Aktivitäten beispielsweise für Kinder zu organisieren. Inzwischen sind wir auch aktiv in die Planung der JavaLand-Konferenz und des Java Forums Nord einbezogen.

Wie viele Veranstaltungen gibt es pro Jahr?
Uwe: Pro Monat sind mindestens zwei Veranstaltungen geplant.
Daniel: Schon seit Beginn der JUGHH im Jahr 2006 hatten wir regelmäßige Treffen mit Vorträgen, die mindestens einmal im Monat stattfanden, zeitweilig jedoch fast zum Erliegen kamen. Vor drei Jahren gab es einen organisatorischen Wandel, der dank Uwes langfristigen Planungen wieder in einen regelmäßigen Rhythmus kam. Nun ist die JUGHH aktiver denn je.

Was bedeutet Java für euch?
Uwe: Für mich in erster Linie „Lohn & Brot“. Es gab Zeiten, in denen ich zu zweifeln begann, ob Java noch auf dem richtigen Weg ist. Aber die wechselvolle Entwicklung der letzten Jahre und die Korrekturen haben bewirkt, dass uns Java wohl noch eine Weile erhalten bleiben wird.

Daniel: Die fünfzehn Jahre, in denen ich nun doch schon mit Java arbeite, empfand ich immer als spannend und abwechslungsreich. Es macht mir nach wie vor großen Spaß, mit der JVM zu entwickeln. Beruflich ist Java für mich die erste Wahl und durch die JUG und Open-Source-Projekte nimmt es auch einen gewissen Raum in meiner Freizeit ein, den ich nicht missen möchte. Umso mehr freut es mich, dass die Verbesserungen in der letzten Version (Lambdas, java.time etc.) Java wieder zu mehr Popularität unter den Entwicklern verholfen haben.

Welchen Stellenwert besitzt die Java-Community für euch?

Uwe: Für mich ist es die beste Quelle für den Austausch mit anderen Entwicklern und eine exzellente Gelegenheit, immer mal wieder über den Tellerrand zu schauen und neue Technologien, Frameworks und Tools kennenzulernen.

Daniel: Ohne die Community würde es die JUGHH nicht geben, viele Sprecher und Unterstützer kommen aus ihr heraus. Als die JUGHH entstand, hatten wir ebenfalls viel Austausch mit anderen JUGs, was unsere eigene Gruppe vorantrieb und positiv beeinflusste. Insofern ist dieser Austausch zwischen den Gruppen, also auch zwischen den Mitgliedern, einer der wichtigsten Aspekte für mich.

Wie sollte sich Java weiterentwickeln?

Uwe: Man sollte sich auch weiterhin von alten Zöpfen trennen, die Trends und Hinweise aus der Community beachten und die neuen Märkte (wie IoT, Big Data) gut im Auge behalten. Java lebt von den Anwendern, und wenn diese zufrieden sind und vermittelt bekommen, dass sie an der Entwicklung teilhaben, kann nicht viel passieren.

Uwe Sauerbrei
uwe@jughh.de
http://www.jughh.de

Zur Person: Uwe Sauerbrei
Uwe arbeitet als selbstständiger Berater in Hamburg. Neben seiner beruflichen Tätigkeit und der Organisation der JUG Hamburg ist es sein Ziel, Möglichkeiten für Kinder und Jugendliche zu schaffen, das Thema „IT" in all seinen Facetten kennenzulernen. Beispiele hierfür sind Formate wie „Devoxx4Kids" und „Jugend hackt", an deren Planung er maßgeblich beteiligt ist.

Daniel Galán y Martins
daniel@jughh.de
http://www.jughh.de

Zur Person: Daniel Galán y Martins
Daniel ist Senior Entwickler bei der Everyjob GmbH und hat mehrere Jahre Erfahrung in der Entwicklung von Java-Anwendungen. Sein Fokus liegt dabei auf Architektur, Middleware, DevOps und Kollaborationswerkzeugen. Wenn er sich nicht gerade um Familie, JUGHH oder Open-Source-Projekte („galan@github") kümmert, trifft man ihn im Dojo an.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?

Uwe: Auch Oracle hat in Bezug auf die Java-Community in den vergangenen Jahren eine Menge dazugelernt. Oracle ist nicht abstrakt, dort sind auch Entwickler, mit denen man sprechen kann und muss. Dies geschieht auf allen Ebenen und liefert gute Orientierungspunkte, wohin die Entwicklung von Java gehen sollte.

Wie sollte sich die Community gegenüber Oracle verhalten?

Uwe: Da muss nicht viel geändert werden. Auch hier ist Kommunikation der wichtigste Faktor. Nach anfänglichen Irritationen funktioniert es im Moment ziemlich gut.

Daniel: Oracle sollte weiter das Feedback aus der Community ernst nehmen, damit die Java-Plattform die Entwickler weiterhin mit Innovation und Wachstum an sich binden kann.

Oracle kündigt mehr als 24 neue Cloud-Dienste an

Mithilfe des Oracle Database Cloud / Exadata Service können Kunden Oracle-Datenbanken mit dem gleichen Funktionsumfang, der gleichen Leistung und Verfügbarkeit in der Cloud nutzen wie in ihrem Rechenzentrum.

Der Oracle Archive Storage Cloud Service stellt Speicherkapazität für Anwendungen und Arbeitsergebnisse bereit, die eine Langzeitspeicherung erfordern.

Mit dem Oracle Big Data Cloud Service erhalten Unternehmen eine hochleistungsfähige, sichere Plattform für verschiedene Aufgaben in Hadoop und NoSQL-Datenbanken, um große Datenmengen speichern und organisieren zu können.

Der Oracle Integration Cloud Service vereinfacht die Integration zwischen Cloud-Anwendungen und denen im eigenen Rechenzentrum sowie Anwendungen von Drittanbietern.

Der Process Cloud Service lässt Geschäftsanwender die digitale Automatisierung von Prozess-Anwendungen unabhängig von der IT-Abteilung selbst weiterentwickeln und einführen.

Der Mobile Cloud Service vereinfacht es Unternehmen, ihre Anwendungen mobil zu machen, indem er die Komplexität der dafür benötigten Backend-Infrastruktur reduziert.

Mit diesen Neuerungen baut Oracle sein Cloud-Geschäft mehr und mehr aus. Erst im Februar dieses Jahres hatte das Unternehmen in Frankfurt am Main und München die ersten Cloud-Rechenzentren in Deutschland eröffnet.

Unbekannte Kostbarkeiten des SDK

Heute: String-Padding

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir stellen in dieser Reihe derartige Features des SDK vor: die unbekannten Kostbarkeiten.

Wie lassen sich Strings mit Leerzeichen zu einer Mindestlänge auffüllen? Wahrscheinlich 99 Prozent aller Java-Entwickler würden antworten: Mit den „StringUtils" aus Apache Commons Lang. Man kann dies aber auch sehr einfach mit SDK-Bordmitteln realisieren.

Motivation

Selbst in Zeiten von Micro-Services und JSON gibt es immer noch Anwendungen, die Textdateien austauschen. Nicht selten gehorchen diese einer Spalten-Struktur, sodass Strings links oder rechts mit Leerzeichen aufgefüllt werden müssen, um diese Spalten-Struktur zu erreichen. Die „StringUtils"-Klasse der Commons-Lang-Bibliothek stellt zur Realisierung die Methoden „leftPad()", „rightPad()" und „center()" zur Verfügung. Da Commons Lang eine sehr populäre Bibliothek ist, spricht nichts gegen die Verwendung, auch wenn andere Klassen und Methoden als die String-Padding-Methoden zum Einsatz kommen. Sollen jedoch lediglich Strings aufgefüllt werden, so kann dies alternativ auch mit SDK-Bordmitteln realisiert werden.

Der SDK-Formatter

Die Klasse „java.util.Formatter" definiert eine durchaus komplexe Formatierungs-Syntax, die an die Formatierungs-Syntax der Funktion „printf()" der Sprache C angelehnt ist. Wir wollen an dieser Stelle nur den kleinen Bereich des String-Padding beleuchten, raten aber, sich einmal etwas intensiver mit dieser Klasse beziehungsweise der darin beschriebenen Formatierungs-Syntax zu beschäftigen, da sie eventuell noch weitere interessante, aber noch unbekannte Möglichkeiten für andere Datentypen enthält.

Die grundlegende Syntax des Formatter ist „%[argument_index$][flags][width][.precision]conversion". Der „argument_index" wird verwendet, um im Formatierungs-String die Parameter über ihre Position in der Argumentliste zu referenzieren. Die Methode „String#format()" verwendet als ersten Parameter einen Term der obigen Syntax. Die weiteren Varargs-Parameter der Methode werden über den Argument-Index im Formatierungs-String ersetzt. Die Nummerierung beginnt bei „1", sodass „%1$" den ersten Parameter, „%2$" den zweiten etc. darstellt.

```
String.format("%1$10s", "hello")
String.format("%1$-10s", "hello")
```

Listing 1

```
public static String leftPad(String str, int size) {
  return String.format("%1$" + size + "s", str);
}

public static String rightPad(String str, int size) {
  return String.format("%1$-" + size + "s", str);
}

public static String center(String str, int size) {
  int padLeft = (size + str.length()) / 2;
  return rightPad(leftPad(str, padLeft), size);
}
```

Listing 2

```
@Test
public void equalWithApache() {
  Assert.assertEquals(StringUtils.leftPad(STRING, SIZE),
                      SdkPadding.leftPad(STRING, SIZE));
  Assert.assertEquals(StringUtils.rightPad(STRING, SIZE),
                      SdkPadding.rightPad(STRING, SIZE));
  Assert.assertEquals(StringUtils.center(STRING, SIZE),
                      SdkPadding.center(STRING, SIZE));
}
```

Listing 3

Das Flag („flags“) „-“ erzeugt eine linksbündige Ausgabe, der Default ist rechtsbündig. Die Breite („width2“) gibt die zu erreichende Gesamtlänge dieses Parameters an. Die Genauigkeit („precision“) ist nur für Gleitkommazahlen relevant. Die Konvertierungsoption („conversion“) bestimmt schließlich, welcher Art der Parameter ist. Für uns ist „s“ für Strings die zu verwendende Konvertierung.

Die Verwendung für ein String-Padding ist damit einfach zu realisieren. Die bereits erwähnte Methode „String#format()“ erwartet als ersten Parameter einen Formatierungs-String, gefolgt von „Varargs“-Parametern für die „$“-Parameter wie im folgenden Beispiel *(siehe Listing 1)*.

Beide Methoden-Aufrufe geben einen String der Länge „10“ zurück, wobei beim ersten Aufruf links von „hello“, beim zweiten rechts von „hello“ mit Leerzeichen aufgefüllt wird. Zu Demonstrationszwecken haben wir die drei „StringUtils“-Methoden „leftPad()“, „rightPad()“ und „center()“ mit dem Formatter implementiert (Klasse „SdkPadding“, siehe *Listing 2*) und ebenfalls gegen „StringUtils“ getestet *(siehe Listing 3)*.

Diese Tests haben bei uns für eine Auswahl von Parametern ein identisches Verhalten beider Implementierungen gezeigt. Einer Verwendung steht nichts mehr im Wege.

Fazit

Die Klasse „Formatter“ definiert eine Syntax für die Formatierung verschiedener Datentypen, unter anderem auch für Strings. Das Auffüllen („Padding“) von Strings ist damit einfach zu realisieren. Die Methode „String#format()“ und weitere Methoden wie „PrintStream#printf()“ verwenden diese Formatierungsmöglichkeit.

Alle unbekannten Kostbarkeiten

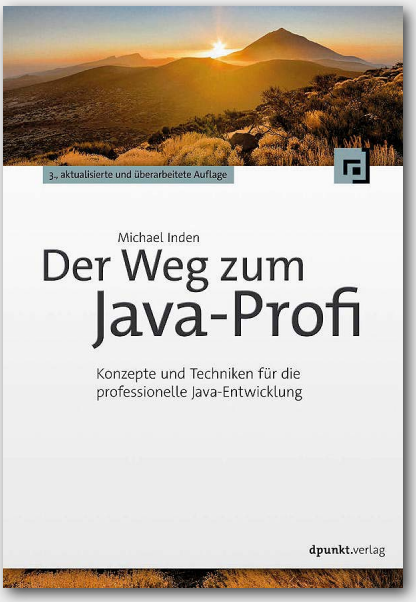
Bernd Müller veröffentlicht seit Jahren die "Unbekannten Kostbarkeiten des SDK" in der Java aktuell. Die früheren Beiträge sind auf seiner Website zu finden *(siehe „http://goo.gl/OJHajS“)*.

Bernd Müller
bernd.mueller@ostfalia.de

Bernd Müller ist Professor für Software-Technik an der Ostfalia (Hochschule Braunschweig/Wolfenbüttel). Er ist Autor mehrere Java-EE-Bücher, Sprecher auf nationalen und internationalen Konferenzen und engagiert sich in der JUG Ostfalen sowie im iJUG.

Der Weg zum Java-Profi

gelesen von Oliver Hock



Mit dem Buch „Der Weg zum Java-Profi“ findet der Autor Michael Inden einen Weg, der endlich da anknüpft, wo Ausbildung, Studium und Schulungen aufhören. Es gehört zwar eine Menge Mut dazu, sich diesen knapp 1.400 Seiten starken Band vorzunehmen, aber es ist ein Vorhaben, das schon bald Früchte tragen wird.

Der Inhalt dieses Buches schließt die Lücke zwischen dem Java-Beginner mit einigen Vorkenntnissen und dem absoluten Profi mit Expertenwissen, wobei die Intention auf dem Verständnis von Konzepten und dem praktischen Nutzen der Themen liegt. Dies wird unter anderem mit Code-Beispielen erreicht, die ergänzend zum Buch als kompilierbare Downloads verfügbar sind. Zudem sind Hintergrund-Informationen und Wissenswertes in entsprechenden Boxen besonders hervorgehoben.

Los geht es zunächst mit ein paar Java-Grundlagen. Getreu dem Motto „Was ich mich nie getraut habe zu fragen“ werden die Methoden „toString()“, „equals()“ und „hashCode()“ von „java.lang.Object“ beschrieben, ohne dabei in Sprach-Semantik oder gar Syntax-Details zu verfallen. Ferner behandelt dieses Kapitel allgemeine Datentypen, String- und Datumsverarbeitung sowie das Lesen und Schreiben von Dateien. Beson-

ders gut wird an dieser Stelle das Fehlerhandling in Java und dessen Konzept anhand von UML-Beispielen erläutert.

Im folgenden Teil kommt Michael Inden dann auf Bausteine zu sprechen, die jede professionelle Java-Applikation beinhalten sollte und die eine Anwendung stabil und qualitativ hochwertig machen. Die Wichtigkeit einer geeigneten Auswahl von Datenstrukturen im Collection-Framework wird hier mit UML hervorgehoben. Arrays, Listen, Mengen und Abbildungen mit deren Unterklassen werden detailliert erklärt. Dabei wird auch auf das interne Speicherverhalten und die implementierten Such- und Sortieralgorithmen eingegangen. Die seit Java 1.5 bekannten „Generics" finden hier ebenso ihren Platz wie viele hervorragend dargestellte Hilfs- und Bequemlichkeitsmethoden.

Aber auch externe Frameworks sind für eine professionelle Software unersetzlich. Bibliotheken wie „Google Guava" und „Apache log4j" erhöhen in modernen Anwendungen die Qualität und die Entwicklungsgeschwindigkeit. Eingehend beschreibt der Autor den Lebenszyklus von „Threads" beim „Multithreading", konkurrierende Datenzugriffe, Synchronisierung sowie die Vermeidung von „Deadlocks". Diese sind prima mit Beispielen aus dem Leben in Szene gesetzt.

Die Kapitel 8 bis 10 behandeln fortgeschrittene Java-Themen wie die GUI-Entwicklung mit Swing oder die Internationalisierung. Diese Bereiche seien an dieser Stelle außen vor gelassen, da es sich überwiegend um überholte Ansätze handelt, die durch spätere Kapitel überarbeitet oder abgelöst werden.

Interessant wird es wieder bei den von der Java-Welt langersehnten Lambdas. Am Beispiel der klassischen Umsetzung mit Java 7 versus dir neue Syntax in Java 8 wird das Lambda- oder auch Closure-Modell illustriert. Wichtig sind hier auch die Parameter-Übergabe und die Auflösung von Rückgabe-Werten. Außerdem kommt es zum Vergleich mit den sogenannten Referenzmethoden und zum Mehrwert von Default-Methoden in Interfaces.

Richtig spannend wird es dann mit den Massen-Operationen auf den zuvor eingeführten Collections. Hier werden Iterationen und Filtermechanismen mit Prädikaten konkretisiert, die leicht verständlich belegt sind und letzten Endes zu den Filtern auf Wertebereiche und dem komplexen „Filter-Map-Reduce" führen.

Kapitel 13 räumt schließlich mit den Vorurteilen auf, die Datums- und Zeitverarbeitung in Java sei „buggy" und kompliziert. Denn Java 8 bringt eine Reihe neuer Klassen und Methoden im Date-API mit und diese werden äußerst einfach und jeweils mit vielsagenden Beispielen gelehrt.

Visuelle Entwickler können sich im folgenden Abschnitt „Einstieg JavaFX 8" einen Überblick über die relativ neue GUI-Programmierung verschaffen. Allerdings handelt es sich tatsächlich um eine knappe Einführung, die für sich allein keiner leistungsfähigen Benutzeroberfläche gerecht wird. Als letzter Teil dieses großen Blockes steht noch ein Sammelsurium an kleinen Java- und JavaScript-Änderungen von Java 8, die aber alle mehr oder weniger schon in den vorangegangenen Kapiteln behandelt wurden.

Stinkig wird es bei den Fallstricken und Problemen der Programmierung. Hier werden extrem hilfreiche „Bad-Smells" identifiziert und deren Hintergründe aufgedeckt. Best-Practice-Ansätze im Programm- und Klassen-Design sind mit klaren Code-Beispielen und UML-Notationen skizziert. Das anschließende Refactoring unterstützt die Software-Entwickler beim Aufräumen von Sourcecode und Beseitigen von Schwachstellen. Umbauwürdige Code-Stellen und das beschriebene Vorgehen beim Refactoring bieten einen echten Mehrwert, wobei zum Ende des Kapitels der Refactoring-Katalog und die defensive Programmierung ein wenig theoretisch anmuten. Es folgt noch die Verwendung von Entwurfsmustern, die zwar zur Strukturierung des Codes von Bedeutung sind, hier aber außen vor bleibt.

Was wäre die Software-Entwicklung ohne einen Qualitätsgedanken? Diesen prägt der Autor in den letzten Bereichen, indem er über Programmier-Stil und Code-Konventionen, Unit-Testing und Code-Reviews berichtet. Es finden sich Richtlinien, die den Code vereinheitlichen und damit lesbarer machen, Vorschläge zur Dokumentation, denn „Dokumentiere für den Nächsten. Du könntest der Nächste sein".

Das Kapitel über „Unit Tests" wird eingehend erarbeitet, indem Test-Strategien und die Einbettung der Entwicklertests in den Gesamtprozess aufgezeigt werden. Ein gründliches Aufbereiten der Testdaten wird als genauso wichtig erachtet wie die kontinuierliche Verbesserung der Testfälle. Bevor sich das Buch mit den Code-Reviews so langsam dem Ende zuneigt, werden noch einige Frameworks zur Test-Parametrisierung und zum „Mocken" aufgezeigt.

Fazit: Wer es bis hier geschafft hat, wurde mit einem außergewöhnlich gut strukturierten Buch belohnt. Die Texte lassen sich herrlich flüssig lesen und fesseln die Leser an die Inhalte. Diese wiederum decken das breite Spektrum der Java-Entwicklung voll und ganz ab. Viele Beispiele auch aus der Praxis helfen immer wieder dabei, das Gelesene und Gelernte zu verstehen und zu vertiefen, und machen Lust auf die Anwendung in der Praxis. Dieser Wegbereiter setzt Akzente in der deutschen Literatur der Java-Entwicklung.

Titel:	Der Weg zum Java-Profi
Untertitel:	Konzepte und Techniken für die professionelle Java-Entwicklung
Autor:	Michael Inden
Verlag:	dpunkt Verlag
Umfang:	1.391 Seiten
Preis:	49,90 Euro
ISBN:	978-3-86490-203-1

Oliver Hock
oliver.hock@videa-services.com

193 Sicherheitsupdates

Mit dem vierteljährlichen Critical Patch Updat (CPU) stellt Oracle insgesamt 193 Sicherheitsupdates für die gesamte Produktpalette bereit. Für Java SE gibt es 25 Patches. Hier wird die höchste Sicherheitseinstufung von 10.0 erreicht und 23 Sicherheitslücken erlauben zudem einen unautorisierten Zugriff ohne Benutzernamen und Passwort. Ob eine vom Sicherheitsanbieter Trend Micro entdeckte Zero-Day-Lücke in der aktuellen Java-Version 8 Update 45 geschlossen wird, ist aktuell nicht abzusehen. Die aktuellen Änderungen gibt es für Java SE 6u95, 7u80 und 8u45, Java SE Embedded 7u75 und 8u33, JavaFX 2.2.80 sowie JRockit R28.3.6.

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e.V.
www.doag.org

Java User Group Stuttgart e.V. (JUGS)
www.jugs.de

Java User Group Köln
www.jugcologne.eu

Java User Group Darmstadt
http://jugda.wordpress.com

Java User Group München (JUGM)
www.jugm.de

Java User Group Metropolregion Nürnberg
www.source-knights.com

Java User Group Ostfalen
www.jug-ostfalen.de

Java User Group Saxony
www.jugsaxony.org

Sun User Group Deutschland e.V.
www.sugd.de

Swiss Oracle User Group (SOUG)
www.soug.ch

Berlin Expert Days e.V.
www.bed-con.org

Java Student User Group Wien
www.jsug.at

Java User Group Karlsruhe
http://jug-karlsruhe.mixxt.de

Java User Group Hannover
www.jug-h.de

Java User Group Augsburg
www.jug-augsburg.de

Java User Group Bremen
www.jugbremen.de

Java User Group Münster
www.jug-muenster.de

Java User Group Hessen
www.jugh.de

Java User Group Dortmund
www.jugdo.de

Java User Group Hamburg
www.jughh.de

Java User Group Berlin-Brandenburg
www.jug-berlin-brandenburg.de

Java User Group Kaiserslautern
www.jug-kl.de

Java User Group Switzerland
www.jug.ch

EuregJUG Maas-Rhine
www.euregjug.eu

Java User Group Görlitz
www.jug-gr.de

Java User Group Mannheim
www.majug.de

Lightweight Java User Group München
www.meetup.com/de/lightweight-java-user-group-munchen

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

iJUG Verbund
www.ijug.eu

Impressum

Herausgeber:
Interessenverbund der Java User Groups e.V. (iJUG)
Tempelhofer Weg 64, 12347 Berlin
Tel.: 030 6090 218-15
www.ijug.eu

Verlag:
DOAG Dienstleistungen GmbH
Fried Saacke, Geschäftsführer
info@doag-dienstleistungen.de

Chefredakteur (VisdP):
Wolfgang Taschner, redaktion@ijug.eu

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

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

Anzeigen:
Simone Fischer
anzeigen@doag.org

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

Druck:
Druckerei Rindt GmbH & Co. KG
www.rindt-druck.de

Titelfoto: © Fiedels / fotolia.com
Foto S. 10: © Tommaso Altamura / 123rf.com
Foto S. 20: © kentoh / fotolia.com
Foto S. 29: © Blueminiu / fotolia.com
Foto S. 47: © MWiner / fotolia.com
Foto S. 53: © AllebaziB / fotolia.com
Foto S. 57: © atoss / fotolia.com
Foto S. 63: © alexmit / 123rf.com

Inserentenverzeichnis

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG *News* und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN

0700 11 36 24 39

ODER BESTELLEN SIE ONLINE

go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr

☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:

Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.