Ausgabe 02/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 02

Java aktuell

iJUG Verbund
www.ijug.eu

High Performance
Ein Blick hinter die Kulissen von Java

Immutability
Vorteile von unveränderlichen Datenstrukturen

Zeitreise
Die Highlights der Java-Versionen 8 bis 13 im Überblick

Das Herz des Java-Universums

DW *Made for minds.*

Coding for freedom.

Mehr Berufung als Beruf:
Dein IT-Job bei der DW.

JETZT BEWERBEN!
dw.com/it-karriere

Du entwickelst passgenau? Dann fordere uns auf der **JavaLand** am Klask heraus. Du willst was footern? Komm mit uns bei einer Tüte Popcorn ins Gespräch.

Du findest uns auf der **JavaLand** direkt gegenüber vom Haupteingang.

Liebe Leser der Java aktuell,

den meisten von Ihnen bin ich als Autor der „Unbekannten Kostbarkeiten des SDK" bekannt – eine bis 2011 zurückreichende und seit 2019 zu neuem Leben erweckte Rubrik dieser Zeitschrift, in der wir gemeinsam die verborgenen Schätze, die das JDK zu bieten hat, heben und beleuchten. Als Mitglied im Redaktionsbeirat unterstütze ich die inhaltliche Erstellung der Fachzeitschrift und möchte mit Ihnen gemeinsam die Inhalte dieser Ausgabe näher beleuchten.

Der Themenschwerpunkt dieser Ausgabe widmet sich den originalen Kernthemen Java und JVM. Dass Java eine hochperformante Sprache ist, wissen wir alle. Warum dies so ist untersucht René Schwietzke in unserem Leitartikel. Der Autor zeigt, warum Javas Just-in-Time-Compilation häufig das Maximum an Performanz herausholen kann und gibt uns einen sehr guten Einblick in Details des Just-in-Time-Compilers. Nicolai Mainiero beschäftigt sich ab Seite 19 mit „immutable" Objekten. Immutabilität ist eine häufig sehr erstrebenswerte Eigenschaft und Mainiero zeigt, wie sie direkt mit Java oder Bibliotheken erreicht werden kann. Er geht auch auf Nachteile verschiedener Lösungen ein.

Mein eigener Artikel auf den Seiten 25 bis 28 greift thematisch die neue virtuelle Maschine GraalVM auf. Diese vermag Java auch Ahead-of-Time zu kompilieren und arbeitet, was diesen Aspekt angeht, wie C oder C++. Der Artikel gibt einen kurzen Überblick der Entwicklung bis hin zu den aktuellen Möglichkeiten des Kompilierens von Java, um erste Gehversuche mit der Erzeugung nativer Images zu machen.

Bei häufigen Releases und der Integration ehemals kommerzieller Features in das OpenJDK ist es leicht, den Überblick zu verlieren. Benjamin Schmid gibt uns eine Zusammenfassung der wichtigsten Änderungen seit Java 8. Apropos neu: André Sept stellt uns auf Seite 38 den neuen, erstmals stattfindenden Thementag im Rahmen der JavaLand 2020 vor.

Obwohl schon häufig Thema früherer Ausgaben, nimmt Mike Betsch noch einmal die Lizenzierung von Java SE durch Oracle unter die Lupe. Er gibt Tipps, wie Unternehmen tatsächliche Installationen erheben und damit die tatsächlichen Kosten ermitteln können. Bei großen Unternehmen mit hunderten oder gar tausenden von Servern, virtuellen Maschinen und PCs kein leichtes Unterfangen.

Benjamin Klatt und Matthias Schulte zeigen Ihnen ab Seite 44 unterschiedliche Integrationsmöglichkeiten mit der Camunda Workflow Engine. Dabei gehen die Autoren auf Legacy-Anwendungen ein und teilen ihre Erfahrungen, wann sich welcher Ansatz eignet.

Als Java-Entwickler freuen wir uns, dass sich JavaScript zumindest teilweise in Richtung Java weiterentwickelt. Wer allerdings nur entfernt damit zu tun hat fragt sich, wie JavaScript, ECMAScript und ES.Next zusammenhängen beziehungsweise was die Unterschiede sind. Marc Teufel beleuchtet diese Thematik in seinem Artikel "JavaScript, ECMAScript, ES.NEXT… noch Fragen?" näher.

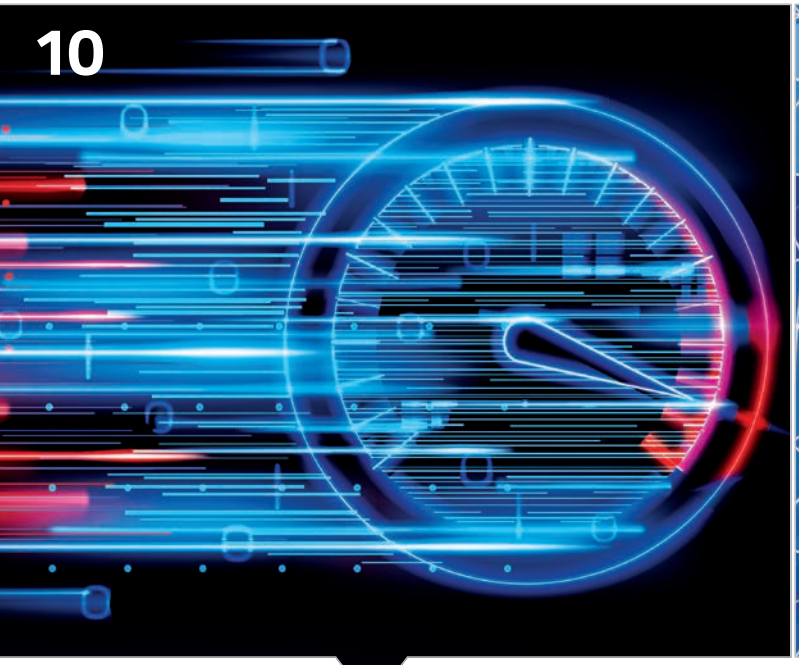
Formulare sind allgegenwärtig und müssen heutzutage bei so ziemlich jeder Online-Buchung ausgefüllt werden. Max Liesegang gibt Ihnen ab Seite 54 Tipps und Tricks, wie Sie ein benutzerfreundliches und effektives Formular gestalten können. Michael Keller und Christian Seifert befassen sich mit der Thematik, wie wir Entscheider von unseren Ideen überzeugen und als Entwickler glücklich sein können.

Zum Abschluss dieses Vorworts gehen wir noch einmal zum Anfang dieser Ausgabe zurück. Wie immer fasst Andreas Badelt im Java-Tagebuch kurz zusammen, was in der letzten Zeit in der Java-Welt passiert ist. Markus Karg, ebenfalls Stammautor der Java aktuell, beleuchtet die Neuigkeiten aus der Eclipse-Welt. Diese sind allerdings leider nicht rosa, sondern eher grau bis schwarz. Markus zeigt uns die Dringlichkeit auf, dass wir, die Community, uns engagieren müssen, um Jakarta EE zum Erfolg zu verhelfen. Los geht's!

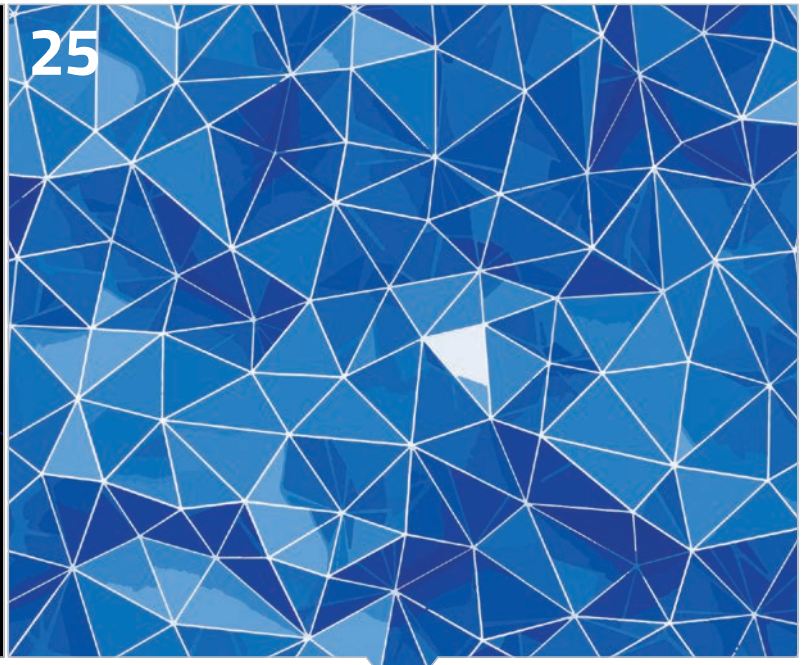
Ihr

Bernd Müller
Redaktionsbeirat Java aktuell

Inhalt



Performance-Tuning und Benchmarking in Java



Einführung in die Erstellung nativer Images mit der GraalVM

Wie gehen Unternehmen mit der Lizenzierung von Java SE um?

Erwartungen und Realität im Entwickleralltag vereinen

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

9. Oktober 2019

Payara-Server offiziell Jakarta-EE-8-kompatibel

Der auf dem GlassFish basierende Payara-Server ist jetzt auch Jakarta-EE-8-kompatibel, genau wie sein Verwandter stromaufwärts. Fast 50.000 Test Suites mussten dafür fehlerfrei „überstanden" werden. Technisch ist das kein Hexenwerk, GlassFish war schließlich die Referenzimplementierung für Java EE, und Jakarta EE 8 ist ja „nur" Java EE 8 im neuen Gewand – und mit neuer Heimat bei der Eclipse Foundation; aber Mike Milinkovich von der EF hebt in seinem Glückwunsch einen wichtigen Punkt hervor: „Payara is a new vendor to this ecosystem, as they were never a Java EE licensee." Die Einstiegshürde ist geringer geworden.

10. Oktober 2019

Big Bang für Jakarta EE 9?

Es gibt nach langen Diskussionen endlich einen umfassenden und konkreten Plan für Jakarta EE 9, den Oracles Bill Shannon gestern in der Entwickler-Mailingliste vorgestellt hat. Darin wird ein Release in maximal zwölf Monaten ab heute vorgeschlagen, das dem „Big Bang"-Ansatz folgt – sprich, alle Packages werden auf einen Schlag in „jakarta.*" umbenannt, egal ob sie für Release 9 modifiziert werden müssen oder nicht. Rückwärtskompatibilität wird den jeweiligen Implementierungen überlassen – wobei ein Open-Source-Projekt vorgeschlagen wird, das als Grundlage der einzelnen Implementierungen dienen kann. Darüber hinaus gibt es in Bills Vorschlag eine längere „Pruning"-Liste von Spezifikationen, die nicht (mehr) in Jakarta EE 9 enthalten sein sollen *[1]*.

17. Oktober 2019

Medium Term Support für Java

Seit der Umstellung auf den halbjährlichen Release Train steckt ja in jedem sechsten Release „Long Term Support". Da es aktuell also die Wahl zwischen den extremen „Upgrades alle sechs Monate" und „jahrelanger Tiefenentspannung mit LTS" gibt, hat sich Azul etwas Neues für sein Zulu Enterprise JDK ausgedacht: „Medium Term Support" Releases. Das sollen immer die ungeraden Nummern zwischen den LTS Releases sein (also aktuell 13 und dann 15). Sie werden bis 18 Monate nach dem folgenden LTS Release mit Updates unterstützt (17 ist das nächste im Jahr 2021). Damit wird die „Kaffee"-Auswahl nicht so schnell langweilig, wie Azul es ausdrückt: „Tall for feature releases, Grande for the new MTS from Azul, and Venti for LTS".

Package Renaming Tool für Jakarta EE

„Es gibt nichts Gutes, außer man tut es." Das muss sich auch IBMs BJ Hargraves (auch CTO der OSGi Alliance) gedacht haben; er hat kurzerhand ein Tool zum Package Renaming in Jar- und War-Files geschrieben und auf GitHub veröffentlicht *[2]*. Das Projekt besteht im Kern aus einer (langen) Transformer-Klasse, die in einem Class Loader eingesetzt werden könnte, und einem File mit Mappings von alten auf neue Package-Namen. Vielleicht sehen wir hier ja schon die Basis für Jakarta-Implementierungen.

21. Oktober 2019

Eclipse MicroProfile 3.1

MicroProfile 3.1 bringt kleinere Updates für die Metrics und Health APIs. Beispielsweise können Health Checks konfigurativ ausgeschaltet werden und Metrics Registries müssen Thread-sicher implementiert sein.

23. Oktober 2019

Amazon tritt dem JCP bei

Der Java Community Process ist für die Standardisierung zumindest von Java SE weiterhin wichtig. Und die Diversifizierung des Ökosystems bringt neue Mitglieder: Amazon, schon seit mehreren Jahren mit seinem eigenen JDK „Corretto" unterwegs, ist jetzt dem JCP beigetreten.

31. Oktober 2019

Microsoft bereit für das OpenJDK

Microsoft hat vergangene Woche das „Oracle Contributor Agreement" unterzeichnet, um am OpenJDK mitwirken zu können (es gehört ja formal immer noch Oracle). Bruno Borges, früher bei Oracle und jetzt Principal Product Manager für Java bei Microsoft, hat nun in der OpenJDK-Mailingliste die nächsten Schritte angekündigt: Das „Java Engineering Team" wird mit kleineren Bug Fixes und Backports loslegen, um sich ins OpenJDK-Projekt einzuleben. Kleine Randnotiz: Auf Twitter gefragt, was denn aus dem „Krieg C# gegen Java" geworden sei, antwortet Borges: „There's no more room for wars in the world." Das lasse ich mal als politisches Statement stehen und spare mir mögliche Kalauer…

13. November 2019

JavaLand 2020: Programm online

Das Programm der JavaLand 2020 ist online und prall gefüllt wie immer. Von Frontalbetankung bis zu ausgiebigen Hands-on-Aktivitäten, ergänzt um jede Menge Spaß drumherum *[3]*.

19. November 2019

2019er Wahlen zum Executive Committee

Die Wahlen zum Executive Committee des Java Community Process

sind abgeschlossen. Der iJUG hatte sich – vielleicht ein bisschen zu spontan – entschieden, seinen Hut in den Ring zu werfen. Am Ende haben sich bei den „Elected Seats" aber Twitter und Tomitribe durchgesetzt, dahinter der iJUG und dann BellSoft. Vielleicht klappt es beim nächsten Mal ja besser. Bei den „Ratified Seats" sind ARM, Fujitsu, IBM und Intel jeweils ohne eine größere Anzahl Gegenstimmen durchgekommen (Fujitsu hatte mit 89 Prozent Ja-Stimmen das schlechteste Ergebnis). Bei den „Associate Seats" haben sich Marcus Biel und Ken Fogel durchgesetzt.

21. November 2019

GraalVM 19.3.0 für Java 11

Das neue Release der polyglotten GraalVM bietet unter anderem Unterstützung für Java 11 (bislang nur Java 8) und damit einhergehend für das Java Platform Module System. Die Native-Image-Unterstützung für Java 11 ist aber noch im Beta-Stadium. Als letztes „Major" Release des Jahres bekommt es Long Term Support (die 19 ist hier nur die Jahreszahl, 19.3 ist dann nach Graal-Interpretation ein Major Release). Long Term Support heißt hier, dass es so lange Bug Fixes erhalten wird, bis das nächste LTS Release (also 20.3 in einem Jahr) veröffentlicht wird. Die anderen Major Releases (19.2, 20.0, …) werden direkt mit der Veröffentlichung des nächsten inaktiv. Details auf der GraalVM-Website *[4]*.

22. November 2019

Die neuen Müllsammler

Wer hat noch den Überblick über die ganzen Garbage Collector, die die JVM inzwischen anbietet (oder nicht mehr anbietet)? Im Java Magazine von Dezember werden die Neuzugänge ausführlich erklärt *[5]*. Auch, wofür der Epsilon Garbage Collector aus Java 11 da ist – auch als „Do Nothing GC" bekannt. Eigentlich soll Epsilon ja nur zum Performance-Tuning genutzt werden. Aber bei kurz laufenden Applikationen kann man sich das Aufräumen am Ende natürlich sparen, solange der Speicher reicht. Bei sehr latenzempfindlichen Applikationen, deren gesamter Speicherbedarf exakt bekannt ist – oder die praktisch komplett Garbage-frei sind, kann er natürlich auch sinnvoll sein.

26. November 2019

Quarkus.io 1.0

Quarkus wird nach nicht einmal neun Monaten in der Öffentlichkeit „erwachsen". Red Hat hat die erste stabile Version für das Microservices-Framework freigegeben, das das Eclipse MicroProfile implementiert und für den Einsatz mit der GraalVM und Kubernetes („Container First") ausgelegt ist. Das Bauen einer nativen Applikation mit Quarkus und der GraalVM kann schon ein paar Minuten länger dauern, aber die Zeile „… startet in 0.008 s" zaubert schnell das Lächeln zurück in die Gesichter und eröffnet neue Möglichkeiten für die Java-Entwicklung. Release 1.0 enthält dem Ziel gemäß viele Fixes, aber keine großen neuen Features. Unter anderem einen verbesserten Spring Compatibility Layer gibt es aber. Achtung: Die GraalVM wird erst in der Version 19.2.1 unterstützt (also Java 8), Abhilfe soll hier Quarkus 1.1 schaffen.

29. November 2019

Java 14 nimmt Gestalt an

Java 14 befindet sich in der Entwicklung und inzwischen sind eine ganze Reihe neuer Features für dieses Release bestätigt worden: Als Preview beziehungsweise Inkubator-Version soll es Pattern Matching für instanceof und ein neues Packaging Tool geben (zum Erzeugen von nativen Paketen wie rpm, dmg oder exe). Text Blocks sind als „zweite Preview" dabei, da hier lieber noch eine Extrarunde gedreht werden soll, bevor sie als stabil deklariert werden. Als stabile Features sollen unter anderem Records und Switch Expressions kommen sowie der Z Garbage Collector für MacOs und eventuell auch für Windows *[6]*.

Und noch 'n Micro-Framework!

Bald gibt es sogar mehr Micro-Frameworks als zu implementierende Standards… Das neueste heißt Azkarra Streams und setzt auf Kafka auf, um „Stand-alone Kafka Streams Applications" zu erzeugen.

Referenzen:

[1] *https://www.eclipse.org/lists/jakartaee-platform-dev/msg00702.html*
[2] *https://github.com/bjhargrave/transformer*
[3] *https://programm.javaland.eu/2020/*
[4] *https://www.graalvm.org/docs/release-notes/19_3/*
[5] *https://blogs.oracle.com/javamagazine/december-2019-v2*
[6] *https://openjdk.java.net/projects/jdk/14/*

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

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

Markus' eclipse-Corner

In der letzten Eclipse Corner habe ich darüber berichtet, dass die Eclipse Foundation inzwischen an Jakarta EE 9 arbeitet und dafür auf die Hilfe der Community angewiesen ist. Auch wenn ich stets versuche, wenig Interpretationsspielraum in meinen Formulierungen zuzulassen, war wohl doch einer Vielzahl von Lesern nach wie vor nicht klar, wer damit gemeint ist. Daher möchte ich diese Ausgabe nutzen, um euch allen kräftig ins Gewissen zu reden.

Fangen wir mit einer kleinen Zeitreise an. Im Jahr 2017 hat Oracle das Eigentum an der Plattform Java EE (also die Quellcodes sowohl der Spezifikationen, TCKs und APIs als auch der Referenzimplementierungen wie beispielsweise Jersey und GlassFish), nicht jedoch die Markenrechte und auch nicht das geistige Eigentum „IP" an die Eclipse Foundation übertragen. Der juristische Akt zog sich lange hin, gefolgt von einer Phase der physischen Übertragung der Quellen. Nach über zwei Jahren war alles bei der Stiftung angekommen und steht seit mehreren Monaten der Öffentlichkeit mittels GitLab-Repositories zur Verfügung. Eine Handvoll Oracle-Mitarbeiter hat in einer letzten Kraftanstrengung, gemeinsam mit einigen anderen Protagonisten, daraus ein Release namens „Jakarta EE 8" geformt, das seit einigen Wochen allgemein zum Download bereitsteht. Mehr kann man von einem Schenkenden nicht erwarten. Und nun?

In der allgemeinen Mailingliste zu Jakarta EE meldeten sich Hunderte von Personen zu Wort, die dort im eigenen Interesse oder als Vertreter einer Organisation ihre Meinung postulierten, wie die zukünftige Ausgabe „Jakarta EE 9" aussehen soll. Manche dieser Aussagen klangen dabei eher wie eine Bestellung als ein Hilfsangebot. Offenbar ist vielen unklar, dass niemand diese Bestellung jemals an die Haustür liefern wird: Selbst kochen ist angesagt! Die *Community* muss sich jetzt an den Herd stehen. Und zwar die *User*-Community.

Ja, die Community seid *ihr*! Nicht Oracle, nicht IBM. Weder Red Hat noch Payara, Tomitribe oder Fujitsu. Die sehen sich zwar gerne als „die Community", wenn es um das Bestimmen geht, und machen auch sicherlich mit Produkten und Support Umsatz, die Jakarta EE zertifiziert sind und teilweise auch Code der ehemaligen Referenzimplementierungen enthalten (teilweise sogar fast vollständig). Doch der physische Beitrag an echter, messbarer Arbeitsleistung im tatsächlichen Quellcode ist gering bis null *[1]* und wird in Kürze noch weiter sinken. Ja, die *bestimmen* über alles und zwar, weil sie für dieses Recht kräftig zahlen *[2]*. Aber die *Arbeit* machen andere *[3]*.

Jetzt könnte man sagen: Das ist unfair. Ja, vielleicht ist es das. Aber es ist nicht zu ändern. Diese Unternehmen lassen sich nicht zur Arbeit zwingen. Weder durch die Eclipse Foundation (die keine Macht über ihre Mitglieder hat und keine eigenen Programmierer beschäftigt) noch durch die Kunden (denen der Support – oder sagen wir es doch ganz ehrlich: die Regressmöglichkeit – durch eine bekannte Firma viel, viel wichtiger ist

als neue Features und weniger Bugs). Was wir aber können, ist, das Heft in die Hand zu nehmen und den Schurz wackeln zu lassen. Ja, auch ihr. Gerade ihr. Wer sonst, wenn nicht ihr? *Ihr wollt* doch Jakarta EE weiter benutzen. Nicht Oracle, und auch nicht IBM.

Wer den Schuss immer noch nicht gehört hat: Ihr müsst euch *jetzt entscheiden*. Entweder ihr helft ab *sofort* bei Jakarta EE tatkräftig mit (zum Beispiel, indem ihr einen Mitarbeiter einen Tag in der Woche an Jersey mitarbeiten lasst) oder ihr steht im Jahr 2020 ohne Jakarta EE da. Denn andere werden es nicht tun. Ja, ihr habt dafür weder Zeit noch Budget. Ja, ja, ja. Aber andere auch nicht. Fakt ist, die Eclipse Foundation diskutiert momentan darüber, was sie alles *jetzt sofort* in den Müll werfen, wenn sich keiner findet, der es pflegt. Darunter EJB, JAXB (also XML), JAX-WS (also SOAP). Ja, alles das gibt es *ab sofort* nicht mehr, wenn *ihr* nicht sofort aktiv werdet. Denn weder Oracle noch IBM noch sonst ein großes Unternehmen wird das für euch pflegen. Noch nicht einmal für Geld! Und nein: Das wird auch *niemand* mit euch diskutieren. Ihr macht es selbst, oder es ist halt weg. Die Zeit der Diskussion ist *lange* vorbei. Wer bezahlt, bestimmt. Wieviel zahlt *ihr*? So ist das nun mal im Kapitalismus!

Habt ihr es jetzt kapiert? Dann meldet euch bei uns. Wir sagen euch, an was ihr jetzt sofort arbeiten könnt.

Referenzen

[1] Offizielle Statistik der Eclipe Foundation: *https://projects.eclipse.org/projects/ee4j.jakartaee-platform/who*

[2] Eclipse Foundation Mitgliedsbeiträge: *https://www.eclipse.org/membership/become_a_member/membershipTypes.php*

[3] GitHub-Statistik: *https://github.com/eclipse-ee4j/jaxrs-api/graphs/contributors*

Markus Karg

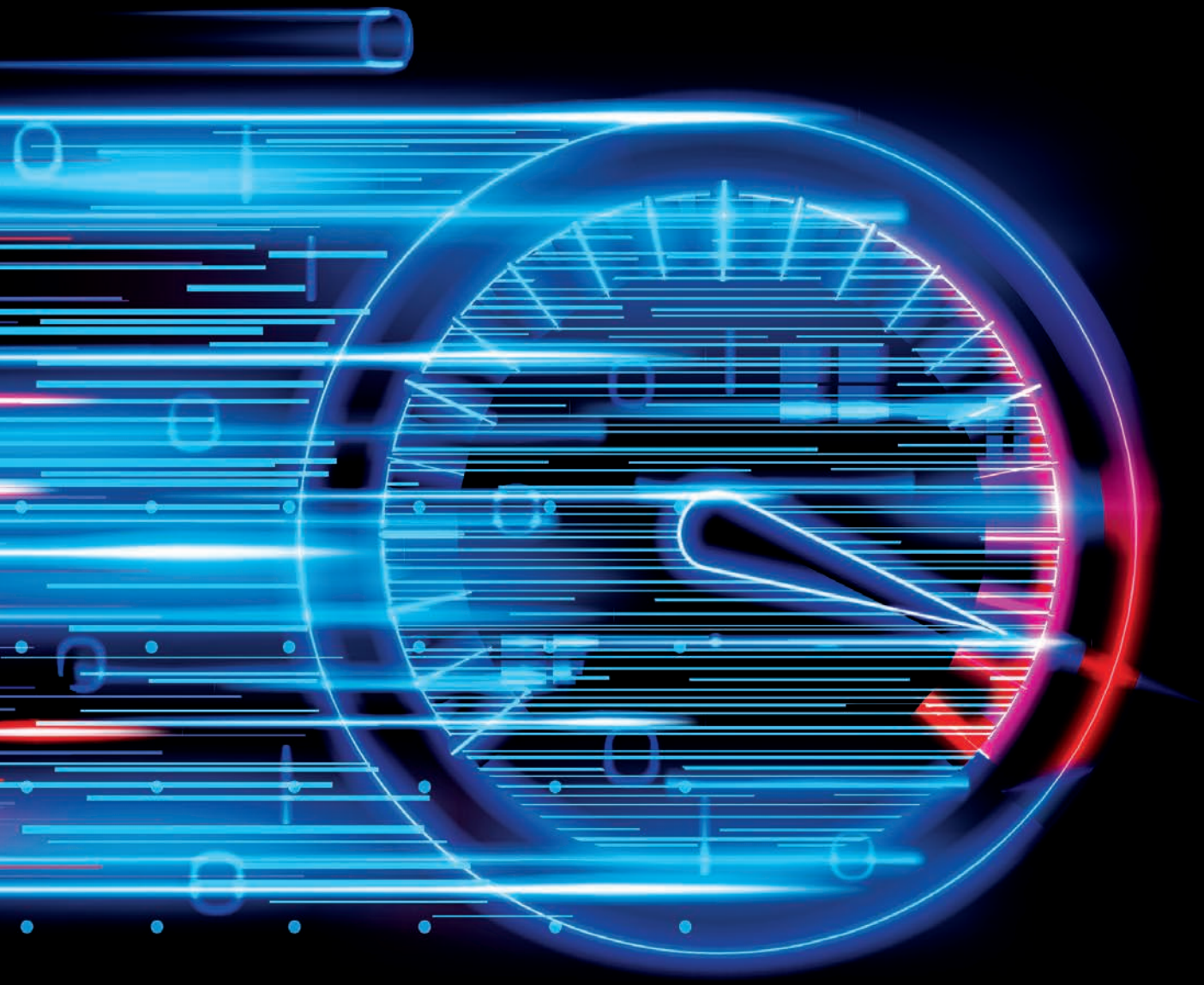
markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

High Performance Java – Hinter den Kulissen von Java

René Schwietzke, Xceptance GmbH

Die Zeiten, in denen Java der langsame Bytecode-Interpreter war, sind lange vorbei. Die JVM nutzt viele Tricks, um Code effizient auszuführen, und transformiert und optimiert Java-Code auf die ausführende Hardware. Mit etwas Wissen über diese Prozesse kann man vermeiden, dass man gegen die JVM arbeitet, und gleichzeitig mehr Geschwindigkeit erreichen. Selbst wenn man nicht die letzte Mikrosekunde jagt, ist es interessant zu sehen, welche Techniken die JVM einsetzt, um die Ausführungsumgebung besser zu verstehen.



Im Rahmen von Performancetests wurde beobachtet, dass viele Java-Entwickler bei Profiling und Debugging eher naiv an Probleme herangehen und oft 50 Prozent bessere Performance erwarten und auch lokal messen, aber am Ende in Produktion keinerlei Unterschiede existieren. Häufig ist das Grundverständnis für die JVM-Laufzeitumgebung nicht ausgeprägt. Zusätzlich ist das Wissen über CPU, Speicher und Hardware im Allgemeinen gering, da moderne Programmiersprachen dies nicht mehr erfordern. Wenn man sich die Effizienzbetrachtungen zu Programmiersprachen *[1]* anschaut, sieht man, dass Java gleich hinter den klassischen kompilierten Sprachen wie C in der Effizienz steht. Wie schafft Java das, obwohl es nicht plattformspezifisch kompiliert wird?

Lernen durch Beobachten

An einem einfachen Beispiel kann man gut erkennen, wie Java mit Code umgeht. Zuerst der Hinweis, dass man natürlich jeden Microbenchmark mit dem Java Measurement Harness (JMH) *[2]* durchführen sollte, das Beispiel in *Listing 1* macht es allerdings bewusst anders.

In *Listing 1* wird eine Methode *append(int)* durch eine einfache Klammer um den Methodenaufruf vermessen (vollständiger Code: *[3]*). Bei Ausführung erhält man Messwerte, die starke Laufzeitunterschiede zwischen der ersten und der hunderttausendsten Ausführung der Methode für verschiedene Laufzeitumgebungen zeigen *(siehe Tabelle 1)*.

```
public class PoorMansBenchmark
{
    // just burn time
    private static String append(final int i)
    {
        final StringBuilder sb = new StringBuilder();
        sb.append(System.currentTimeMillis());
        sb.append(i);
        sb.append("lkJAHJ AHSDJHF KJASHF HSAKJFD");

        char[] ch = sb.toString().toCharArray();
        Arrays.sort(ch);

        return new String(ch);
    }

    public static void main (final String[] args)
    {
        final int SIZE = 100_000;
        final long[] result = new long[SIZE];

        int sum = 0;
        for (int i = 0; i < SIZE; i++)
        {
            final Timer t = Timer.startTimer();

            sum += append(i).length();

            result[i] = t.stop().runtimeNanos();
        }

        // pretend we need it but we really don't
        // just to avoid optimizations (spoilers, duh!)
        if (sum < 0)
        {
            System.out.println(sum);
        }

        // more code here to output the measurements
    }
}
```

Listing 1: Ein naiver Benchmark – org/sample/PoorMansbenchmark.java

Der Java-Skeptiker wird natürlich sofort die Garbage Collection (GC) als Grund ausmachen, während der C-Fan auf den Interpreter zeigen wird. Um diese Skepsis teilweise zu beseitigen, schließt Spalte 2 „NoOp" die GC mit dem Epsilon GC aus *[10]* und für Spalte 3 „Graal AOT" wurde das Programm mit der GraalVM nativ kompiliert. Ein kurzer Blick auf *Tabelle 1* zeigt, dass der Java-Code schneller wird, je länger er läuft. Die GC hat keinen Einfluss und der Code läuft sogar etwas langsamer. Der nativ kompilierte Code via Graal AOT ist anfangs schneller, wird aber am Ende von der Java Virtual Machine (JVM) trotzdem unterboten. Schaut man sich die Spalte 1 „G1" im Chart an, sieht man, wie stark die Laufzeiten schwanken *(siehe Abbildung 1)*.

Man sieht, dass die Laufzeit diskontinuierlich sinkt, gelegentlich aber sogar steigt. Java kompiliert und optimiert den Code wiederholt, bis ein optimales Laufzeitverhalten vorliegt. Dies führt zu den drei Teilthemen für diesen Artikel:

- Compiler
- Javas Trickkiste
- CPU, Memory und Cache

Compiler

Bei dem Begriff „Java Compiler" fällt vielen nur *javac* ein. Dieser ist allerdings nur einer von drei Compilern im Standard OpenJDK mit Hotspot. Hotspot ist der Just-in-time-Compiler (JIT), der wiederum aus zwei Compilern (C1 und C2) besteht und vier verschiedene Möglichkeiten besitzt, Code zu kompilieren. Bei dieser Übersetzung handelt es sich auch um die Transformation von Bytecode zu Maschinencode, in diesem Fall erfolgt diese jedoch erst während der Laufzeit des Programms.

javac übersetzt den Java-Code in plattformunabhängigen Bytecode. Wichtig ist, dass javac nur minimale Optimierungen vornimmt, und zwar nur, wenn Ergebnisse bereits zur Compile-Zeit berechenbar sind. Hierzu zählen Arithmetik, String-Operationen und Konstanten.

C1 und C2 übersetzen nur ganze Methoden und Schleifen (for, while), abhängig von der Häufigkeit der Verwendung. Je öfter Code ausgeführt wird, desto „heißer" und damit wahrscheinlicher ist eine Übersetzung in Maschinencode. Die Übersetzung kostet Zeit, deswegen wird nicht pauschal alles übersetzt. Man kann Java zwingen, mehr zu übersetzen, das ist jedoch ein eigener Forschungsgegenstand, da immer eine Kosten-Nutzen-Betrachtung erforderlich ist.

Wie bereits erwähnt, haben C1 und C2 vier mögliche Übersetzungsergebnisse. Ein Blick in den Sourcecode des JDK (in src/share/vm/runtime/advancedThresholdPolicy.hpp) erklärt, welche Varianten existieren:

Durchlauf	1 - G1	2 - NoOp	3 - Graal AOT
1	595.433 ns	549.924 ns	8.791 ns
100	40.456 ns	34.017 ns	1.989 ns
1.000	6.923 ns	7.365 ns	1.620 ns
10.000	2.743 ns	2.738 ns	1.309 ns
100.000	755 ns	990 ns	1.278 ns

Tabelle 1: Vergleich der unterschiedlichen Umgebungen

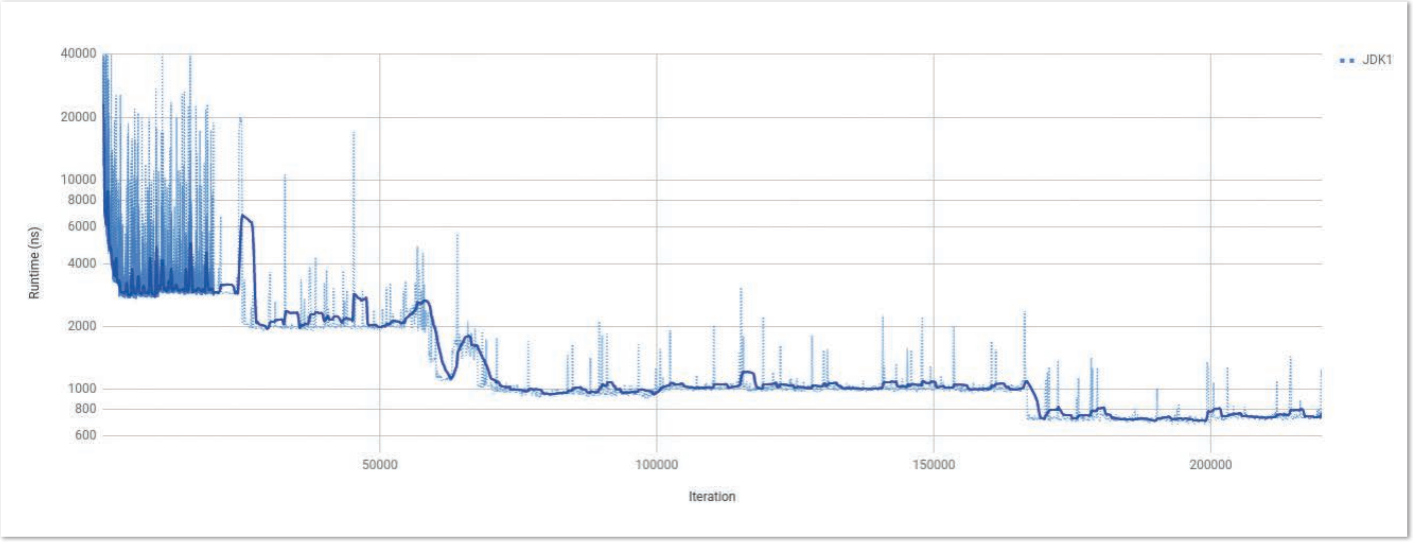


Abbildung 1: Laufzeitentwicklung von append(int) (© René Schwietzke, [4])

- Level 0: Interpreter
- Level 1: C1 with full optimization (no profiling)
- Level 2: C1 with invocation and back-edge counters
- Level 3: C1 with full profiling (level 2 + MDO)
- Level 4: C2 fully optimized

Während Level 0 lediglich Bytecode interpretiert, produziert C1 Code, der mit verschiedenen Instrumentierungen versehen wird, um messen zu können, ob der gebaute Code effizient ist oder weiter verfeinert werden kann.

Im Idealfall springt die JVM von Level 0 nach 3 und nach einer weiteren Beobachtung wird finaler Code mit C2 optimiert produziert. Aber natürlich ist die Welt nicht so einfach, deswegen kann es sein, dass C2 gerade beschäftigt ist und der C1 zunächst Level-2-Code produziert, bevor Level 3 und 4 genutzt werden können. Eventuell ist der Code so trivial, dass Level 1 ausreichend ist. Darum erklärt sich auch, warum Code schneller wird, je länger er läuft. Die JVM beobachtet kontinuierlich die Performance und kompiliert den Code gegebenenfalls nochmals mithilfe der gewonnenen Informationen.

Im Zusammenhang mit dem Wissen aus dem folgenden Kapitel kann es aber zu unerwarteten Seiteneffekten kommen, die den Compiler zwingen, seine Code-Optimierungen zurückzunehmen.

Javas Trickkiste

Nachfolgend einige Beispiele für Optimierungen, die die JVM während der Programmausführung durchführt.

Dead Code Elimination

Das folgende Beispiel von Douglas Hawkins zeigt, wie die JVM Laufzeitinformationen nutzt, um überflüssigen Code zu entfernen *(siehe Listing 2)*. Bei der Ausführung entsteht folgendes Bild *(siehe Abbildung 2)*.

Die Laufzeit ist zunächst hoch, verbessert sich dramatisch mit jeder Iteration, bis zur Iteration 400, wo ein einziges Mal die Variable `trap` nicht `null` ist. In diesem Moment verwirft der Compiler den bisherigen Code. Kurz wird der Interpreter wieder benutzt, bis der Compiler neuen nativen Code verfügbar hat. Dieser erreicht nicht wieder das vorherige Laufzeitniveau.

```
public static void main(String[] args)
{
    Object trap = null;
    Object o = null;

    for (int i = 0; i < 1_000; i++)
    {
        final Timer t = Timer.startTimer();

        for (int j = 0; j < 1_000; j++)
        {
            // burn time and train that null is normal
            o = new Object();

            if (trap != null)
            {
                System.out.println("Got you." + o);
                trap = null;
            }
        }

        // Give me a Non-Null, Vasily.
        // One Non-Null only, please.
        if (i == 400)
        {
            trap = new Object();
        }

        System.out.println(
            MessageFormat.format("{1} {0, number, #}",
            t.stop().runtimeNanos(), i));
    }
}
```

Listing 2: AllocationTrap.java

Der Compiler hat über die Zeit erkannt, dass `trap` immer `null` ist, und entschieden, ab circa 200 Iterationen das `IF` zu streichen. Er verlässt sich stattdessen auf das Betriebssystem und riskiert einen Segmentation Fault, um sich benachrichtigen zu lassen, falls `trap` doch einmal einen anderen Wert annimmt. Das ist die übliche Vorgehensweise in Java, weil sonst jeder Objekt-Zugriff mit einem `if != null else NPE` umgeben werden müsste. Die Iteration 400 zeigt dem Compiler jedoch, dass die der Optimierung zugrunde liegende Annahme falsch war, sodass die aggressive Optimierung zurückgenommen wird.

Bei der Optimierung des eigenen Codes ist daher wichtig, dass Ausnahmen dieser Art nicht vorkommen oder früher behandelt werden.

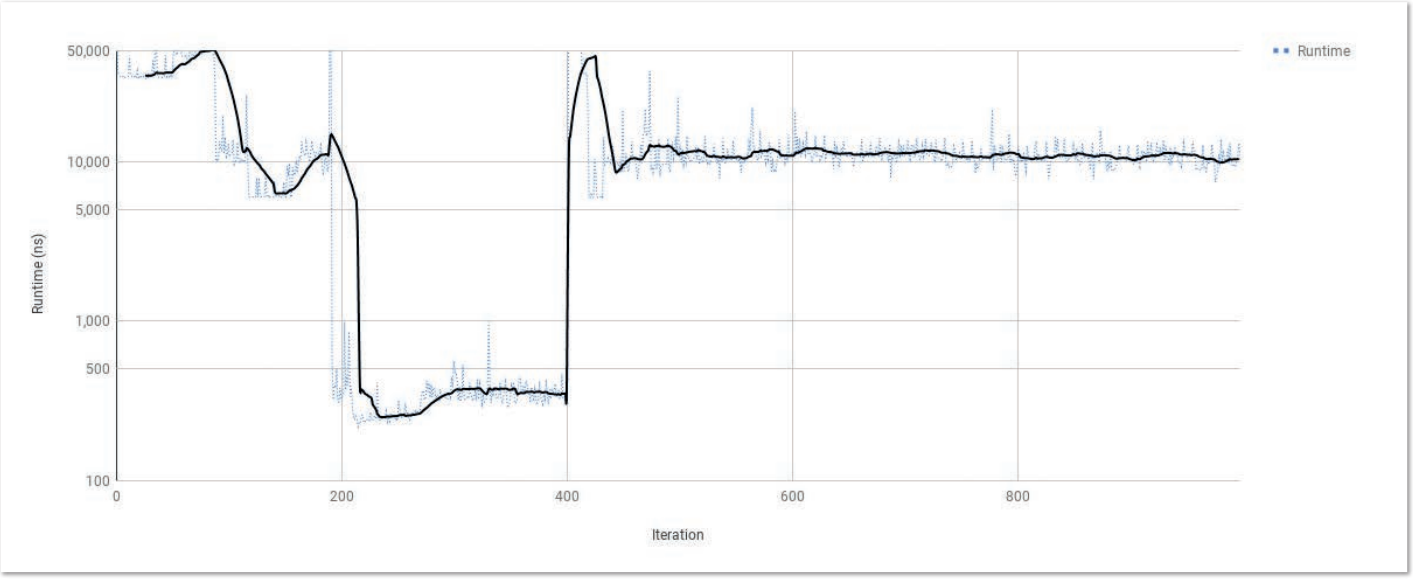


Abbildung 2: AllocationTrap.java Laufzeiten (© René Schwietzke, [4])

Oft sind dazu algorithmische Änderungen nötig, aber so kann man das stark optimierte Laufzeitverhalten erhalten.

Loop Unrolling

Dieses Beispiel zeigt, wie Java Schleifen durch Zerlegen optimiert. Dabei reduziert der Compiler die Anzahl der Schleifendurchläufe und baut dafür die in der Schleife auszuführenden Anweisungen mehrfach in den Anweisungsblock des Schleifenkörpers ein. So kann die JVM Bereichsüberprüfungen und Sprünge einsparen *(siehe Listing 3)*. Die Laufzeiten zeigen, dass die Methode `variable()` etwas mehr als 30 Prozent langsamer ist. Die Gegenprobe mit dem Interpreter beweist, dass der Gesamtcode gleich ist, wenn der Compiler nicht optimiert *(siehe Listing 4)*.

```
private int next()
{
    int i = r.nextInt(1) + 1;
    return i;
}

@Benchmark
public int classic()
{
    int sum = 0;
    int step = next();
    for (int i = 0; i < ints.length; i = i + 1)
    {
        sum += ints[i];
        step = next();
    }
    return sum + step;
}

@Benchmark
public int variable()
{
    int sum = 0;
    int step = next();
    for (int i = 0; i < ints.length; i = i + step)
    {
        sum += ints[i];
        step = next();
    }
    return sum + step;
}
```

Listing 3: Optimizing Loops Example (LoopUnroll.java)

Der JIT-Compiler nutzt hier die Information der konstanten Laufweite von 1 und baut den Code entsprechend um. Für den interessierten Nutzer ist ein Blick in den Code von Hotspot (src/share/vm/opto/loopTransform.cpp) beziehungsweise der GraalVM (org/graalvm/compiler/loop/phases/LoopTransformations.java) zu empfehlen. Die VM zerlegt die Schleife übrigens noch weiter, inklusive der Möglichkeit der Auto-Vectorization von Schleifen (siehe *[12]*).

Inlining und Virtual Methods

Methodenaufrufe stellen Overhead dar, da Parameter bestimmt und übergeben werden müssen und aus Sicht der CPU-Caches auch nicht optimal genutzt werden. Viele Methoden, gerade Getter und Setter sowie statische Hilfsmethoden, werden häufig aufgerufen. Aus diesem Grund bettet der JIT-Compiler Methodencode direkt ein. Er kopiert den Anweisungsblock des Aufrufziels (callee) in die Quelle ein (callsite). Diese Optimierung erfolgt je nach Aufruffrequenz und Größe der Zielmethode. Gleichermaßen können mehrfach verschachtelte Inlines erzeugt werden. Große Methoden werden nicht eingebettet, ebenso beschränkt sich Inlining auf Methoden am Ende des Callstack bis zu einer bestimmten Tiefe.

Da viele Methoden in Java virtuelle Methoden sind, deren Auflösung zur Laufzeit erfolgt, wendet Java hier ähnliche Optimierungen wie bei Dead Code an, um festzustellen, wie viele Varianten einer Methode es gibt. Sollte für lange Zeit keine weitere Variation der Methode existieren, dann wird der Code direkt eingebettet. Sollte der Code in mehr als den bisher bekannten Variationen auftreten, dann kann die JVM auch den Code wieder de-optimieren.

```
Benchmark            (size)  Mode  Cnt       Score  Units
LoopUnroll.classic    10000  avgt    2      18,871  ns/op
LoopUnroll.variable   10000  avgt    2      27,433  ns/op

# Verification of identical behavior with -Xint
Benchmark            (size)  Mode Cnt         Score  Units
LoopUnroll.classic    10000  avgt   2     2,650,812  ns/op
LoopUnroll.variable   10000  avgt   2     2,449,845  ns/op
```

Listing 4: Laufzeiten LoopUnroll.java

```
# -XX:+PrintCompilation
@ 54   java.lang.Math::min (11 bytes)   (intrinsic)
@ 57   java.lang.System::arraycopy (0 bytes)   (intrinsic)
```

Listing 5: Compiler-Debug-Output

Es gibt allerdings noch Grenzfälle, wenn es zwei, drei oder mehrere mögliche Methoden gibt. Normalerweise muss hier die Methode erst über eine Virtual Method Table (VMT) aufgelöst werden. Um diesen Overhead zu reduzieren, bettet Java bei einer Variante den Code direkt ein – bei zwei Varianten über ein einfaches `IF`, das via Branch Prediction optimiert werden kann. Sollte es drei oder mehr Varianten der Methode geben, dann geht Java den klassischen Weg über die VMT. Mehr zum Thema kann man bei Aleksey Shipilëv lesen *[11]*.

Intrinsics

Intrinsics sind optimierter und mitgelieferter Code, auf den Java zurückgreift, um Code schneller zu kompilieren. Wenn die Plattform den bereitliegenden Code unterstützt, dann wird der Intrinsics-Native-Code direkt an die notwendige Stelle kopiert, ohne dass Methoden-Aufrufe oder Native-Calls erfolgen. Intrinsics sind im Java-Quellcode nicht erkennbar. Einige Stellen mit nativem Code (Native-Keyword) werden genauso ersetzt wie purer Java-Code, zum Beispiel `Math.min()`.

Beim Start der JVM wird die Umgebung einschließlich der CPU ausgewertet, um zu entscheiden, welche Library-Methoden direkt ersetzt werden. Einzig der Debug-Output des Compilers gibt Aufschluss darüber *(siehe Listing 5)*.

Escape Analysis

Mit Blick auf CPU und Speicher wird klar, dass die Verwendung des lokalen Stacks effizienter ist als die Nutzung von Heap im Hauptspeicher. Auf dem Stack ist keine GC nötig. Die JVM versucht daher zu vermeiden, dass der Heap benutzt wird, und arbeitet mit CPU-Registern und Stack. Dabei besteht das Problem jedoch darin, zu erkennen, wann ein Objekt aus einer Methode entkommt, also von anderen Objekten auf dem Heap referenziert wird. Ein Escape Analysis genanntes Verfahren versucht, die Gültigkeit von Objekten zu erkennen. *Listing 6* zeigt ein einfaches Beispiel.

Die Methode `array64()` läuft circa 22 ns pro Aufruf, während `array65()` circa 42 ns läuft. Java erkennt, dass das Array „a" nicht außerhalb der Methode genutzt wird, also weder zurückgegeben noch über globale Objekte referenziert wird. Mit der „-prof gc"-Option des JMH sieht man, dass der Heap für `array64()` nicht genutzt wird *(siehe Listing 7)*. Die JVM besitzt ein Array-Limit für lokale Allokationen, das man mit `-XX:EliminateAllocationArraySizeLimit` anpassen kann.

CPU, Memory und Cache

Bei eigenen Optimierungsversuchen hat man sich gegebenenfalls schon gefragt, warum das rechenintensive, wiederholte Parsen ei-

```
@Benchmark
public long array64()
{
    int[] a = new int[64];

    a[0] = r.nextInt();
    a[1] = r.nextInt();

    return a[0] + a[1];
}
@Benchmark
public long array65()
{
    int[] a = new int[65];

    a[0] = r.nextInt();
    a[1] = r.nextInt();

    return a[0] + a[1];
}
```

Listing 6: Lokale Objekte – org/sample/EscapeAnalysis.java

```
array64                          avgt  2    22.804 ns/op
array64:·gc.alloc.rate           avgt  2    ≈ 10⁻⁴ MB/sec
array64:·gc.alloc.rate.norm      avgt  2    ≈ 10⁻⁶ B/op
array64:·gc.count                avgt  2      ≈ 0 counts

array65                          avgt  2    41.890 ns/op
array65:·gc.alloc.rate           avgt  2  5795.571 MB/sec
array65:·gc.alloc.rate.norm      avgt  2   280.000  B/op
array65:·gc.count                avgt  2    93.000 counts
```

Listing 7: GC Profile für Listing 6

```
final int SIZE = 1_000_000;
final int[] src = new int[SIZE];

@Benchmark
public int step1()
{
    int sum = 0;
    for (int i = 0; i < SIZE; i++)
    {
        sum += src[i];
    }

    return sum;
}

@Benchmark
public int step20()
{
    int sum = 0;
    for (int i = 0; i < SIZE; i = i + 20)
    {
        sum += src[i];
    }

    return sum;
}
```

Listing 8: Hauptspeicherzugriff (org.sample.ArraysAndHardware.java)

```
Benchmark  Mode  Cnt    Score  Units
step1      avgt    2  325,159  ns/op
step20     avgt    2   97,402  ns/op
```

Listing 9: Benchmark-Ergebnisse ArraysAndHardware

nes Strings schneller ist, als das Ergebnis zu cachen. Der verringerte CPU-Bedarf müsste doch zu einer kürzeren Laufzeit führen, oder?

In modernen Architekturen ist die CPU um ein Vielfaches schneller als der Speicher. Um Daten aus dem Hauptspeicher zu bekommen, muss die CPU im ungünstigsten Fall bis zu 200 Zyklen warten. Bekommt sie die Daten aus einem L1- bis L3-Cache, dann dauert es zwei bis 60 Zyklen. Wenn man jetzt bedenkt, dass eine moderne CPU pro Zyklus (und das bedeutet nicht Multi-Core!) eine Vielzahl von parallelen Execution-Units nutzt (siehe *[6]*), sodass circa drei bis vier Microbefehle gleichzeitig ausgeführt werden können (solange es keine Datenabhängigkeit gibt), verliert man bei einem einzigen Hauptspeicherzugriff 600 bis 800 mögliche ausgeführte Instruktionen.

Für alle modernen Rechner gilt, dass der Speicherzugriff teuer, aber CPU-Zyklen günstig und reichlich vorhanden sind. Auch diese Tatsache lässt sich für die Optimierung nutzen. Das Beispiel in *Listing 8* zeigt das auf einfache Art. Es summiert die Inhalte eines Arrays. Dabei greift die Methode `step20()` nur auf jedes 20. Element zu. Rein theoretisch sollte also `step20()` etwa 20 Mal schneller sein als `step1()` *(siehe Listing 9)*.

`Step20()` ist jedoch nur etwas mehr als drei Mal schneller, obwohl 20 Mal weniger Speicherzugriffe und Additionen durchführt werden. Ein Blick auf die Hardware-Statistiken erklärt das Problem *(siehe Listing 10)*.

Es ist zu sehen, dass `step1` die CPU besser auslastet (`insn per cycle`), auch wenn die Auslastung vom Optimum (`3.5 insn per cycle`) weit entfernt ist. Nur 6,25 Prozent aller L1-Cache-Zugriffe treffen nicht (`L1-dcache-load-miss`). `Step20()` greift eigentlich immer am L1-Cache vorbei und löst mehr LLC-Zugriffe (Low Level Cache, L2/L3) aus. Ein L1-Hit kostet circa zwei Zyklen, ein L2/L3-Miss ungefähr 20 bis 60 Zyklen. Dieses einfache Beispiel zeigt, dass sich Code nicht immer so verhält, wie man es erwartet, wenn man die Laufzeitumgebung inklusive der CPU und des Speichers ignoriert.

Branch Prediction and Speculative Execution

Wenn man über CPUs redet, sind zwei Features wichtig: Branch Prediction und Speculative Execution – Fähigkeiten der CPU, Code im Voraus auszuführen, um die Pipelines und Execution Units besser auszulasten. Java selbst wendet ebenfalls Branch Prediction an, um Sprünge im Programmcode zu reduzieren und Verzweigungen umzusortieren. *Listing 11* zeigt ein Beispiel, wo die Branch Prediction von Java und der CPU zum Tragen kommen.

Der Test läuft über ein Array mit Zufallszahlen. Abhängig vom Vorzeichen wird Zweig A oder B durchlaufen. Sind die Zufallszahlen sortiert im Array vorhanden, läuft der Code nahezu doppelt so schnell *(siehe Listing 12)*.

Wenn man die technischen Informationen vergleicht, wird klar, dass ein unsortiertes Array mehr Branch-Prediction-Misses hat und damit langsamer läuft. Die Gesamtzahl der Instruktionen ist gleich, aber der Code auf dem unsortierten Array kann die CPU nur zur Hälfte auslasten. Die sortierten Arrays schaffen fast das Optimum an Auslastung der CPU mit 3,2 Instruktionen pro Cycle (IPC).

Aus dem reinen Benchmark geht übrigens nicht hervor, ob die CPU die Vorhersage vorgenommen hat oder ob der JIT-Compiler den

Code optimiert hat und die Reihenfolge der Branches geändert wurde, um Sprünge zu vermeiden (siehe *[9]*).

Synchronization

Der Autor möchte noch kurz auf zwei weitere Techniken, um Kontextwechsel der CPU und Memory-Zugriffe zu vermeiden, eingehen: Lock Elision und Lock Coarsening.

Lock Elision: Wenn der JIT-Compiler Objekte als lokal erkennt, diese jedoch Synchronized-Statements enthalten, dann werden diese Locks nicht ausgeführt, da ohnehin kein paralleler Zugriff möglich ist.

Lock Coarsening: Der JIT-Compiler versucht, nah zusammenliegende Locks des gleichen Objekts zusammenzufassen, um Kontextwechsel der CPU und Schreiboperationen auf den Hauptspeicher zu reduzieren.

Weiterhin darf der JIT-Compiler jederzeit auch Code von außerhalb eines Synchronized-Blocks hineinziehen, wenn es die Performance verbessert.

Wichtiger Hinweis

Die JVM ist absolut frei, mit neuen Versionen ihr Verhalten zu ändern. Der Compiler javac kann anderen Bytecode erzeugen, die JVM kann andere Intrinsics mitbringen, Inlining und Escape Analysis können sich ändern. Deswegen ist es wichtig, Tests mit neuen JDKs und neuer Hardware wiederholt auszuführen und das Ergebnis zu bewerten. In *Listing 13* sind drei Ergebnisse des gleichen Tests auf unterschiedlicher Hardware (IntrinsicsArrayCopy_AverageTime.java) zu sehen.

Garbage Collection kann die Performance verbessern

Garbage Collection wird meist als Problem gesehen, kann jedoch auch überraschend helfen. Der folgende Test zeigt, wie das Layout von Objekten im Speicher die Geschwindigkeit bestimmt (org.sample.GCAndAccessSpeed *[3]*). Im Beispiel werden Strings in der Reihenfolge ihrer Erzeugung in eine Liste gelegt sowie eine zweite Liste mit den gleichen Strings, aber in zufälliger Reihenfolge, erzeugt. Im Test wird die Liste abgelaufen und die Länge aller Strings aufsummiert. Zusätzlich werden zehn Mal mehr Strings erzeugt und gehalten, als in den Listen später genutzt werden.

In *Listing 14* zeigt die Spalte (drop), ob die temporären String-Referenzen aus dem Setup freigegeben wurden, und (gc) zeigt, ob nach dem Aufwärmen des Benchmarks eine Garbage Collection angestoßen wurde.

Reihenfolge: Man kann deutlich sehen, dass das geordnete Ablaufen des Hauptspeichers deutlich schneller ist (*Listing 14*, 1 vs. 2), denn die L1- bis L3-Caches sind optimal gefüllt. Werden die Referenzen gelöst, aber kein GC ausgeführt, dann bleibt das Ergebnis gleich, denn die Caches wissen nichts von den Inhalten des Speichers und darüber, ob die Daten wirklich benötigt werden (*Listing 14*, 5 und 6).

Unordnung: Wird eine GC ausgeführt, aber die temporären Strings weiter gehalten, dann wird auch der geordnete Zugriff langsamer, denn die GC hat die Ordnung, die nach dem Anlegen im Speicher existierte, nicht bewahrt. Damit wird die geordnete Liste aus Sicht des Speicherlayouts auch zu einer ungeordneten Liste (*Listing 14*, 3 und 4).

```
Linux Perf for step1
=============================================================
11,620,571,786  instructions         # 1.04 insn per cycle
 9,482,675,316  L1-dcache-loads      # 2114.210 M/sec
   593,181,935  L1-dcache-load-miss  # 6.26% of all hits
    21,448,491  LLC-loads            # 4.782 M/sec
     5,856,980  LLC-load-misses      # 27.31% of all hits

Linux Perf for step20
=============================================================
11,046,881,256  instructions         # 0.35 insn per cycle
   979,546,079  L1-dcache-loads      # 219.829 M/sec
   860,310,077  L1-dcache-load-miss  # 87.83% of all hits
   746,566,165  LLC-loads            # 167.544 M/sec
   330,798,682  LLC-load-misses      # 44.31% of all hits
```

Listing 10: Hardware-Statistiken JMH mit -prof perf

```
private static final int COUNT = 10_000;

// Contain random numbers from -50 to 50
private int[] sorted;
private int[] unsorted;
private int[] reversed;

public void doIt(int[] array, Blackhole bh1, bh2)
{
    for (int v : array)
    {
        if (v > 0)
        {
            bh1.consume(v);
        }
        else
        {
            bh2.consume(v);
        }
    }
}
```

Listing 11: org.sample.BranchPrediction1.java

```
Benchmark                        Mode         Score   Units
=============================================================
reversed                         avgt        35,182   ns/op
reversed:IPC                     avgt         3.257    #/op
reversed:branch-misses           avgt        13.767    #/op
reversed:branches                avgt    55,339.912    #/op
reversed:cycles                  avgt   110,993.223    #/op
reversed:instructions            avgt   361,508.018    #/op

sorted                           avgt        35,183   ns/op
sorted:IPC                       avgt         3.257    #/op
sorted:branch-misses             avgt        14.738    #/op
sorted:branches                  avgt    55,884.946    #/op
sorted:cycles                    avgt   112,912.763    #/op
sorted:instructions              avgt   367,820.131    #/op

unsorted                         avgt        65,770   ns/op
unsorted:IPC                     avgt         1.577    #/op
unsorted:branch-misses           avgt      3795.737    #/op
unsorted:branches                avgt    56,008.188    #/op
unsorted:cycles                  avgt   211,089.268    #/op
unsorted:instructions            avgt   333,105.231    #/op
```

Listing 12: Runtimes und Branch-Informationen BranchPrediction1.java

Umordnung: Werden allerdings die überflüssigen Objekte de-referenziert und eine GC angestoßen, dann führt die Umordnung der Daten im Hauptspeicher dazu, dass die Zugriffe optimal erfolgen und unnötige Daten (nicht benötigte Objekte) die Caches nicht mehr belasten. Die Objekte sind zusammengerückt.

```
# Lenovo T450s, JDK 11
Benchmark                    Mode  Cnt       Score  Units
IntrinsicsArrayCopy.manual   avgt    2  56,108.312  ns/op
IntrinsicsArrayCopy.system   avgt    2  55,748.067  ns/op
IntrinsicsArrayCopy.arrays   avgt    2  53,769.876  ns/op

# AWS c5.xlarge, JDK 11
Benchmark                    Mode  Cnt       Score  Units
IntrinsicsArrayCopy.manual   avgt    2  83,066.975  ns/op
IntrinsicsArrayCopy.system   avgt    2  63,422.029  ns/op
IntrinsicsArrayCopy.arrays   avgt    2  64,748.500  ns/op

# GCP n1-highcpu-8, JDK 11
Benchmark                    Mode  Cnt       Score  Units
IntrinsicsArrayCopy.manual   avgt    2  82,865.258  ns/op
IntrinsicsArrayCopy.system   avgt    2  62,810.793  ns/op
IntrinsicsArrayCopy.arrays   avgt    2  61,237.489  ns/op
```

Listing 13: Ergebnisse dreier Plattformen, jeweils Linux und JDK11

```
Benchmark          (drop)   (gc)      Score    Units
1:walkNonOrdered    false  false   1,903,731   ns/op
2:walkOrdered       false  false   1,229,945   ns/op

3:walkNonOrdered    false   true   2,026,861   ns/op
4:walkOrdered       false   true   1,809,961   ns/op

5:walkNonOrdered     true  false   1,920,658   ns/op
6:walkOrdered        true  false   1,239,658   ns/op

7:walkNonOrdered     true   true   1,160,229   ns/op
8:walkOrdered        true   true     403,949   ns/op
```

Listing 14: Ergebnisse GCAndAccessSpeed Tests mit G1-GC

Die GC kann das Hauptspeicherlayout verbessern und die CPU-Cache-Effizient steigern. Gleichzeitig werden jedoch Annahmen über die Speicherordnung (siehe *Listing 14*, Messungen 3 und 4) ungültig.

Fazit und Warnung

Performance-Tuning und Benchmarking machen viel Spaß, aber solange man nicht weiß, welche Probleme der Code hat, sollte man nicht auf diesem niedrigen Level optimieren. Um Donald Knuth zu zitieren: „Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." *[8]*. In 90 Prozent der Fälle ist eine algorithmische Optimierung des Programmcodes hilfreicher, um Performanceprobleme zu beheben. Erst wenn diese Möglichkeiten ausgeschöpft sind, dann kann man sich Änderungen zuwenden, die das Wissen über den JIT, die JVM und die Hardware nutzen.

Weiterführende Themen und Material

Dieser Artikel erhebt keinen Anspruch auf Vollständigkeit und soll als Inspiration verstanden werden, sich näher mit den komplexen Themen Compiler, Memory, CPU und Caches zu beschäftigen. Viele Zusammenhänge lassen sich ohne umfangreiche Codelistings und Messergebnisse schwierig darstellen.

Empfohlen seien an dieser Stelle Douglas Q Hawkins Talks auf YouTube *[5]*. Dough ist ein Core-JVM-Entwickler. Das Buch „Optimizing Java" ist ebenfalls sehr empfehlenswert *[6]*, da es detailliert auf Themen rund um Compiler, GC und JVM- Interna eingeht. Die Ausgangsbasis dieses Artikels als vollständige Präsentation *[4]* liefert mehr Messergebnisse und mehr Codebeispiele *[3]*.

Quellen

[1] Energy Efficiency across Programming Languages: How does Energy, Time and Memory Relate?, accepted at the International Conference on Software Language Engineering (SLE) - Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva, 2017; *https://sites.google.com/view/energy-efficiency-languages/home*

[2] *https://openjdk.java.net/projects/code-tools/jmh/*

[3] GIT Repository für den Code dieses Artikels: *https://github.com/Xceptance/jmh-jmm-training*

[4] René Schwietzke, High Performance Java Präsentation, 2019: *https://training.xceptance.com/java/420-high-performance.html*

[5] Douglas Q Hawkins, Java Performance Puzzlers, Azul Systems, 2017: *https://www.youtube.com/watch?v=wgQBz2Ldhvk*

[6] Evans/Gough/Newland, Optimizing Java, O'Reilly Media, 2018

[7] *https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake*

[8] Donald Knuth, Turing Award Lecture 1974

[9] Java on Steroids: 5 Useful JIT Optimization Techniques; *https://blog.overops.com/java-on-steroids-5-super-useful-jit-optimization-techniques/*

[10] An Introduction to Epsilon GC, Attila Fejér, *https://www.baeldung.com/jvm-epsilon-gc-garbage-collector*

[11] Aleksey Shipilëv, JVM Anatomy Quark #16: Megamorphic Virtual Calls *https://shipilev.net/jvm/anatomy-quarks/16-megamorphic-virtual-calls/*

[12] Fasih Khatib, JVM JIT - Loop Unrolling, *http://fasihkhatib.com/2018/05/20/JVM-JIT-Loop-Unrolling/*

René Schwietzke
Xceptance GmbH
r.schwietzke@xceptance.com

René Schwietzke ist Mitgründer und Geschäftsführer der Xceptance GmbH. Er arbeitet seit der Version 1.0 mit Java und beschäftigt sich seit 2004 intensiv mit den JVM-Interna für Analyse und Performance-Tuning von SaaS-Commerce-Applikationen. Das Xceptance-Lasttest-Tool XLT basiert ebenfalls auf Java. Effiziente und zuverlässige Ausführungen großer Tests machten es nötig, die JVM genau zu verstehen.

Immutable Java

Nicolai Mainiero, sidion

Viele Java-Entwickler schauen neidisch zu den Kollegen, die mit funktionalen Sprachen wie Scala oder Clojure Software schreiben. Diese müssen sich viel weniger Gedanken darüber machen, ob ihre Anwendungen auch dann noch korrekt sind, wenn sie parallel ausgeführt werden oder ob sie eine Datenstruktur bedenkenlos an eine Bibliothek übergeben können. Diese und noch weitere Vorteile ergeben sich daraus, dass die funktionalen Kollegen unveränderliche Datenstrukturen zur Verfügung haben. In diesem Artikel geht es darum, welche Vorteile sich daraus ergeben, wie unveränderliche Datenstrukturen in Java realisiert werden können und was es dabei zu beachten gibt.

Wenn man einen Java-Entwickler nach immutable Objekten in der Standardbibliothek fragt, fällt diesem wahrscheinlich sofort die `String`-Klasse ein. Dass es sich hierbei nicht um die einzige Klasse handelt, vergessen viele. Neben der `String`-Klasse sind auch alle „geboxten" primitiven Typen, wie zum Beispiel `Integer`, `Boolean` oder `Float`, immutable. Das moderne „Date and Time API" von Java, auch bekannt unter JSR-310 *[1]*, wurde explizit unter dem Gesichtspunkt der Immutability designt und implementiert. Darüber hinaus gibt es noch viele weitere in der Standardbibliothek verstreute Klassen, die immutable sind. Welche Vorteile ergeben sich nun daraus?

Vorteile von immutable Klassen

Immutable Objekte können Code vereinfachen und leichter verständlich machen. Sie führen oft auch zu einfacherem, sicherem und korrekterem Code, sind zuverlässige Schlüssel in HashMaps, sind Thread-safe und müssen nie kopiert oder geklont werden. In *Listing 1* ist das API eines Scheduler zu sehen, der eine Aufgabe (`task1`, `task2`) und einen Zeitpunkt, wann die entsprechende Aufgabe auszuführen ist, als Parameter akzeptiert. Je nach Implementierung kann der Scheduler die Aufgaben tatsächlich um einen Tag versetzt ausführen oder beide Aufgaben werden zum selben Zeitpunkt ausgeführt. Hier ist es also nötig, die Implementierung des Scheduler genau zu kennen, um zu beurteilen, ob der Code sich richtig verhält.

```
Date d = new Date();
Scheduler.scheduleTask(task1, d);
d.setTime(d.getTime() + ONE_DAY);
Scheduler.scheduleTask(task2, d);
```

Listing 1: Probleme beim Teilen von veränderbaren Objekten

Auch bei der Programmierung mit Threads kann man mit mutable Objekten Fehler machen, die nicht sofort offensichtlich sind. *Listing 2* demonstriert ein typisches Problem von nicht synchronisiertem Code.

In diesem Fall ist die `Sampler`-Klasse veränderbar und wird von beiden Threads gleichzeitig gelesen und beschrieben. Das führt dazu, dass die `incrementValue()`-Methode zwar insgesamt zwei Millionen Mal aufgerufen wird, der Wert von `value` aber in den meisten Fällen unter zwei Millionen bleiben wird. Ein etwas ungewöhnlicheres Problem, bei dem ein immutable Objekt Fehler verhindert, ist in *Listing 3* zu sehen. Wenn ein veränderbares Objekt als Schlüssel für eine HashMap verwendet wird, kann

```
public class Threads {

    public static void main(String[] args) throws InterruptedException {
        Sampler sampler = new Sampler();

        Thread thread1 = new Thread(new SamplerIncRunnable(sampler));
        thread1.start();

        Thread thread2 = new Thread(new SamplerIncRunnable(sampler));
        thread2.start();

        thread1.join();
        thread2.join();

        System.out.println(sampler.getValue());

    }

    static class SamplerIncRunnable implements Runnable {
        private Sampler sampler;

        public SamplerIncRunnable(Sampler sampler) {
            this.sampler = sampler;
        }

        public void run() {
            for ( int i=0; i<1_000_000; i++ ) {
                sampler.incrementValue();
            }
        }
    }

    static class Sampler {
        private int value = 0;

        void incrementValue(){
            this.value++;
        }

        public int getValue(){
            return this.value;
        }
    }
}
```

Listing 2: Probleme mit Nebenläufigkeit bei veränderbaren Objekten

es sein, dass nach einer Änderung des Felds des Schlüssels der zugehörige Wert nicht mehr gefunden wird. HashMap verwendet das Ergebnis der `hashCode()`-Methode, um den internen Schlüssel zu bestimmen.

Das Ändern des Namens der Person `jim` führt zu einer Änderung des `hashCode` dieses Objekts. Dadurch wird der Wert mit diesem Objekt als Schlüssel nicht mehr gefunden. Das Auflisten aller Einträge in der HashMap zeigt jedoch, dass das Objekt sich noch in der HashMap befindet.

Immutable Objekte erstellen

Die Vorteile von immutable Datenstrukturen sind nachvollziehbar und es bietet sich an, sie im täglichen Gebrauch zu nutzen. Java liefert alle notwendigen Mittel, um ein Objekt immutable zu definieren. In *Listing 4* sind zwei Klassen definiert, `Person` und `Address`. Eine Person hat einen Namen und eine Liste von Adressen. Die Adresse besteht in diesem vereinfachten Beispiel nur aus den Feldern „Stadt" und „Land".

Die Klassen enthalten Getter und Setter, wie sie üblich sind und von jeder IDE erzeugt werden können. Um eine Klasse immutable zu machen, sind folgende Schritte notwendig:

1. Deklariere die Klasse als `final`
2. Mache alle Felder `private` und `final`
3. Verhindere, dass der interne Zustand veränderbar nach außen gegeben wird
4. Mache im Konstruktor Kopien von veränderbaren Datenstrukturen

Listing 5 zeigt dieselben Klassen, nachdem diese vier Regeln angewendet worden sind, um die daraus entstehenden Objekte immutable zu machen. Den wesentlichen Teil der Arbeit kann uns hier die IDE abnehmen, wenn wir alle Felder der Klasse korrekt mit `final` deklariert haben. Dann werden zum Beispiel keine Setter mehr erzeugt und der erzeugte Konstruktor enthält alle Felder auch als Parameter.

Erwähnenswert in *Listing 5* ist der Konstruktor der Person, in dem mit `List.copyOf()` eine unveränderliche Kopie der übergebenen Liste erstellt wird. Diese Methode ist seit Java 10 verfügbar und ist der Methode `Collections.unmodifiableCollection()` vorzuziehen. Letztere liefert nur eine unveränderliche Ansicht auf die übergebene Collection zurück. Das bedeutet, dass wenn sich die darunterliegende Collection ändert, sich diese Änderung auf die erzeugte Ansicht auswirkt. In Java 9 kann man auf das Statement `List.of(collection.toArray())` ausweichen. Dies entspricht bis auf eine Speicheroptimierung der Implementierung von `List.copyOf()`.

```
import java.util.HashMap;
import com.google.common.truth.Truth;

public final class BadHashKey {

  final static class Person {
    private String name;

    public Person(String name) {
      this.name = name;
    }

    public String getName() {
      return name;
    }

    public void setName(String name) {
      this.name = name;
    }

    public int hashCode() {
      return name.hashCode();
    }

    public boolean equals(Object otherObj) {
      Person otherPerson = (Person) otherObj;
      return (name.equals(otherPerson.getName()));
    }
  }
    public static void main(String[] args) {
        Person jim  = new Person("James Holden");
        HashMap<Person, String> hashMap = new HashMap<>();
        hashMap.put(jim, "ice freighter");
        Truth.assertThat(hashMap.get(jim)).contains("ice freighter");
        jim.setName("Julie Mao");
        hashMap.entrySet().forEach(
          e -> System.out.println(e.getKey().getName() + ", " + e.getValue())
        );
Truth.assertThat(hashMap.get(jim)).contains("ice freighter"); // fails
    }
}
```

Listing 3: Probleme mit veränderbaren Objekten als Schlüssel

Veränderungen während der Softwareentwicklung

Immutable Objekte haben viele Vorteile und sind mit Java-Bordmitteln leicht zu realisieren. Allerdings erfordern sie ein gewisses Umdenken bei der Softwareentwicklung. Durch das bewusste Entfernen der Setter können bestehende Objekte nicht mehr manipuliert werden. Um dennoch Änderungen an Objekten, also neue Objekte mit geänderten Werten, vorzunehmen, wird eine andere Methode erforderlich. Üblich ist, die sogenannten Wither-Methoden als Pendants zu Settern zu schreiben, um eine Kopie eines Objekts zu bekommen, in dem ein Feld geändert worden ist. In *Listing 6* wird dieses Pattern vorgestellt. Das Prinzip dieser Wither-Methoden ist simpel: Für jedes Feld der Klasse wird eine `withX()`-Methode erzeugt. In dieser wird ein neues Objekt vom selben Typ erstellt und alle Felder bis auf X werden aus dem aktuellen Objekt übernommen. Der neue Wert für X wird der Methode als Parameter übergeben.

```
import java.util.List;

public class Person {
    private String name;
    private List<Address> addresses;

    public Person() {
        super();
    }

    public Person(String name, List<Address> addresses) {
        this.name = name;
        this.addresses = addresses;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }


    public List<Address> getAddresses() {
        return addresses;
    }

    public void setAddresses(List<Address> addresses) {
        this.addresses = addresses;
    }
}

public class Address {

    private String city;
    private String country;

    public Address() {
        super();
    }

    public Address(String city, String country) {
        this.city = city;
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

Listing 4: Zwei einfache veränderbare Klassen

Das Erzeugen neuer Objekte wie hier ist nicht langsamer, als ein bestehendes Objekt zu verändern. Tatsächlich sind die aktuellen Garbage Collector darauf optimiert, mit einer großen Menge kurzlebiger Objekte umzugehen. Dieser Code-Stil hat daher kaum Auswirkung auf die Performance *[3]*. Eine weitere Anpassung ist die konsequente Verwendung des Builder-Patterns bei Objekten mit mehr als drei Feldern. Durch die Verwendung von benannten Methoden zur Initialisierung der Objekte werden Fehler, zum Beispiel durch eine falsche Reihenfolge der Parameter, vermieden. Wenn gewünscht, kann dieser Stil auch mithilfe des Projekts Lombok *[4]* erreicht werden. Es müssen dann folgende Annotationen verwendet werden, um dasselbe Ergebnis zu erreichen. Erstens `@Value`, um die Klasse immutable zu machen. Lombok deklariert dann bei der Erzeugung der Implementierung alle Felder `private` und `final`, egal wie sie im Quellcode definiert sind. Die Klasse wird ebenfalls `final` markiert. Zweitens `@Builder`, um einen Builder für die Klasse zu erzeugen. Dieser macht den Konstruktor von `@Value` dann ebenfalls `private`, da dieser dann nur noch vom Builder verwendet werden soll. Und zuletzt `@With`, um die Wither-Methoden zu erzeugen, die es erlauben, Kopien des Objekts mit einzelnen geänderten Feldern zu erzeugen.

Immutable Objekte mit Immutables

Allerdings gibt es auch Vorbehalte gegenüber Lombok, da es den Quellcode während des Kompilierens erzeugt beziehungsweise verändert. Da diese Änderungen durch nicht öffentliche APIs gemacht werden, kann es sein, dass es bei einem neuen JDK Release nicht mehr funktioniert. Ebenfalls muss für jede IDE eine Extra-Erweiterung geschrieben werden, damit Funktionen wie Auto-Complete nutzbar sind.

Immutables *[5]* versucht, genau diese Probleme zu vermeiden und einfache, sichere und konsistente immutable Objekte zu erzeugen. Dazu integriert es sich als Annotation-Prozessor und erzeugt aus annotierten Interfaces konkrete Implementierungen, die dann eben auch inspiziert werden können. Es ist dasselbe Prinzip, das auch MapStruct *[6]* verwendet. In *Listing 7* ist zu sehen, wie kompakt sich ein Mitarbeiter mit Namen und Alter mit Immutables definieren lässt.

Der Annotation-Prozessor erzeugt aus dieser Definition neben einem `Builder` für jedes Attribut eine Wither-Methode und natürlich sinnvolle `hashCode()`-, `equals()`- und `toString()`-Methoden. Die Verwendung der so erzeugten Klassen ist einfach und unkompliziert, wie in *Listing 8* zu sehen ist.

Einen Nachteil darf man dennoch nicht verschweigen. Beim Zusammenspiel mit APIs von anderen Frameworks muss man oft wieder auf veränderbare Objekte ausweichen. So müssen zum Beispiel alle Entitäten bei JPA einen Default-Konstruktor besitzen. Dies verhindert, dass alle Felder als `final` deklariert werden können, und macht die Klasse mutable. Hier bietet es sich an, eine klare Grenze zwischen den unveränderlichen und den veränderlichen Objekten zu definieren.

```
public class Address {

    private final String city;
    private final String country;

    public Address(String city, String country) {
        this.city = city;
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public String getCountry() {
        return country;
    }
}
public class Person {
    private final String name;
    private final List<Address> addresses;

    public Person(final String name, final List<Address> addresses) {
        this.name = name;
        this.addresses = List.copyOf(addresses); // <1>
    }

    public String getName() {
        return name;
    }

    public List<Address> getAddresses() {
        return addresses;
    }
}
```

Listing 5: Immutable Person und Address

```
public class Person {
    private final String name;
    private final String address;


    public Person(final String name, final String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public Person withName(final String newName){
        return new Person(newName, address);
    }
}
```

Listing 6: Immutable Objekt mit Wither-Methoden

```
@Value.Immutable
abstract static class Employee {
public abstract String name();
public abstract Integer age();
}
```

Listing 7: Immutable Employee mit Immutables

Fazit

In „Effective Java“ *[7]* schreibt Josuha Bloch: „Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, limit its mutability as much as possible.“ Genau das sollte auch der Anspruch sein, wie in Java mit Immutability umgegangen wird. So kann man einen Anwendungskern entwickeln, der die Fachlogik enthält und immutable ist. Um diesen herum werden die Bibliotheken von Dritten, wie zum Beispiel JPA oder eine REST-Schnittstelle, angeordnet. Dieses Vorgehen führt zu einer sauberen Architektur, die von den Vorteilen unveränderlicher Klassen profitiert, ohne dass Workarounds gebaut werden müssen, um nicht immutable Bibliotheken an den eigenen Code anzupassen.

Quellen

[1] JSR 310: Date and Time API (2014), *https://jcp.org/en/jsr/detail?id=310*

[2] The Java Tutorials > Immutable Objects, *https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html*

[3] Brian Goetz (2003): Urban performance legends, *https://www.ibm.com/developerworks/library/j-jtp04223/index.html*

[4] Projekt Lombok, *https://projectlombok.org/*

[5] Immutables, *https://immutables.github.io/*

[6] MapStuct, *https://mapstruct.org/*

[7] Joshua Bloch (2017): Effective Java. Addison-Wesley Professional, Boston

[8] Quellcode unter *https://github.com/nicolaimainiero/immutable-java*

Nicolai Mainiero

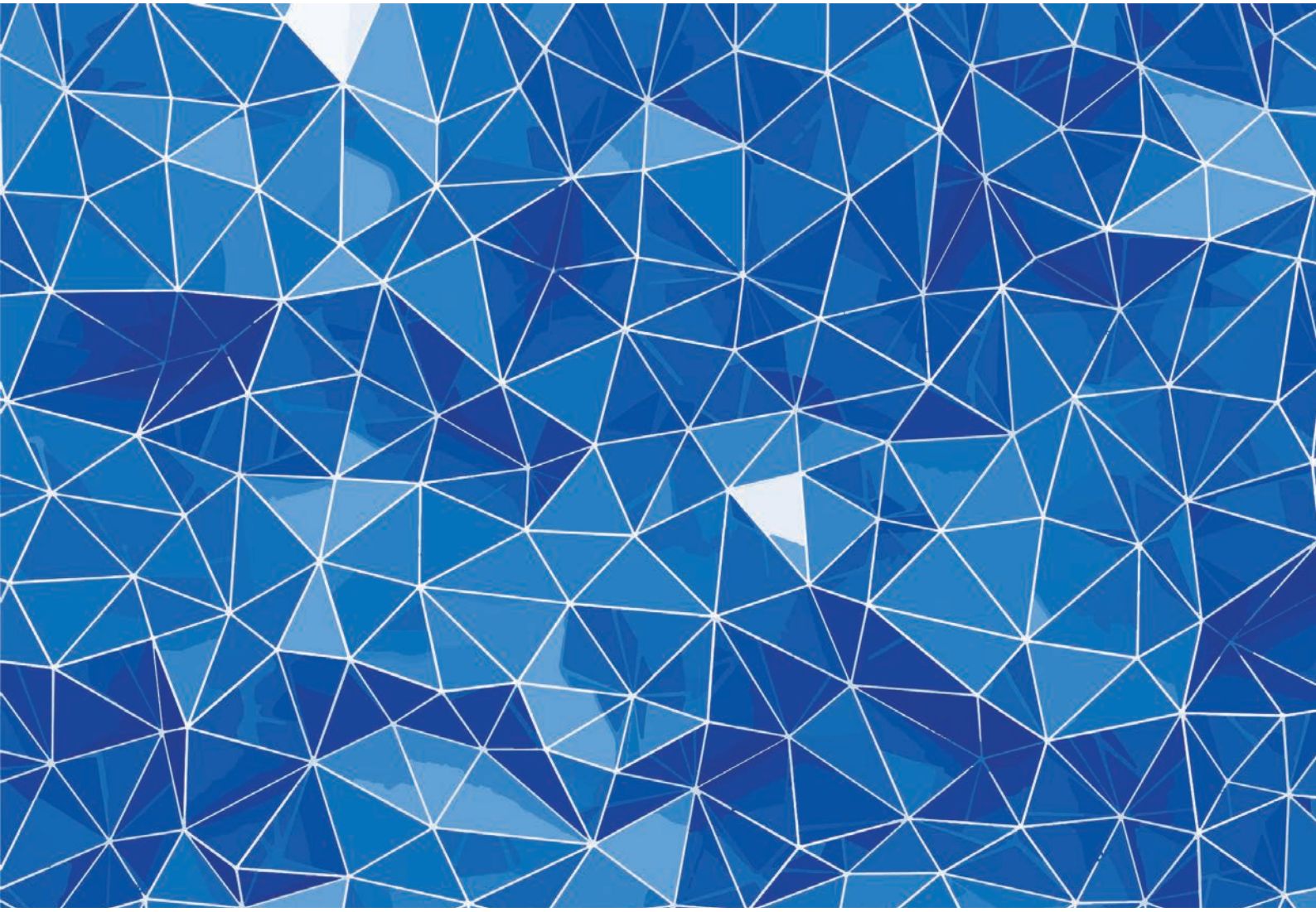
sidion

nicolai.mainiero@sidion.de

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

```
final Employee julie = ImmutableEmployee.builder().name("Julie Mao").age(28).build();
final Employee james = ImmutableEmployee.copyOf(julie).withName("James Holden");
Truth.assertThat(julie.age()).isEqualTo(28);
Truth.assertThat(james.name()).isEqualTo("James Holden");
```

Listing 8: Immutable Employee mit Immutables



Native Images mit GraalVM

Bernd Müller, Ostfalia

Dieser Artikel gibt einen ersten Einblick in die Erzeugung nativer Images mit der GraalVM. Neben der Möglichkeit, in verschiedenen Programmiersprachen erstellte Programmteile in ein und derselben virtuellen Maschine ausführen zu können, ist die Erzeugung nativer Images einer der Gründe, warum die GraalVM innerhalb kürzester Zeit so viel Aufmerksamkeit erlangt hat. Der Autor stellt die Grundlagen der Erzeugung nativer Images vor und zeigt, was schon geht – aber auch, was nie gehen wird. Neben diesem Artikel wird es noch weitere geben, da das Thema „native Images mit der GraalVM" sehr facettenreich ist.

Oracle, die Firma hinter der GraalVM, beschreibt das Produkt als *„A universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Groovy, Kotlin, Clojure, and LLVM-based languages such as C and C++"*. Neben den schon immer in Bytecode übersetzten Sprachen Java, Scala, Groovy, Kotlin und Clojure ist die Möglichkeit einer gemeinsamen Ablaufumgebung für zusätzlich JavaScript, Python, Ruby, R sowie C und C++ das Alleinstellungsmerkmal dieser virtuellen Maschine. Für die beiden letztgenannten Sprachen gilt dies allerdings nur, falls sie nicht nativ, sondern in LLVM-Code übersetzt wurden. Die Möglichkeit des Kompilierens von Java in nativen Code ist in der Standarddistribution der GraalVM nicht enthalten, kann aber als optionale Komponente nachinstalliert werden, genauso wie die Interpreter für Python, Ruby und R.

GraalVM: Überblick und Installation

GraalVM ist mit dem Slogan „Run Programs Faster Anywhere" *[1]* zu finden und wird als Open-Source-Projekt *[2]* entwickelt. Wenn es allerdings um den Download geht, wird zwischen Community und Enterprise Edition unterschieden. Die letztgenannte Variante ist dann nicht mehr quelloffen und muss über das Oracle Technology Network heruntergeladen werden. Für Evaluationszwecke, wie für diesen Artikel, ist die Verwendung jedoch frei möglich und wird von uns so praktiziert. Wir verwenden die aktuelle Version 19.2.1 für Linux. Die heruntergeladene Datei ist ein komprimiertes TAR, das an beliebiger Stelle entpackt wird. Wie bereits erwähnt, ist die Native-Image-Erzeugung eine optionale Komponente und muss nachinstalliert werden. Nachdem diese Komponente als JAR-Datei heruntergeladen wurde, wird sie mit dem GraalVM Component Updater (gu) installiert, und zwar mit `gu -L install <component-jar>`. Das Programm `gu` befindet sich im bin-Verzeichnis der Installation, so wie die meisten Befehle eines normalen JDK, etwa `javac`, `java`, `jar` etc. In der GraalVM Version 19.2.1 ist Java noch in der Version 8 enthalten, an der Version 11 wird gearbeitet. Nach dieser Aktualisierung steht mit `native-image` ein weiterer Befehl im bin-Verzeichnis zur Verfügung und wartet auf seinen ersten Einsatz.

Zunächst ein wenig Java-Compiler-Geschichte

In der Java aktuell 04/2019 *[3]* haben wir Javas Just-in-time-Compiler beleuchtet. Die beiden JIT-Compiler, C1 und C2 genannt, sind etwas in die Jahre gekommen und Oracle arbeitet an einem Nachfolger mit dem Namen „Graal". Während C1 und C2 wie fast alle Module der HotSpot-Virtual-Machine in C++ geschrieben sind, ist Graal in Java implementiert. Grund hierfür waren die Erfahrungen mit C1 und C2, die praktisch nicht mehr wartbar waren und deshalb neu implementiert werden sollten. Historisch gesehen starteten die Compiler-Arbeiten im OpenJDK zunächst mit dem JEP 296: *Ahead-of-Time Compilation* *[4]* bereits im Jahr 2016. Ziel war es, Bytecode vor dem Start der JVM in Maschinencode umzuwandeln. Auf Basis dieses JEP wurde 2017 der JEP 317: *Experimental Java-Based JIT-Compiler* *[5]* formuliert. Der zu entwickelnde Compiler wurde Graal getauft. Sowohl der AoT-Compiler, der Graal als Code-Erzeugungs-Backend verwendet, als auch Graal selbst sind seit Java 9 in der Linux-Version des OpenJDK enthalten, bei Windows ab Version 10.

Spracheigenschaft	Unterstützung
Dynamic Class Loading/Unloading	Not supported
Reflection	Supported (Requires Configuration)
Dynamic Proxy	Supported (Requires Configuration)
Java Native Interface	Mostly supported
Unsafe Memory Access	Mostly supported
Class Initializers	Supported
InvokeDynamic & Method Handles	Not supported
Lambda Expressions	Supported
Synchronized, wait and notify	Supported
Finalizers	Not supported
References	Mostly supported
Threads	Supported
Identity Hash Code	Supported
Security Manager	Not supported
JVMTI, JMX, other native VM interfaces	Not supported
JCA Security Services	Supported

Tabelle 1: Einschränkungen der Native-Image-Erzeugung [7]

Wie geht AoT-Kompilierung?

Der neue Befehl jaotc (Java Ahead-of-Time Compiler) im `bin`-Verzeichnis des JDK erlaubt die Kompilierung einer Klasse oder eines Moduls in nativen Bibliothekscode, also `.so` unter Linux bzw. `.dll` unter Windows. Das Kompilieren einer Klasse erfolgt mit: `jaotc --output libhelloworld.so HelloWorld`. Voraussetzung ist hier, dass der Java-Quellcode der Klasse zuvor in Bytecode übersetzt wurde. Die entstehende Datei ist bei einem normalen Hello-World-Programm etwa 420 Bytes groß. Die Option `--verbose` führt dazu, dass die kompilierten Methoden ausgegeben werden, hier der Default-Konstruktor und die `main()`-Methode.

Um das Kompilat zu verwenden, muss mit `java` die Verwendung der Bibliothek angezeigt werden: `java -XX:AOTLibrary=./libhelloworld.so HelloWorld`. Das Interessante dabei ist die interne Funktionsweise. Die Bytecode-Datei muss weiterhin vorhanden sein, da die JVM versucht, diese zu laden. Beim Laden werden dann die Methodendeskriptoren auf den Code in der Bibliothek umgelenkt. Die Angabe der Option `-XX:+PrintAOT` beim letzten Befehl macht dies explizit.

Wie oben erwähnt, können nicht nur Klassen, sondern auch Module AoT-kompiliert werden. So kann etwa das gesamte Basismodul mit `jaotc --output libjavabase.so --module java.base` kompiliert werden. Das Kompilieren ist durchaus zeitaufwendig und resultiert in einer Bibliotheksgröße von 323 MB. Durch eine Compiler-Option, die die zu kompilierenden Dateien einschränkt, kann dies reduziert werden. Die Option `jaotc ... --compile-commands compileCommands.txt` verweist auf die Datei compileCommands.txt, die lediglich `compileOnly java.lang.*` als Inhalt enthält. Das Kompilieren benötigt dann nur noch einen Bruchteil der ursprünglichen Zeit, die Bibliotheksgröße reduziert sich auf 40 MB. Beim Programmstart müssen beide Bibliotheken verwendet werden: `java -XX:AOTLibrary=./libhelloworld.so,./libjavabase.so HelloWorld`.

Nachdem das OpenJDK bereits seit Version 9 ahead-of-time kompilieren kann, muss jetzt im Hinblick auf die Erzeugung nativer Images nur noch verhindert werden, dass komplette Module – im Worst Case sogar alles, was sich im Klassenpfad befindet – übersetzt werden, um ein natives Image zu erzeugen. Kein leichtes Unterfangen, wie wir sehen werden.

Wie funktioniert die Native-Image-Erzeugung mit der GraalVM?

Szenenwechsel: GraalVM. Beim Erzeugen eines nativen Image mit der GraalVM analysiert der Compiler die Klassen der Anwendung einschließlich aller Abhängigkeiten. Diese Analyse findet zur Compile-Zeit statt und kann daher Informationen, die die HotSpot VM nur zur Laufzeit hat, nicht verwenden. Grundlage der Analyse ist eine Closed-World-Assumption: Alles, was in der statischen Analyse erreichbar ist, wird auch verwendet, sprich kompiliert. Was nicht

erreichbar ist, wird nicht kompiliert. Falls etwa `Class.forName()` über einen Ausdruck lädt, der erst zur Laufzeit feststeht, ist die so zu ladende Klasse nicht zur Compile-Zeit bestimmbar und wird daher nicht kompiliert. Es ist in diesem Fall allerdings möglich, den Klassennamen dem `native-image`-Befehl als Parameter zu übergeben und somit auch zu kompilieren, wie wir in einem späteren Beispiel sehen. Da Java eine hochdynamische Sprache ist und viele Dinge erst zur Laufzeit feststehen, können eine ganze Reihe von Spracheigenschaften nicht statisch kompiliert werden. *Tabelle 1* gibt einen Überblick darüber, wie weit verschiedene Spracheigenschaften von Java bei der nativen Image-Erzeugung unterstützt werden.

Was kann ein natives Image?

In *Tabelle 1* sind einige Spracheigenschaften mit *„not supported"* gekennzeichnet. Dies ist in großen Teilen nicht nur eine Momentaufnahme des aktuellen Zustands, sondern als endgültige Entwurfsentscheidung zu verstehen. Das bedeutet, dass ein Java-Programm, das mit der GraalVM in ein natives Image übersetzt wurde, nicht alle Möglichkeiten Javas unterstützt. Etwas härter ausgedrückt: Die Sprache, die durch native Images unterstützt wird, ist *nicht* Java, sondern eine echte Teilmenge von Java!

Das hört sich jetzt erst einmal sehr schlimm an, ist es aber in der Praxis nicht unbedingt, wie viele Beispiele zeigen. Quarkus, Heldion, Micronaut, Spring Boot, Spark und weitere setzen auf die GraalVM beziehungsweise überlegen, wie weit sie dabei gehen wollen. Sie müssen sich daher einschränken, dürfen also nicht den gesamten Sprachumfang von Java nutzen. Wie es aber scheint, ist dies kein K.o.-Kriterium und tut der Popularität der GraalVM keinen Abbruch. Um die notwendigen Beschränkungen besser verstehen zu können, muss man sich klarmachen, wie ein natives Image und dessen Erzeugung funktioniert. Die Definition von Java erfolgt nicht ausschließlich über die Spezifikation der Sprache *[8]*, sondern auch über die Spezifikation der virtuellen Maschine *[9]*. Existiert diese nicht, entfällt ein großer Teil der Spezifikation von Java und natürlich auch der Möglichkeiten von Java. So ist zum Beispiel das Kapitel 5 der JVM-Spezifikation *„Loading, Linking, and Initializing"* hinfällig. Bei einem nativen Image gibt es keine HotSpot VM, die diesem Kapitel fünf gehorcht, sondern die sogenannte Substrate VM. Diese realisiert grundlegende Dinge wie Speicher-Management, Thread Scheduling und Garbage Collection, nicht aber das Laden von Klassen. Die Substrate VM ist selbst in Java geschrieben und wird in das native Image kompiliert. Das besagte Kapitel fünf beschreibt, wie Bytecode in die JVM (HotSpot, nicht Substrate) geladen, gelinkt und initialisiert wird. Die Methode `loadClass()` der Klasse `ClassLoader` und deren Unterklassen erzeugen aus einem Byte-Array eine Instanz der Klasse `Class`. Damit kann eine Substrate VM nichts anfangen. Darum das *„not Supported"* in der ersten Zeile von *Tabelle 1*.

Die ersten Gehversuche mit `native-image`

Nachdem der Autor den Leser nun eventuell etwas desillusioniert hat, beginnen wir mit den motivierenden Dingen, dem obligatorischen „Hello World". Dies kann mit `native-image HelloWorld` übersetzt werden. Es fällt auf, dass dies der Klassenname und nicht

```
public class HelloWorld {

  public static void main(String[] args) throws Exception {
    Method method = HelloWorld.class
        .getDeclaredMethod("helloWorld", new Class[] {});
        method.invoke(null, new Object[] {});
  }

  private static void helloWorld() {
    System.out.println("Hello World");
  }
}
```

Listing 1: Hello World mit Reflection

der Dateiname ist. Der Befehl `native-image` arbeitet auf Bytecode, nicht auf Java-Quellcode, sodass zuvor der normale Java-Compiler (`javac`) aufzurufen ist. Optional kann als weiterer Parameter der Image-Name angegeben werden, der im Default-Fall zum kleingeschriebenen Klassennamen wird. Das erzeugte Image ist unter Linux ein normales, dynamisch gelinktes ELF-Executable mit einer Größe von 2,4 MB. Für eine Quellcode-Größe von 116 Bytes (wir verzichten auf die Angabe des Hello-World-Quell-Codes) recht beachtlich. Dies relativiert sich jedoch etwas, wenn man sich klarmacht, dass die Substrate VM mit Speicher-Management, Thread Scheduling und Garbage Collection enthalten ist. Zusätzlich müssen alle Klassen, die transitiv erreichbar sind, kompiliert und gelinkt werden. Ein Hello-World verwendet auf oberster Ebene lediglich `java.lang.String` und `java.lang.System`. Der interessierte Leser kann einen Blick in die Import-Listen dieser beiden Klassen werfen, um einschätzen zu können, wie viele Imports diese beiden Klassen enthalten und wie die transitive Hülle aussehen könnte.

Mit der Option `--shared` kann eine (shared) Bibliothek, also eine `.so`-Datei erzeugt werden. Zusätzlich werden auch noch Include-Dateien für C bzw. C++ erzeugt. Wir gehen hierauf jedoch nicht ein und zeigen daher nicht, wie der mögliche Aufruf einer Java-Methode aus C bzw. C++ heraus ohne die Verwendung von JNI möglich ist.
Die Option `--static` erzeugt ein statisch gelinktes ELF-Executable mit einer Größe von 4,7 MB. Diese Form des Executable ist je nach Konfiguration des Linux-Systems zu empfehlen, da neben den typischen Verdächtigen, wie etwa `libc` und `libm`, auch `libz` verwendet wird, die eventuell nicht auf allen Systemen vorhanden ist.

Reflektives Hello World

In *Tabelle 1* ist vermerkt, dass die Verwendung von Reflection ein gewisses Maß an Konfigurationsaufwand benötigt. Dies ist für den allgemeinen Fall korrekt, der Trivialfall funktioniert aber ohne weiteres Zutun. Der in *Listing 1* gezeigte Code bildet ein Hello World mit Reflection ab. Dieser Code kann ganz normal kompiliert und ausgeführt werden. Eine Konfiguration ist nicht vonnöten. Der Autor schließt hier seine ersten Beispiele zur Erzeugung nativer Images mit der GraalVM und verweist auf seinen nächsten Artikel in einer zukünftigen Ausgabe der Java aktuell.

Zusammenfassung

Nach einem kurzen Überblick über die GraalVM haben wir uns mit Java-Compilern beschäftigt; dem allseits bekannten und nicht näher beleuchteten `javac`, den JIT-Compilern C1 und C2 sowie vor allem Graal. Dieser wurde als zukünftiger Ersatz für C2 entwickelt und ist im JDK unter Linux seit Version 9, unter Windows seit Version 10 verfügbar. Graal ist so flexibel einsetzbar, dass er in der GraalVM als Ahead-of-time-Compiler verwendet wird, um native Images zu erzeugen. Wir haben dann die ersten Gehversuche bei der Erzeugung nativer Images auf Hello-World-Niveau erfolgreich absolviert. In einem zukünftigen Artikel werden wir diese ersten Gehversuche hinter uns lassen und neue Möglichkeiten der GraalVM bei der Erzeugung nativer Images kennenlernen. Stay tuned!

Referenzen

[1] *https://www.graalvm.org*
[2] *https://github.com/oracle/graal/*
[3] Bernd Müller. Unbekannte Kostbarkeiten des SDK – Heute: Just-in-Time-Compilation. Java aktuell 04/2019.
[4] JEP 296: Ahead-of-Time Compilation. *http://openjdk.java.net/jeps/295*
[5] JEP 317: Experimental Java-Based JIT Compiler. *http://openjdk.java.net/jeps/317*
[6] JEP 243: Java-Level JVM Compiler Interface. *http://openjdk.java.net/jeps/243*
[7] *https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md*
[8] The Java Language Specification, Java SE 13 Edition, 2019.
[9] The Java Virtual Machine Specification, Java SE 13 Edition, 2019.

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Ja schlägt's denn schon 13?

Benjamin Schmid, eXXcellent solutions GmbH

Seit der Umstellung auf eine halbjährige Release-kadenz purzeln die neuen Java-Versionen wie nie zuvor. Verstärkt kommen diese nun im Alltag der Projekte und Entwickler an. Dieser Artikel gibt einen gesammelten und praxisorientierten Überblick über alle wichtigen Änderungen seit Java 8.

Es war das Letzte seiner Art: Als Oracle Ende 2017 nach gut dreieinhalb Jahren Java 9 vorstellte, begann eine neue Zeitrechnung. Für viel Aufruhr sorgte dabei insbesondere die Umstellung des Lizenzierungsmodells: Es gibt nun immer nur genau eine öffentlich unterstützte Version von Oracles Java. Diese ist exakt sechs Monate lang aktuell und verfällt mit Veröffentlichung der Folgeversion. Das gilt auch für die nun alle drei Jahre erscheinenden LTS-Versionen. Für kommerziellen Produktionseinsatz oder längere Supportzeiten muss nun generell bezahlt werden. Im April 2019 endete auch die öffentliche Update-Versorgung für Oracle Java 8.

Dieser Wandel löste enorme Verunsicherung aus. Manch einer witterte gar den drohenden Untergang der Plattform. Aber weit gefehlt, denn sie vergessen eine weitere wichtige Änderung: Oracle selbst veröffentlicht inzwischen OpenJDK Builds und hat zur LTS-Version 11 alle ehemals kommerziellen Features wie Java Flight Recorder, Application Class-Data Sharing (CDS) oder Z-Garbage Collector (ZGC) an das OpenJDK übergeben. Ab Java 11 sind OpenJDK-Versionen somit funktional komplett identisch und vollwertiger Ersatz für ein Oracle JDK. Die Unterschiede beschränken sich auf kosmetische Details wie zum Beispiel abweichende Versionsstrings.

Länger als sechs Monate freien Support gibt es aber auch für das OpenJDK von Oracle nicht: In die Bresche springen zahlreiche alternative OpenJDK-Distributionen wie Amazon Corretto, Alibaba

Dragonwell, SAP SapMachine, Red Hat OpenJDK und andere. Die Empfehlung der ersten Wahl ist das AdoptOpenJDK *[1]*. Unterstützt von zahlreichen hochrangigen Sponsoren bietet es vier Jahre kostenfreien Support der LTS-Versionen für alle wichtigen Plattformen und Betriebssysteme und ist damit die ideale Anlaufstelle für jede neue Java-Installation.

Java 9 bis 13 im Überblick

Mit insgesamt 91 *JDK Enhancement-Proposals* (JEP) sticht Java 9 als das mit Abstand größte Release heraus. Seitdem landet man aufgrund des neuen Release-Zyklus eher bei fünf bis fünfzehn Neuerungen pro Release. Die Termine sind nun fix. Nur was fertig ist, kommt mit in die nächste Version. Eine Übersicht der Java-Versionen und ihren Highlights ist in *Tabelle 1* zu sehen.

Ein Dauerthema sind Performance-Verbesserungen. Mit jeder Version hat Java bei Garbage Collectors, Memory Management und Nebenläufigkeit Verbesserungen nachgelegt. Auch Bestandsanwendungen können davon profitieren, sodass sich ein Upgrade auch ohne Code-Änderungen auszahlen kann. Neuere JVM-Versionen erkennen nun beispielsweise die geltenden CPU- und Memory-Limits von Docker-Containern, statt zu erwarten, dass die Maschine ihnen exklusiv gehört. Das hilft, entsprechende Trashing-Effekte zu vermeiden.

Eine wichtige Sonderrolle spielt Java 11: Als Long-Term-Support(LTS)-Version brachte sie ein breites Spektrum an Produktpflege und Aufräumarbeiten mit: Java EE, JavaFX und CORBA wurden aus dem JDK entfernt, ein neues HTTP/2- und Websocket-API überarbeitet und finalisiert, Infrastruktur wie die eingesetzte Unicode-Version oder die verfügbaren Verschlüsselungsverfahren einer Frischzellenkurz unterzogen. Bis zum Erscheinen der nächsten LTS-Version 17 im Jahr 2021 dürfte Java 11 in vielen Produktionsumgebungen gesetzter Standard sein. Features, die neuere Versionen voraussetzen, sind daher im Artikel explizit abgegrenzt.

Neue Sprachmittel

Zu den neuen Dingen, die Entwicklern große Freude bereiten, dürfte zweifelsohne die Typinferenz für lokale Variablen gehören. Der reservierte Typ `var` erlaubt es, bei der Deklaration von Variablen innerhalb von Methoden die vollständige Typangabe entfallen zu lassen. Auf Wunsch erschließt sich der Compiler den effektiven Datentyp anhand der ersten Zuweisung nun selbst. Das gelingt sowohl für Klassen als auch für primitive Datentypen, wie *Listing 1* illustriert.

Insbesondere bei Rückgabewerten, die Generics mit Wildcards kombinieren, verhilft die lokale Typinferenz zu erheblich besser und

Version	Eckdaten		Highlights
8 (LTS)	Release: 2014-03, EOL: 2019-04 (2023 bei AdoptOpenJDK)		
9	Release	2017-03	▪ **Java Modulsystem** (Jigsaw) ▪ *API:* Project Coin Milling, Stream, … ▪ *Tooling:* `jshell`, `jlink`, Unified JVM Logging ▪ *Neue Plattformen:* AArch64, s390x, Arm32/Arm64
	EOL	2019-04	
	JEPs	91	
10	Release	2018-03	▪ **Typinferenz für lokale Variablen mit** `var` ▪ *Performance:* GC, Class-Data Sharing, Threads, … ▪ *Experimentell:* Graal Ahead-of-Time -Compiler
	EOL	2019-09	
	JEPs	12	
11 (LTS)	Release	2018-09	▪ **Oracle JDK ≅ OpenJDK** *Ex-payware:* Flight Recorder, ZGC, CDS, … ▪ Neuer HTTP/2- & Websocket-Client ▪ HTML5 Javadoc, Unicode 10, … ▪ *Performance:* Low-latency ZGC ▪ *Kryptographie:* TLS 1.3, neue Ciphers & Kexs ▪ *Entfernt:* Java EE, JavaFX, CORBA
	EOL	2023-09	
	JEPs	17	
12	Release	2019-03	▪ Low-pause GC *Shenandoah* ▪ Microbenchmark Suite ▪ *Preview:* **Switch Expressions** ▪ *Performance:* CDS, G1 GC, Constants
	EOL	2019-09	
	JEPs	8	
13	Release	2019-09	▪ *Preview:* **Text Blocks**, Switch-Expression (überarbeitet) ▪ *Performance:* AppCDS, ZGC, Socket
	EOL	2020-03	
	JEPs	5	
14 (Plan)	Release	2020-03	▪ Switch-Expression, **selbstbeschreibende NullPointerException** ▪ *Tools:* JFR Event Streaming *Potentiell:* ▪ *Preview:* `instanceof` Pattern Matching, *Record*-Typen, Text Blocks ▪ *Performance:* ZGC für Windows & macOS, G1 für NUMA ▪ *Wegfall:* CMS GC, Pack2000 ▪ *Tools:* Java Installationspakete (`.msi`, `.deb`, `.dmg`, …)
	EOL	2020-09	
	JEPs	8~14	
17 (LTS)	Release: 2021-09, EOL: 2026-09		

Tabelle 1: Java-Versionen und ihre Highlights im Überblick

einfach lesbarem Code. Technisch motivierte Ausdrücke wie `List<? extends Foo<? extends Number>>` lassen sich auf ein elegantes `var` verkürzen und geben somit den Blick auf das Wesentliche frei.

Wo die Grenzen liegen, zeigt *Listing 2*. Da der Typ zum Zeitpunkt der ersten Zuweisung vollständig bekannt sein muss, ist es nicht zulässig, eine `var`-Deklaration erst später, mit null oder nur als Arrayliteral zu initialisieren. Es gilt der Typ zum Deklarationszeitpunkt, das heißt, zum Diamond-Operator (`<>`) verkürzte Generics-Parameter werden als `Object` aufgelöst. Methodenreferenzen und Lambda-Ausdrücke akzeptiert `var` nur in Form anonymer Klassen, sie können aufgrund des individuellen Typs danach nicht neu zugewiesen werden.

Dies sind jedoch triviale Einschränkungen, derer man sich als Entwickler nur bewusst sein muss. Ein wenig Augenmaß beim Einsatz von `var` und den entsprechenden Variablennamen ist angebracht, schließlich soll die neue Kürze den Code nicht zum Puzzle werden lassen. Der offizielle Styleguide gibt hier Empfehlungen *[2]*.

Sprachverbesserungen eher kleiner Natur liefert das *Milling Project Coin (Listing 3)*. Automatisch zu schließende Ressourcen können nun auch außerhalb des `try()`-Blocks definiert werden. Die Möglichkeit, private Methoden in Interfaces zu nutzen, erlaubt die Wiederverwendung von Code zwischen `default`-Methoden.

Experimentierfreude

Um neue Sprachfeatures besser ausprobieren zu können und Feedback aus der Praxis zu gewinnen, gibt es seit Java 12 die *Preview Features*. Das sind experimentelle Sprachelemente, die im Compiler mittels `javac --release <javaversion> --enable-preview` freigeschaltet werden können. Damit sie sich nicht unbewusst einschleichen, müssen sie auch bei der Ausführung über `java --enable-preview` … nochmals explizit aktiviert werden.

In Java 12 wurde erstmals eine Alternative zu einem der sperrigsten und fehlerträchtigsten Konstrukte erprobt, bevor sie in Java 13 nochmals leicht überarbeitet und mit Java 14 endgültig in die Sprache Einzug halten wird: die `switch`-Expression als moderne Alternative zum klobigen `switch`-Statement.

Wie elegant und knackig nun Fallunterscheidungen im Vergleich zu früher *(Listing 4)* gelingen, zeigt *Listing 5*. Neue Arrow-Labels `->` erlauben nun die Verwendung von Blöcken und Ausdrücken. `switch` kann nun selbst als Ausdruck fungieren und damit direkt Ergebnisse zurückliefern. Sperrige Interimsvariablen und `break`-Statements sind passé. In Blöcken markiert `yield` den Rückgabewert. Versteht der Compiler zum Beispiel bei Enums, dass alle möglichen Fälle behandelt wurden, ist auch der bisher zwingend erforderliche `default`-Zweig nicht mehr notwendig. So machen Fallunterscheidungen wieder Spaß!

Als Preview ab Java 13 kommen die neuen *Text Blocks*. Sie erlauben mehrzeilige String-Literale, umrahmt von drei Anführungszeichen `"""`. Enthaltene Strings können sich über mehrere Zeilen erstrecken und dürfen auch einzelne Anführungszeichen enthalten. Entfernt werden führende Leerzeichen bis zu Höhe des öffnenden beziehungsweise schließenden `"""` sowie unnötige Leerzeichen am Zeilenende. In *Listing 6* hat `hello()` also effektiv keine Einrückung.

```
var primitiveVal = 5;   // int
var doubleVal    = 5d;  // Typ via Literal

final var sl = new ArrayList<String>();
var letters = "";       // String
for (var s : sl) {
  letters += s.toLowerCase();
}
```

Listing 1: Typinferenz für lokale Variablen

```
// var wontCompile;
// var wontCompile = null;
// var wontCompile = {-1, 1};

var myMap = new HashMap<>();
myMap.put(42, "The answer");
// var wontCompile = myMap.get(42).trim();

// var wontCompile = String::toUpperCase;
var myPredicate = new IntPredicate() {
  public boolean test(int v) { return v > 0; } };
// myPredicate = (int i) -> (i % 2) == 0;
```

Listing 2: Randfälle der Typinferenz

```
var inputStream = new FileInputStream(".gitignore");
try (inputStream) { … }

interface Version {
  byte[] digits();
  default String text() { return text(digits()); }
  private String text(byte[] version) { … }
}
```

Listing 3: Milling-Project-Coin-Verbesserungen

```
enum Direction {N, S, W, E}

String java8SwitchStatement(Direction way) {
  String result;
  switch (way) {
    case N:
      result = "Up";
      break;
    case S:
      result = "Down";
      break;
    case E:
    case W:
      result = "Somewhere left or right";
      break;
    default:
      throw new IllegalStateException("Huh?");
  }
  return result;
}
```

Listing 4: Altbekannt: das fehlerträchtige switch-Statement

```
String java14SwitchExpression(Direction way) {
  return switch (way) {
    case N -> "Up";
    case S -> { yield "Down"; }
    case E, W -> "Somewhere left or right";
    // default -> "Huh?"
  };
}
```

Listing 5: Eleganter switch-Ausdruck ab Java 12 bzw. 14

Glückloses Modulsystem Jigsaw

Eine der umfassendsten und tiefgreifendsten Änderung der Kernsprache haben wir bislang noch nicht erwähnt: das Modulsystems Jigsaw aus Java 9. Module bündeln eine oder mehrere Java-Packages und bieten eine erheblich strengere Kapselung als JAR-Dateien, denn `public` deklarierte Klassen sind anderen Modulen erst einmal nicht zugänglich. Die entsprechenden Packages müssen zuvor explizit zur Weitergabe an Dritte freigegeben werden.

Listing 7 zeigt ein beispielhaftes `module-info.java`, das im Quellcode-Rootverzeichnis abgelegt wird. Darin definiert das Modul über `requires` benötigte Drittmodule und nennt unter `exports` die anderen Modulen zugänglichen Packages von sich selbst. Daneben bietet Jigsaw noch einen einfachen Service-Provider-Mechanismus, sodass Module Implementierungen für Service-Interfaces bereitstellen können, ohne dass ihre Anwender die Implementierungsklassen kennen müssen. Ein einfaches `ServiceLoader.load(java.sql.Driver.class)` instanziiert die bereitgestellten Implementierungen.

Trotz des massiven und grundlegenden Umbaus: So richtig durchsetzen konnte sich Jigsaw bislang nicht. Das mag einerseits am Fehlen von Features wie einer Versionierung der Module liegen, vermutlich aber auch an der fehlenden Notwendigkeit, denn das Modulsystem ist ein vollständig nebenläufiger Ansatz. Entweder man kompiliert über `javac -mp modulepath …` seine Quellen als Module und erhält dadurch `.jmod` statt der altbekannten `.jar`-Dateien. Diese nutzt man dann über `java -mp modulepath -m modulename/moduleclass`. Alternativ bleibt man beim bewährten Classpath und arbeitet weiterhin wie gewohnt. Dazu rät selbst Java-Altmeister Joshua Bloch *[3]* und empfiehlt abzuwarten, ob sich das Java-Modulsystem auch außerhalb des JDK durchsetzen kann, es sei denn, es liegen überzeugende Vorteile auf der Hand. Doch dazu später mehr.

Neues bei den APIs

Neben den Erweiterungen an der Sprache hat auch die Standardbibliothek zahlreiche Neuerungen im Reisegepäck der neuen Versionen.

```
Object obj = engine.eval("""
    function hello() {
       print("Hi, world!");
    }

    hello();
    """);
```

Listing 6: Textblöcke (Preview)

```
module com.mysql.jdbc {
  // Benötigte Dritt-Module
  requires java.sql;

  // Exportierte Pakete des Moduls
  exports com.mysql.jdbc;

  // SPI: Bereitstellung Service-Implementierung
  provides java.sql.Driver
    with com.mysql.jdbc.Driver;
}
```

Listing 7: Moduldefinition als /module-info.java

Klein, aber oho!

Die oft selbst implementierten Helferlein, um Collections zu erzeugen, können eingemottet werden. Neue Factory-Methoden an den Collection-Interfaces erlauben zum Beispiel, mit `List.of(1,2,3)` oder `Map.of("q", -1, "a", 42)` nun direkt schreibgeschützte, vorbefüllte Collections zu erzeugen. `null` ist dabei ebenso wenig erlaubt wie doppelte Schlüsselwerte für `Set` und `Map`. An gleicher Stelle finden sich auch die neuen `copyOf()`-Methoden, mit denen schreibgeschützte Kopien der verschiedenen Collection-Typen entstehen. Anders als bei `Collections.unmodifiableXXX()` handelt es sich dabei um echte Kopien, sodass spätere Veränderungen an der Ausgangsliste keine Auswirkung haben.

Den Fluss bändigen

Bei den Java Streams *(Listing 8)* sind `dropWhile()` und `takeWhile()` dazugekommen. Anhand des übergebenen Kriteriums erlauben sie, eine *zusammenhängende* Kette an Elemente zu verwerfen beziehungsweise auszuschneiden. Dank zusätzlicher Abbruchbedingung kann `iterate()` nun auch endliche Reihen erzeugen. Die neue Factory-Methode `ofNullable()` erzeugt einwertige oder leere Streams abhängig davon, ob sie einen Wert oder `null` erhält.

Zuwachs gab es auch bei der `Collectors`-Factory, die nun das Sammeln in unveränderbare Collections unterstützt (`toUnmodifiableList/Set/Map()`), unerwünschte Elemente mit `filtering()` aussortiert und dank `flatMapping()` verschachtelte Streams on-the-fly auflöst.

Spannend ist die Ergänzung ab Java 12: Der `teeing()`-Kollektor ermöglicht es, einen Stream aufzuspalten und diesen an zwei Kollektoren gleichzeitig weiterzugeben. Im Beispiel `rms()` aus *Listing 8* berechnen wir darüber das quadratische Mittel: Der erste Kollektor summiert die Quadrate der einzelnen Werte, während der zweite Kollektor nur die Anzahl zählt. Um dem `Collector`-Interface gerecht zu werden, müssen zum Schluss die beide Teilergebnisse wieder zu einem Gesamtwert zusammengeführt werden.

```
var stream = Stream.of(-1, 0, 1, 21, 42);

// zusammenhängende Teilketten ausfiltern
stream.dropWhile(i -> i < 2)  // → [ 21, 42   ]
stream.takeWhile(i -> i < 2)  // → [ -1, 0, 1 ]

// Endliche Reihen erzeugen
Stream.iterate(2, i -> i < 999,  i -> i*i);
// → [2, 4, 16, 256]

Stream.ofNullable(null);    // → [ ]
Stream.ofNullable("Hi");    // → [ "Hi" ]

// JDK12: teeing - Streams aufsplitten
double rms(Stream<Integer> numStream) {
  return numStream.collect(
    Collectors.teeing(
      Collectors.summingDouble(i -> i*i), // → sum
      Collectors.counting(),              // → n
      (sum, n) -> Math.sqrt(sum / n)      // Join
    )
  );
}
```

Listing 8: Neues bei den Streams

Programm online

JavaLand 2020

17. - 19. März 2020 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Sein oder nicht sein

`Optional` ist der späte Versuch in Java, die Probleme mit `null`-Referenzen zu lindern. Mit `ifPresentOrElse()` lassen sich die beide Fälle nun kompakter behandeln *(Listing 9)*. Mit `or()` können Optionals verkettet werden, während `stream()` diese komfortabel in einen Stream überführt. Das neue `isEmpty` ist das Gegenstück zum altbekannten `isPresent()` und unterstützt eine bessere Lesbarkeit.

Aufräumen – diesmal aber wirklich!

Eigentlich soll eine Deprecated-Markierung Entwickler von der weiteren Verwendung obsoleter APIs abhalten. Die jahrzehntelange Abwärtskompatibilität von Java hat ihrer Glaubwürdigkeit allerdings keinen Gefallen getan. Die neuen, zusätzlichen Felder `forRemoval` und `since` leisten nun mehr Nachdruck in der Sache. So weist das `@Deprecated(since="11", forRemoval=true)` an Pack2000.Packer nun unmissverständlich darauf hin, dass man auch tatsächlich vorhat, diese Teile zu entfernen. So wird `Pack2000` in Java 14 tatsächlich nicht mehr enthalten sein. Die API-Teile, die zur Entfernung vorgemerkt sind, werden in der Javadoc gesondert aufgeführt. Eine zusätzliche Stütze bietet das `jdeprscan`-Tool: Es listet die aktuell zu Disposition stehenden Kandidaten des JDK auf und kann im Bestandscode nach deren Verwendung suchen *(siehe Abbildung 1)*.

Dass mit Java 9 auch `Object.finalize()` als deprecated markiert wurde, muss allerdings noch niemanden verunsichern. Finalizer sind inhärent problematisch für Garbage Collectors, denn ein `finalize()`-Aufruf kann dazu führen, dass ein zum Abräumen bereitstehendes Objekt wieder zu komplett neuem Leben erwacht. Bei der verstärkten Nebenläufigkeit moderner CPU-Architekturen bereitet das viel Kopfzerbrechen und Probleme. Die Deprecation-Markierung soll daher also vor allem Entwickler auf die besseren Alternativen wie das `AutoCloseable`-Interface oder die neue `Cleaner`-Infrastruktur hinweisen. Vorerst ist das `forRemoval`-Flag nicht gesetzt und dies wird vermutlich recht lange so bleiben.

Neue Befehlsgewalten

Komfortablere und feingranulare Kontrolle über laufende und gestartete Prozesse bietet das neue `ProcessHandle` *(Listing 10)*. Über `parent()` und `children()` kann man sich einfach in der Prozesshierarchie bewegen. Besonders interessant ist `onExit()`, über das ein `CompletableFuture` registriert werden kann, das dann bei Prozessende ausgeführt wird, um zum Beispiel die Ergebnisse eines gestarteten Hintergrundprozesses abzuholen.

Komfortabler Netzwerken

Eine erhebliche Erleichterung im Zeitalter vernetzter Anwendungen stellt der neue HTTP/2- und Websocket-taugliche Client dar. Mit einem zeitgemäßen Fluent-API löst der neue `HttpClient` die sperrige Handhabung der `UrlConnection` durch ein sprechendes Builder-Pattern ab. Über das API können alle praxisrelevanten Aspekte wie Proxies, Cookies, Authentifizierung und Timeouts einfach gesteuert werden.

Mitunter am spannendsten ist die komfortable Möglichkeit für asynchrone Requests: Über `sendAsync()` starten Requests im Hintergrund, während man sich derweil sofort anderen Dingen zuwenden kann. Dafür erhält man ein `CompletableFuture`, das sich mit Verfügbarkeit des Response auflöst *(Listing 11)*.

```
var maybeInt = Optional.ofNullable(
     (Math.random()<0.5) ? 42 : null );

// Wert oder nicht Wert? Erledige dies oder das
maybeInt.ifPresentOrElse(
        (n) -> System.out.println(n),
        () -> System.out.println("Nada")   );

// Leeres Optional? → Baue Ersatz on-the-fly …
var secondTry = maybeInt.or( () -> Optional.of(-1) );

// Erzeuge daraus einen Stream → [] or [42]
Stream<Integer> intStream = maybeInt.stream();

// Neues isEmpty(): Ergänzt isPresent()
assert maybeInt.isPresent() == !maybeInt.isEmpty();
```

Listing 9: Neues bei Optional

```
Process sleeper =
  new ProcessBuilder("sleep", "9s").start();
ProcessHandle sleepHandle = sleeper.toHandle();

// Führe nach Prozessende ein Runnables aus
sleepHandle.onExit().thenRun(
  () -> out.println("`sleep` process exited") );

// Navigieren im Prozessbaum
ProcessHandle jvm = ProcessHandle.current();
jvm.children().forEach(out::println);

sleepHandle.destroy();  // Prozessende erzwingen
```

Listing 10: Komfortable & feinere Prozess-Steuerung mit ProcessHandle

```
HttpClient client = HttpClient.newBuilder()
  .version(HttpClient.Version.HTTP_2)
  .connectTimeout(Duration.ofSeconds(3)).build();

URI uri = URI.create("https://www.exxcellent.de/");
HttpRequest req = HttpRequest.newBuilder(uri)
  .header("Useragent", "MyDemo").GET().build();

var future =
  client.sendAsync(req,
        HttpResponse.BodyHandlers.ofString())
  .thenApply(HttpResponse::body)
  .thenAccept(System.out::println);

// … während der Request läuft: Harte Arbeit!
new BigInteger(9999, 9, new Random());

// … abschließendes Warten auf den Request
future.get();
```

Listing 11: Moderner, asynchroner HTTP/2-Client

Gegenüber Java 8 schafft es Java 13 auf insgesamt knapp 1.400 hinzugefügte und circa 110 entfernte API-Elemente. Lesern, die einen vollständigen Überblick möchten, empfiehlt der Autor das Java-Almanac-Projekt *[4]*. Dieses pflegt eine strukturierte Gegenüberstellung der verschiedenen Java-API-Versionen.

Tools

Zum Abschluss wollen wir noch einen Blick auf die Neuerungen im Bereich des JDK Tooling werfen.

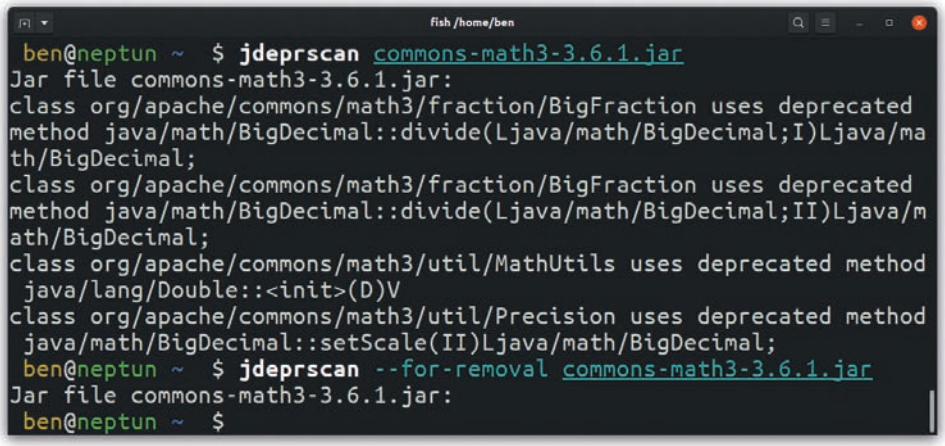


Abbildung 1: jdeprscan hilft beim Aufspüren von Altlasten (Quelle: Benjamin Schmid)

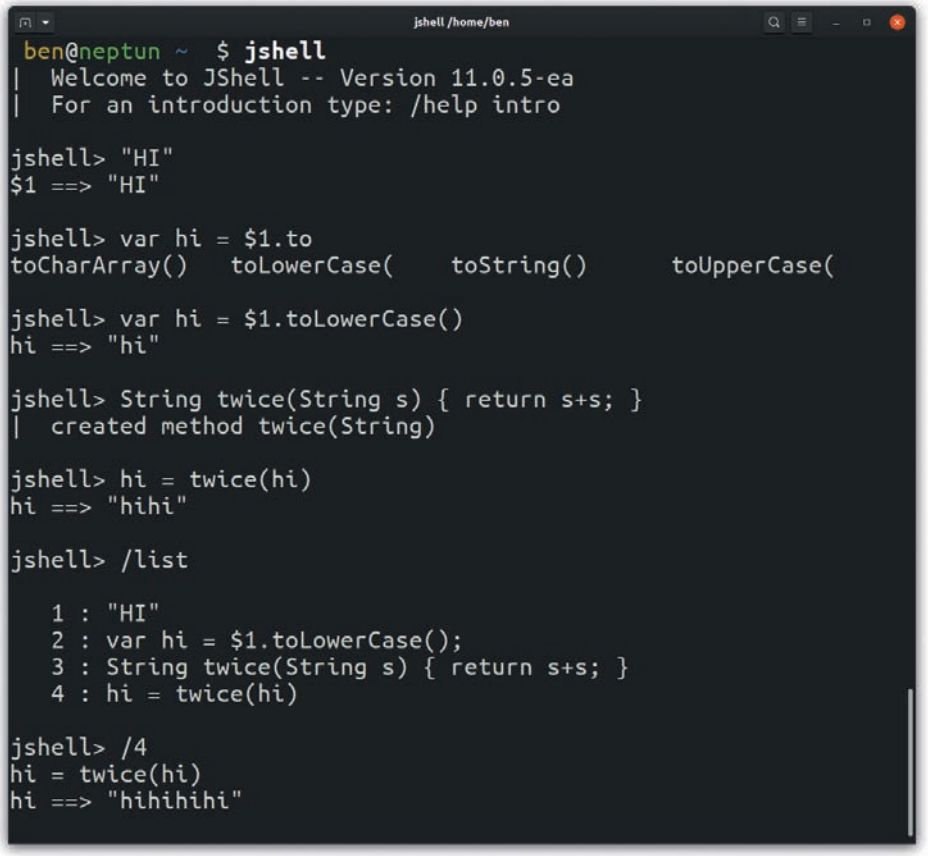


Abbildung 2: Schnelle Experimente mit der REPL jshell (Quelle: Benjamin Schmid)

```
ben@neptun ~ $ java -Xlog:os="debug,task*=info:stdout:pid,level,tags" \
                   SayHello.java java_aktuell!
[25347][debug][os] Initial active processor count set to 4
[25347][info ][os] Use of CLOCK_MONOTONIC is supported
[25347][info ][os] Use of pthread_condattr_setclock is supported
[25347][info ][os] Relative timed-wait using pthread_cond_timedwait is assc
[25347][info ][os] HotSpot is running with glibc 2.30, NPTL 2.30
[25347][info ][os] SafePoint Polling address, bad (protected) page:0x00007f
[25347][info ][gc,task] GC(0) Using 4 workers of 4 for evacuation
Hello java_aktuell!
```

Abbildung 3: Einfaches Logging & Scripting (Quelle: Benjamin Schmid)

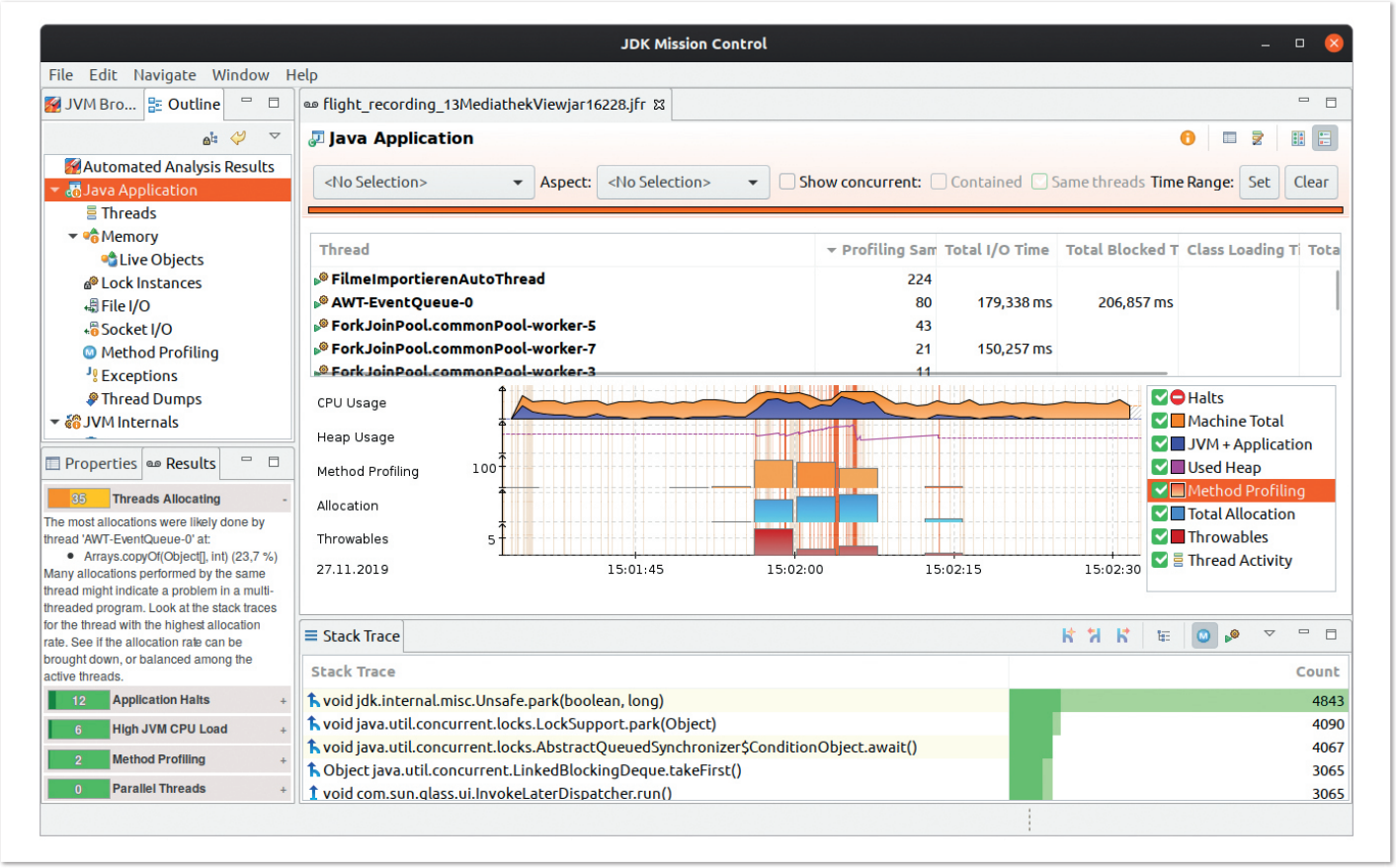


Abbildung 4: Performance-Profiling in Produktion mit JMC (Quelle: Benjamin Schmid)

Experimentierkasten

Schnelles und unkompliziertes Experimentieren mit Java erlaubt das neue REPL-Tool JShell und verdient damit, in das Standardrepertoire eines jeden Entwicklers aufgenommen zu werden. Die Codevervollständigung hilft beim Formulieren neuer Snippets. Mit `$1` beziehungsweise `/1` lassen sich Ergebnisse und Statements weiterverwenden. Dank `/edit` können die eigenen Entwürfe in einem externen Editor noch weiter verfeinert werden, bevor man sie für die weitere Verwendung als Datei exportiert *(siehe Abbildung 2)*.

Auch Java selbst kann nun direkt und ohne Umweg über den Compiler mit Sourcecode umgehen, falls dieser in einer einzigen Quelldatei liegt. Ein einfaches `java SayHello.java World!` führt die Main-Methode samt Parameter direkt aus. Auf unixoiden Betriebssystemen geht es sogar noch komfortabler: Umbenannt in `SayHello` (ohne `.java`-Dateiendung) und ergänzt um ein einleitendes „Shebang"-Statement wie `#!/usr/bin/java --source 11`, können die eigenen „Java-Skripte" dort direkt ausgeführt werden.

Vereintes Logging

Gut gemeint, aber bislang recht erfolglos war die im JDK enthaltene Logging-Infrastruktur (JUL). Ursachen dafür waren sicherlich die recht umständliche Konfiguration und der bruchstückhafte Einsatz im JDK selbst. Die Überarbeitungen könnten JUL allerdings zu einem späten Erfolg verhelfen. Alle Teile des JDK nutzen endlich die zentrale Logging-Infrastruktur. Kommandozeilenoptionen erlauben nun eine einfache und feingranulare Konfiguration. Mittels API können die Logger plattformweit durch eigene Implementierungen ersetzt werden. Damit steht einer späten Wiedervereinigung der vielen verschiedenen Logging-Lösungen kaum noch etwas entgegen *(siehe Abbildung 3)*.

Houston, wir haben ein Problem!

Der *Java Flight Recorder* (JFR) und das separate Analyse-Werkzeug *JDK Mission Control* (JMC) gehören zu den ehemals kommerziellen Bestandteilen des Oracle JDK. Eine Besonderheit dieser Profiling- und Monitoring-Werkzeuge ist ihre Ausrichtung auf den Einsatz in Produktion. Das verspricht Ad-hoc-Einsatz und geringen Performance-Overhead. Der Flight Recorder ist fester Bestandteil der JVM und kann beim Auftreten von Problemen jederzeit über die Kommandozeile oder JMC gestartet werden. Das Mission Control Tool analysiert den darüber erhaltenen Mitschnitt und unterstützt über zahlreiche Darstellungen von CPU, Memory und I/O bei der Suche nach Engpässen. Über JMX kann sich JMC auch live an laufende Prozesse anbinden und zum Beispiel bei Erreichen von Schwellwerte eigene Trigger-Aktionen starten, wie zum Beispiel das Versenden einer warnenden E-Mail bei lang anhaltender CPU-Überlastung *(siehe Abbildung 4)*.

Trennungsgründe

Ohne triftige Gründe ist das Java-Modulsystem bislang noch keine heiße Empfehlung. Für dessen Verwendung sprechen allerdings die Werkzeuge `jdeps` und `jlink`. Mit `jlink` ist es möglich, schlanke, auf das notwendigste reduzierte Laufzeit-Images für modulare Java-Anwendungen einschließlich JRE zu erzeugen. Die schlanke Linie ist insbesondere für Java-Anwendungen im IoT-/Embedded-Bereich und in Docker- beziehungsweise Kubernetes-Umgebungen interessant. Das Werkzeug `jdeps` unterstützt bei der Analyse von Bestandsanwendungen und nennt für die dazugehörigen deren Abhängigkeiten. Das hilft zum Beispiel bei den Vorbereitungen für die Aufspaltung in neue Module und damit dem Umstieg.

Fazit und Ausblick

Der Umfang an Neuerungen in den aktuellen Java-Versionen ist beachtlich. Aufgrund des neuen Innovationstempos lohnt es sich, auch jenseits der LTS-Version 11 am Ball zu bleiben. Wer möchte, kann dazu mit den gezeigten Beispielen *[5]* experimentieren.

Im Mai erscheint bereits die nächste Java-Version 14. Zum Zeitpunkt des Artikels lagen noch gut zwei Wochen bis zum Feature Freeze der *Rampdown Phase One*. Den genauen Funktionsumfang dokumentieren die Projektseiten des OpenJDK *[6]*. Absehbar ist, dass neben dem endgültigen Einzug der Switch-Expression über Pattern Matching und Record-Typen zwei weitere innovative Sprachfeatures in die Preview-Erprobung eintreten dürften. Spannend sind auch Themen wie zum Beispiel die polyglotte GraalVM *[7]* oder leichtgewichtige Nebenläufigkeit ohne Threads des Projekts Loom, die sich bereits am Horizont abzeichnen. Für ausreichende Neuerungen in kommenden Java-Versionen ist also bestens gesorgt!

Referenzen

[1] *https://adoptopenjdk.net/*
[2] *http://openjdk.java.net/projects/amber/LVTIstyle.html*
[3] Joshua Bloch 2018: Effective Java: Third Edition. Addison-Wesley Professional.
[4] *https://javaalmanac.io/*
[5] *https://github.com/bentolor/java9to13*
[6] *https://openjdk.java.net/projects/jdk/14/*
[7] *https://www.graalvm.org/*

Benjamin Schmid
eXXcellent solutions GmbH
benjamin.schmid@exxcellent.de

Benjamin Schmid ist Portfolio-Manager und Technology Advisor bei der eXXcellent solutions und erster Ansprechpartner in allen technologischen und methodischen Fragestellungen. Seine Schwerpunkte liegen im Bereich von Java-, Web- und Cloud-Architekturen. Auf der stetigen Suche nach innovativen, soliden und nachhaltigen Lösungen gibt er seine praxisnahen Erfahrungen und Aha-Momente als Redner und Autor immer wieder gerne weiter.

JavaLand Thementag: Microservices hautnah erleben

André Sept, DOAG

Mit dem erstmals stattfindenden Thementag *[1]* zieht auf der JavaLand 2020 *[2]* ein neues Format ein. Vier Organisationen berichten am Donnerstag, den 19. März über ihre bisherigen Erfahrungen mit Microservices, und zeigen den Besuchern, welche Höhen und Tiefen sie bei der Implementierung erlebt haben. Am Vormittag stellt jede Firma ihr Projekt praxisnah vor. Dabei spielen nicht nur die technischen Aspekte eine Rolle, gerade die Wandlung der Organisation wird von allen vier Firmen als größte Herausforderung gesehen.

REWE digital entschied sich 2014 dazu, ihre monolithische Shop-Applikation in eine Microservices-Architektur zu überführen. Fünf Jahre später bereut dies niemand – im Gegenteil: Die Entwickler genießen ihre Autonomie, Product Owner die schnellen Entwicklungszyklen und die Geschäftsführung die transparente Planbarkeit neuer Features. Eine wichtige Erkenntnis dabei: Microservices sind keine rein technische Lösung, sondern vor allem auch eine organisatorische. Es wird exemplarisch gezeigt, wie eine Migration zu Microservices gelingen kann, worauf dabei zu achten ist und welche Vorteile zu erwarten sind.

Die **Deutsche Bahn** renoviert ihre Vertriebsplattform für den Personenverkehr. Also die IT-Landschaft, mit der die Kunden Zugverbindungen abfragen und Tickets erwerben können. Eine Großbaustelle mit über 300 Projektbeteiligten und dem Vorhaben, eine der umsatzstärksten E-Commerce-Plattformen Deutschlands zu modernisieren. Es werden grundlegenden Architekturkonzepte und Entscheidungskriterien für die neue Plattform dargestellt; etwa warum keine Microservices-Architektur vorgegeben wurde (aber trotzdem viele Microservices entstanden sind) oder wie SOA eingesetzt wird (ohne einen ESB einzusetzen). Technisch detaillierter wird die Ausgestaltung einer der Domänen gezeigt. Neben Architekturprinzipien und Auswahl der Technologie wird auch auf organisatorische Herausforderungen im Hinblick auf Architektur und Technik eingegangen.

Bereits seit 2011 wird **otto.de** als verteiltes System von Self-contained Systems und Microservices entwickelt. Mittlerweile besteht das System aus rund 200 lose gekoppelten Microservices, die von 27 Teams verantwortet und kontinuierlich (circa 100 Mal pro Tag) deployed werden. Die Beteiligten zeigen die Reise von eher monolithischen Anwendungen hin zum Otto-Shop im Jahre 2020.

Die **Bundesagentur für Arbeit** hat im Jahr 2015, im Zuge eines Relaunches des Internetangebots, grundlegende Veränderungen bezüglich Organisation, Prozessen, Architektur und Technik in Gang gesetzt. Die Entwicklung hin zu einer Microservices-artigen Architektur war hierbei mehr logische Folge dieses Veränderungsprozesses als Auslöser. Das Zusammenspiel von Organisation, Prozessen und Architektur wird näher beleuchtet. Neben den Vorteilen werden auch wesentliche Herausforderungen wie Integration und Betrieb dargestellt.

Am Nachmittag wird die Thematik in acht parallel stattfindenden Deep Dive Sessions noch weiter vertieft. Dies ermöglicht den Teilnehmern, die ganze Bandbreite der aktuellen Microservices-Themen zu erleben. Zum Abschluss stehen alle Experten den Teilnehmern in einer Panel-Diskussion für Fragen zur Verfügung. Mit diesem speziellen Thementag wollen wir das Programm der JavaLand bunter gestalten. Das Ziel ist es, den Teilnehmern das geballte Praxiswissen der Anwender mitzugeben. Und eben auch die Möglichkeit, die Berge und Täler, die mit dem Microservices-Ansatz einhergehen, näher zu erforschen.

Referenzen:

[1] *https://www.javaland.eu/de/programm/thementag/*

[2] *https://www.javaland.eu/de/home/*

André Sept

Leiter der DOAG Java Community

andre.sept@doag.org

André gehört seit 2010 aktiv der DOAG an. Dort startete er als Regioleiter Nürnberg und organisierte von 2012 bis 2014 die Special Interest Group (SIG) Java. André ist Teil des Redaktionsbeirats der Java aktuell und vertritt die DOAG auch im iJUG e.V. Seit März 2015 leitet er die DOAG Java Community und ist Mitglied im Vorstand.

Herausforderungen für die Lizenzierung von Oracle Java SE

Mike Betsch, LiGenius Solutions AG

Seit März 2018, als Oracle die Lizenzänderungen für Java SE bekannt gab, ist eine Welle der Entrüstung, Unsicherheit und Angst um die Welt gegangen. Was wird nun? Stirbt Java? Eine Menge Spekulationen über Risiken, Lizenzregeln und geeignete Maßnahmen waren plötzlich überall im Internet zu finden. Dienstleister boten ihre Hilfe an, doch auch sie bekamen von Oracle unterschiedliche Antworten auf ihre Fragen. Verbindliche Aussagen bekam man selbst von Oracle nicht. Die Ungewissheit führte mit der Zeit bei Kunden zu Resignation oder Aktionismus. Mittlerweile sind die Änderungen der Versionsnummern, Release-Zyklen und die Angleichung an OpenJDK vollzogen. Was bleibt, ist die Frage, mit der sich alle Unternehmen beschäftigen müssen, auch die, die bisher gar keine Oracle-Kunden waren: Wie gehe ich in meinem Unternehmen mit dem Thema Oracle Java SE um? Darauf haben die Konzerne ganz unterschiedliche Antworten für sich gefunden. Einige stellt der Autor hier vor.

Zunächst muss der Autor die Annahme vieler Kunden widerlegen, die sich im letzten Jahr durch die vielen Headlines zu dem Thema in den Köpfen festgesetzt hat; „Oracle macht Java SE Anfang 2019 kostenpflichtig!", hieß es überall in den Fachmedien. Diese Aussage ist insofern verwirrend, da Oracles Java SE unter bestimmten Voraussetzungen schon seit 2011 lizenzpflichtig ist. Eine kostspielige Ansicht, wie sich dann in der Praxis zeigt.

Die Java-Lizenzgeschichte

Nachdem Oracle im Jahr 2010 den Spark- und Solaris-Hersteller Sun Microsystems übernommen hat, wurde für Java zunächst eine eigene Ausführung der GNU-GPLv2-Lizenz verfasst, das sogenannte „Binary Code License Agreement for Java SE Platform Products and JavaFX" (BCLA). Darin wird die Nutzung für Oracle Java SE für alle kostenfrei erlaubt, die Nutzung für zusätzliche sogenannte Commercial Features jedoch unter Lizenzpflicht gestellt. Mit dieser Definition war es Oracle möglich, eigene zusätzliche Funktionalitäten in das Java SE einzubauen und Lizenzgebühren dafür zu verlangen. Zum Beispiel die „Auto Update OFF"-Versionen. Das sind Java SE Downloads, die sich im Gegensatz zur kostenfrei nutzbaren Version nur in einem Punkt unterscheiden. Wie der Name schon vermuten lässt, ist der Prozess zum automatischen Update Check nicht aktiviert. Wer diese Version installiert hatte, benötigte schon damals eine Lizenz. Das ist nur ein Beispiel für die Komplexität dieser Oracle-Java-SE-Produktlizenzierung. Oracle macht es seinen Kunden nicht immer leicht, die richtigen Produkte in der korrekten Menge einfach selbst zu ermitteln.

Im Jahr 2014 erfand Oracle ein Java-SE-Desktop-Lizenzprodukt, die „Java SE Advanced Desktop"-Lizenz. Schon damals zeichnete sich der Weg zu den bekannten Metriken der Oracle-Technologie-Produkte wie Datenbank und WebLogic ab. Es gab damit unterschiedliche Lizenzen und Preise für Server und Desktop-PCs.

Im November 2018 hat Oracle mit der neuen Java SE Roadmap neue Benutzer-Kategorien geschaffen, um besser zwischen Lizenzpflicht und freier Nutzung unterscheiden zu können. Diese Benutzer-Kategorien hier einmal vom Autor interpretiert:

1. Oracle Customers: Alle Kunden, die für Java SE bezahlen.
2. Commercial Users: Alle Kunden, die nicht zur Kategorie 1 gehören und Java SE kostenfrei als Teil einer Java-Applikation eines Drittanbieters oder einer Eigenentwicklung nutzen.
3. Personal Users: Einzelpersonen, die Java SE für private Zwecke kostenlos nutzen.

Wenn einem also beispielsweise bei einem Download einer älteren Version des Oracle Java SE der Hinweis ins Auge sticht, dass dieser Download nur für Oracle Customers erlaubt ist, sollte man vielleicht zweimal darüber nachdenken, den Download fortzusetzen.

Das „Oracle Technology Network License Agreement for Oracle Java SE"

Seit Januar 2019 ist nun der nächste Schritt abgeschlossen. Mit dem neuen „Oracle Technology Network License Agreement for Java SE" haben sich einige wichtige Änderungen ergeben, die man kennen sollte.

Zunächst ist die Nutzung von Java-SE-Versionen, bis auf definierte Ausnahmen, unter dieser Vereinbarung für alle kostenpflichtig. Damit kehrt Oracle die Regeln aus dem BCLA um. Zudem kommt nun, wie bei allen Standardlizenz-Bedingungen, ein Auditrecht zum Einsatz. Oracle darf also, auch wenn es nach dem Wissen des Autors noch keinen Audit-Prozess für Java SE gibt, die Lizenznutzung für ihr Java SE prüfen. Dieser Prozess dürfte bei Oracle in Arbeit sein, auch wenn sich dieser voraussichtlich vom etablierten Auditvorgehen unterscheiden wird. Denn auch für Oracle ist diese Situation neu. Denkt man an herkömmliche Lizenz-Produkte von Oracle, sind es in der Regel Softwareinstallationen, die bei den Kunden in den Rechenzentren in einer geringen oder zumindest definierten und dokumentierten Anzahl vorhanden sind. Das heißt zum Beispiel, dass die 100 Datenbanken eines Unternehmens aktiv vom DBA-Team operativ verwaltet und administriert werden. Für solch eine Anzahl kann man mit interaktiven Skripten noch arbeiten. Für Java SE dürfte es deutlich aufwendiger sein, alle Desktop-PCs, Server und virtuelle Maschinen zu überprüfen.

Mit dem „Audit; Termination"-Eintrag regelt Oracle auch die Pflichten des Kunden, falls dieser sein Abonnement für Java SE nicht verlängern möchte. Wir kennen noch die Regel, dass Technologie-Produkte auch ohne Support-Vertrag weiterbetrieben werden können, solange alle Lizenzen eines Lizenz-Sets denselben Supportlevel haben. Bei Java SE muss der Kunde nach Beendigung seiner Subscription auch die kostenpflichtige Version deinstallieren. Das ist notwendig, da es sich bei der Subscription um ein zeitlich limitiertes Nutzungsrecht handelt und nicht um eine Lizenz. Jetzt fragt man sich zu Recht, wie denn mit Java-Lizenzen umgegangen wird, die ein Kunde vielleicht schon unter dem BCLA gekauft hat. Wir werden sehen, wie Oracle sich da aufstellen wird, grundsätzlich gelten aber die Bedingungen zum Zeitpunkt des Lizenzkaufs. Für Besitzer von Java-SE-Lizenzen sollte es also möglich sein, den Support auslaufen zu lassen und die Deployments weiterzubetreiben. Allerdings dürften dann keine Updates mehr eingespielt werden, was aus Sicht des Autors den Sinn verfehlt.

Mittlerweile kann die Nutzung von Java SE nur noch als jährliche Subscription erworben werden. Jährliche Subscription ist das nächste Stichwort, das viele Kunden missverstehen. In der Preisliste der Java SE Subscription ist von Monatspreisen pro Desktop oder Prozessor die Rede. Das impliziert bei den Kunden ein monatliches Kündigungsrecht. Dem ist leider nicht so! Wie sich herausgestellt hat, wird die Subscription immer für mindestens ein Jahr fällig. Aber wahrscheinlich hört sich 2,50 US-Dollar pro Monat einfach weniger an als 30 US-Dollar pro Jahr.

„Dann ist es eben so!"

Es ist erstaunlich, wenn Menschen sich emotional Luft machen, schimpfen und sich über die aus ihrer Sicht „unverschämten Machenschaften" beklagen. Je nach Grad der Wut ist es vorgekommen, dass kurzerhand der Entschluss gefasst wurde, Oracle Java SE komplett aus dem Unternehmen zu entfernen. Software-Lieferanten wurden Ultimaten gestellt, ihre Programme anzupassen oder aussortiert zu werden. Binnen kurzer Zeit wurde zu einem auf OpenJDK basierenden Anbieter gewechselt. Dort zahlt der Kunde auch für Updates, jedoch aufgrund anderer Bemessungsregeln deutlich weniger.

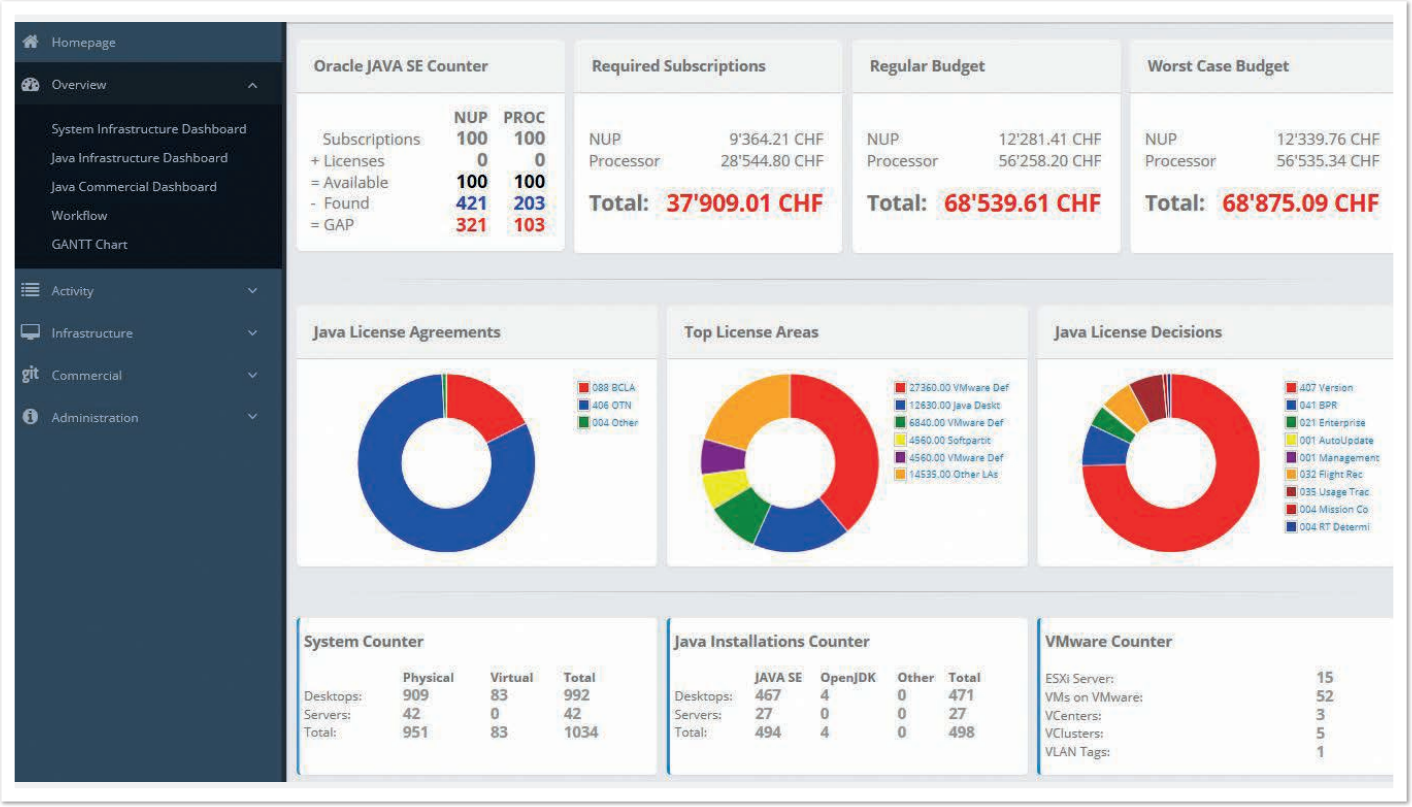


Abbildung 1: Java Commercial Dashboard (© LiGenius Solutions AG, Java-Genius)

Auf der anderen Seite gibt es einige, vor allem größere Firmen, die relativ gelassen auf die Auswertung ihres Java-SE-Subscription-Bedarfes reagieren, den sie in Zukunft zusätzlich in ihr Budget einplanen müssen. Sechsstellige Beträge sind keine Seltenheit. Auf Nachfrage kommt dann oft die Antwort, dass eine Optimierung einfach zu aufwendig sei und unter dem Strich ähnlich viel kosten würde. Das ist zwar gut nachvollziehbar, aber das kleine Detail „jährlich" macht eine Optimierung aus Sicht des Autors immer sinnvoll.

Die meisten Firmen, mit denen der Autor in den mittlerweile fast zwei Jahren seit der Änderungsankündigung im März 2018 gesprochen hat, versuchen, die Nutzung von Oracle Java SE auf ein Minimum zu reduzieren.

Mühsam ernährt sich das Eichhörnchen

Um diesen Optimierungsprozess erfolgreich zu durchlaufen, sind relativ große Anstrengungen zu bewältigen. Die Ermittlung des Lizenzbedarfes inklusive aller Abhängigkeiten und Commercial Features ist mithilfe von Tools gut zu automatisieren. Mehrere SAM-Tool-Hersteller (Software Asset Manager Tool) versprechen eine saubere Lizenzbilanz, auch der Java-Genius der Firma des Autors *(siehe Abbildung 1)*.

Was allerdings sehr viel Zeit raubt, ist die Kommunikation mit den Software-Herstellern. Denn als alle Firmen noch der Meinung waren, Java sei per se kostenfrei, hat sich niemand Gedanken gemacht, wo der Java-Download gestartet wird. Man hat einfach die Oracle-Java-SE-Version genommen. Auch für die Server-Applikationen, die kein Java Web Start oder JavaFX brauchten. Das bedeutet, in der Regel wurde überall Oracle Java SE installiert. Das gilt für die Server ebenso wie für die Clients. Um nun zu entscheiden, ob die Java-Applikation Oracle Java SE benötigt, müssen für jede Applikation Informationen vom Hersteller eingeholt und gegebenenfalls ein alternatives OpenJDK getestet werden. Wenn die Tests erfolgreich waren, kann in einem Wartungsfenster die Java-Installation ausgetauscht werden.

Der Autor empfiehlt, immer mit den Server-Installationen zu beginnen, da auf Serversystemen meist nicht so viele Applikationen installiert sind. Außerdem laufen die Server-Applikationen meist im Dauerbetrieb, sodass die Zuordnung von Java SE zur Applikation einfacher als bei einem Client mit 100 Anwendungen ist. Über die Applikation können oft auch die Benutzergruppen identifiziert werden, die mit dem Programm arbeiten. Mit der Information vom Hersteller, ob Oracle Java SE für die Server-Anwendung oder die Clients benötigt wird, kommt man nach Erfahrung des Autors am besten voran. Wenn diese Arbeit geschafft ist, geht es mit der Planung zügig vorwärts. Aktionspläne können erstellt und Infrastrukturen den Bedürfnissen angepasst werden. Eine Alternative zu Oracle Java SE wird nach Entscheidungsmatrix ausgewählt und auf die Server in den benötigten Versionen verteilt.

Für die Clients sind die Firmen auf verschiedene Ideen für den Change-Prozess gekommen. Während die einen dieselben Tests wie für die Server-Anwendungen planen, gehen andere radikaler an die Sache heran. Sie haben beispielsweise über vorhandene Software-Verteilungsmechanismen sämtliche Java Deployments einfach entfernt und auf Applikationsfehler die ausgesuchte Alternative zu Oracle Java SE installiert. Ihre Funktion wurde mithilfe der Mitarbeiter getestet. Dieses Vorgehen ist schnell und wenn die Mitarbeiter zuvor informiert werden, funktioniert dieser Weg. Der Vorteil ist, dass am Ende nur noch Versionen eines alternativen Java-Anbieters im Unternehmen existieren. Andere

haben nur alle Oracle-Java-SE-Versionen auf den Clients deinstalliert und gegen die Alternative ausgetauscht. Es gibt viele Möglichkeiten, sich individuell richtig aufzustellen und eine gute Strategie für sich zu erarbeiten. Aus Sicht des Autors ist es aber auf jeden Fall sinnvoll, die individuelle Entscheidung auf einer vollständigen Informationsgrundlage zu treffen. Es wäre nicht das erste Mal, dass fehlende Informationen zu einem bösen Erwachen führen.

Fazit

Egal, wie Sie in Ihrem Unternehmen handeln wollen, suchen Sie sich einen Spezialisten, der Ihnen diese Datenbasis beschafft und Ihnen alle Fragen beantwortet. Nur dann wissen Sie sicher, welche Auswirkungen Ihre Entscheidungen haben können. In diesem Sinne wünscht der Autor allen Lesern erfolgreiche Zeiten und effiziente Projekte.

Referenzen

[1] Oracle Website 2019: „Oracle Binary Code License Agreement for the Java SE Platform Products and JavaFX". *https://www.oracle.com/downloads/licenses/binary-code-license.html*

[2] Oracle Website 2019: „Oracle Technology Network License Agreement for Oracle Java SE". *https://www.oracle.com/downloads/licenses/javase-license1.html*

Mike Betsch
LiGenius Solutions AG
Mike.Betsch@ligenius-solutions.com

Mike Betsch ist 1973 in Hamburg geboren und auch dort aufgewachsen. Nach einigen Versuchen der Selbstständigkeit in verschiedenen Bereichen bekam Mike 1999 die Möglichkeit, sich mit Oracle-Datenbanken technisch und kaufmännisch in verschiedenen Positionen auseinanderzusetzen. Wegen eines Job-Angebots als Oracle-Lizenzberater ist Mike mit seiner Familie nach Zürich gezogen. Nach Stationen bei Trivadis und SoftwareONE hat Mike im Juni 2019 mit der LiGenius Solutions AG sein eigenes Unternehmen gegründet und entwickelt mit seinem Partner Oracle-SAM-Lösungen.

Hysterisch gewachsen, trotzdem orchestriert – connected by Camunda

Benjamin Klatt und Mathias Schulte, viadee

Geschäftsprozesse sind die Algorithmen von Unternehmen. Ihre Automatisierung bedeutet in der Regel jedoch eine gute Portion Integrationsherausforderungen. Besonders anspruchsvoll sind hier Anwendungslandschaften mit Legacy-Systemen, die keine idealtypischen, externen Schnittstellen wie beispielsweise REST-Microservices bereitstellen. Camunda als leichtgewichtige Open Source Workflow Engine verspricht hier hohe Flexibilität. In diesem Artikel stellen die Autoren die verschiedenen technischen Möglichkeiten vor. Zudem teilen sie ihre Erfahrungen, bei welchen Architekturanforderungen sich welcher Integrationsansatz besonders empfiehlt.

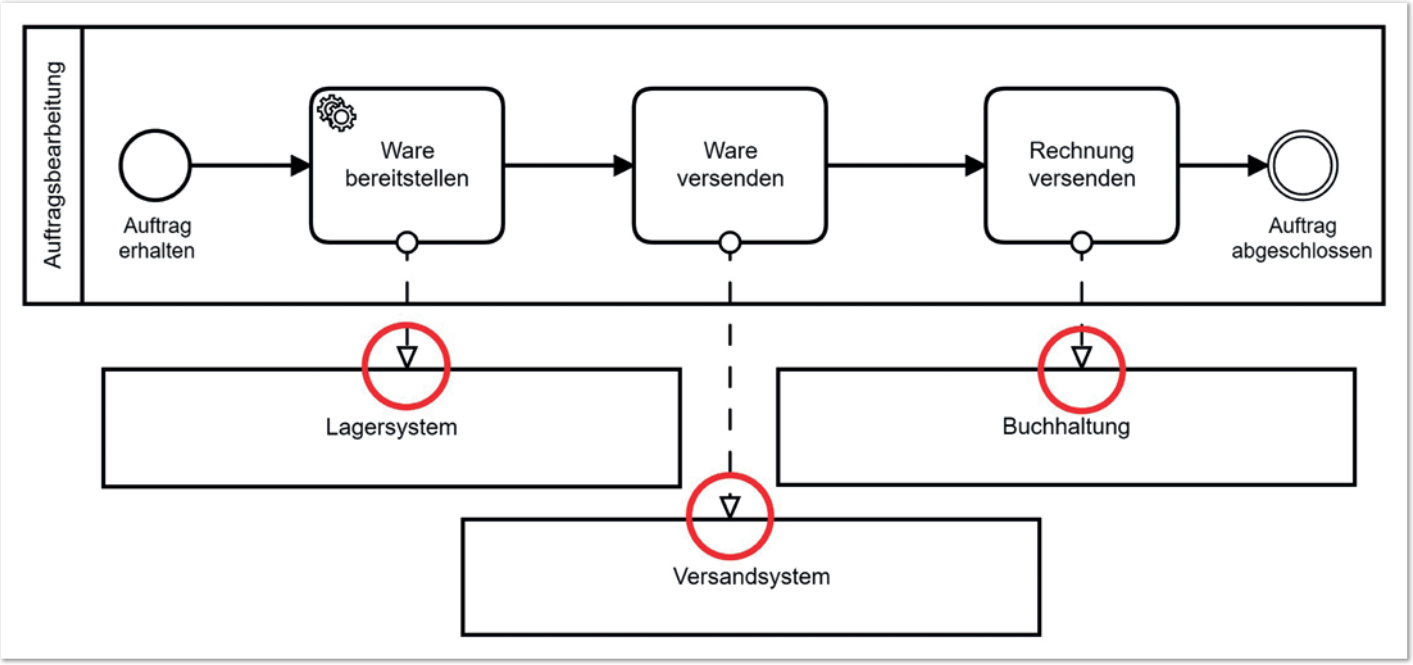


Abbildung 1: Prozessautomatisierung mit Integrationspunkten (© Benjamin Klatt)

Viele Unternehmen arbeiten daran, ihre bestehenden Geschäftsprozesse zu automatisieren, um den Anteil sogenannter „Dunkelverarbeitung" zu erhöhen. Ist der gewünschte Geschäftsprozess entworfen, so warten oft unterschätzte Herausforderungen bei der Integration bestehender Legacy-Systeme, wie in *Abbildung 1* gekennzeichnet. Nur selten stehen hier wohldefinierte Schnittstellen wie moderne REST-Services bereit, sondern es gilt, diverse Remote-Schnittstellen von gewachsenen SOAP-Webservices bis HOST-Strecken anzubinden und teilweise sogar Programmbibliotheken zu integrieren. Camunda bietet hierfür eine Bandbreite von Techniken, die die Autoren nachfolgend vorstellen und Tipps bei der Auswahl geben.

Architektur-Dimensionen

Bevor die Autoren genauer auf die Camunda-Integrationstechniken schauen, werfen sie einen Blick auf Architektur-Dimensionen, die hilfreich bei der Wahl der richtigen Alternative sind. Als Vorbereitung der Integration empfiehlt es sich, sich für jedes zu integrierende System bewusst zu machen, welche Ausprägungen der folgenden Dimensionen gefordert sind:

synchron – asynchron
Während bei synchroner Integration die Prozessanwendung auf eine Rückantwort des integrierten Systems wartet, kann bei einer asynchronen Integration die Arbeit der Prozessanwendung prinzipiell fortgeführt werden. Letzteres ist auch der Fall, wenn die Prozessanwendung erst mal nur bis zu einem vorgesehenen Wartezustand weiterläuft.

push – pull
Während bei einem Push die Prozessanwendung die führende Komponente ist und das integrierte System aufruft, so ist es bei einer Pull-Ausprägung das integrierte System, das einen Aufruf an die Prozessanwendung schickt.

nativ – wrapper
Stellt das zu integrierende System eine Schnittstelle bereit, die man direkt von der Prozessanwendung ansprechen kann und möchte, so spricht man von einer nativen Integration. Ein Wrapper bedeutet hingegen, dass man ein Stück Software zwischen Prozessanwendung und System platziert, um sie miteinander zu integrieren.

remote - embedded
Bei einer Remote-Integration wird ein eigenständig laufendes System angebunden, während bei einer embedded Variante das System, beispielsweise in Form einer Programmbibliothek, direkt mit der Prozessanwendung verpackt wird.

Die vorgestellten Dimensionen können unabhängig voneinander betrachtet werden. Die identifizierten Ausprägungen helfen, die Liste der potenziellen Camunda-Integrationsmöglichkeiten einzuschränken, da nicht jede Ausprägung von allen Alternativen unterstützt wird. Zunächst stellen die Autoren aber die Möglichkeiten vor.

Integrationsmöglichkeiten

Camunda bietet vier grundlegende technische Möglichkeiten, die man für eine Systemintegration nutzen kann. Es können auch mehrere Möglichkeiten gleichzeitig für verschiedene Integrationen in einer Prozessanwendung genutzt werden. Die Autoren haben mit der Robot-Process-Automation(RPA)-Möglichkeit eine fünfte Variante hinzugefügt, die zwar auf die ersten vier Alternativen aufbaut, die sie aber aufgrund ihrer Aktualität mit einordnen möchten:

- Camunda Connector
- Plain Old Java Delegate
- Send and Receive Pattern
- External Tasks
- BPM-RPA-Kombination

Im Folgenden skizzieren wird kurz die verschiedenen Varianten. Detaillierte Implementierungsanleitungen sind auf den Camunda-Seiten zu finden.

```
<dependency>
  <groupId>org.camunda.bpm</groupId>
  <artifactId>camunda-engine-plugin-connect</artifactId>
</dependency>
<dependency>
  <groupId>org.camunda.connect</groupId>
  <artifactId>camunda-connect-connectors-all</artifactId>
</dependency>
```

Listing 1: Maven Dependency für Camunda Standard-Connectoren

Camunda Connector

Camunda Connectoren bieten eine Infrastruktur, über die Integrationen direkt mit dem Camunda Modeler im Prozessmodell konfiguriert und mit ihm zusammen auch versioniert werden können *[2]*. Hierfür können Standard Connectoren, wie für REST oder SOAP einfach als Maven-Abhängigkeit eingebunden werden *(siehe Listing 1)*. Anschließend wird ein Service Task angelegt, in dessen Konfiguration der gewünschte Connector samt Aufruf und Script-basierter Datenverarbeitung hinterlegt wird *(siehe Abbildung 2)*. Die Projekterfahrung zeigt, dass die Connectoren nur für sehr einfache Use Cases und Integrationstests geeignet sind. Insbesondere die Handhabung von Daten ist in Programmcode mit entsprechender Programmierumgebung deutlich besser zu handhaben. In den Camunda Best Practices selbst wird zur Integration mit einem Java-Client im Vergleich zu den Camunda REST- und SOAP-Connectoren geraten *[3]*. Auch wenn die Möglichkeit besteht, eigene Connectoren zu entwickeln, bieten sich hier meist eigene Java Delegates an, die im nächsten Abschnitt näher beschrieben werden.

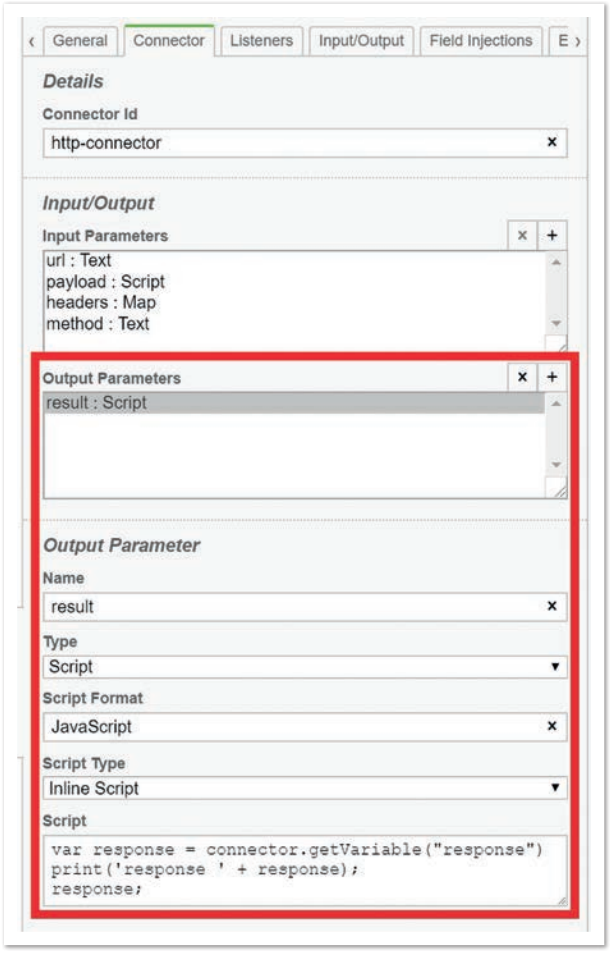


Abbildung 2: Konfigurationsbeispiel Camunda http-Connector (© Benjamin Klatt)

Plain Old Java Delegate

Die rudimentärste und gleichzeitig flexibelste Möglichkeit, externe Systeme in einer Camunda-Prozessanwendung zu integrieren, ist die Implementierung eines klassischen Delegates mit "Plain Old Java". Dazu erstellt man einen Service Task im Modell und konfiguriert ihn als Delegate-Implementierung samt gewünschter Delegate ID *(siehe Abbildung 3)*. Anschließend annotiert man die eigene JavaDelegate-Implementierung mit der gleichen, eindeutigen ID als Komponente wie in *Listing 2* gezeigt. Die eigentliche Integration kann dann in der `execute`-Methode nach dem IPO-Pattern (Input-Process-Output) mit der vollen Java-Mächtigkeit gestaltet werden. Auch wenn die volle Java-Mächtigkeit bereitsteht, gibt es gleichzeitig aber keine Camunda-seitige Unterstützung, die einem Integrationsherausforderungen wie eingeschränkte Verfügbarkeit oder Failovers abnimmt.

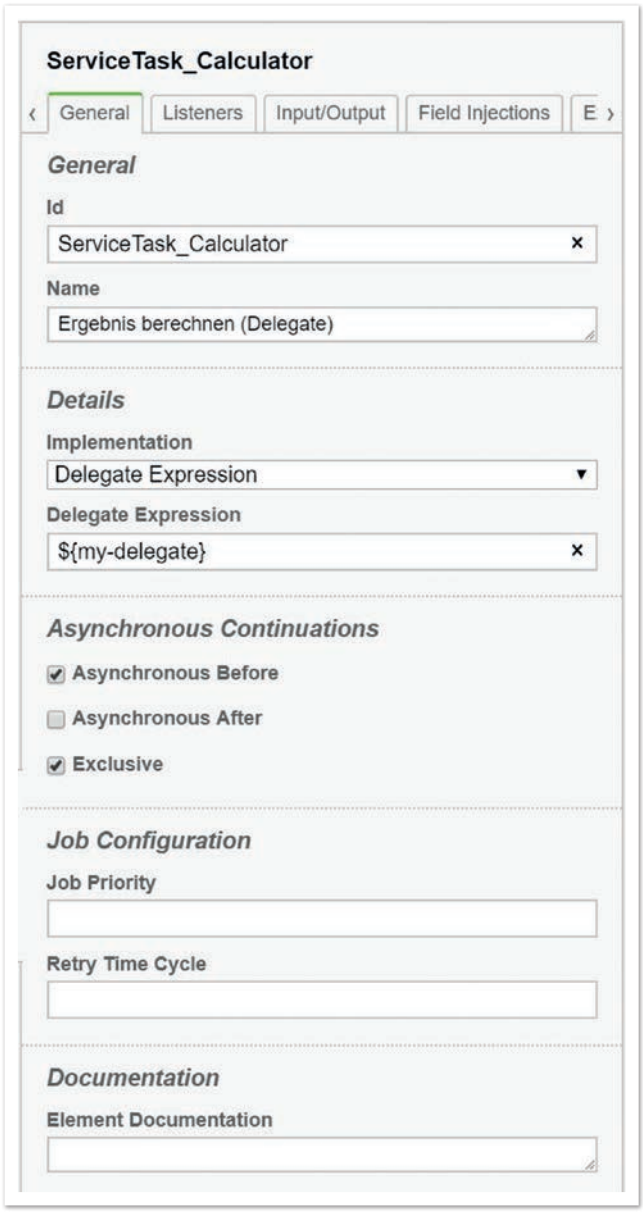


Abbildung 3: Delegate Konfiguration im Model, id=my-delegate (© Benjamin Klatt)

```java
@Component("my-delegate")
public class CalculatorDelegate implements JavaDelegate {

public void execute(DelegateExecution execution) throws Exception{
     // Read from context
     LongValue num1 =  execution.getVariableTyped("num1");
     // Write to context
     execution.setVariable("result", 1+2);
  }
}
```

Listing 2: Beispiel-Implementierung eines Delegate

Send & Receive

Der Send-and-Receive-Ansatz erlaubt, eine fachliche Asynchronität im Modell explizit zu modellieren. Zunächst wird im Send Task, zum Beispiel mit einem JavaDelegate, ein externes System aufgerufen und eine eindeutige Correlation ID übergeben. Danach wartet der Prozess im folgenden Catch-Event auf eine Eingangsnachricht vom passenden Message-Typ und mit der zuvor übergebenen Correlation-ID zur aktuellen Prozessausführung. Die Eingangsnachricht kann von dem integrierten System direkt an die Camunda REST API geschickt werden. Wichtig hierbei ist nur, dass die übergebenen Daten für Message Name und Correlation-ID passen *(siehe Abbildung 4)*. Camunda übernimmt dann automatisch die Zuordnung von eingehender Nachricht und wartender Prozessinstanz.

External Tasks

External Tasks sind eine explizite, asynchrone Integrationsmöglichkeit von Camunda. Besonderheit der Technik ist, dass die Implementierung eines Service Tasks nicht in der Prozessanwendung selbst liegt, sondern von einem externen Programm, einem sogenannten Worker, übernommen wird. Erreicht eine Prozessausführung einen solchen Service Task, dann steht seine Bearbeitung als Job über die Camunda API bereit und kann von einem Worker abgeholt werden. Das Ergebnis sendet ein Worker ebenfalls über die API an die Prozessanwendung zurück. Vorteil dieser Technik ist, dass sie eine asynchrone Verarbeitung mit einer sehr guten Skalierbarkeit ermöglicht, denn dank eines Fetch-and-Lock-Mechanismus können beliebig viele Worker parallel Jobs abarbeiten. Sind die Worker selbst einmal nicht verfügbar, dann puffert Camunda automatisch die Jobs, bis sie wieder abgeholt werden. Ein weiterer großer Vorteil ist, dass Jobs, die nicht erfolgreich abgearbeitet werden konnten, wahlweise einen Incident auslösen oder einfach für einen erneuten Bearbeitungsversuch zurückgegeben werden können. Viele weitere Features wie Topics und Priorisierung helfen, den Umgang mit External Tasks zu vereinfachen *[4]*. *Abbildung 5* zeigt die Konfiguration eines External Tasks im Modell sowie Beispielaufrufe über die REST API. Die Implementierung der Worker kann frei mit jeder beliebigen Technologie gestaltet werden. Wir nutzen oft auch eine Kombination mit Apache Camel als Integration Layer und/oder Spring wahlweise über die Camunda REST- oder Java-API.

BPM-RPA-Kombination

Als Letztes möchten die Autoren auf eine Möglichkeit eingehen, die es erlaubt, Systeme zu integrieren, die nur über eine grafische Benutzeroberfläche verfügen. Robotic Process Automation (RPA) ist eine Technik, bei der eine Software die Benutzeroberfläche wie ein menschlicher Benutzer verwendet, indem Felder gefüllt oder Buttons geklickt werden. Schon vor vielen Jahren waren Automatisierungen, beispielsweise mit Windows-Makros, möglich. Heutzutage gibt es jedoch deutlich reifere Werkzeuge, die die entwickelten Roboter-Automatisierungen direkt über eine Programmierschnittstelle zugänglich machen. Hat man eine gewünschte Interaktion mit einem zu integrierenden System mittels RPA erstellt, dann kann diese über die Programmierschnittstelle von der Prozessanwendung aus angesprochen werden. Letzteres kann wiederum über eine der zuvor vorgestellten Möglichkeiten realisiert werden. Die Autoren selbst greifen für RPA-Anforderungen unter anderem auch auf

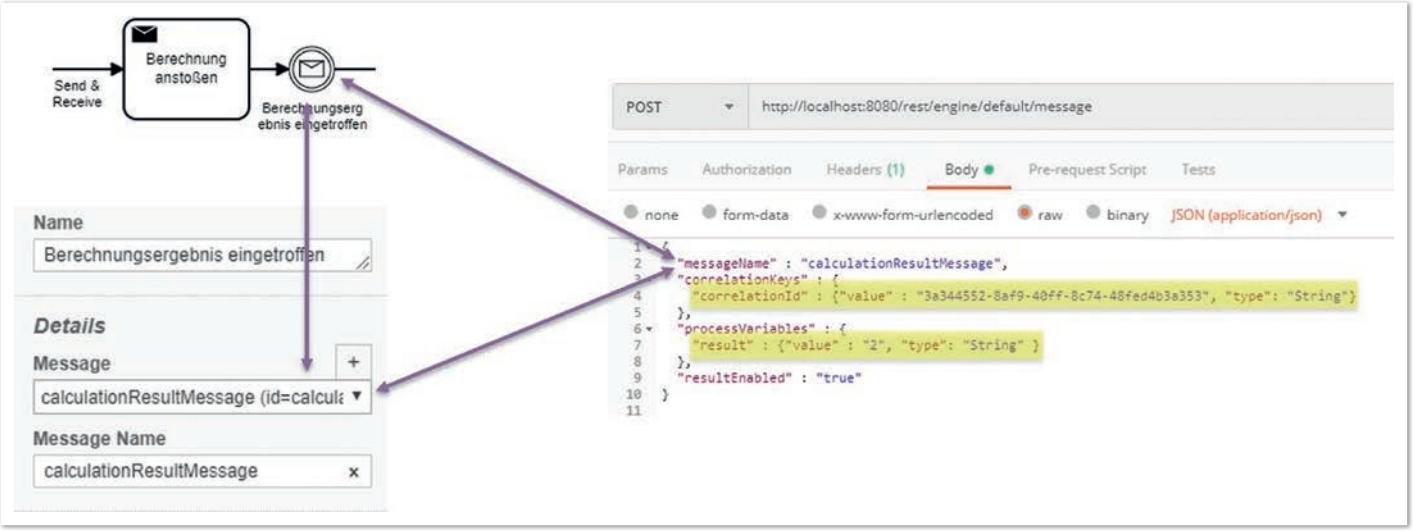


Abbildung 4: Send & Receive Zusammenspiel von Modell und Rückantwort (© Benjamin Klatt)

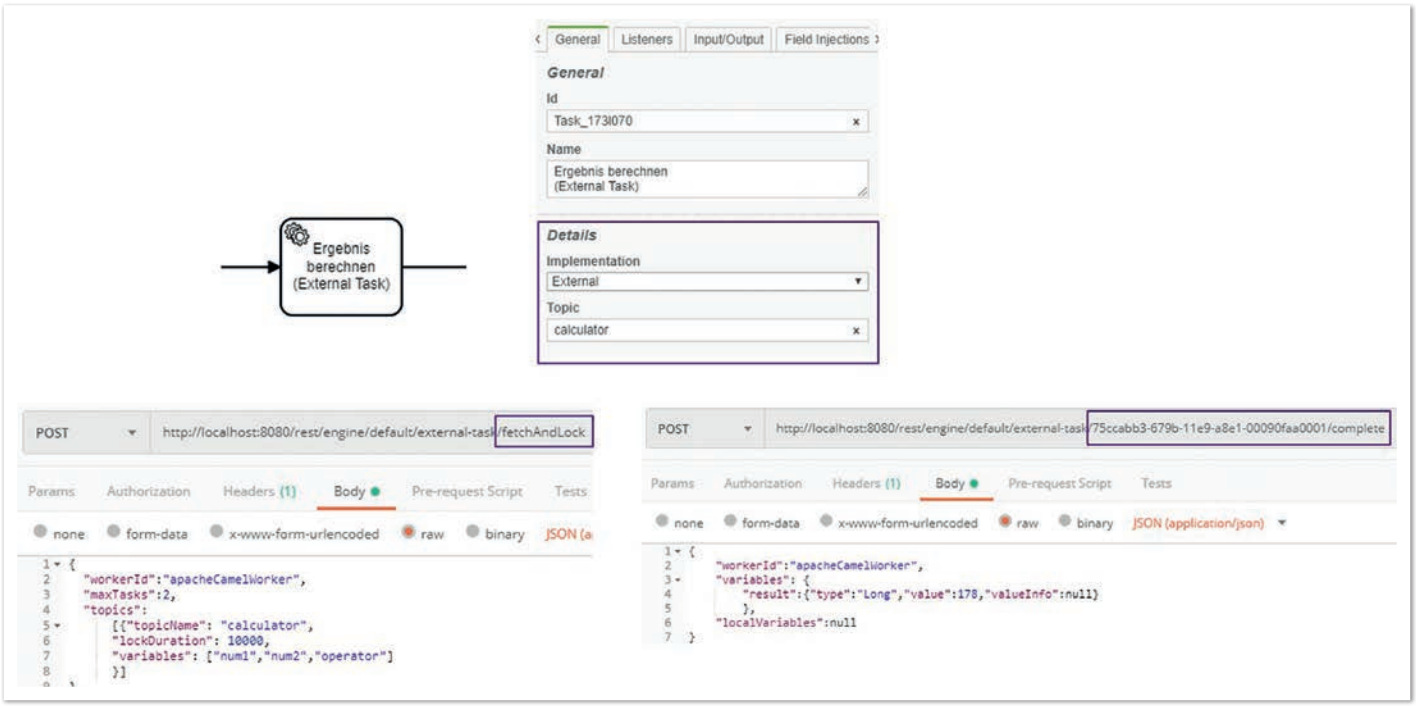


Abbildung 5: External Tasks Konfiguration und REST API Basisaufrufe (© Benjamin Klatt)

ein viadee RPA-Werkzeug zurück *[5]*. Darüber hinaus gibt es aber auch noch viele andere Werkzeuge am Markt. Generell sollte man sich jedoch bewusst sein, dass eine RPA-Lösung zwar kurzfristig die Erstellung einer technischen Schnittstelle zu dem entsprechenden Legacy-System ersparen kann, mittel- und langfristig warten aber potenziell sehr hohe Wartungsaufwände.

Auswahl und Architektur-Dimensionen

Gilt es nun, die passende Integrationstechnik zu wählen, können die anfänglich vorgestellten Architektur-Dimensionen weiterhelfen. *Abbildung 6* zeigt eine Übersicht, in der die Autoren die verschiedenen Techniken den jeweiligen Architekturanforderungen gegenüberstellen und ihre Unterstützung mit einem grünen Haken kennzeichnen. So fallen Camunda Connector und Plain Old JavaDelegate (POJD) raus, wenn eine asynchrone Verarbeitung von der Integration gefordert ist. Auf der anderen Seite ist ein Push bei External Tasks vom Mechanismus selbst erst mal nicht vorgesehen. Hierfür gibt es zwar Optimierungsstrategien, sie würden aber den Umfang dieses Artikel überschreiten.

Eignung für die Legacy Integration

Wie gezeigt ist eine Integration von Legacy-Systemen mit allen Camunda-Techniken prinzipiell möglich. Es kommt jedoch auf die jeweiligen konkreten, architekturellen Anforderungen an. In der Übersicht in *Abbildung 7* haben die Autoren noch mal einige Vor- und Nachteile der verschiedenen Techniken gegenübergestellt.

So bietet ein Camunda Connector zwar ein schnelles Setup und eine Versionierung mit dem Modell, seine eingeschränkte Testbarkeit, beispielsweise in Unit-Tests, und seine limitierte Unterstützung komplexerer Integrationen erschweren jedoch einen produktiven Einsatz. JavaDelegates dagegen erlauben maximale Flexibilität durch die volle Java-Mächtigkeit, man bekommt jedoch keinerlei Integrationsunterstützung geboten. Der Send-and-Receive-Ansatz bietet eine explizite fachliche Darstellung der Asynchronität im Modell, kommt aber mit einer starken Abhängigkeit von der Stabilität des Transportmediums und mit einer eingeschränkten Fehlerdarstellung bei Verarbeitungsproblemen. Letztendlich bieten die External Tasks sehr gute Möglichkeiten für

	synchron / asynchron	push / pull	nativ / wrapper	remote / embedded
Connectors	✓ ✗	✓ ✗	✓ ✓	✓ ✗
POJD	✓ ✗	✓ ✗	✓ ✓	✓ ✓
Send & Receive	✗ ✓	✓ ✗	✓ ✓	✓ ✗
External Tasks	✗ ✓	✗ ✓	✓ ✓	✓ ✗
RPA	✓ ✓	✓ ✓	✗ ✓	✓ ✗

Abbildung 6: Gegenüberstellung der Techniken zu Architektur-Anforderungen (© Benjamin Klatt)

	Pro	Kontra
Connectors	Schnelles Setup Versionierung mit Modell	Eingeschränkte Tests Limitierte Komplexität
POJD	Volle Java-Mächtigkeit	Entwicklung von Grund auf
Send & Receive	Explizite fachl. Asynchronität	Eingeschränkte Fehlerdarstellung Modellgröße
External Tasks	Robustheit & Skalierung Technische Asynchronität	Es muss jemanden geben der „abholt“
RPA	Integration auch über GUI Makro- und Mikroprozess	Wartung sehr aufwändig Technische Schulden

Abbildung 7: Vor- und Nachteile der Integrationstechniken (© Benjamin Klatt)

robuste und gut skalierende Anwendungen und sind empfehlenswert für technische Ansynchronität. Größte Herausforderung der External Tasks ist sicherlich die Implementierung der Worker zur Job-Verarbeitung.

RPA als Sonderfall hat das Alleinstellungsmerkmal, auch Systeme integrieren zu können, wenn außer einer Benutzeroberfläche keine weitere Schnittstelle existiert. Nicht zu vergessen sind hier aber die aufwendige Wartung und die technischen Schulden, die man sich hier einkauft.

Fazit und Good Practices

Als generelles Fazit lässt sich festhalten, dass die Camunda Workflow Engine ihr Flexibilitätsversprechen in Bezug auf Systemintegrationen einhält. Die vorhandenen technischen Möglichkeiten erlauben es, geeignete Lösungen für jedes anzubindende System zu entwickeln. Nicht zu vergessen ist jedoch, dass bei der Integration von Legacy-Systemen viele Herausforderungen unabhängig von Camunda sind. Als Beispiel seien hier instabile oder fachlich ungünstig geschnittene Schnittstellen genannt. Zum Abschluss noch drei Good Practices für das eigene Projekt:

- Nachhaltigkeit bei der Architekturentscheidung beachten
- Zu hohe Heterogenität bei der Implementierung vermeiden
- Leichtgewichtige Integration Layer, beispielsweise mit Apache Camel, in Betracht ziehen

Außerdem haben die Autoren Code-Beispiele für die verschiedenen Integrationsmöglichkeiten im GitHub Repository bereitgestellt *[6]*.

Quellen

[1] *https://camunda.com/de/solutions/*
[2] *https://docs.camunda.org/manual/latest/reference/connect/*
[3] *https://camunda.com/best-practices/invoking-services-from-the-process*
[4] *https://docs.camunda.org/manual/latest/user-guide/process-engine/external-tasks/*
[5] *https://www.viadee.de/loesungen/rpa-robotic-process-automation*
[6] *https://github.com/viadee/camunda-integration*

Dr. Benjamin Klatt
viadee Unternehmensberatung AG
benjamin.klatt@viadee.de

Dr. Benjamin Klatt ist Integrationsarchitekt und Agiler Coach bei der viadee AG. Als Integrationsarchitekt beschäftigt er sich vorrangig mit der Digitalisierung von Geschäftsprozessen und Produkten.

Matthias Schulte
viadee Unternehmensberatung AG
Matthias.Schulte@viadee.de

Matthias Schulte ist Senior Berater für BPM- und Integrationsprojekte. Sein Schwerpunkt liegt in der Einführung von Prozesssteuerungsplattformen in großen Unternehmen sowie dem (agilen) Projektmanagement von Digitalisierungsprojekten.

JavaScript, ECMAScript, ES.NEXT… noch Fragen?

Marc Teufel, hama GmbH & Co KG

Was ist eigentlich der Unterschied zwischen JavaScript und ECMAScript? Ist JavaScript wirklich so schlecht wie sein Ruf? Und was hat es mit ES.NEXT auf sich? Mit diesem Artikel möchte der Autor Sie auf eine kleine Tour in die aufregende Welt von JavaScript mitnehmen, die oben erwähnten Fragen klären, Ihnen zeigen, wie vital JavaScript wirklich ist und wie es sich weiterentwickelt. Sein Ziel hat er erreicht, wenn er bei Ihnen Interesse für das Thema wecken konnte. Denn eines ist sicher: Wenn Sie für die Zukunft gewappnet sein möchten, dann dürfte eine nähere Betrachtung dieser Sprache sicher nicht die schlechteste Option sein.

Dieses JavaScript ist schon so eine Sache, gar nicht so einfach. Eine Sprache voller Missverständnisse. Einige hassen sie, andere lieben sie. Und die Entwicklergemeinde? Ja, die Entwicklergemeinde ist geteilter Meinung, was sie von dieser mittlerweile über zwanzig Jahre alten Sprache halten soll. Und trotzdem ist es die Sprache des Webs und das wird sie wohl auch bleiben. JavaScript bildet die Grundlage und das Fundament. Gleichzeitig ist JavaScript eine sogenannte „Glue Language", denn ihre Eigenschaften, ihre Flexibilität und ihre starke Verbreitung macht sie maximal attraktiv und bringt ganz nebenbei unterschiedlichste Entwickler-Communities zusammen.

Grundlage und Fundament

Ob man sich als Entwickler später für Werkzeuge und Spracherweiterungen wie Flow oder TypeScript entscheidet oder gar auf ganz neue Sprachen wie ReasonML oder Elm wechselt – die Grundlage bildet immer JavaScript, denn sie alle kompilieren (ein anderes Wort hierfür wäre übrigens „transpilieren") in genau diese Sprache.

Dass JavaScript nichts mit Java zu tun hat, dürften viele wissen; dass der korrekte Name von JavaScript jedoch eigentlich „ECMAScript" (ES) ist, wissen dagegen schon weniger. JavaScript und ECMAScript sind also ein und dasselbe. ECMAScript ist eine dynamisch typisierte Sprache, sie ist objektorientiert, obwohl sie im Kern keine Klassen kennt. Diese Eigenschaften werden bei ECMAScript allerdings auch sehr oft kritisiert und stattdessen Spracherweiterungen wie TypeScript empfohlen. Aber gerade die Dynamik von ECMAScript ist es doch, was die Sprache so unglaublich flexibel macht und ihr die Eigenschaft der zu Beginn erwähnten „Glue Language" verleiht. Wer klassisch objektorientiert programmieren will, kann dies unter Verwendung von Klassen tun, die mit ES2015 eingeführt wurden. Alternativ kann man natürlich auch TypeScript einsetzen, mit dem man neben der klassischen objektorientierten Programmierung strenge Typisierung bekommt, wenn diese im Projekt erforderlich ist. Stattdessen kann man auch gänzlich in ECMAScript bleiben und beispielsweise die Flow-Erweiterung von Facebook nutzen, um strenge Typisierung zu bekommen.

Der vorige Abschnitt macht es hoffentlich deutlich: JavaScript – sorry, ECMAScript – ist ziemlich flexibel und genau das ist die große Stärke der Sprache. Wir können dynamisch oder stark typisiert arbeiten, alle Stufen dazwischen sind natürlich auch enthalten. Wir können klassisch objektorientiert entwickeln oder auf die im JavaScript-Sprachkern enthaltene, sogenannte „prototypische Objektorientierung" zurückgreifen. Dabei handelt es sich auch um Objektorientierung – aber eben auf eine ganz andere Art und Weise. Folgt man aktuellen Trends, ist Objektorientierung schon wieder ein alter Hut (– na ja, ist es ja eigentlich auch). Funktionale Programmierung wird heutzutage immer wichtiger und auch damit hat ECMAScript keine Probleme, was die Flexibilität nochmals unterstreicht.

JavaScript versus TypeScript

Ja, JavaScript ist wirklich so eine Sache. Soll man es nun damit versuchen? Oder doch besser auf TypeScript setzen, wie viele sagen? Eine schwierige Frage, die man nicht einfach beantworten kann. Der größte Vorteil von TypeScript ist sicherlich die Tatsache, dass es sich um eine sogenannte „statisch typisierte" und vollständig objektorientierte Sprache handelt. Das bedeutet, dass bereits zum Zeitpunkt der Programmierung Datentypen festgelegt werden können. Eigentlich genau so, wie wir es von populären Sprachen wie Java und C# schon seit Jahren gewöhnt sind. Diese Tatsache und auch der Umstand, dass TypeScript in dem Sinne objektorientiert ist, wie wir es heute kennen, sind vermutlich die Gründe dafür, in größeren Projekten auf TypeScript statt auf JavaScript zu setzen.

Vielleicht haben die Leute, die solche Empfehlungen aussprechen, auch recht – nein, sie haben ganz bestimmt recht! Trotzdem möchte der Autor Sie bitten, nochmal kurz innezuhalten. Gehen Sie in sich, hören Sie auf Ihr eigenes Bauchgefühl und entscheiden Sie für sich. Es muss nicht immer unbedingt alles richtig sein, was andere behaupten und das schließt diesen Artikel (und auch die Meinung des Autors) nicht aus!

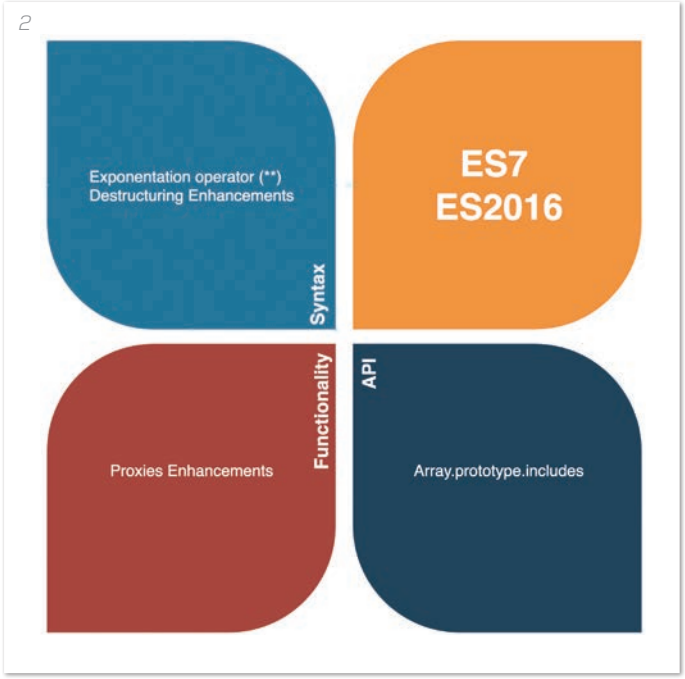
JavaScript oder ECMAScript – beide Bezeichnungen sind ja synonym – ist eine interpretierte Programmiersprache. Klassische Objektorientierung ist in neueren Versionen (ab ES2015 aufwärts) durchaus vorhanden, jedoch bei Weitem nicht so ausgeprägt wie in TypeScript. Mit der sogenannten „prototypischen Objektorientierung" verfügt ECMAScript über die gleichen Möglichkeiten, nur funktioniert es im Kern anders, als wir es von der klassischen objektorientierten Programmierung (OOP) gewohnt sind.

Mit TypeScript kann man ganz klassisch objektorientiert programmieren und die Resultate werden nach JavaScript kompiliert oder „transpiliert" und laufen dort in Form von prototypischer Objektorientierung ab. So schräg kann JavaScript doch gar nicht sein, wenn es die Zielplattform für TypeScript-Programme ist? Und ergibt es nicht doch Sinn, sich mit prototypischer Objektorientierung zu beschäftigen? Hält man bestimmte Entwurfsmuster ein oder bedient sich Features neuerer JavaScript-Versionen, lässt sich auch mit JavaScript gut arbeiten und man kommt da relativ schnell rein. Und wenn ich schon eine neue Sprache erlernen muss, ganz egal ob ich nun von Java oder sonst woher komme, wieso nicht gleich die Sprache erlernen, die die Laufzeitumgebung darstellt? Wieso sind die meisten wichtigen Frameworks, die es mittlerweile gibt, in JavaScript und nicht in TypeScript geschrieben? Wieso findet man im Netz Unmengen an Informationen, Tutorials und Videos zu JavaScript, aber weitaus weniger zu TypeScript? Warum sich in die Abhängigkeit eines Herstellers begeben (obwohl dieser in den letzten Jahren durch seine Aktivitäten für die Entwickler-Community durchaus Pluspunkte sammeln konnte), wenn man doch auch gleich die standardisierte und unabhängige Ablaufumgebung nutzen kann?

Fragen über Fragen, auf die ein jeder seine eigenen Antworten finden und für sich selbst entscheiden muss. Ein ganz neuer Gedanke: **Was halten Sie davon, JavaScript und TypeScript gar nicht als Konkurrenten zu sehen, sondern als sich ergänzende Partner?** Die Grundlage, die Basis, das Fundament bildet JavaScript. TypeScript ist ein weiteres Werkzeug im Werkzeugkasten, das wir zusätzlich einsetzen können, wenn wir es brauchen. Oder eben auch nicht.

ECMAScript-Versionen

Weiter oben fiel mit „ES2015" bereits ein wichtiger Begriff. Hierbei handelt es sich um einen wichtigen Meilenstein in der Geschichte der Sprache. Vor ES2015 schritt die Entwicklung eher gemächlich dahin. Da vergingen zwischen einzelnen Versionen schon mal ein paar Jahre. Seit ES2015 (ECMAScript, Version 6) gibt es jedes Jahr eine neue Version. Dabei markiert ES2015 insofern ei-



Abbildungen 1-4: Features und Neuerungen in den letzten ECMAScript-Versionen (© Marc Teufel, hama GmbH & Co KG)

nen entscheidenden Schritt, als mit dieser Version eine sehr große Anzahl Verbesserungen, Fehlerbehebungen und neue Funktionen in die Sprache eingezogen sind. In den Abbildungen 1 bis 4 kann man sehen, wie sich JavaScript in den letzten Jahren in den Bereichen API- und Syntax-Verbesserung sowie in Bezug auf neue Funktionen weiterentwickelt hat. Auf einzelne Features gehen wir nicht weiter ein, da dieser Artikel sich eher mit JavaScript/ECMAScript im Allgemeinen beschäftigt. Die interessierte Leserin und der interessierte Leser können jedoch im Internet nach den in den Abbildungen genannten Stichwörtern suchen und sich so zusätzliche Informationen besorgen. Ganz bestimmt lohnen dürfte sich auch ein Blick in die ECMAScript-Versionsübersicht/Kompatibilitätstabelle, die von GitHub-User „kangax" gepflegt wird *[1]*. Auf dieser Seite kann man sich durch die einzelnen ECMAScript-Versionen klicken und bekommt eine Auflistung aller in dieser Version enthaltenen Features. Jeder Eintrag ist mit einem Link versehen, der wieder zu weiteren Detail-Informationen, Code-Beispielen sowie der Dokumentation und Spezifikation des jeweiligen Features führt. Empfehlenswert ist diese Seite auch, weil sie anzeigt, welche der genannten Funktionen in welcher der vielen möglichen Ablaufumgebungen für ECMAScript verfügbar ist. Browser sind dabei nur eine Umgebung, in denen JavaScript-Programme laufen können. Die Seite informiert auch darüber, welches Feature in den Bereichen Compiler/Transpiler, Server-Runtimes und im mobilen Umfeld verfügbar ist oder nicht.

Seit dem Mega-Release im Jahr 2015 bekommen wir jedes Jahr eine neue JavaScript-Version. Dabei fällt auf, dass die Anzahl und Dichte der Neuerungen ziemlich überschaubar geworden ist. Woran liegt das?

TC39

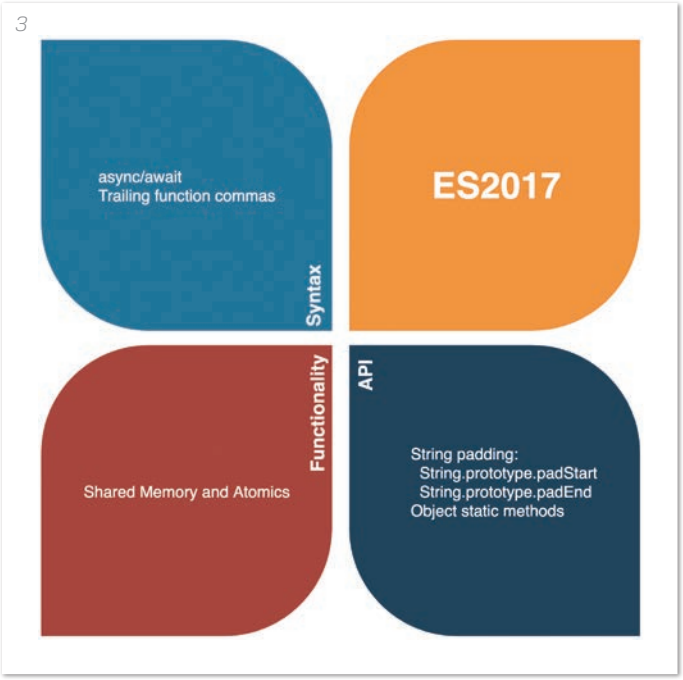
Seit dem Release ES2015 gibt es jährlich eine neue ECMAScript-Version. Dabei ist es wichtig zu verstehen, dass nur die neuen Features in das nächste Release aufgenommen werden, die einen langwierigen und aufwendigen Entwicklungsprozess durchlaufen haben. Sie müssen letztlich von einem Gremium, dem TC39, freigegeben werden. Die TC39 ist eine Gruppe von Entwicklern unterschiedlicher Firmen und setzt sich aus Mitarbeitern von Google, HP, Microsoft, IBM, Intel, PayPal, Apple, Facebook, Netflix, diversen Universitäten und vielen weiteren Firmen zusammen. JavaScript wird also von einer Gemeinschaft vorangetrieben und die TC39 ist ihr Kontrollgremium. Jedes (zahlende) Mitglied von TC39 darf einen Repräsentanten benennen. Die Repräsentanten dürfen wiederum an den alle zwei Monate stattfindenden Meetings teilnehmen, bei denen einzelne Feature-Proposals diskutiert werden. Alle dürfen Ideen für neue Funktionen in JavaScript einbringen. Hierzu muss ein sogenanntes „Feature Proposal" eingereicht werden. In den soeben erwähnten Meetings der TC39 werden diese Feature Proposals diskutiert: Ist das Proposal sinnvoll? Welche Auswirkungen hat das Proposal? Bringt es die Sprache grundsätzlich weiter? Ein neu eingereichtes Proposal landet erstmal im Zustand „Stage 0". Entscheidungen werden im Gremium stets über Abstimmungen getroffen. Erst, wenn alle zustimmen, erreicht ein Proposal seinen nächsten Level oder Stage.

Auf der Website des TC39 *[2]* kann man sich über das Gremium, vor allem aber über den aktuellen Stand der einzelnen Proposals, informieren. Was ist gerade in Entwicklung und wie wird das nächste ECMAScript-Release aussehen? Diese und weitere Informationen bekommt man dort. Der wichtigste Bereich hier dürfte vermutlich „Spec → Active Proposals" ein. Dieser Link leitet Sie auf eine GitHub-Seite weiter. Ja, JavaScript wird auf GitHub entwickelt und gehostet. Hier findet sich eine komplette Übersicht über alle bestehenden Proposals und ihren derzeitigen Zustand (Stage).

Kleine Orientierungshilfe

Proposals, die im „Stage 0 (New)" vorliegen, sind ganz neu. Genau genommen ist es ein Pseudo-Stage, ein Platzhalter. Es existiert erstmal nur die Idee, aber noch keine wirkliche Implementierung. Wenn ein Proposal von der TC39 akzeptiert wurde, geht es über in den „Stage 1 (Discussion)". Das bedeutet allerdings noch lange nicht, dass es dieses Feature in ECMAScript schaffen wird – mitnichten! Es bedeutet zunächst einmal nur, dass sich die TC39 wei-

3



4

ter mit diesem Vorschlag beschäftigt, ihn auseinandernehmen und intensiv diskutieren wird. Erst, wenn dies zu einem allgemeinen Konsens führt, geht es mit dem Vorschlag weiter. Hat ein Proposal den „Stage 2 (Dev)"-Status erreicht, dann könnte das Feature tatsächlich in einem der nächsten jährlichen Releases landen. Eine genaue Spezifikation wird entwickelt und das Verhalten des Features genau festgelegt. Im „Stage 3 (Proof of Concept)" ist das Feature so weit vorangeschritten, dass einige JavaScript-Engines und Browser dieses bereits implementieren. Features im Stage 3 beweisen durch ihre Referenzimplementierung also, dass sie funktionieren. Wenn ein Feature „Stage 4 (Final)" erreicht hat, dann müssen alle wichtigen JavaScript-Umgebungen und Browser die neue Funktion implementieren, die Final Specification ist zu diesem Zeitpunkt auch fertiggestellt. Einmal im Jahr, meistens im März, schaut das TC39-Gremium über alle Features, die sich im „Stage 4" befinden, und entscheidet, welches letztlich ins nächste finale ECMAScript-Release einziehen darf. Im Juni wird die neue ECMAScript-Version final veröffentlicht.

... Und was bitteschön ist jetzt dieses ES.NEXT?

Nochmal zurück zu der bereits erwähnten GitHub-Seite *[3]*, die den Überblick über die aktuellen Proposals liefert. Im separat verlinkten Dokument „Finished Proposals" sieht man, welche neuen Funktionen mit welchem Jahres-Release von ECMAScript veröffentlicht wurden beziehungsweise noch veröffentlicht werden.
Dort findet man auch ES.NEXT. ES.NEXT ist nicht nur ein Wort, sondern man kann ganz genau mitverfolgen, wie sich JavaScript weiterentwickelt und wo die Reise hingeht. Mehr noch: Zukünftige Features ab „Stage 3" kann man bereits frühzeitig ausprobieren, auch schon dann, wenn es noch gar nicht offizieller Bestandteil eines jährlichen ECMAScript-Release ist. Schöne, neue offene Welt (und das ist nicht mal sarkastisch gemeint).

Und wie geht's jetzt weiter?

Der Autor hofft, dass er Sie mit diesem Text ein bisschen neugierig machen konnte. Im JavaScript-/ECMAScript-Dunstkreis passiert zurzeit so unglaublich viel, dass all dies unmöglich in einem Artikel erwähnt werden kann. Im Rahmen des JavaLand-Schulungstags behandelt und vertieft der Autor die Thematik in seinem Seminar „ECMAScript ist anders!" am Schulungstag der JavaLand 2020 *[4]*.

Quellen

[1] ECMAScript-Feature-Matrix: *http://kangax.github.io/compat-table/es6/*
[2] TC39: *http://tc39.es*
[3] Übersicht über aktuell in Entwicklung befindliche JavaScript-Features (ES.NEXT): *https://github.com/tc39/proposals*
[4] ECMAScript ist anders! *https://www.javaland.eu/de/programm/schulungstag/*

Marc Teufel
hama GmbH & Co KG
marc.teufel@hama.de

Marc Teufel arbeitet bei hama und ist dort für die Entwicklung großer Anwendungen im Logistikumfeld verantwortlich. Er ist Autor einer Vielzahl von Artikeln und Büchern rund um Software-Entwicklung. Nach so vielen Jahren ist er immer noch „Driven by passion".

Das Design von Formularen

Maximilian Liesegang, esentri

Formulare sind der Mittelpunkt fast aller Interaktionen. Sie stehen zwischen uns und dem, was wir versuchen zu erledigen. Täglich sind wir mit einer großen Anzahl von Formularen konfrontiert, wenn wir zum Beispiel privat oder beruflich mit IT-Anwendungen eine Tätigkeit durchführen wollen. Egal ob man ein Ticket kaufen, ein Hotelzimmer buchen oder etwas online bestellen möchte, höchstwahrscheinlich musste man währenddessen ein Formular ausfüllen. Formulare sind aber nur ein Mittel zum Zweck. Die Benutzer sollten in der Lage sein, sie schnell und verwirrungsfrei abzuschließen. In diesem Artikel lernen Sie praktische Tipps kennen, die Ihnen helfen, ein effektives Formular zu gestalten.

Formulare sind bescheiden. Es ist einfach ihr Ziel, ausgefüllt zu werden. Zwei Faktoren haben dabei einen großen Einfluss auf die Rate, in der Menschen dieses Ziel erfüllen. Zum einen die Wahrnehmung der Komplexität und zum anderen die Interaktionskosten eines Formulars.

Das erste, was Benutzer tun, wenn sie ein neues Formular sehen, ist abzuschätzen, wie viel Zeit zum Ausfüllen insgesamt benötigt wird. Benutzer tun dies, indem sie das Formular überfliegen. Die Wahrnehmung spielt eine entscheidende Rolle im Prozess der Schätzung. Je komplexer ein Formular aussieht, desto wahrscheinlicher ist es, dass der Benutzer den Prozess abbrechen wird.

Die Interaktionskosten sind die Summe der kognitiven und auch der physischen Anstrengungen, die Benutzer in die Interaktion mit dem Formular stecken, um ihr Ziel zu erreichen. Sie haben einen direkten Zusammenhang mit seiner Benutzerfreundlichkeit. Je mehr Aufwand die Benutzer aufwenden müssen, um ein Formular auszufüllen, desto weniger benutzbar ist das Formular. Hohe Interaktionskosten können das Ergebnis von erschwerter Eingabe, der Unfähigkeit, die Bedeutung einiger Fragen zu verstehen, oder Verwirrung über Fehlermeldungen sein.

Diese Faktoren sind durch folgende Grundsätze positiv beeinflussbar: Reduziert die kognitive Belastung, macht es schwerer, Fehler zu begehen, und macht es menschlicher.

Schlechte Formulare kosten viel Geld

Diese Erfahrung durfte das amerikanische Online-Reisebüro Expedia machen. Wie üblich, suchten und fanden Kunden auf ihrer Website die gewünschten Hotels, gaben ihre Rechnungsinformationen ein und drückten auf die „Kaufen"-Schaltfläche. Expedia konnte jedoch die große Mehrheit dieser Zahlungen nicht durchführen. Was war der Grund dafür?

Analysten begannen, die fehlgeschlagenen Zahlungen zu untersuchen, um festzustellen, welche Eigenschaften sie gemeinsam hatten. Die Antwort, so stellte sich heraus, war ganz einfach. Das Web-Formular hatte unter dem Feld „Name" ein optionales Feld „Firma". Diese Konstellation verwirrte einige Kunden, die das Feld „Firma" mit ihrem Banknamen ausfüllten. Nach der Eingabe ihres Banknamens gaben diese Kunden dann in das Adressfeld die Adresse ihrer Bank und nicht ihre Privatadresse ein. Als es um die Adressverifizierung zur Verarbeitung der Zahlung ging, scheiterte diese, weil in dem Feld nicht die Adresse des Kreditkarteninhabers eingetragen wurde. Nachdem Expedia dies bemerkte, entfernten sie das entsprechende Feld sofort. Über Nacht stieg die Zahl der erfolgreichen Transaktionen an. Nach Angaben der Firma konnte so pro Jahr 12 Millionen Dollar mehr Umsatz gemacht werden.

Reduziert die kognitive Belastung

Im folgenden Abschnitt finden sich vier Tipps, um die kognitive Belastung der Nutzer zu reduzieren. So können die Interaktionskosten und die Komplexität der Formulare vermindert werden.

Eine Sache pro Seite bedeutet, einen komplizierten Prozess in kleine Stücke zu zerlegen und diese auf eine einzelne Seite zu verteilen. Anstatt beispielsweise Lieferadresse, Lieferoptionen und Zahlungsformulare auf einer langen Seite darzustellen, werden sie auf separaten Seiten dargestellt. Es geht dabei nicht zwingend darum, auf jeder Seite ein einziges Formularfeld zu haben, obwohl das auch möglich ist. Durch den ständigen Fortschritt wird die Vervollständigungsrate der Benutzer gefördert. Außerdem kann mit Fortschrittsbalken die Unsicherheit der Benutzer reduziert werden. Diese Balken schaffen Transparenz darüber, wie weit man von der Vervollständigung entfernt ist. Ein kleiner Hack ist es, den Fortschrittsbalken initial nicht bei 0% starten zu lassen. Untersuchungen haben ergeben, dass so die Vervollständigungsrate gesteigert werden kann.

Formularfelder gehören in *eine Spalte*. Tests zeigen, dass mehrspaltige Formularlayouts anfällig für Fehlinterpretationen sind. Die Folgen sind, dass Benutzer Felder überspringen, in denen sie tatsächlich Daten eingeben müssen, Daten in die falschen Felder eingeben oder einfach zum Stillstand kommen und rätseln, wie sie mit der Eingabe fortfahren sollen. Anstatt eine visuelle Neuorientierung des Benutzers zu erfordern, halten Sie ihn im Fluss, indem Sie sich an eine einzelne Spalte mit einer separaten Zeile für jedes Feld halten. Ausnahmen von dieser Regel gibt es auch. Kurze und logisch zusammenhängende Felder wie Stadt, Bundesland und Postleitzahl können in derselben Zeile angezeigt werden.

Es ist empfehlenswert, ***Beschriftungen über die entsprechenden Formularfelder zu setzen***. Obwohl dies die Gesamtlänge des Formulars

Abbildung 1: Teilen Sie den Prozess auf mehrere Seiten auf (Quelle: Adam Silver) [1]

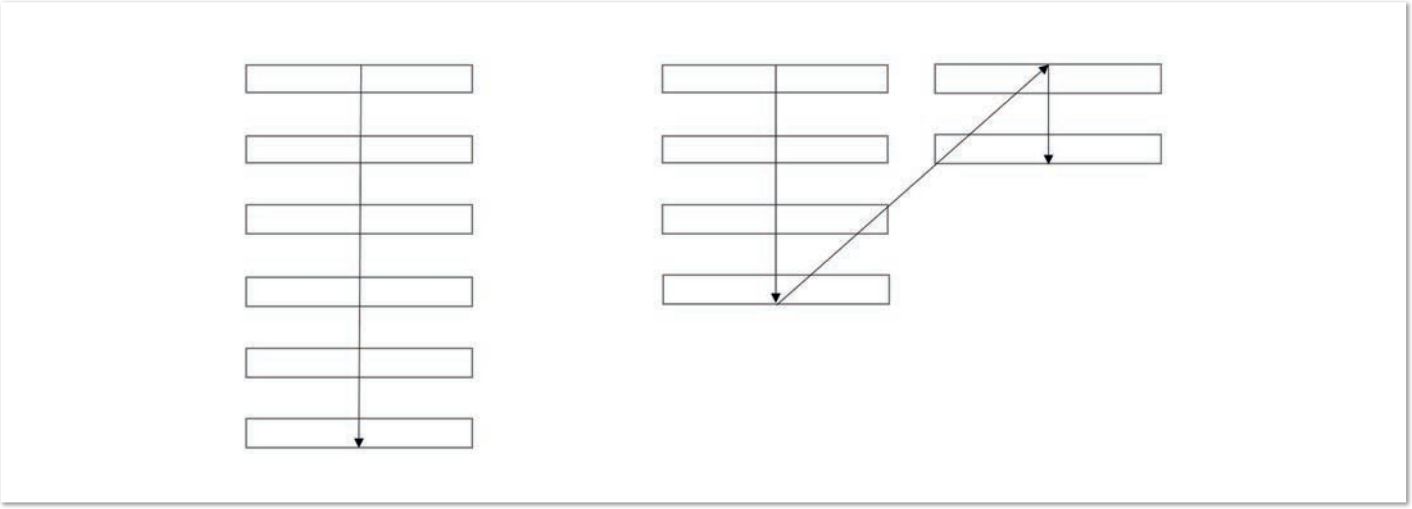


Abbildung 2: Einspaltige Layouts, wie links, sind angenehmer (Quelle: Maximilian Liesegang)

verlängert, erleichtert es das Überfliegen des Formulars, da der Benutzer das Feld in der gleichen Linie wie die Feldbeschriftungen sehen kann. Außerdem erspart es den Augen des Benutzers, zwischen der Beschriftung und dem Eingabefeld hin- und herzuspringen *(siehe Abbildung 3)*. Diese Platzierung ermöglicht auch längere Feldbeschriftungen, da horizontaler Abstand kein Problem darstellt. Wenn es um die Formularlänge geht, kann man links neben den Textfeldern Feldbezeichnungen platzieren. Achten Sie jedoch in diesem Fall darauf, dass die Etiketten gleich lang sind und so nah wie möglich an den Textfeldern platziert werden. Wenn die Etiketten zu weit links liegen, kann es schwierig sein, das richtige Etikett dem entsprechenden Feld zuzuordnen. Wenn die Nähe ein solches Problem ist, warum platziert man dann nicht die Beschriftungen in den Feldern? Wenn man die Beschriftung als Platzhalter in das Formularfeld einfügt, verschwindet es, wenn Benutzer ihren Text eingeben. So müssen sie es sich merken, wenn sie das Feld ausfüllen wollen. Dies führt insbesondere dann zu einem Problem, wenn Benutzer mit der Tabulatortaste durch ein Formular navigieren. Wenn sie mit der Tabulatortaste zum nächsten Feld navigieren, ohne zu schauen, verpassen sie, welche Informationen benötigt werden.

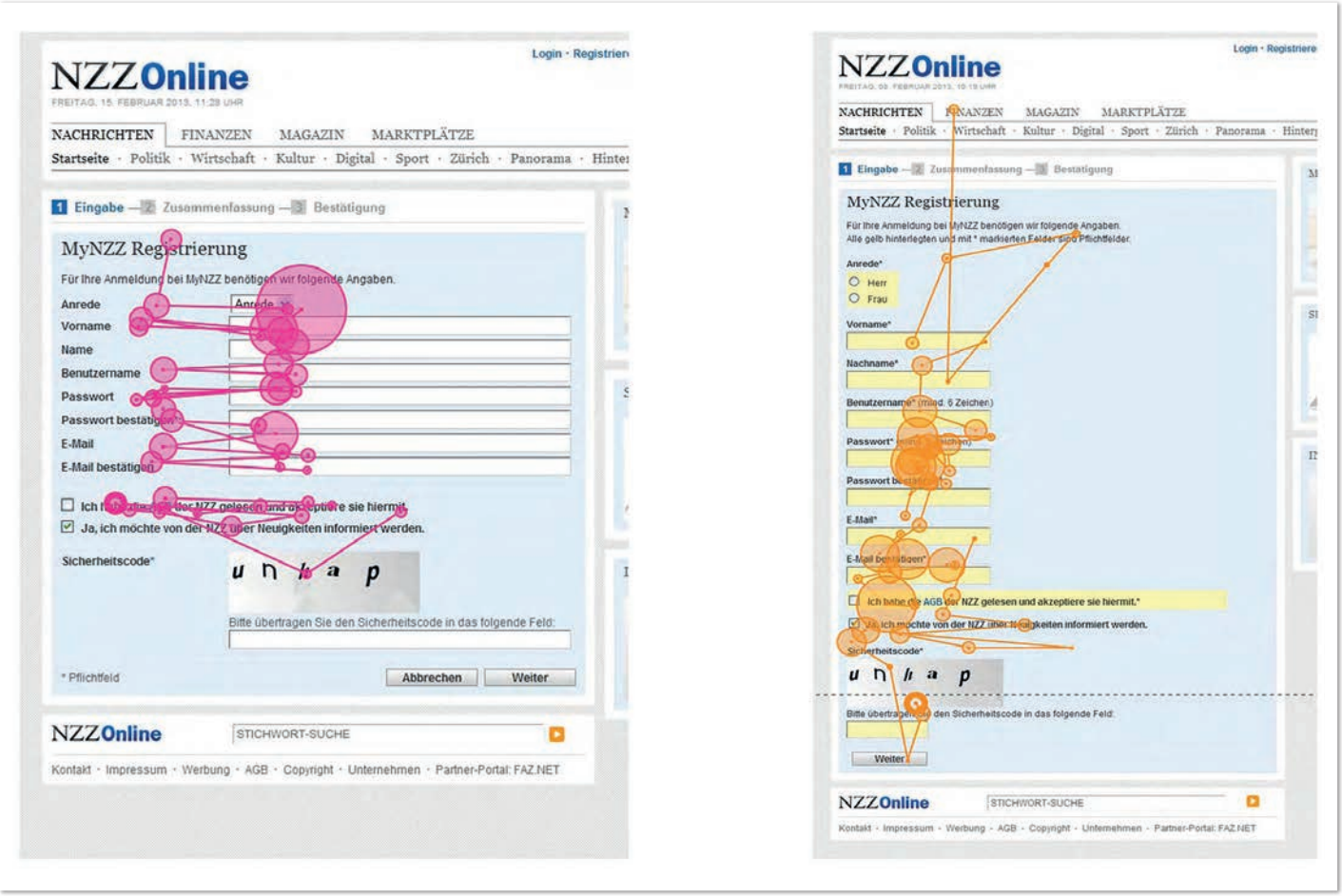


Abbildung 3: Eye-Tracking-Ergebnisse im Vergleich von Beschriftungen (Quelle: Mirjam Seckler et al.) [2]

Dropdowns können bei richtiger Verwendung sehr hilfreich sein. Sie sparen Platz auf dem Bildschirm und verhindern, dass Benutzer fehlerhafte Daten eingeben, da sie nur zulässige Optionen zeigen. Sie verfügen über viele nützliche Funktionen wie Gruppierungen und eine leichte Tastatur-Navigation. Allerdings werden Dropdowns auch sehr häufig in den falschen Situationen eingesetzt. Der Benutzer leidet initial unter einem Mangel an Informationen. Er muss erst klicken, um alle verfügbaren Optionen zu sehen. Solange es fünf oder weniger Einträge gibt, sollte man eine Radio-Komponente verwenden. Mit dieser können Benutzer sofort sehen, wie viele Optionen sie haben und was diese Optionen sind, ohne eine weitere Aktion durchführen zu müssen. Besonders Ihre mobilen Nutzer werden es Ihnen danken. Auf der anderen Seite der Skala gibt es auch zu viele Optionen für ein Dropdown. Wenn es größer als 15 Optionen ist, wird es für den Benutzer schwierig, die Informationen zu scannen und sie zu navigieren. Riesige Dropdown-Listen können für Ihre Benutzer ein echter Alptraum sein, da sie anfangen müssen, innerhalb der Liste zu scrollen, weil sie den Teil der Optionen außerhalb des Sichtbaren verbergen. Diese miserable Benutzererfahrung verlangsamt den gesamten Prozess. Stattdessen sollte man bei langen Listen das Dropdown um ein Eingabefeld erweitern. Mit diesem Feld filtert der Benutzer die Einträge und kommt so schnell und zuverlässig an sein Ziel.

Macht es schwerer, Fehler zu begehen

Benutzer werden Fehler machen. Es ist unvermeidlich. Deshalb ist es wichtig, ein Formular zu entwerfen, das die Benutzer in Momenten, in denen Probleme auftreten, unterstützt. Da das Thema „Fehler und Validierung" einen eigenen Artikel verdient, sind im Folgenden nur zwei Dinge aufgelistet, die zu tun sind, um den Benutzern zu helfen.

Indem Sie *sinnvolle Standardantworten* auf Fragen bereitstellen, sparen Sie den Benutzern kognitive Arbeit. Er muss nicht mehr den Aufwand betreiben, über die Antwort nachzudenken oder sie einzugeben. Das Ausfüllen von Formularen macht nie Spaß, aber wenn dieses Muster die Zeit halbiert, die ein Benutzer braucht, um es durchzuarbeiten, wird er dankbar sein. Möglicherweise werden die meisten Benutzer auf eine bestimmte Weise antworten. Oder der Benutzer hat bereits genügend Kontext-Informationen für die Benutzeroberfläche bereitgestellt, um genau vorherzusagen, welche Antwort er geben wird. Wenn beispielsweise jemand eine Postleitzahl eingibt, können Sie aus genau dieser Nummer die Stadt ableiten. Genauso kann man den Namen einer Person bei Adressen oder Zahlungsinformationen wiederverwenden. Bei semi-relevanten Fragen kann man vielleicht nicht erwarten, dass der Benutzer die Antwort weiß oder sich um sie kümmert, und „was auch immer das System vorgibt", reicht völlig aus. Wählen Sie aber keinen Standardwert, nur weil Sie der Meinung sind, dass Sie kein Feld leer lassen sollten. Tun Sie dies nur, wenn Sie sich ziemlich sicher sind, dass die meisten Benutzer dieses nicht ändern werden. Sonst wird man natürlich unnötigerweise zusätzliche Arbeit schaffen.

Ein kurzer *Hilfetext* in vollständigen Sätzen, der dauerhaft an der Oberfläche sichtbar ist und der eine Funktion oder das Ergebnis von Aktionen erklärt, kann ein sehr hilfreiches Konzept sein. Es wird neuen Benutzern helfen, die Dinge besser zu verstehen, und bestehende Benutzer bei ihrer Arbeit unterstützen. Überraschenderweise haben viele Anwendungen heutzutage extrem kurze Bezeichnungen ohne Erklärungen neben ihren Formularfeldern. Wenn die Benutzer das Feld jedoch nicht verstehen, sind sie komplett verloren. Eine weitere Ausprägung der Hilfetexte sind präzisere Bezeichnungen der Formularfelder. So sollte man in einem Login-Formular das Feld „Benutzername" durch „E-Mail-Adresse" ersetzen, sollte das System nur E-Mail-Adressen zum Login benutzen. Allgemein gilt es, vieles, was implizit ist, explizit zu machen und auszuschreiben. Die Nutzer werden es Ihnen danken.

Gestaltet es menschlicher

Es ist wichtig, dass Formulare barrierefrei sind, um allen Benutzern gleichen Zugang und gleiche Chancen zu bieten. Ein zugängliches Formular kann Menschen mit Einschränkungen helfen, sich aktiver als je zuvor zu beteiligen.

Verlassen Sie sich bei der Kommunikation nicht auf Farbe. Es gibt eine große Anzahl von Menschen, die einen gewissen Grad an Farbenblindheit besitzen. Achten Sie deshalb bei der Anzeige von Validierungsfehlern oder Erfolgsmeldungen darauf, dass das Feld nicht ausschließlich grün oder rot markiert ist. Wo immer Farbe verwendet wird, versuchen Sie auch zusätzlich Text und Symbole anzuzeigen, um dem Benutzer die entsprechende Nachricht zu übermitteln.

Das „Required"-Attribut aus HTML sorgt dafür, dass viele Formulare das Symbol * benutzen. Aber die Formulierung „Optional" nach einer Beschriftung ist viel klarer als jedes visuelle Symbol, das Sie je verwenden könnten. Es wird immer Nutzer geben, die sich fragen, was dieses Sternchen bedeutet. Und diese Nutzer werden nach einer Legende suchen, die Dinge erklärt.

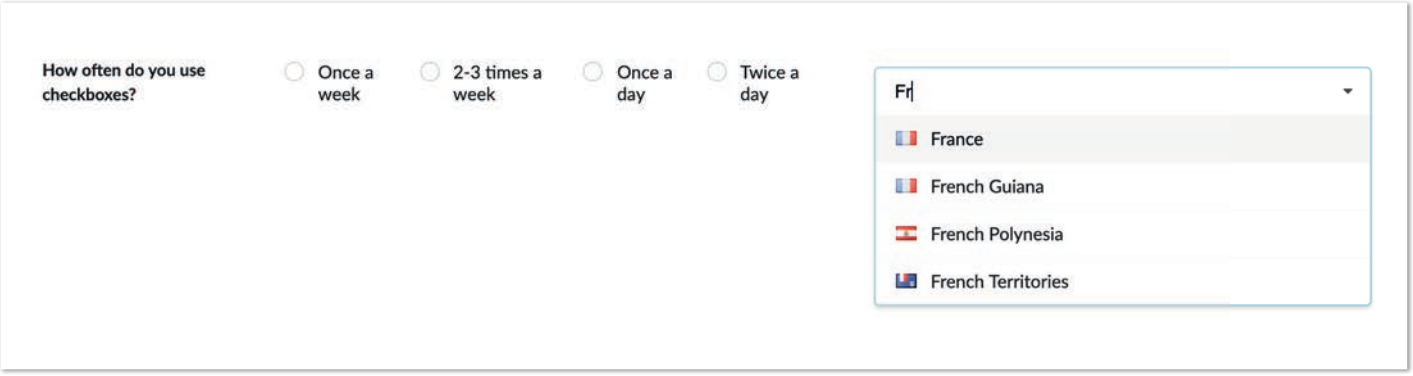


Abbildung 4: Links eine Radio-Komponente, rechts ein Input-Dropdown (Quelle: Maximilian Liesegang)

Abbildung 5: Komplette Sätze als Hilfetext sind sehr hilfreich (Quelle: Maximilian Liesegang)

Stellen Sie sicher, dass Ihr gesamtes Formular mit der Tabulatortaste navigierbar ist. Während viele Benutzer die Tabulatortaste verwenden, um schneller durch Formulare zu navigieren, ist dies besonders wichtig für manche Benutzer, die sich auf eine Software verlassen, die die Tabulatorfunktion verwendet, um von einem Feld zum nächsten zu wechseln.

Wenn Sie Fehlermeldungen schreiben, konzentrieren Sie sich darauf, die Frustration zu minimieren, die Benutzer empfinden, wenn sie ein Problem mit einem Formular haben. Hier sind ein paar Regeln zum Schreiben effektiver Fehlermeldungen: Die Art und Weise, wie Sie eine Fehlermeldung übermitteln, kann einen enormen Einfluss darauf haben, wie Benutzer sie wahrnehmen. Eine Fehlermeldung wie „Sie haben eine falsche Nummer eingegeben" gibt dem Benutzer die ganze Schuld, was dazu führen kann, dass der Benutzer frustriert ist. Schreiben Sie stattdessen etwas, das neutral oder positiv klingt. Eine neutrale Meldung könnte sein: „Diese Zahl ist falsch." Des Weiteren vermeiden Sie Fehlermeldungen wie „User Input Error: 1034". Diese sind kryptisch und beängstigend. Schreiben Sie wie ein Mensch, nicht wie ein Roboter. Verwenden Sie menschliche Sprache und erklären Sie, was genau der Benutzer oder das System falsch gemacht hat und was genau der Benutzer tun sollte, um das Problem zu lösen.

Fazit

Wie wir sehen können, ist es nicht leicht, ein gutes Formular zu entwerfen. Riskieren Sie nicht, dass Benutzer enttäuscht werden oder wertvolle Zeit damit verschwenden, herauszufinden, wie Ihr Formular funktioniert. Der Autor hofft, mit dem Artikel ein paar Tipps gegeben zu haben, die es Ihnen in Zukunft erleichtern, effektivere Formulare zu gestalten. Es ist nicht so, dass man besonders künstlerisch begabt sein muss, um die Oberfläche eines Formulars besser zu machen. Das Wichtigste ist, dass man bei der Entwicklung von Anwendungen niemals vergisst, dass Menschen diese am Ende benutzen sollen. Zumindest solange wir die Singularität noch nicht erreicht haben.

Quellen

[1] Adam Silver (2017): One Thing Per Page. Smashing Magazine, *https://www.smashingmagazine.com/2017/05/better-form-design-one-thing-per-page/*

[2] Mirjam Seckler et al. (2014): Designing Usable Web Forms – Empirical Evaluation of Web Form Improvement Guidelines. Conference on Human Factors in Computing Systems

Maximilian Liesegang
esentri
maximilian.liesegang@esentri.com

Als Consultant bei esentri ist Maximilian dafür verantwortlich, digitale Lösungen für Menschen erlebbar zu machen. Dabei liegt sein Fokus besonders auf der Schnittstelle zwischen Mensch und Maschine. Außerhalb von esentri engagiert er sich für das Thema User Experience innerhalb der DOAG.

Mitmachen und Autor werden!

Sie kennen sich in einem bestimmten Gebiet aus dem Java-Themenbereich bestens aus und möchten als Autor Ihr Wissen mit der Community teilen?

Nehmen Sie Kontakt zu uns auf und senden Sie Ihren Artikelvorschlag zur Abstimmung an **redaktion@ijug.eu.**

Wir freuen uns, von Ihnen zu hören!

iJUG Verbund

Als Entwickler glücklich sein – Tipps und Tricks

Christian Seifert, BetterDoc

Der Softwareentwickler, der sein Hobby zum Beruf gemacht hat – so würden sich sicherlich viele von uns charakterisieren. Doch angekommen im Beruf sieht die Welt nicht selten plötzlich gar nicht mehr so rosig aus. Aus den verschiedensten Gründen sind wir als Entwickler gefrustet und denken uns: „So hatte ich mir das aber nicht vorgestellt!". Häufig liegt das an unseren Erwartungshaltungen und Vorstellungen, die mit der Realität nicht oder nur wenig übereinstimmen. In diesem Artikel wollen wir uns einige „Real Life Happiness Patterns" ansehen, die dabei helfen können, den ganz normalen Alltagswahnsinn ein wenig leichter zu überstehen und den Spaß an der eigenen Arbeit wiederzufinden.

Der typische Softwareentwickler sitzt spät abends allein im spärlich beleuchteten Keller, neben ihm die leere Pizzabox und zwei Flaschen Cola. Die Realität von professioneller Softwareentwicklung hat mit dieser idealisierten Vorstellung nur sehr, sehr wenig zu tun. Und dennoch ist es dieses Bild, das wir häufig im Kopf haben und das für viele eine Idealvorstellung ist: Ungestört vom Rest der Welt, nur fokussiert auf die Technik, durch das Schreiben von Code kniffelige Probleme lösen. Stattdessen haben wir ganz andere Herausforderungen: quengelige Kunden, fordernde Projektmanager, enge Zeitpläne und Kollegen, die vollkommen andere Vorstellungen davon haben, was am Ende herauskommen soll, als man selbst. Das Ergebnis ist nicht selten Frust, Resignation und in immer mehr Fällen auch Burnout.

Mit einem frischen Blick auf die eigene Umwelt und das eigene Selbstverständnis finden sich aber durchaus Auswege, den Spaß und die Freude an der Arbeit wiederzufinden und Softwareentwicklung nicht nur als notwendiges Übel zu sehen, das am Ende des Monats dafür sorgt, dass Geld auf dem eigenen Konto ist, sondern es wieder als spannende Herausforderung zu sehen, für die man gerne morgens aus dem Bett steigt.

Erwartungen und die Realität

Wie in vielen anderen Bereichen gibt es auch von Softwareentwicklung ein idealisiertes Bild, das mit der Realität nur selten übereinstimmt. Eine grundlegende Frage, die wir uns alle stellen sollten, ist daher: Was will ich eigentlich wirklich in meinem Job? Womit möchte ich mich beschäftigen und (vielleicht ebenso wichtig) womit möchte ich mich nicht beschäftigen? Nur wer für sich selbst definiert hat, worauf er seinen Fokus legen möchte und wo seine Grenzen liegen, kann aktiv nach eben den Dingen suchen, die für ihn selbst interessant sind.

Gerade der bereits beschriebene Weg vom Hobby zum Beruf ist häufig eher hinderlich als förderlich: Software allein in der Freizeit zu schreiben erlaubt (und erfordert) eine andere Denk- und Herangehensweise, als Software im Team mit Kollegen für Geld zu entwickeln und diese zu warten. Als Tipp gilt daher: „Wer sein Hobby zum Beruf gemacht hat, tut gut daran, sich als Erstes ein neues Hobby zu suchen". Ein Hobby sollte nicht zuletzt auch dazu dienen, abzuschalten und den Kopf freibekommen zu können. Wer sich aber sowohl beruflich als auch in seiner Freizeit mit den immer gleichen Dingen beschäftigt, für den wird es zunehmend schwieriger, den „Akku aufzuladen".

Stakeholder kennen und verstehen

Professionelle Softwareentwicklung ist bereits seit Langem keine One-Man-Show mehr. Wir finden uns eigentlich immer in einem Team mit den verschiedensten Rollen und Persönlichkeiten wieder. Viele der größten Missverständnisse und Enttäuschungen entstehen für uns als Softwareentwickler dadurch, dass wir davon ausgehen, alle anderen haben die gleiche Denkweise, verfolgen die gleichen Ziele und nutzen die gleichen Methoden wie wir selbst. Interesse, über den eigenen Tellerrand blicken zu können und zu wollen, ist hierbei einer der Schlüssel zu einer besseren, reibungsloseren und damit stressfreieren Zusammenarbeit. Nur wer wirklich versteht, wie sein Gegenüber tickt, was seine Ziele und Motivationen sind, der wird in der Lage sein, auf Augenhöhe zu kommunizieren und seine eigenen Wünsche und Probleme zu äußern.

Viele Projekte, nicht nur, aber auch und gerade im Software-Umfeld, scheitern nicht an den technischen Rahmenbedingungen. Sie scheitern nicht daran, dass man das falsche Framework oder die falsche Datenbank ausgewählt hat, sondern daran, dass das eigentliche Problem nicht oder nur schlecht verstanden wurde. Die Fähigkeit und die Bereitschaft, die Sprache unseres Gegenübers zu verstehen und aktiv sprechen zu können, ist dabei einer der größten Hebel, die wir haben, um auf unsere Probleme aufmerksam zu machen und unsere Sichtweise zu artikulieren. Wird allein mit technischem Jargon versucht, einen Fachverantwortlichen davon zu überzeugen, Dinge zu ändern oder Probleme anzugehen, der wird sich früher oder später frustriert zurückziehen, weil „der mir sowieso nicht zuhört und nicht versteht, was ich ihm zu sagen versuche".

Hier hilft es, sich selbst in einer ähnlichen Situation zu beobachten: Stellen wir uns vor, nach einem Sportunfall bekommen wir von einem Arzt zu hören, dass wir uns eine Radiuskopffraktur, eine Scaphoidfraktur und eine laterale Tibiakopffraktur zugezogen haben. Vollkommen zu Recht würde jeder von uns erwarten, diese Diagnose für einen Nicht-Mediziner verständlich übersetzt zu bekommen: Knochenbrüche im Ellenbogen, der Handwurzel und seitlich am Schienenbein. Ebenso sollten wir als Softwareentwickler in der Lage sein, unsere Themen für Nicht-Fachleute übersetzen zu können.

Die Konsequenzen sind hierbei enorm und direkt zu spüren: Nur wenn andere Projektbeteiligte unsere Probleme verstehen und nachvollziehen können, sind sie auch in der Lage, uns unterstützen zu können, was uns wiederum in die Lage versetzt, unsere eigene Arbeit besser, leichter und motivierter zu erledigen.

Wertschätzung der eigenen Arbeit

Die Softwareindustrie entwickelt sich mit einer atemberaubenden Geschwindigkeit. Technologien, die buchstäblich gestern noch als hochinnovativ und fortschrittlich galten, sind heute schon überholt. Doch nicht jedes Projekt und nicht jede Softwarelösung kann (und will) mit diesem enormen Tempo mithalten. Viele Banken und Versicherungen entwickeln heute noch aktiv mit Technologien, die ihre Ursprünge in den 1950er und 1960er Jahren haben. Wer einmal in die großen Stellenportale schaut, wird verblüfft sein, wie viele Stellenangebote es auch 2019 noch für COBOL-Entwickler gibt.

In Entwicklerkreisen werden solche Legacy-Anwendungen gerne belächelt und Entwickler, die nicht ganz vorne auf der Innovationswelle reiten, als zweitklassig eingestuft. Wer nicht mit den neusten Technologien umgehen kann, wird ungern als vollwertiges Mitglied des Clubs angesehen.

Doch sollten wir dabei nicht vergessen, dass viele dieser Altanwendungen sicherstellen, dass essenzielle Bestandteile unseres täglichen Lebens einwandfrei funktionieren. Natürlich mag ein System zur Verwaltung von Rentenversicherungen von außen eher langweilig und verstaubt wirken, doch wird jeder von uns hoffen und davon ausgehen, dass seine Versicherung Sorge dafür trägt, die ihr anvertrauten Gelder korrekt zu verwalten. Ein Teil des Entwicklerteams zu sein, das eben diese Software pflegt und weiterentwickelt, sollte daher nicht per se dazu führen, als Entwickler zweiter Klasse abgestempelt zu werden.

Eine interessante Analogie hierzu ist die Dombauhütte Köln, deren 60 Mitarbeiter den Kölner Dom in Stand halten. Keiner dieser Handwerker wird seine Aufgabe beschreiben mit „Es ist furchtbar, diese alte Kirche mit alter Bauweise und altem Material am Leben zu erhalten". Im Gegenteil: Jeder wird mit stolzgeschwellter Brust davon berichten, dass er ein Teil der Gruppe ist, die das Weiterbestehen dieses Teil des Weltkulturerbes sicherstellt.

Nun ist es sicherlich ein wenig weit hergeholt, ein Stück Software mit einer Hunderte Jahren alten Kirche vergleichen zu wollen, aber dennoch: Auch Entwickler, die sich mit „langweiliger" Legacy-Entwicklung beschäftigen, sind ein essenzieller und wichtiger Bestandteil unserer Entwickler-Community, die stolz auf ihre Arbeit sein können (und sollten).

Karriere-Management

Als Softwareentwickler liegt unser Fokus selten auf einer Karriere im klassischen Sinne: dem Sammeln von Privilegien und dem Erklimmen immer neuer Stufen in der firmeninternen Hierarchie. Dennoch lohnt es sich, hin und wieder innezuhalten und zu überlegen: Wo stehe ich im Moment? Was sind meine Optionen und möglichen nächsten Schritte? Was sind Dinge, die ich bisher gelernt habe? Worauf möchte ich mich in Zukunft konzentrieren und fokussieren? Welche Pfade stehen mir offen und welche davon möchte ich bewusst und aktiv wählen?

Eine Beobachtung, die man hier häufig machen kann, ist: Wer seine Karriere nicht aktiv gestaltet, für den wird sie von jemand anderem gestaltet. In vielen Köpfen existiert immer noch „der" klassische Weg eines Softwareentwicklers, der nach einigen Jahren vorsieht, weniger Code zu schreiben und dafür mehr organisatorische Aufgaben zu übernehmen. Während für manch einen genau das ein wünschenswerter Entwicklungspfad ist, so finden sich auch immer wieder Beispiele von vormals exzellenten Technikern, die nach einer Beförderung auf eine Management-Position weder glücklicher noch produktiver waren als vorher. Die harte Wahrheit ist auch hier: Nicht jeder Softwareentwickler ist automatisch ein guter Manager.

Es ist daher mehr als hilfreich, ein realistisches Verständnis davon zu haben, was die eigenen Stärken, aber auch die eigenen Schwächen sind. Karriereschritte sollten nicht danach ausgewählt werden, was im Lebenslauf am besten aussieht oder welche Zusatzleistungen damit verbunden sind, sondern vielmehr danach, ob sie den eigenen Interessen entsprechen und interessante und vor allem gewollte Herausforderungen mit sich bringen.

Continuous Improvement

In agilen Entwicklungsteams ist die regelmäßige Retrospektive ein wichtiger Bestandteil von kontinuierlicher Verbesserung. Was sind Dinge, die gut liefen? Was sind Dinge, die verbessert werden sollten? Welche kleinen Schritte können wir ab morgen unternehmen, um noch besser zu werden?

Es lohnt sich, diese Denkweise auch für die eigene Arbeitsweise anzusetzen und sich selbst in den Mittelpunkt zu stellen. Was sind Dinge, die in der letzten Zeit gut gelaufen sind? Was sind Dinge, die mir besonders Spaß gemacht haben und/oder die ich neu gelernt habe? Was davon lohnt es sich, weiterzuverfolgen und was davon hat sich als nicht interessant herausgestellt?

Große und radikale Änderungen umsetzen ist immer schwierig und aufwendig – das gilt für Softwareprojekte ebenso wie für eigene Verhaltensweise. Agile Entwicklung hat uns „Inspect and adapt" gelehrt: Viele kleine Anpassungen lassen sich deutlich einfacher umsetzen.

Der Blick nach vorne

Die Schnelllebigkeit der Softwareindustrie birgt sowohl Chancen als auch Risiken. Chancen dadurch, dass sich der eigene Aufgabenbereich und die eigene Expertise innerhalb weniger Jahre grundlegend ändern kann. Risiken dadurch, dass man häufig kaum dazu kommt, innezuhalten und zu reflektieren, ob das, was man gerade macht, für einen selbst noch interessant und sinnstiftend ist. Wer sich aber immer wieder bewusst dazu entscheidet, einen offenen und ehrlichen Blick auf sich selbst zu riskieren, der wird auch lernen, sicher durch diese rauen Gewässer zu navigieren.

Christian Seifert
BetterDoc GmbH
christian.seifert@betterdoc.de

Christian Seifert ist Software Engineer mit fast 20 Jahren Erfahrung in der Entwicklung und dem Support von Individualsoftware. Ob beim Schreiben von Code, der Analyse von Anforderungen oder dem Mentoring im Team – für ihn geht es immer darum, genau die Lösung zu finden, die auch tatsächlich von Anwendern benötigt und gewollt ist. Auch wenn ihn ursprünglich die Faszination an der Technik zur Softwareentwicklung trieb, so verbringt er heute mindestens genauso viel Zeit mit Menschen und wirbt leidenschaftlich für ein besseres Verständnis zwischen Entwicklungsteams und anderen Stakeholdern.

Wie Sie Entscheider von Ihrer Idee überzeugen

Michael Keller, salegro GmbH

Sie haben einen Termin. Die Bühne ist bereitet. Es warten der Entscheider oder vielleicht aber auch eintausend Menschen hungrig auf Ihren Input. Jetzt kommt es drauf an: Wie gut sind Sie vorbereitet? Wie gut ist Ihre Performance? Wie gut können Sie den oder die Menschen in Ihren Bann ziehen? Die in diesem Artikel beschriebene Methode hilft Ihnen, überzeugender zu sein.

Den Entscheider oder ein Entscheider-Gremium von der eigenen Idee zu überzeugen, kann sich manchmal schwierig gestalten. Doch es gibt Strategien, Ihren Vorschlag in den Köpfen Ihrer Gesprächspartner erfolgreicher zu positionieren. Die hier vorgestellte Methode hilft Ihnen, Ihren Einfluss auszuweiten, mehr Selbstsicherheit zu erlangen und mit einem souveränen Auftritt zu überzeugen. Häufig erzielen nur State-of-the-Art-Vorstellungen die gewünschte Sogwirkung, die den Erwartungen, dem Zeitgeist oder schlicht den Anforderungen an einen gekonnten Vortrag oder professionellen Pitch entsprechen. Nutzen Sie die Methode doch in den nächsten Meetings, hinterlassen Sie zukünftig in Gesprächen mehr Eindruck oder bestehen Sie in Diskussionen, wenn Sie unter Druck geraten. Der Artikel veranschaulicht zunächst einige grundsätzliche Prinzipien zum Erzeugen einer Sogwirkung und zeigt danach konkrete Sprachmuster auf, die Sie anwenden können, um die Sogwirkung Ihrer Vorschläge und Ideen unwiderstehlich zu machen.

Wir befinden uns, wenn man von einem Sales Pitch oder von einer Konzept-Präsentation spricht, im letzten Teil eines vierstufigen Methodengefüges. Vorangehend haben Sie Ihr Thema bereits eingeleitet, die Aufmerksamkeit des Zuhörers auf Ihr Anliegen fokussiert, Ihren Lösungsvorschlag systematisch strukturiert und nun folgt der hier behandelte Schritt, mit dem Sie den Entscheider überzeugen.

Zuerst zu den grundsätzlichen Prinzipien. Erinnern wir uns nochmal an die Eingangssituation: Wie wäre es jetzt, vor tausend Leuten eine Rede zu halten? Ist Ihnen mulmig? Verspüren Sie Vorfreude? Sicher haben Sie schon einmal in einem Raum gesessen und einem Präsentator zugehört. Wie wäre es, einmal zu tauschen? In die Rolle des anderen zu schlüpfen? Einen Perspektivwechsel zu machen? Genau diese Änderung des Blickwinkels, das Schlüpfen in die Schuhe des anderen, können Sie sich zunutze machen, wenn Sie einen oder mehrere Menschen, den Entscheider oder das ganze Gremium, überzeugen wollen. Dieser Perspektivwechsel ist der Schlüssel, um den Entscheidungsprozess positiv zu beeinflussen.

Damit stellt sich nun die Frage, wie der Entscheidungsprozess überhaupt vonstattengeht. Das lässt sich gut anhand einer kleinen Geschichte aufzeigen, die Sie vermutlich so oder so ähnlich auch schon erlebt haben: Als ich mir das letzte Mal neue Schuhe für die Arbeit kaufen wollte, habe ich mir überlegt, dass ich um die 100 Euro ausgeben und ein Paar schwarze Halbschuhe erstehen möchte. Das nächste Mal, als ich in der Stadt war – gar nicht auf der Suche nach neuen Schuhen –, sind mir in einem Schaufenster schwarze Halbschuhe aufgefallen. Das Design fand ich klasse, der Schuh schien auch hochwertig zu sein. Der Preis: 180 Euro. Es fand eine kurze Diskussion in meinem Kopf statt, ob ich nun hineingehen und die Schuhe genauer anschauen sollte – schließlich wollte ich nur 100 Euro ausgeben. Kennen Sie das? Ich vermute mal, ja. Schlussendlich habe ich den Laden betreten und die Schuhe anprobiert. Und sie fühlten sich super an! Top Qualität, angenehmer Tragekomfort und gut sahen sie auch noch aus. Fünf Minuten später waren die Schuhe gekauft. Ein echter Lustkauf.

Was glauben Sie, was in meinem Kopf jetzt vorgegangen ist? *„Du wolltest doch nur 100 Euro ausgeben. Wie erklärst du das deiner Frau?“* Aber der kleine Mann im Ohr hatte gleich eine Antwort parat: *„Durch die super Qualität wird der Schuh aber viel länger halten als einer, den man für weniger Geld bekommen hätte. Und ich habe auch viel mehr Freude an dem Schuh. Das hat sich auch zeitlich günstig ergeben, da ich eh gerade in der Stadt war. Wie oft habe ich sonst schon stundenlang gesucht, um den richtigen Schuh zu finden? Außerdem transportiert der Schuh genau das, was ich auf der Bühne darstellen will: Top-Qualität, modern, etwas ausgefallen, aber nicht zu sehr, wunderbar!“* Wenn man die Situation nun genauer betrachtet, drängt sich die Frage auf: Wer hat überhaupt die Entscheidung gefällt, das Geschäft zu betreten? Wäre ich rein rational vorgegangen, hätte ich draußen bleiben müssen, denn ich wollte ja maximal 100 Euro ausgeben. Aber Entscheidungen werden eben oft emotional getroffen, die rationale Rechtfertigung kommt erst danach.

Für den Aufbau unserer Argumentation sind diese Erkenntnisse elementar. Denn es bedeutet, dass wir bei der Argumentation idealerweise die Motive des Entscheiders bedienen sollten und im besten Fall auch gleich die rationalen Vorteile für den kleinen Mann im Ohr mitliefern. Im Bereich der rationalen Vorteile kann hierbei eine Auswahl der vier bekanntesten Vorteile angesprochen werden: Qualität, Geld, Bequemlichkeit und Zeit. Auf jeden dieser vier Vorteile triggert Ihr Kunde, Ihr Entscheider mit einer bestimmten Präferenz. Und die gilt es auf jeden Fall zu bedienen. Wenn Sie kurz überlegen, werden Sie von den meisten Entscheidern kontextbezogen die persönliche Präferenz einschätzen können. In unserer Geschichte lassen sich die rationalen Vorteile wie folgt zuordnen: Das Argument mit der guten Verarbeitung des Schuhs spricht die Felder *Qualität* und *Geld* an. Die Rechtfertigung,

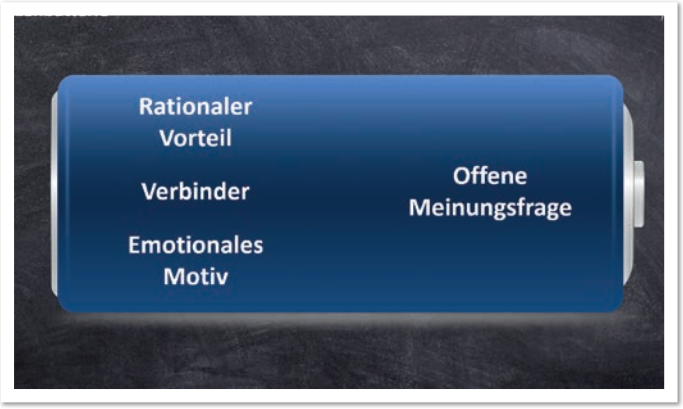


Abbildung 1: Nutzenbatterie Beispiel (© salegro GmbH)

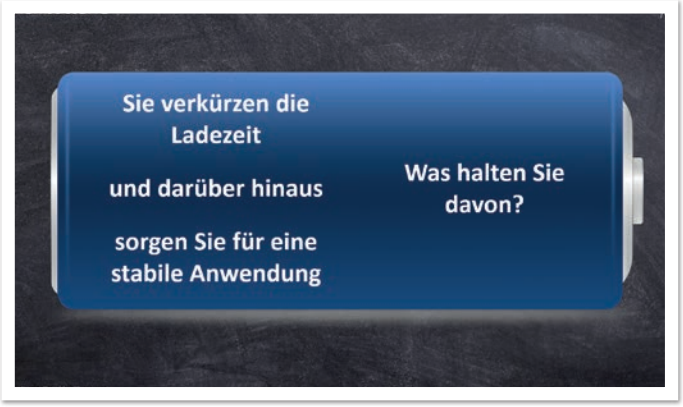


Abbildung 2: Nutzenbatterie narrativer Aufbau (© salegro GmbH)

dass sich die Situation günstig ergeben hat, gehört dem Bereich *Bequemlichkeit* an, die kurze Suche lässt sich der Sparte *Zeit* zuordnen. Machen Sie sich zunächst schlau: Möchte Ihr Kunde eine möglichst kostengünstige Lösung? Strebt er nach der höchsten Qualität bei seinen Produkten? Oder ist ihm am wichtigsten, dass alles reibungslos und schnell vonstattengeht? Haben Sie das ermittelt, können Sie das von ihm präferierte Feld der rationalen Vorteile für Ihre Argumentation nutzen, wenn Sie dem Entscheider gegenübertreten.

Wieso habe ich das Schuhgeschäft aber überhaupt betreten? Das geschah aufgrund von *emotionalen Motiven*. Von ihnen gibt es sehr viele, die, wenn Sie sich die Motive als Orte auf einer Landkarte vorstellen, mal näher aneinander, mal weiter voneinander entfernt liegen. Beispielsweise liegen Design, Neugier und Kreativität nahe beieinander, genauso wie Geborgenheit, Sicherheit und Familie an einer anderen Stelle der Landkarte unweit voneinander liegen. Da es nicht praktikabel ist, mit so vielen Motiven zu arbeiten, arbeiten wir meist der Einfachheit halber mit nur drei Motivpolen. Bewährt haben sich folgende Pole: Macht, Sicherheit und Neuartigkeit. Dass der Schuh mich auf der Bühne ins rechte Licht rückt, gibt mir das Gefühl von *Macht* und *Sicherheit*. Sein modernes, leicht ausgefallenes Design bedient darüber hinaus den Bereich der *Neuartigkeit*.

Die emotionalen Motive erzeugen den stärksten Sog und sollten folglich in Ihrer Argumentation möglichst präzise angesprochen werden. Es gilt dabei im Vorfeld abzuklären, auf welches Motiv der Entscheider den größten Wert legt. Möchte er den nächsten Karriereschritt machen, seine Marktsegmente ausbauen oder bei seinem Chef in einem positiven Licht erscheinen? Vielleicht versucht er aber auch, in einem schrumpfenden Markt seine Anteile zu behaupten, oder ist eher darauf bedacht, keinen Ärger mit Kunden oder innerhalb der Firma heraufzubeschwören? Oder strebt er immer nach den State-of-the-Art-Prozessen und -Dienstleistungen und schätzt die modernsten Produkte und Auftritte am meisten? Mit diesen Informationen können Sie Ihre Argumente nach den emotionalen Motiven des Entscheiders ausrichten und individuell anpassen.

Wie können Sie die Informationen zu rationalen Vorteilen, emotionalen Motiven und dem Perspektivwechsel jetzt konkret für sich nutzen, beispielsweise mit vorbereiteten Sätzen? Ein optimales Sprachmuster, um von den oben genannten Vorüberlegungen zu profitieren, ist die sogenannte *Nutzenbatterie* *(siehe Abbildung 1)*. Sie ist wie folgt aufgebaut: Ein *rationaler Vorteil* wird durch einen *Verbinder* mit einem *emotionalen Motiv* verflochten. Abgeschlossen wird die Ausführung durch eine offene *Meinungsfrage*.

Ein Beispiel zur Verbesserung einer Software könnte folgendermaßen lauten: *Mit diesem Programm verkürzen Sie die Ladezeit* (rationaler Vorteil der Zeit) und sorgen darüber hinaus (Verbinder) *für eine stabilere Anwendung* (emotionales Motiv der Sicherheit). *Wie interessant ist das für Sie* (offene Meinungsfrage)*?* *(siehe Abbildung 2)* Haben Sie das Entscheider-Gremium gerade emotionalisiert, ist es erfolgskritisch, sofort eine Antwort zu erfragen und nicht weiter und weiter zu reden. Stellen Sie dabei eine **offene** Frage, um kein einfaches *Nein* zu erhalten. Haben Sie die erste positive Resonanz erhalten, können Sie mit einer **geschlossenen** Frage den Deal festmachen: *Dann machen wir das so?* Jetzt können Sie sich Ihre eigene Nutzenbatterie bauen und damit nicht nur effektiv auf Ihren nächsten Pitch vorbereiten, sondern Sie platzieren gleichzeitig Ihre Idee zielsicher im Kopf des Entscheiders! Was halten Sie davon?

Michael Keller
Salegro GmbH
m.keller@salegro.de

Seit September 2000 ist Michael Keller im internationalen Vertrieb von Projekten und Dienstleistungen und deren Umsetzungen tätig und hat dabei in über 15 Ländern die Projektverantwortung getragen. Zunächst als Change Berater, später zusätzlich als Trainer und heute als geschäftsführender Gesellschafter dient er seinen Kunden in den Feldern Change Management und Persönlichkeitsentwicklung für Vertriebsorganisationen und Vertriebsmitarbeiter. Motto im Change Management: Sachlich analysieren, menschlich umsetzen! Motto in der Persönlichkeitsentwicklung: Erfolg ist nur die Folge!

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen

iJUG Verbund

www.ijug.eu

Impressum

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

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

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

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Mylène Diacquenod
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: Bild © vanatchanan | *https://de.123rf.com*
S. 10: Bild © Sensvector | *https://stock.adobe.com*
S. 19: Bild © alphaspirit | *https://de.123rf.com*
S. 25: Bild © Yuriy Kirsanov | *https://de.123rf.com*
S. 29: Bild © Dragos Nicolae Dragomirescu | *https://de.123rf.com*
S. 38: Bild © Samantha Craddock | *https://stock.adobe.com*
S. 40: Bild © Le Moal Olivier | *https://de.123rf.com*
S. 44: Bild © stokkete | *https://de.123rf.com*
S. 50: Bild © blankstock | *https://de.123rf.com*
S. 54: Bild © bloomua | *https://de.123rf.com*
S. 60: Bild © belchonock | *https://de.123rf.com*
S. 63: Bild © Dmitrii Shironosov | *https://de.123rf.com*

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

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

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

Inserentenverzeichnis

HAST DU ES SCHON MIT EINEM NEUSTART VERSUCHT?

Come and join the CI Crowd.

JAVA DEVELOPER BI BERATER UX DESIGNER MOBILE DEVELOPER DU

Hier neu starten:
karriereportal.
cologne-intelligence.de

COLOGNE INTELLIGENCE

QAWARE
SOFTWARE ENGINEERING

IT-Probleme lösen.
Digitale Zukunft gestalten.

Mit Erfindergeist
und Handwerksstolz.

Great Place To Work.
2019
Beste Arbeitgeber™
Information & Telekommunikation

kununu.com/qaware
qaware.de/karriere