Ausgabe 03/2021

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 03

Java aktuell

iJUG Verbund
www.ijug.eu

Clojure
Einstieg und weiterführende Informationen

Kotlin
Mit Koroutinen asynchron programmieren

Soft Skills
Das Streben nach Expertentum und Digital Leadership

JVM Sprachen

iJUG Verbund

Werden Sie Mitglied im iJUG!

Ab 15,00 EUR im Jahr erhalten Sie

30 % Rabatt auf Tickets der JavaLand

Jahres-Abonnement der Java aktuell

Mitgliedschaft im Java Community Process

www.ijug.eu

Liebe Leser/innen der Java aktuell,

wir freuen uns sehr, euch mitteilen zu dürfen, dass in diesem Jahr statt der ursprünglich geplanten vier Ausgaben eine fünfte Java aktuell erscheinen wird!

Im Editorial der Java aktuell 5/20 gaben wir zuletzt bekannt, dass wir aufgrund notwendiger Kosteneinsparungsmaßnahmen statt der regulären sechs Ausgaben im Jahr 2020 auf fünf und 2021 auf vier reduzieren mussten. Unter anderem die erfolgreiche Online-Premiere der JavaLand 2021 ermöglicht es uns, nun doch fünf Ausgaben zu publizieren. Also könnt ihr euch nach diesem Heft noch auf zwei weitere freuen! Apropos JavaLand, unseren Nachbericht zur Veranstaltung findet ihr ebenfalls in diesem Heft ab Seite 16.

Der Schwerpunkt dieser Ausgabe ist das Thema JVM-Sprachen. Dazu findet ihr im vorderen Bereich der Zeitschrift drei aufeinander aufbauende Artikel zur Programmiersprache Clojure. Den Einstieg gibt Tim Zöller ab Seite 18. Darin gibt er einen Überblick für Java-Entwickler, die Clojure gerne einmal ausprobieren möchten. Nachfolgend erklärt uns Nicolai Mainiero Concurrency und State in Clojure. Zum Abschluss runden Manuel Herzog und Dominik Galler die Clojure-Thematik ab und betrachten Clojure und Java in Kombination. Ab Seite 35 widmet sich Thomas Künneth einer weiteren Programmiersprache – Kotlin, die mit ihren Koroutinen plattformübergreifend einsetzbar ist. Und genau zu diesen Koroutinen gibt Künneth in seinem Artikel einen Überblick.

Abseits unseres Themenschwerpunkts erwartet euch wie gewohnt eine bunte Mischung wissenswerter Artikel aus dem Java-Themenkosmos.

Im März 2021 wurde das aktuelle Major-Release 16 von Java veröffentlicht. Falk Sippach wirft mit uns ab Seite 11 einen Blick darauf und nimmt die neuen Features genauer unter die Lupe. Bernd Müller zeigt in seinem Artikel ab Seite 40 die Umsetzung zweier beliebter Features mit WildFly – das Deployment als JAR sowie ein Entwicklermodus mit Hot Deployment bei Änderungen an Code-Artefakten.

Wie wir auf Mikroarchitekturebene in Java eine erhöhte Flexibilität erreichen, zeigt uns Holger Tiemeyer. Er stellt die Möglichkeiten unter Verwendung des Java-9-Modulsystems sowie OSGi dar. Die weitverbreitete Verwendung von Frameworks und Bibliotheken von Drittanbietern sorgt für erhöhte Angriffsfläche von Anwendungen. Ein Tool, mit dem Open-Source-basierte Sicherheitslücken in Cloud-basierten Java-Anwendungen aufgedeckt und vor der Produktivstellung behoben werden können, stellt Mathias Conradt ab Seite 48 vor.

Zum Abschluss haben wir wieder nicht-technische Artikel, die weiche Kompetenzen (oder „Soft Skills") thematisieren. So befasst sich Anika Zohren ab Seite 57 mit der Frage, ob es neben Experten nur Pfeifen gibt und ob es überhaupt erstrebenswert ist, ein Experte zu werden. Dabei zieht sie einige überraschende Schlussfolgerungen. Dr. Dominic Linder thematisiert ein derzeit immer wichtiger werdendes Thema: Digital Leadership. Das Remote-Arbeiten hat Führungskräfte auf eine harte Probe gestellt. Er beschreibt, wie Führungskräfte auch während der ausschließlich virtuell stattfindenden Zusammenarbeit Mitarbeiter leiten und motivieren können.

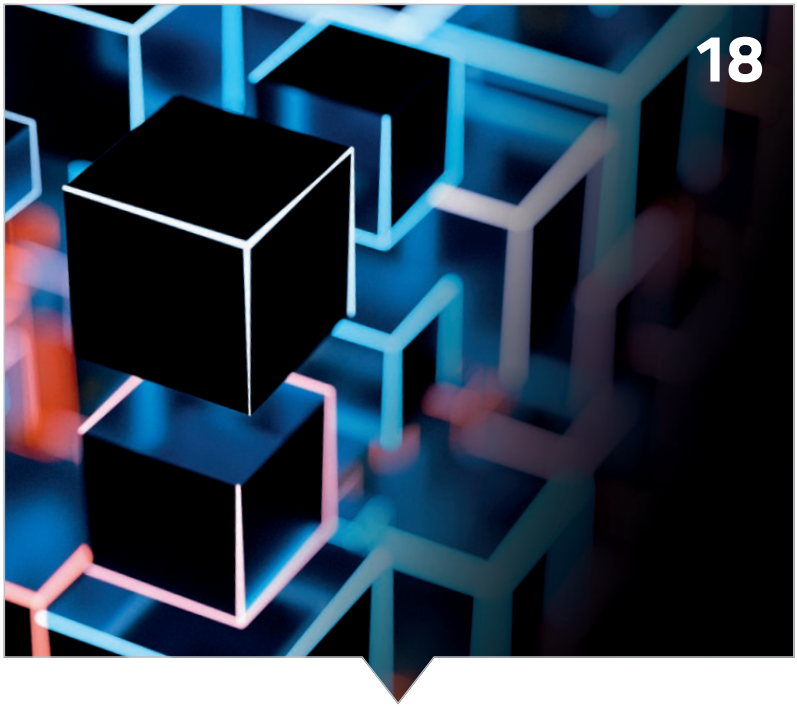
Wir wünschen euch viel Spaß beim Lesen!

Eure

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Java 16: Die neue Major-Version und ihre Features



Einstieg in die JVM-Sprache Clojure

OSGi für mehr Flexibilität bei Microservices in Java

Ist es erstrebenswert, ein Experte zu sein?

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

10. Dezember 2020

Kotlin und Deep Learning

Ein weiteres Deep-Learning-API für die JVM: JetBrains, die Firma hinter (unter anderem) IntelliJ und Kotlin, hat jetzt KotlinDL 0.1.0 veröffentlicht. Inspiriert vom Python-DL-API Keras und aufbauend auf – beziehungsweise abstrahierend von – TensorFlow, soll KotlinDL das Erstellen und Nutzen von Deep-Learning-Modellen vereinfachen. Wichtige TensorFlow-Features, etwa die Nutzung von (NVIDIA) GPUs, bleiben verfügbar. Aktuell wird unter der Haube TensorFlow 1.15 genutzt, die geplante Migration auf die Version TensorFlow 2 soll aber das höhere API nicht beeinflussen *[1]*.

20. Dezember 2020

Scala 3

Wo bleibt eigentlich Scala 3? Im Frühjahr 2018 war das Compiler-Projekt Dotty, an dem damals schon mehrere Jahre gearbeitet worden war, als Scala 3 für Anfang 2020 angekündigt worden. Vor einem Jahr wurde dann als Ziel Ende 2020 angegeben. So ganz hat das auch nicht funktioniert, aber immerhin gibt es seit Mai einen „Feature Freeze", und vor ein paar Tagen hat das Team die Community um Hilfe gebeten, insbesondere durch Feedback in einer Umfrage zur „Developer Preview". Einen Monat will das Team auf Rückmeldungen warten, bevor die letzte Phase für das Release gestartet wird.

Eine gewisse Sorge ist wohl da, mit dem großen Umbruch und damit einhergehenden inkompatiblen Änderungen viele Scala-Entwickler/innen zu verlieren. Aber mit der in den Compiler selbst eingebauten Migrationsunterstützung wurde ein guter Weg gefunden, die Hürden möglichst niedrig zu halten. Und da ein Hauptziel von Scala 3 war, die Sprache nicht nur sicherer, sondern auch leichter erlernbar zu machen, könnte das auch neuen Zulauf bescheren.

21. Dezember 2020

Eclipse Adoptium mit kreativer Namensgebung

Aus AdoptOpenJDK, das inzwischen auch bei der Eclipse Foundation untergeschlüpft ist, wird Eclipse Adoptium – weiterhin mit dem Ziel, in Zusammenarbeit mit anderen Projekten eine möglichst umfassende Menge von Java Runtimes zur Verfügung zu stellen.

Sehr kreativ ausgetobt haben sich die Mitglieder bei der Benennung der neuen Unterprojekte, die Teil von Adoptium sind: Die Testaktivitäten inklusive Entwicklung der „AQA Test Suite" finden in Eclipse AQAvit statt. Auch das Erstellen der Releases erfolgt unter einem getränketechnisch interessanten Namen: Eclipse Temurin – nicht nur ein Anagramm von „Runtime", sondern auch der Name eines chemischen Elements, das verwandt mit, aber verträglicher als Koffein ist. Das letzte Teilprojekt, Eclipse Temurin Compliance, wurde in Verhandlungen mit Oracle geschaffen, um den Zugang zu den TCKs für die Kompatibilitätszertifizierungen zu ermöglichen, die weiterhin Eigentum von Oracle sind und nicht entsprechenden Zugangsbeschränkungen unterliegen.

6. Januar 2021

Cloud Native for Java Alliance

Aus den „Sondierungsgesprächen" (so wäre wohl die politische, nicht notwendigerweise korrekte Bezeichnung) zwischen Jakarta EE und MicroProfile ist jetzt die „Cloud Native for Java (CN4J) Alliance" hervorgegangen. Zumindest schon mal als Mailing-Liste bei der Eclipse Foundation (cn4j-alliance auf *eclipse.org*). CN4J ist ja im vergangenen Jahr schon ein paar Mal als Begriff aufgetaucht, insbesondere beim eigenen „CN4J Day" auf der CloudNativeCon.

8. Januar 2021

Oracle nun offiziell bei MicroProfile dabei

Oracle hat sich nun offiziell der MicroProfile Working Group in der Ecplise Foundation angeschlossen. Mit Helidon hat „Big Red" ja eine eigene MicroProfile-Implementierung, die aber – nicht unbedingt technisch, aber was die Aufmerksamkeit angeht – nicht mit Quarkus vom Konkurrenten Red Hat mithalten kann. Die engere Zusammenarbeit könnte so beiden Seiten zugutekommen.

9. Januar 2021

Von Java 15 auf Java 8

Auch mal eine Idee: Migration rückwärts. In seinem Video-Blog schreibt Markus Karg eine mit Java 15 implementierte Applikation um, damit sie auf Java 8 läuft. Um zu zeigen, dass man mit Java 8 alles machen kann und die neueren Versionen gar nicht braucht. Ok, sorry, Markus, ich versuch's noch mal: Um zu zeigen, wie vergleichsweise umständlich, unverständlich und fehleranfällig der Code mit Java 8 ist, und um zur Migration auf moderne Versionen zu ermutigen *[2]*.

14. Januar 2021

Eclipse Foundation ist europäisch

Der Umzug der Eclipse Foundation nach Brüssel ist nun vollständig abgeschlossen und die Foundation damit europäisch – wobei sie, wie früher erwähnt, die Eclipse Foundation Inc. in Delaware/USA als Tochter behält. Die neue Gesellschaft heißt Eclipse Foundation AISBL („Association internationale sans but lucratif", zu Deutsch „Internationale Vereinigung ohne Gewinnerzielungsabsicht").

18. Januar 2021

IntelliJ IDEA wird 20

Die Entwicklungsumgebung IntelliJ IDEA wird auch schon 20 Jahre alt. Und wird laut Hersteller JetBrains wirklich weltweit genutzt, sogar in der Antarktis! Na ja, wenn es einen Ort gibt, an dem sich ungestört programmieren lässt, dann vermutlich dort.

20. Januar 2021

GraalVM 21

Release 21 der GraalVM ist da. Einige der Neuerungen sind noch als „experimentell" gekennzeichnet, zum Beispiel der Support für Linux auf ARM 64-Bit und eine alternative JVM-Implementierung auf Basis des Truffle Interpreter anstelle von Hotspot, sodass Java auf die gleiche Weise wie andere Sprachen in die GraalVM eingebunden wird. Damit ist vollständige Interoperabilität mit den anderen Sprachen möglich, aber auch die Nutzung der kompletten Werkzeug-Palette von Truffle und eine bessere Isolierung beziehungsweise Ausführung in einem von der Host-JVM separaten Kontext *[3]*.

4. Februar 2021

Tomcat 10 und Jakarta EE

Tomcat 10.0.2, die erste stabile Version von Tomcat 10, ist freigegeben worden. Mit ihr wird die Migration von Java EE auf Jakarta EE vollzogen. Sie hat die TCK-Tests der unterstützten Jakarta-EE-9-Spezifikationen bestanden – „mit Ausnahme einiger erwarteter Fehler, die die Spezifikationskonformität nicht beeinflussen", so die Release Notes. Ein Migrations-Tool für die Umbenennung der javax.*-Packages zu jakarta.* ist schon länger angekündigt, aber noch nicht fertig.

Die Liste aller kompatiblen Implementierungen ist auf der Jakarta-EE-Website *[4]* zu sehen. Für EE 9 befinden sich dort bislang aber nur Open Liberty 21 und GlassFish 6 (nach den neuen Regeln nicht mehr „die" Referenz-Implementierung, sondern einfach „eine" kompatible Implementierung). Tomcat wird dort aber nicht gelistet, da es nur einen Teil des Umfangs unterstützt (vielleicht wäre das eine eigene Auflistung wert).

5. Februar 2021

„Und – ich muss verrückt sein – (Gr)aal gibt's kostenlos obendrauf"

Kunden mit einer kommerziellen Lizenz für Java SE packt Oracle jetzt noch die Enterprise-Lizenz für die polyglotte GraalVM dazu *[5]*.

12. Februar 2021

Jakarta EE und MicroProfile

Die Frage, wie sich Jakarta EE und MicroProfile (MP) in Zukunft zueinander verhalten sollen, ist weiter ungeklärt. Reza Rahman, Jakarta EE „Ambassador", auch für MicroProfile aktiv, hat eine Zusammenfassung der Optionen, die vorher in der Mailing-Liste der neuen CN4J Alliane disktutiert wurden, gepostet und eine Online-Umfrage hinzugefügt. Dass MicroProfile schneller, mit kürzeren Release-Zyklen unterwegs ist und Jakarta EE etwas langsamer agiert, dafür aber größten Wert auf Abwärtskompatibilität legt, soll und wird sich wohl nicht ändern.

Wie aber soll Jakarta die Nutzung von MP-Spezifikationen ermöglicht werden, um Synergien zu nutzen? Bislang läuft es nur in umgekehrter Richtung. Die Optionen sind: Ausgewählte MP-Spezifikationen werden ins Jakarta-Projekt überführt, ohne Anpassung (Option A1) oder mit Anpassung (Option A2) von Package-Namen; MP-Spezifikationen werden aus Jakarta heraus referenziert (B); oder es werden in Jakarta EE eigene Versionen von MP-Spezifikationen entwickelt – mit entsprechend höherem Aufwand (C). Alle Möglichkeiten haben eine Reihe von Vor- und Nachteilen, die ich hier gar nicht wiedergeben kann. Wer mag, kann sie direkt bei Reza nachlesen *[6]*.

24. Februar 2021

Quarkus 1.10, 1.11, 1.12

Was für Java der Release-Train, ist für Quarkus der Release-Bus (U-Bahn, was auch immer). Jedenfalls, was die Taktung angeht. Seit Quarkus 1.10 gibt es zwei neue Versionen: 1.11 hat zum Beispiel RESTEasy Reactive (eine reaktive JAX-RS-Implementierung) sowie eine experimentelle Dev-UI gebracht. Letztere zeigt alle geladenen Quarkus-Extensions in einer Web-basierten Konsole. Die Entwickler/innen der Erweiterungen können darüber dann nicht nur Informationen auflisten, sondern auch Aktionen erlauben (etwa das wiederholte Ausführen von Flyway-Skripten – „[the] sky is the limit" heißt es dazu in den Release Notes). Weitere Verbesserungen gibt es zum Beispiel bei der Micrometer- oder der Spring-Data-REST-Unterstützung.

Das gerade freigegebene Quarkus 1.12 bringt unter anderem HTTP-Multipart-Unterstützung für RESTEasy Reactive und macht das in Quarkus 1.5 eingeführte „Fast Jar" für schnellere Startzeiten zum Standard.

28. Februar 2021

Die nächsten Jakarta EE Releases

Jakarta EE 9.1 ist auf Kurs und soll Ende März freigegeben werden – vorausgesetzt GlassFish als kompatible Implementierung besteht die TCK-Tests bis Mitte März (die Ergebnisse für EE 9: *[7]*; für 9.1 dürften sie dann wohl auch bald dort erscheinen).

Dann geht's direkt weiter mit EE 10: Bis 15. April müssen die Komponenten-Spezifikationen Pläne einreichen. Diese werden dann auf ihre Umsetzbarkeit binnen sechs Monaten geprüft – damit das neue Release im Oktober fertig ist.

6. März 2021

Ausblick auf Java 17

Im September wird laut Release Train das neue Long-Term-Support Release (LTS) Java 17 herauskommen, in dieser Rolle Nachfolger von Java 11 aus 2018. Was die Features angeht, ist das Release aber noch recht überschaubar *[8]*. Ein verbesserter Zufallszahlen-Generator soll dabei sein sowie eine neue 2D-Rendering-Pipeline für MacOs (auf Basis des Apple-Metal-API anstelle des OpenGL-API). Aber Java 16 hat ja auch noch ein paar Features im Inkubator – mal sehen, welche davon die Produktionsreife erhalten.

17. März 2021

JavaLand als Online-Konferenz

Zwei Tage JavaLand vergingen wie im Flug. Nach der pandemiebedingten Absage 2020 gab es dieses Jahr eine reine Online-Konferenz. Die erste große Konferenz auf der neu geschaffenen Online-Plattform (basierend auf BigBlueButton und Vimeo für die Sessions, sowie Gather.Town für alles „dazwischen") hatte am ersten Tag noch so seine Problemchen, aber dank einer Nachtschicht des Technik-Teams lief es am zweiten Tag reibungslos. Dank gebührt auch den Teilnehmer/innen, die viel Geduld bewiesen und reichlich Feedback gegeben haben!

Damit auch online das JavaLand-Feeling aufkommt, wurde das Phantasialand mit viel Liebe in einer 2D-Landschaft nachgebaut. Hier konnten die Teilnehmer/innen ihre Avatare auch zwischen den virtuellen Vortragsräumen bewegen und dabei mit anderen in Kontakt treten – wie auf der „richtigen" Konferenz. Physische Treffen sind derzeit nun mal nicht möglich, doch eine Plattform, die mehr als Vortragsstreams und Chats bietet, kommt dem Konferenzerlebnis schon deutlich näher.

Dennoch hoffe ich auf eine Zukunft, in der ich wieder die Wahl habe, denn Gather.Town und speziell der Video-Chat sind zweifelsohne super, allerdings ist ein Zusammentreffen an der Matamba Bar in der physischen Welt für mich auch immer noch Teil der „JavaLand Experience" – die Digital Natives mögen mir das verzeihen. Nähere Informationen und Eindrücke zur JavaLand 2021 findet ihr auch in unserem Nachbericht ab Seite 16.

Referenzen

[1] *https://github.com/JetBrains/KotlinDL*
[2] *https://headcrashing.wordpress.com/2021/01/09/java-15-vs-java-8-modern-java-features-head-crashing-informatics-25/*
[3] *https://www.graalvm.org/reference-manual/java-on-truffle/*
[4] *https://jakarta.ee/compatibility/*
[5] *https://blogs.oracle.com/java/ways-graalvme-adds-value-to-java-se-subscription*
[6] *https://reza-rahman.me/2021/02/09/*
[7] *https://glassfish.org/compatibility*
[8] *openjdk.java.net/projects/jdk/17*

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

Seit mehreren Jahren schreibe ich an dieser Stelle meine Kolumne zu den Geschehnissen in der Eclipse Foundation; aufgrund der Wichtigkeit und Aktualität des Themas hauptsächlich meine pointierte Meinung zu Jakarta EE. Der eine oder andere von euch fragt sich vielleicht, wieso mir Herausgeber (iJUG e.V.) und Verlag (DOAG Dienstleistungen GmbH) eigentlich in jeder Ausgabe diesen wertvollen Platz einräumen. Daher geht es heute mal weniger um Jakarta EE (außer der Warterei auf 9.1 tut sich da grad eh nichts), sondern um Politik.

Ja genau, eigentlich geht es nämlich in dieser Kolumne eh schon immer um Politik und nicht um die wohl immer noch bekannteste Java IDE. Vor einigen Jahren sah es ja so aus, dass Oracle allein das Sagen hatte im Java-Universum, sowohl über Java SE als auch über Java EE. Wir als Nutzer dieser exzellenten Entwicklungsplattform waren auf Gedeih und Verderb dem Willen eines recht verschlossen agierenden Industrieriesen ausgeliefert, und damit waren wir, die Mitglieder der Java User Groups, nicht sonderlich zufrieden. Daher gründeten wir 2010 (ich selbst kannte den iJUG da noch gar nicht, aber es schreibt sich lockerer) den iJUG, den Interessenverbund der Java User Groups e. V., als Dachverband der deutschsprachigen Java User Groups, was so viel bedeutet wie: als Lobbyvereinigung der vielen tausend Java-Programmierer aus der DACH-Region gegenüber Oracle *[1]*.

Die Java aktuell war ein erstes gemeinsames Projekt, die JavaLand ein zweites, und beide sind bis heute sehr beliebt! Zudem pflegten wir sehr frühzeitig direkte Kontakte zu Oracle – bis heute. Ziel des iJUG ist es, im Java-Universum mitzureden, entgegen der Werbemacht der Konzerne die Meinungshoheit mitzuprägen und die Zukunft von Java mitzubestimmen. Wir machen also Politik aus Programmierersicht, und das ziemlich erfolgreich.

Mit der Übergabe von Java EE an die Eclipse Foundation (EF), also an eine quasi demokratisch agierende Stiftung, sahen wir (und in diesem Zusammenhang stimmt das „wir" nun tatsächlich) die Chance, die Interessen unserer Mitglieder noch direkter zu vertreten und mehr Macht auf das zukünftige Jakarta EE auszuüben. Der iJUG trat in die Eclipse Foundation ein (und diese in den iJUG), und die damaligen „Unterhändler" des iJUG, Jan Westerkamp und meine Wenigkeit, agieren seither als Botschafter des iJUG in der EF, machen also aktive Politik. Doch auch dies war uns nicht genug, womit wir (endlich!) zum heutigen Thema kommen.

In der EF gibt es zwei für uns sehr wichtige Arbeitsgruppen. Die eine kümmert sich um die Zukunft von Jakarta EE, die andere um die Zukunft der bislang von der AdoptOpenJDK-Gruppe herausgegebenen JDK-Distribution. Letztere heißt neuerdings Eclipse Adoptium und wird in einer kommenden Ausgabe von mir noch näher beleuchtet. Und in exakt diese beiden Arbeitsgruppen ist der iJUG kürzlich eingetreten.

Was bedeutet dies nun für euch? Die Mitgliedschaft im iJUG verschafft euch eine Stimme in diesen wichtigen Gremien! Der iJUG hat nun Stimmrecht bei Jakarta EE und bei Eclipse Adoptium, womit wir, die Java-Programmierer, nun endlich mitbestimmen können, und zwar sowohl bei der Zukunft der Enterprise-Plattform als auch bei der Zukunft einer sehr beliebten JDK-Distribution. Hinter den Kulissen werkelte an dieser neuen Verbindung neben Jan und mir diesmal vor allem Hendrik Ebbers, der sich bereits für AdoptOpenJDK engagierte und nun ebenfalls in leitender Rolle bei Adoptium aktiv ist.

Jetzt, da wir ein Stimmrecht in diesen Gremien haben, können wir endlich für euch aktive Politik betreiben, und daher mein Aufruf diesmal an alle Leser: Wer noch nicht Mitglied im iJUG ist (entweder direkt oder über eine der vielen Mitglieds-JUGs), hat nun zwei weitere, attraktive Gründe, um beizutreten. Und wer schon Mitglied ist, der möge uns doch bitte schreiben (office@ijug.eu), was er oder sie von unserer Lobbyarbeit erwartet, worauf wir unser Augenmerk legen sollen und wie die Zukunft von Eclipse Adoptium und Jakarta EE aussehen soll!

Die Macht ist nun endlich mit uns – nutzen wir sie weise, mit Bedacht und vor allem zu unserem gemeinsamen Vorteil!

Referenzen

[1] Zeitungsartikel zur Gründung: *https://www.datacenter-insider.de/interessenverbund-der-java-user-groups-gegruendet-a-245669/*

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.


16

Java 16 – die Änderungen im Überblick

Falk Sippach, embarc Software Consulting GmbH

Oracle hat ab Java 10 vor etwa drei Jahren damit begonnen, die Major-Releases halbjährlich auszuliefern. Das hörte sich zunächst verwegen an, hat sich aber als ein sehr sinnvoller Schachzug herausgestellt. Mittlerweile kommen alle sechs Monate immer genau im Zeitplan die neuen JDK-Versionen mit genau der Anzahl an Features heraus, die eingeplant waren und bis dahin fertiggestellt werden konnten. Wobei auch gezielt Previews von neuen Funktionen herausgegeben werden, damit sehr früh Feedback dazu eingesammelt werden kann. Vorbei sind also die Zeiten, in denen man drei, vier oder sogar fünf Jahre auf das nächste Java-Release warten musste und dann von der großen Menge an Neuerungen geradezu erschlagen wurde. Mitte März wurde nun das OpenJDK 16 finalisiert. Tatsächlich ist Java 16 die letzte Major-Version vor dem nächsten Long-Term-Support(LTS)-Release, das als OpenJDK 17 im September 2021 erscheinen wird.

Die Liste der für das OpenJDK 16 umgesetzten JEPs (Java Enhancement Proposals) sieht auf den ersten Blick relativ lang aus *[1]*. Allerdings sind auch einige alte Bekannte aus den vergangenen Releases wieder mit von der Partie, die diesmal entweder abgeschlossen oder als weitere Previews erneut verprobt werden.

- 338: Vector API (Incubator)
- 347: Enable C++14 Language Features
- 357: Migrate from Mercurial to Git
- 369: Migrate to GitHub
- 376: ZGC: Concurrent Thread-Stack Processing
- 380: Unix-Domain Socket Channels
- 386: Alpine Linux Port
- 387: Elastic Metaspace
- 388: Windows/AArch64 Port
- 389: Foreign Linker API (Incubator)
- 390: Warnings for Value-Based Classes
- 392: Packaging Tool
- 393: Foreign-Memory Access API (Third Incubator)
- 394: Pattern Matching for instanceof
- 395: Records
- 396: Strongly Encapsulate JDK Internals by Default
- 397: Sealed Classes (Second Preview)

Einige der Punkte sind für Java-Entwickler jedoch nicht direkt relevant. Dazu zählen beispielsweise die Migration zu Git beziehungsweise GitHub, die Aktivierung der C++14-Sprachfeatures und auch das Foreign-Linker-API als zukünftiger Ersatz für das Java Native Interface (JNI). Wir werfen am Ende des Artikels trotzdem einen Blick darauf.

Schauen wir aber zunächst auf die für Entwickler relevanten Funktionen. Dem aufmerksamen Beobachter der vergangenen Java-Releases werden hier allerdings keine bahnbrechenden Neuerungen ins Auge springen. Das hängt hauptsächlich mit dem bevorstehenden LTS-Release zusammen, das im Herbst 2021 erscheinen wird. Bis Java 17 werden die neuen Features der vergangenen Monate stabilisiert, um dann für die folgenden Jahre eine gute Ausgangsbasis für die notwendigen Updates und Patches zu schaffen.

Bereits das dritte Mal dabei: Records

Die vor einem Jahr in Java 14 eingeführten Record-Datentypen wurden mit dem JEP 395 nun finalisiert. Es gab seit der zweiten Preview (JDK 15) nur noch einige kleine Änderungen, die sich aus den Rückmeldungen der letzten Monate ergeben haben.

Bei den Records handelt es sich um eine eingeschränkte Form der Klassendeklaration, ähnlich zu den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Die kompakte Syntax wird Bibliotheken wie Lombok in Zukunft zumindest zum Teil obsolet machen. Die einfache Definition einer Person mit zwei Feldern kann man in *Listing 1* sehen. Eine erweiterte Variante mit einem zusätzlichen Konstruktor ist erlaubt. Dadurch lassen sich neben Pflichtfeldern auch optionale Felder abbilden *(siehe Listing 2)*.

```
public record Person(String name, Person partner) {}
```

Listing 1

```
public record Person(String name, Person partner) {
  public Person(String name) {
    this(name, null);
  }
  public String getNameInUppercase() {
    return name.toUpperCase();
  }
}
```

Listing 2

Vom Compiler wird eine unveränderbare (immutable) Klasse erzeugt, die neben den beiden Attributen und den eigenen Methoden natürlich auch noch die Implementierungen für die Accessoren, den Konstruktor sowie `equals/hashCode` und `toString` enthält. In *Listing 3* sieht man den Pseudo-Code, den man dafür hätte schreiben müssen. Dann werden Records wie normale Java-Klassen verwendet. Der Aufrufer merkt also gar nicht, dass ein spezieller Typ instanziiert wird *(siehe Listing 4)*.

Records sind tatsächlich keine klassischen Java Beans, da sie keine echten Getter enthalten. Man kann auf die Membervariablen jedoch über die gleichnamigen Methoden zugreifen (`name()` statt `getName()`). Sie können im Übrigen auch Annotationen oder JavaDocs enthalten. Im Body dürfen zudem statische Felder sowie

```
public final class Person extends Record {
  private final String name;
  private final Person partner;
  public Person(String name) { this(name, null); }
  public Person(String name, Person partner) {
    this.name = name; this.partner = partner;
  }
  public String getNameInUppercase() {
    return name.toUpperCase();
  }
  public String toString() { /* ... */ }
  public final int hashCode() { /* ... */ }
  public final boolean equals(Object o) { /* ... */ }
  public String name() { return name; }
  public Person partner() { return partner; }
}
```

Listing 3

```
var man = new Person("Adam");
var woman = new Person("Eve", man);
woman.toString(); // ==> "Person[name=Eve, partner=Person[name=Adam, partner=null]]"
woman.partner().name(); // ==> "Adam"
woman.getNameInUppercase(); // ==> "EVE"
// Deep equals
new Person("Eve", new Person("Adam")).equals( woman ); // ==> true
```

Listing 4

Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Die Definition weiterer Instanzfelder außerhalb des Record-Headers ist nicht erlaubt.

Java goes Pattern Matching

Bereits seit einiger Zeit schwebt das Thema Pattern Matching im Raum und hält nach und nach Einzug in Java. Dazu sind zur Vorbereitung diverse Änderungen in der Sprache selbst notwendig, deshalb erfolgt die Einführung nur schrittweise. Los ging es bereits im JDK 12 mit den Switch Expressions. Seit Version 14 gab es zudem bereits zwei Previews zu „Pattern Matching for instanceof". Das wurde nun mit Java 16 abgeschlossen.

Ein Pattern ist übrigens eine Kombination aus einem Prädikat (das auf eine Zielstruktur passt) und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist letztlich die Destrukturierung von Objekten, also das Aufspalten in die Bestandteile und Zuweisen in einzelne Variablen zur weiteren Bearbeitung.

Die Spezialform des Pattern Matching beim `instanceof`-Operator spart unnötige Casts auf die zu prüfenden Ziel-Datentypen. Wenn `o` ein String oder eine Collection ist, dann kann direkt mit den neuen Variablen (`s` und `c`) mit den entsprechenden Datentypen weitergearbeitet werden. Das Ziel ist es, Redundanzen zu vermeiden und dadurch die Lesbarkeit zu erhöhen *(siehe Listing 5)*.

Der Unterschied zum zusätzlichen Cast mag marginal erscheinen. Für die Puristen unter den Java-Entwicklern spart das allerdings eine kleine, aber dennoch lästige Redundanz ein. Laut Brian Goetz soll die Sprache dadurch prägnanter und die Verwendung sicherer gemacht werden. Erzwungene Typumwandlungen werden vermieden und stattdessen implizit durchgeführt. Bereits die zweite Preview, die im JDK 15 erschienen war, hatte keine nennenswerten Änderungen mehr mit sich gebracht. Deswegen wurde das Feature jetzt als JEP 394 finalisiert. In zukünftigen Java-Versionen wird es jedoch noch weitere Neuerungen rund um das Pattern Matching geben, zum Beispiel in Zusammenarbeit mit den Switch Expressions.

Versiegelte Klassen

Tatsächlich erst das zweite Mal dabei sind die Sealed Classes. Sie wurden in Java 15 als Preview-Feature eingeführt und verbleiben als JEP 397 auch im JDK 16 im Vorschau-Modus. Es gibt ein paar kleine Ergänzungen gegenüber der letzten Version und vermutlich werden sie dann im LTS-Release des OpenJDK 17 finalisiert. Bis dahin möchten die Macher allerdings noch Rückmeldungen einsammeln.

Dieses Feature wurde übrigens im Rahmen von Projekt Amber entwickelt und gehört ebenfalls zu einer Reihe von vorbereitenden Maßnahmen für die Umsetzung von Pattern-Matching-Mechanismen in Java. Ganz konkret soll es bei der Analyse von Mustern unterstützen. Aber auch für Framework-Entwickler bieten die Sealed Classes einen interessanten Mehrwert. Die Idee ist, dass versiegelte Klassen und Interfaces entscheiden können, welche Sub-Klassen oder -Interfaces von ihnen abgeleitet werden dürfen. Bisher konnte

```
boolean isNullOrEmpty(Object o) {
    return o == null ||
      o instanceof String s && s.isBlank() ||
      o instanceof Collection c && c.isEmpty();
}
```

Listing 5

man als Entwickler die Ableitung von Klassen nur durch Zugriffsmodifikatoren (private, protected, ...) einschränken oder durch die Deklaration der Klasse als `final` komplett durch den Compiler untersagen. Sealed Classes bieten nun einen deklarativen Weg, um gezielt bestimmten Subklassen die Ableitung zu erlauben *(siehe Listing 6)*.

```
public sealed class Vehicle
       permits Car, Bike, Bus, Train {
}
```

Listing 6

Vehicle darf nur von den vier genannten Klassen überschrieben werden. Damit wird auch dem Aufrufer deutlich gemacht, welche Subklassen erlaubt sind und überhaupt existieren. In Zukunft sollen Sealed Classes auch bei Switch-Expressions eingesetzt werden können (im Rahmen des Pattern Matching). Wenn man dann je `case`-Zweig alle erlaubten Subklassen verwendet, kann der Einsatz des `default`-Blocks entfallen. Durch die Information aus der `permit`-Anweisung kann der Compiler sicherstellen, dass mindestens einer der Zweige aufgerufen wird *(siehe Listing 7)*.

Subklassen bergen immer die Gefahr, dass beim Überschreiben der Vertrag der Superklasse und damit das Liskov'sche Substitutionsprinzip verletzt wird. Zum Beispiel ist es unmöglich, die Bedingungen der `equals`-Methode aus der Klasse `Object` zu erfüllen, wenn man Instanzen von einer Super- und einer Subklasse miteinander vergleichen will. Weitere Details dazu kann man in der API-Dokumentation *[2]* unter dem Stichwort Äquivalenzrelationen (konkret Symmetrie) nachlesen. Als Autor einer Superklasse hat man nun mit Sealed Classes mehr Kontrolle darüber, welche Subklassen abgeleitet werden und wo man dementsprechend auf das Einhalten der geerbten Verträge achten muss.

Sealed Classes funktionieren auch mit abstrakten Klassen. Es gibt jedoch ein paar Einschränkungen. Eine Sealed Class und alle erlaubten Sub-Klassen müssen im selben Modul existieren. Im Falle von Unnamed Modules müssen sie sogar im gleichen Package liegen. Außerdem muss jede erlaubte Sub-Klasse direkt von der Sealed Class ableiten. Die abgeleiteten Klassen dürfen übrigens wieder selbst entscheiden, ob sie weiterhin versiegelt, final oder komplett offen sein wollen. Die zentrale Versiegelung einer ganzen Klassenhierarchie von oben bis zur untersten Hierarchiestufe ist nicht möglich.

Zwischen Sealed Classes und den Record-Typen gibt es übrigens eine Integration, wie das Beispiel in *Listing 8* zeigt.

Eine Familie von Records kann vom gleichen Sealed Interface ableiten. Die Kombination aus Records und versiegelten Klassen führt uns zu algebraischen Datentypen, die vor allem in funktionalen Sprachen wie Haskell zum Einsatz kommen. Konkret können wir jetzt mit Records sogenannte Produkttypen und mit versiegelten Klassen Summentypen abbilden. Und auch hier schließt sich wieder der Kreis zum Pattern Matching, das ebenfalls vor allem in funktionalen Sprachen zum Einsatz kommt. Die Modernisierung von Java entwickelt sich also weiterhin in die vielversprechende Richtung der funktionalen Programmierung. Aber bitte nicht erwarten, dass Java in ein paar Jahren eine rein funktionale Sprache sein wird. Da wird es auch in Zukunft besser geeignete Kandidaten (Haskell, Clojure, etc.) geben. Nichtsdestotrotz wird man auch bei der Entwicklung mit Java von einigen dieser Möglichkeiten profitieren können.

Was sonst noch so geschah ...

Bisher wurden die Quellen des OpenJDK in Mercurial verwaltet, einem nicht so verbreiteten Versionsverwaltungssystem. Dadurch war die Hürde für neue Entwickler relativ hoch, sich an der Entwicklung des JDK zu beteiligen. Im Rahmen des JEP 357 wurde der Source Code nun in ein Git Repository migriert und sogar noch nach GitHub *[3]* umgezogen (JEP 369). Dabei gab es drei Hauptgründe für die Migration:

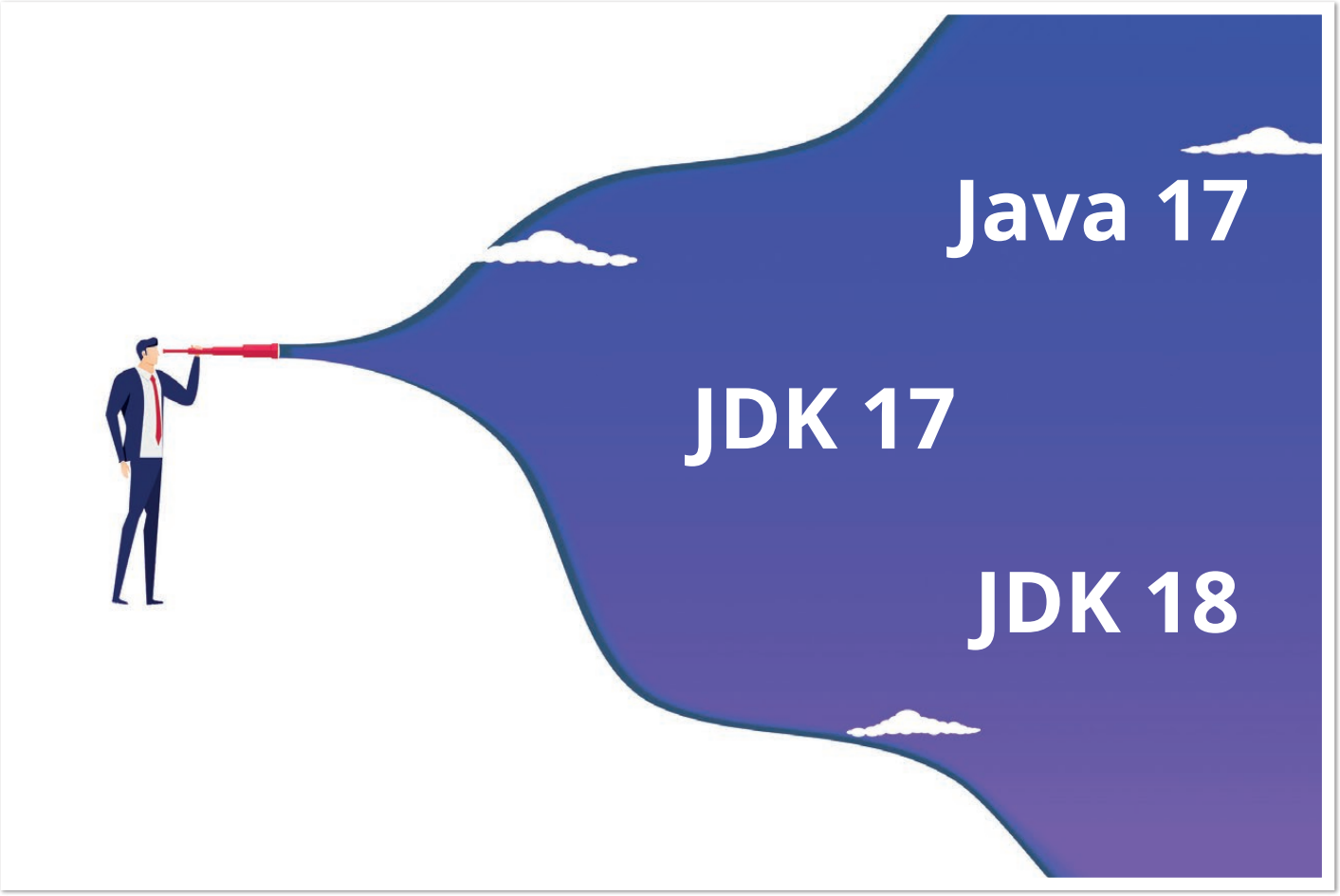
1. Größe der Metadaten des Versionsveraltungssystems
2. Verfügbare Werkzeuge für die Versionsverwaltung
3. Angebote an Hosting-Optionen

```
// noch kein gültiger Code, kommt erst in späteren Java-Versionen
public BigDecimal calculateExpense(Vehicle vehicle) {
  return switch(vehicle) {
    case Car c -> calculateCarExpense(c);
    case Bike b -> calculateBikeExpense(b);
    case Bus b -> calculateBusExpense(b);
    case Train t -> calculateTrainExpense(t);
  }
}
```

Listing 7

```
public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr {
  ...
}
public record ConstantExpr(int i) implements Expr {...}
public record PlusExpr(Expr a, Expr b) implements Expr {...}
public record TimesExpr(Expr a, Expr b) implements Expr {...}
public record NegExpr(Expr e) implements Expr {...}
```

Listing 8



Bei den Metadaten kam es immerhin zu einer Reduktion von 1,2 GByte auf 300 MByte im .git-Ordner. Zudem wird bei vielen Tools wie IDEs oder Texteditoren Git bereits standardmäßig unterstützt oder lässt sich leicht über Plug-ins erweitern. Git hat einfach den Markt für verteilte Versionsverwaltungssysteme in den letzten Jahren nahezu überrollt. Der Schritt, die Quellen des OpenJDK nach GitHub umzuziehen, ist also nachvollziehbar. Um alle relevanten Informationen wie die Historie und Tags mit zu übertragen, wurde eigens ein kleines Tool geschrieben. Dieses hat die Mercurial Commit Messsages in das Git-Format überführt.

Wer schon sehr lange in der Java-Welt unterwegs ist, wird sich vielleicht noch an das Java Native Interface (JNI) erinnern. Damit kann man nativen C-Code aus Java heraus aufrufen. Der Ansatz ist jedoch relativ aufwendig und fragil. Das Foreign-Linker-API (JEP 389) bietet nun einen statisch typisierten, rein Java-basierten Zugriff auf nativen Code. Zusammen mit dem Foreign-Memory-Access-API (JEP 393) kann diese Schnittstelle den bisher fehleranfälligen Prozess der Anbindung einer nativen Bibliothek beträchtlich vereinfachen. Mit Letzterer bekommen Java-Anwendungen die Möglichkeit, außerhalb des Heap zusätzlichen Speicher zu allokieren.

Wer häufig mit den primitiven Wrapper-Klassen (Integer, Boolean etc.) arbeitet, wird ab dem JDK 16 womöglich über neue Deprecation-for-Removal-Warnungen stolpern. Das betrifft die Konstruktoren sowohl mit dem String-Parameter als auch mit dem jeweiligen primitiven Datentyp als Argument (int bei Integer). Hinter dieser Maßnahme steckt auch das Projekt Valhalla. Dort strebt man die Erweiterung des Java-Programmiermodells in Form von primitiven Klassen an. Diese primitiven Klassen sollen keine Identität besitzen und dadurch auch leicht vom Compiler beziehungsweise dem Laufzeit-Interpreter „ge-inlined" werden können. Dadurch lassen sie sich frei zwischen Speicherorten kopieren und als Werte von Instanzfeldern kodieren.

Ebenfalls dem Vorschau-Feature entwachsen ist mit dem OpenJDK 16 das jpackage-Werkzeug. Es unterstützt native Paketformate, um den Nutzern eine einfache Installation zu ermöglichen, inklusive der Angabe von Startparametern zum Zeitpunkt der Paketierung. Zu den Formaten gehören msi und exe unter Windows, pkg und dmg unter MacOS sowie deb und rpm unter Linux. Das Tool kann direkt über die Befehlszeile oder auch programmatisch aufgerufen werden.

Durch das in Java 9 eingeführte Java-Plattform-Modul-System (JPMS) können nun JDK-interne Klassen vor dem Zugriff von außen geschützt werden. Bisher galt zum Übergang die eingeschränkte Kapselung als Default, das heißt, interne APIs konnten weiterhin verwendet werden. Dieses Schlupfloch wird ab dem OpenJDK 16 geschlossen. Die neue Standardeinstellung ist die strikte Kapselung der JDK-Interna, außer für sehr kritische interne APIs wie misc. Unsafe. Das Ziel dieser Maßnahme ist die Erhöhung der Sicherheit und der Wartbarkeit des JDK. Man möchte die Entwickler ermutigen, alte, auf Interna basierende Lösungen zukünftig für den Zugriff auf Standard-APIs umzubauen. Somit sollen sowohl Java-Entwickler als auch Endbenutzer viel problemloser auf zukünftige Versionen updaten können.

Neben den prominenten JEPs gibt es in jeder neuen JDK-Version noch viele kleine Änderungen, zum Beispiel an der Java-Klassenbibliothek. Mit dem Java-Version-Almanac *[4]* kann man sehr einfach und kompakt die Differenzen zwischen den Releases, aber auch bei Versionssprüngen von zum Beispiel JDK 8 auf 16, einsehen. Aus Entwicklersicht sind besonders zwei Neuerungen an der Klasse Stream interessant. `Stream.toList()` bietet eine prägnantere und im Einsatz mit `parallel()` meist auch effizientere Alternative zu `Stream.collect(Collectors.toList()`. Als Ergebnis wird eine nicht veränderbare (unmodifiable) ArrayList zurückgegeben. Weitere Informationen kann man der API-Doc *[5]* oder dem Artikel von Donald Raab *[6]* entnehmen. Die zweite Neuerung in der Klasse Stream ist die in diversen Ausprägungen hinzugekommene Methode `mapMulti(BiConsumer)`. Sie stellt eine imperative und schnellere Alternative zu `flatMap` dar. Nicolai Parlog hat die neue Funktion in einem Blog-Post näher unter die Lupe genommen und zum Vergleich Performance-Messungen durchgeführt *[7]*.

Ausblick

Schon kurz vor der Fertigstellung des OpenJDK 16 wurden die ersten Features für das JDK 17 angekündigt. Unter anderem soll es eine neue Rendering Pipeline für MacOS und die Erweiterung des Pseudo-Zufallszahlen-Generators geben. Mit dem Erscheinen dieses Artikels wird die Liste der Neuerungen sicher schon länger geworden sein, ein Blick auf die Projektseite des OpenJDK 17 lohnt also auf jeden Fall *[8]*. Höchstwahrscheinlich werden wir jedoch keine weiteren großen, prominenten Änderungen sehen. Schließlich werden mit Java 17 die Entwicklungen der vergangenen 36 Monate abgeschlossen und es wird eine Version bereitgestellt, die wiederum für die nächsten Jahre mit Updates und Sicherheit-Patches versorgt werden muss. Die wirklich spannenden Neuerungen und Syntax-Erweiterungen werden also voraussichtlich erst wieder in der Version 18 erscheinen. Ab da hat Oracle dann wieder zweieinhalb Jahre Zeit, Feedback zu Preview-Features einzuarbeiten und diese abzurunden.

Der 2018 eingeschlagene Weg, der anfangs nicht unumstritten war, wird weiterhin konsequent verfolgt. Die halbjährlichen Updates der Programmiersprache und der Plattform Java geben uns Entwicklern die einfache Möglichkeit, regelmäßig neue Funktionen ausprobieren zu können. Potenziell kann man sogar sein Produktivsystem alle sechs Monate aktualisieren und vermeidet langwierige Migrationsaufwände, die sonst alle paar Jahre anfallen würden. Konservative Kunden können aber trotzdem LTS-Versionen einsetzen, die wie frühere Releases (vor Java 9) über mehrere Jahre mit Updates versorgt werden. Dabei hat man mittlerweile die freie Wahl zwischen verschiedenen Anbietern. Neben kommerziellen Versionen (Oracle JDK und andere) gibt es auch genügend JDKs mit freien Updates (allen voran AdoptOpenJDK).

Das Java-Ökosystem ist also lebendiger denn je und weiterhin sehr innovativ. Konkurrenz, wie beispielsweise Python (hauptsächlich wegen Machine/Deep Learning), Go und die C-basierten Sprachen beleben das Geschäft. Java, das besonders stark im Unternehmensanwendungsumfeld vertreten ist, wird allerdings weiterhin ein gehöriges Wort mitreden.

Referenzen

[1] *https://openjdk.java.net/projects/jdk/16/*

[2] *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals%28java.lang.Object%29*

[3] *https://github.blog/2020-09-30-github-welcomes-the-open-jdk-project/*

[4] *https://javaalmanac.io/*

[5] *https://download.java.net/java/early_access/jdk16/docs/api/java.base/java/util/stream/Stream.html#toList()*

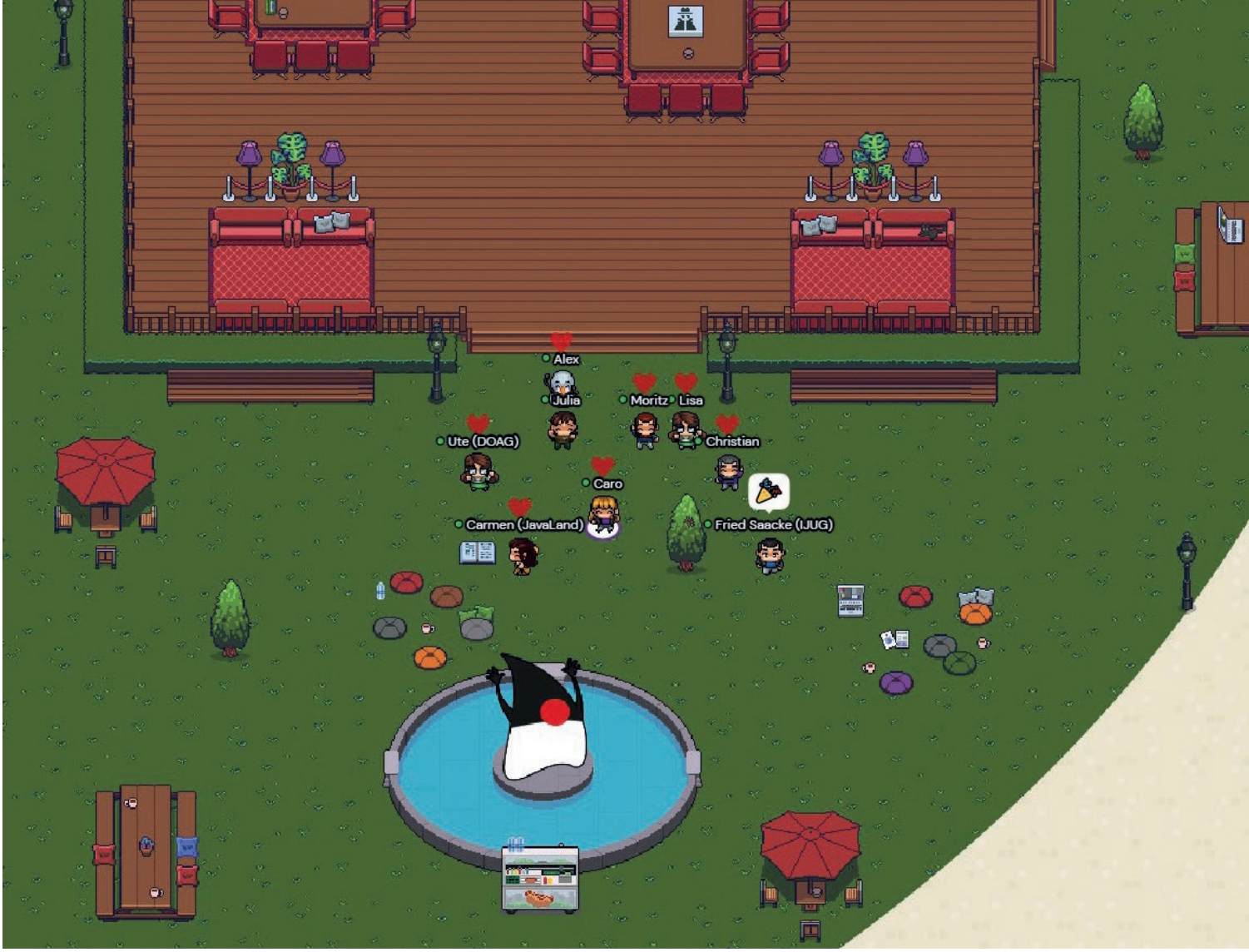
[6] *https://medium.com/javarevisited/stream-tolist-and-other-converter-methods-ive-wanted-since-java-2-c620500cb7ab*

[7] *https://nipafx.dev/java-16-stream-mapmulti/*

[8] *https://openjdk.java.net/projects/jdk/17/*

Falk Sippach
embarc Software Consulting GmbH
falk@jug-da.de

Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Bereits seit über 15 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blog-Beiträgen sowie bei Vorträgen auf Konferenzen oder User-Group-Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen. Falk twittert unter @sippsack.



JavaLand 2021: Online-Premiere punktet mit einmaligem Community-Erlebnis

Christian Luda, DOAG Dienstleistungen GmbH

Ihren siebten Geburtstag feierte die JavaLand in diesem Jahr auf besondere Art: Sie lud erstmals ausschließlich online zum großen Wiedersehen der Java-Community ein.

Nachdem die Community-Konferenz zuvor sechs Jahre lang erfolgreich im Phantasialand in Brühl stattfand, musste sie im vergangenen Jahr pandemiebedingt ausfallen und fand aufgrund der andauernden Beschränkungen 2021 erstmals online statt.

Mehr als 1.300 Java-Fans versammelten sich am 16. und 17. März vor ihren Rechnern, Laptops, Tablets und Smartphones, um zwei Tage lang mehr als 120 Vorträge zu 13 Themengebieten rund um ihre Lieblingsprogrammiersprache zu verfolgen. Auch auf spannende Workshops und Community-Aktivitäten mussten die Besucher online nicht verzichten.

Während die Teilnehmer auf der Konferenzplattform zwischen acht parallelen Streams wählen konnten, sorgte ein weiteres Online-Tool dafür, dass ein besonders wichtiges Element der JavaLand, die so-

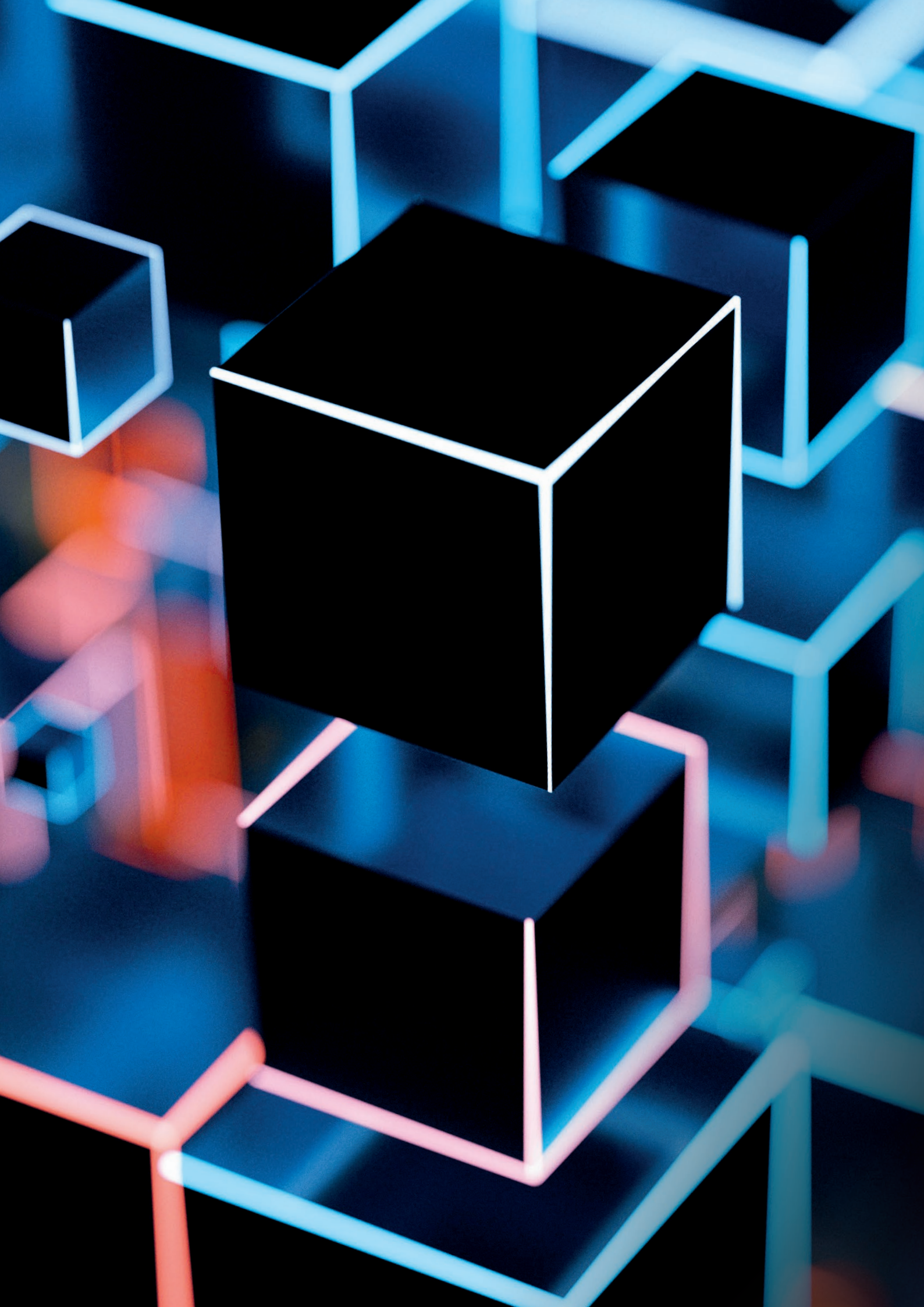
ziale Interaktion, nicht zu kurz kam: In Gather.Town fanden die Java-Fans eine virtuelle Nachbildung des Phantasialands vor. Hier konnten sich die Teilnehmer mit einem Avatar durch den Park bewegen und mit Gleichgesinnten austauschen. An ihren virtuellen Ständen empfingen die Austeller Interessierte mit spannenden Aktionen und Gesprächen. In den virtuellen Vortragsräumen gab es nach den Vorträgen außerdem Gelegenheit, mit dem Referenten ins Gespräch zu kommen und Fragen loszuwerden.

Dank Gather.Town stellte sich auch in der virtuellen Welt schnell das vielzitierte Community-Gefühl der JavaLand ein. Verzichten mussten die Teilnehmer in diesem Jahr neben dem Catering allerdings auf den OpenPark mit seinen Fahrgeschäften. Stattdessen gab es am ersten Konferenztag für alle Beteiligten eine Achterbahnfahrt der Gefühle: Am Vormittag kam es in Folge einer Überlastung der Webserver zu Ausfällen auf der Konferenzplattform. Auch nach dem Hinzubuchen weiterer Server gab es bei vielen Teilnehmern noch Probleme, die Streams zu verfolgen. Über Nacht gelang es schließlich der Technik, die Probleme zu lösen, sodass der zweite Konferenztag reibungslos über die virtuelle Bühne gehen konnte.

Im Raum Silverado, der virtuellen Hauptbühne der JavaLand, bekamen Java-Fans neben ausgewählten Vorträgen wie der Eröffnungskeynote „The Value of Learning and Sharing" von Bert-Jan Schrijver von morgens bis abends ein unterhaltsames Rahmenprogramm live aus dem JavaLand-Studio geboten. Zwischen den Vorträgen begrüßten die Moderatoren spannende Interviewgäste und führten mit jeder Menge Unterhaltung durch das Programm. Darüber hinaus empfingen Hendrik Ebbers und Sandra Parsick die Zuschauer an beiden Tagen zum JavaLand-Frühstücksfernsehen, bei dem es neben viel Spaß diverse Preise zu gewinnen gab. Zum Ausklang des ersten Tages gingen die beiden dann mit einer JavaLand-Ausgabe der CyberLand Late Night noch einmal auf Sendung.

Besonders überwältigt waren wir, insbesondere nach den anfänglichen technischen Schwierigkeiten, vom positiven Feedback der Teilnehmer, das uns im Laufe der Konferenz über den Chat, auf Twitter sowie in Gather.Town erreichte. So stand am Ende der beiden Tage ein äußerst positives Fazit: Die Online-Premiere der JavaLand ist gelungen. Auch an dieser Stelle möchten wir uns nochmal bei allen Mitwirkenden und Teilnehmern bedanken!

Während viele sich nach einem baldigen Wiedersehen vor Ort sehnen, brachte die Online-Variante auch Vorteile mit sich. Adam Bien etwa war erstmals als Speaker auf der JavaLand dabei – der Java-Champion hatte es bis dato aufgrund der langen Anreise nie auf die JavaLand-Bühne geschafft. Ein weiterer Pluspunkt: Alle Vorträge wurden aufgezeichnet und werden nun im Nachgang den Teilnehmern exklusiv zur Verfügung gestellt, sodass jeder die Möglichkeit hat, seine persönlichen Lieblingsvorträge noch einmal zu erleben und Verpasstes nachzuholen – die perfekte Einstimmung auf die JavaLand 2022. Apropos, JavaLand 2022 – Save the Date: Wir hoffen, euch vom 15. bis 17. März 2022 in alter Frische vor Ort begrüßen zu dürfen!



Clojure: Einstieg und Überblick

Tim Zöller, lambdaschmiede GmbH

„That one's a pretty easy one, and that's Clojure. And the reason for that is, I think Clojure is one of the few production quality, serious JVM languages that isn't just ‚let's do Java better'", so beantwortete Brian Goetz, Java Language Architect bei Oracle, im Interview mit Nicolai Parlog die Frage nach seiner Lieblings-JVM-Sprache – abgesehen von Java, versteht sich [1].

Damit bringt er die Unterschiede auch ziemlich gut auf den Punkt: Zwar läuft Clojure auch (unter anderem) auf der JVM und kann mit Java-Klassen interagieren, die zugrunde liegenden Paradigmen sind jedoch grundverschieden. Während Java eine objektorientierte Sprache ist, deren Syntax eher von der C-Familie beeinflusst wurde, handelt es sich bei Clojure um einen Lisp-Dialekt. Da viele Entwickler Lisp-Dialekte ausschließlich aus dem Studium kennen, hängt ihnen der Ruf nach, eher akademisch und wenig praxisbezogen zu sein – ein Irrglaube. Dieser Artikel soll ein schneller Überblick zu Clojure für Java-Entwickler sein, die gerne über den Tellerrand schauen und bereit sind, sich auf andere Programmierparadigmen einzulassen.

Syntax und Ausdrücke

Clojure evaluiert, simpel ausgedrückt, Listen. Bevor wir näher auf die Syntax eingehen können, ist es zunächst hilfreich, anhand von *Listing 1* zu verstehen, wie dies geschieht. Listen werden durch runde Klammern ausgedrückt.

```
(+ 1 2)
=> 3
```

Listing 1: Ein simpler Funktionsaufruf in Clojure

Der Code beschreibt also eine Liste mit drei Elementen: dem Symbol `+` und den Nummer-Literalen `1` und `2`. Wird eine solche Liste als Ausdruck interpretiert, muss das erste Element stets eine Funktion sein, sämtliche weiteren Elemente stellen die Argumente dar. Der Ausdruck `(+ 1 2)` ruft also die Funktion hinter dem Symbol `+` mit den Argumenten `1` und `2` auf; das Ergebnis ist dementsprechend `3`. Dieses Prinzip gilt konsequent: alle Ausdrücke in Clojure folgen der Präfix-Notation. Dies gilt auch für verschachtelte Ausdrücke, etwa `(+ (* 2 3) (- 5 1))`. Hier werden zuerst die inneren Ausdrücke evaluiert, dann der äußere. Die Präfix-Notation ist für die meisten Java-Entwickler zunächst ungewohnt (und die vielen Klammern erst!), jedoch ergibt sich durch sie ein sehr striktes und einheitliches Codebild: Während es in den meisten Sprachen neben Funktionen auch festverdrahtete Operatoren gibt, die jeweils rechts- oder linksassoziativ sind, existieren in Clojure nur Funktionen.

Gängige Literale

Die gesamte Clojure-Syntax kann sich sehr kompakt zusammenfassen lassen. Aus diesem Baukasten können wir unsere gesamten Programme schreiben *(siehe Listing 2)*. Aus den numerischen und alphanumerischen Literalen sticht zunächst die Ratio hervor. Das Literal `2/3` beschreibt tatsächlich einen Bruch. Mit diesem lässt sich rechnen, ohne Gleitkommafehler zu riskieren. Der Ausdruck `(* 5 2/3)` wird zum Ergebnis 10/3 ausgewertet, und nicht zu „ungefähr 3,3333333".

Interessant sind darüber hinaus die Collections. Soll eine Liste als Datenstruktur verwendet und nicht ausgeführt werden, so wird sie mit einem „Single Quote" versehen. Listen in Clojure ähneln am ehesten Linked Lists – Elemente am Anfang der Liste hinzuzufügen ist eine sehr schnelle Operation. Vektoren werden durch eckige Klammern ausgedrückt, sie bieten Elementzugriffe durch Indizes, neue Elemente werden am Ende angehängt. Sets werden mit geschweiften Klammern, die einer Raute folgen, ausgedrückt. Maps bestehen aus geschweiften Klammern mit einer geraden Anzahl von Elementen – jeweils die Schlüssel gefolgt von ihren Werten. Es ist erlaubt, Elemente in Collections mit optionalen Kommas zu trennen, etwa `{"a" 1, "b" 2, "c" 3}` oder `[1, 2, 3, 4]`. In der Praxis wird dies jedoch eher selten praktiziert.

Neben Symbolen (die beispielsweise auf Funktionen referenzieren können, wie vorher schon gesehen, oder auf Spezialformen wie `true`, `false`, `null`) gibt es schließlich noch Keywords. Aus Java-Perspektive lassen sie sich am ehesten mit Enumerationen vergleichen, die nicht extra deklariert werden müssen, sondern direkt im Code verwendet werden. Sie eignen sich beispielsweise hervorragend als Schlüssel für Maps: `{:name "Harry Hirsch" :age 65}`.

Definition von Variablen und Funktionen

Natürlich bestehen Clojure-Programme nicht nur aus Listen, die die bestehenden Funktionen der Kernsprache benutzen, und Literalen. Es besteht die Möglichkeit, eigene Variablen und Funktionen zu definieren *(siehe Listing 3)*. Globale Variablen werden (jeweils für einen Namespace) mit dem Symbol `def` definiert. Das Symbol `defn` definiert Funktionen, die mit dem darauffolgenden Bezeichner referenziert werden können. Es folgt ein optionaler Docstring, der die Funktion dokumentiert. An dritter Stelle steht ein Vektor mit Parametern.

Java-Entwicklern wird zuerst auffallen, dass der Parameter nicht typisiert ist. In der Tat ist Clojure eine dynamisch typisierte Sprache. Zuletzt folgt der eigentliche Code der Funktion. Die hier definierte Version berechnet die Fakultät des übergebenen Parameters. Um dies nachzuvollziehen, liest man den Ausdruck der Funktion von innen nach außen: `(range n)` erzeugt eine Liste mit n Einträgen, be-

```
;; Alphanumerisch
"Ich bin ein String"    ;; String
\c                      ;; Character
#"[a-z]*"               ;; Regular Expression

;; Numerisch
-100                    ;; Integer
32.7                    ;; Floating Point
2/3                     ;; Ratio
5.3M                    ;; High Precision Floating Point (benutzt BigDecimal)

;; Collections
'(1 2 3 4)              ;; List
[1 2 3 4]               ;; Vector
#{1 2 3 4}              ;; Set
{"a" 1 "b" 2 "c" 3}     ;; Map

;; Symbols & keywords
+                       ;; Symbol
nil                     ;; Null
true false              ;; booleans
:keyword                ;; Keyword
:namespaced/keyword     ;; Keyword mit Namespace
```

Listing 2: Clojure-Syntax im Überblick

```
(def current-year 2021)
(def birth-year 1989)

(def age (- current-year birth-year))

age
=> 32

(defn faculty
  "Calculates the faculty for the given number"
  [n]
  (reduce * (map inc (range n))))

;; Aufruf
(faculty 6)
=> 720
```

Listing 3: Definition von Variablen und Funktionen

ginnend bei 0. Die Funktion `map` wird, genau wie bei Java, verwendet, um eine Funktion auf jedes Element einer Collection anzuwenden – dies lässt unsere Liste bei 1 starten. Die Funktion `reduce` wendet eine Funktion auf die ersten beiden Parameter einer Collection an, dann auf das Ergebnis und den dritten Parameter, und so weiter – in diesem Fall die Funktion `*`. Alle Elemente der Liste werden miteinander multipliziert, was in der Fakultät unseres übergebenen Parameters resultiert. Was an diesem Beispiel sehr schön zu sehen ist: Die neue Funktion wird durch die Komposition fünf weiterer Funktionen erzeugt: `reduce`, `*`, `map`, `inc`, `range`. Der gesamte Code beschäftigt sich ausschließlich mit dem zu lösenden Problem.

Unveränderliche Datenstrukturen

Unveränderliche Datenstrukturen sind eine der wichtigsten Eigenschaften von Clojure. Werden Operationen auf diesen Datenstrukturen ausgeführt, wird eine manipulierte Kopie der ursprünglichen Datenstruktur zurückgegeben. Im folgenden Beispiel wird der global definierte Vektor `numbers` im Ausdruck `(map inc numbers)` benutzt, um einen neuen Vektor zu erzeugen, bei dem jedes Element um 1 erhöht wurde. Der eigentliche Vektor wird von dieser Aktion nicht beeinflusst, wie das *Listing 4* verdeutlicht. Dies hat einen entscheidenden Vorteil: Wenn Entwickler im Code eine Referenz auf eine Datenstruktur halten und diese als Parameter in eine Funktion geben, können sie sich sicher sein, dass die Datenstruktur niemals von der Funktion manipuliert werden kann. Das gilt auch für verschachtelte Datenstrukturen – etwa eine Liste in einer Map in einem Vektor

```
(def numbers (range 5))

(map inc numbers)
=> (1 2 3 4 5)

numbers
=> (0 1 2 3 4)
```

Listing 4: Datenstrukturen in Clojure sind unveränderlich

– und unterscheidet sich fundamental vom bekannten Verhalten in Java. In Java kann durch die Benutzung des `keyword final` und durch unveränderliche Collections wie ImmutableList zwar auf der obersten Ebene sichergestellt werden, dass Datenstrukturen nicht verändert werden sollen, dies bezieht sich jedoch nicht auf weitere, verschachtelte Referenzen.

Die fest eingebauten unveränderlichen Datenstrukturen machen Clojure zu einer enorm robusten Sprache. Selbst wenn Funktionen mit denselben Parametern in unterschiedlichen Threads parallel ausgeführt werden, können keine Race Conditions auftreten, solange die Funktionen keine sonstigen Seiteneffekte auslösen. Diesen Vorteil gegenüber Referenzen auf veränderliche Datenstrukturen stellte Rich Hickey, der Kopf hinter Clojure, in seinem Talk „Clojure Made Simple" sehr ausführlich und unterhaltsam dar *[2]*.

Destructuring

Da Clojure, wie bereits angesprochen, nicht statisch typisiert ist, sind Entwickler nicht darauf angewiesen, dieselben Datenformate (sprich „Objekte") als Parameter von Funktionen wiederzuverwenden. Um zu vermeiden, dass beim Aufruf von Funktionen kontinuierlich manuelle Datentransformationen vorgenommen werden müssen, hilft uns Destructuring.

In *Listing 5* ist Destructuring für Vektoren in der Funktion `format-winners` dargestellt. Auch wenn in der Definition der Funktionsparameter drei Werte aufgeführt sind, akzeptiert die Funktion nur einen Collection-Parameter, etwa eine Liste oder einen Vektor. Die ersten drei Elemente der übergebenen Datenstruktur werden den Variablen `gold`, `silber` und `bronze` zugewiesen und können unter diesen Namen im weiteren Code verwendet werden. Angezeigt wird dies durch die eckigen Klammern, die den Parameter umschließen. Die Zuordnung erfolgt hier nach Reihenfolge.

```
;; Destructuring für Vektoren
(defn format-winners [[gold silver bronze]]
  (format "Gold für %s, Silber für %s und Bronze für %s" gold silver bronze))

(format-winners ["Maria" "Max" "Marla" "Marlon" "Mareike"])
=> "Gold für Maria, Silber für Max und Bronze für Marla"

;; Desctructuring für Maps
(defn format-person [{:keys [firstname lastname city age]}]
  (format "%s %s aus %s, %s Jahre alt" firstname lastname city age))

(format-person {:firstname "Max"
                :lastname "Mustermann"
                :city "Berlin"
                :age 30
                :height 183})
=> "Max Mustermann aus Berlin, 30 Jahre alt"
```

Listing 5: Beispiele für Destructuring in Clojure

In unserem Beispielfall enthält der übergebene Vektor fünf Werte, die letzten werden ignoriert. Dies hat den positiven Nebeneffekt, dass die Signatur der Funktion konstant bleibt, auch wenn sich die Funktion dahin ändern würde, dass die ersten fünf Plätze angezeigt werden sollen. Natürlich könnte man die Platzierungen auch manuell aus der Rangliste extrahieren, dieser manuelle Aufwand entfällt aber durch die kürzere Schreibweise mit Destructuring.

Die Funktion `format-person` zeigt, wie assoziatives Destructuring, also bei Maps, aussehen kann. Der Parameter wird zunächst in geschweifte Klammern „verpackt". Dem Keyword `keys` folgt eine Liste von Variablennamen, die den Schlüsseln der übergebenen Map entsprechen. Diese Funktion wird also mit sämtlichen Maps als Parameter funktionieren, die die Schlüssel `:firstname`, `:lastname`, `:city` und `:age` enthalten. Weitere Schlüssel werden ignoriert.

Dies hat zur Folge, dass die Funktion in verschiedenen Domänen genutzt werden kann, da die übergebenen Werte nur diese Schlüssel als gemeinsamen Nenner benötigen, ohne dass sie etwa durch ein gemeinsames Interface aneinandergekoppelt sind. Auch hier ist ein manuelles Mappen der Parameter nicht nötig, der Code bleibt kompakt und von Zeremonie befreit.

Interaktion mit Java

Viele Sprachen, die eher eine Nische ausfüllen und keine flächendeckende Verbreitung in der Industrie erfahren, können nur auf ein eingeschränktes Ökosystem zurückgreifen, etwa OCaml oder Erlang. Da Clojure auf der JVM ausgeführt wird, können sich Clojure-Entwickler sämtlicher Funktionen und Bibliotheken aus dem Java-Umfeld bedienen – sei es die Integration von Jetty oder Undertow als HTTP-Server, Umgang mit den Standard-Datentypen wie String, BigDecimal oder LocalDateTime oder der Einsatz komplexer Bibliotheken wie zum Beispiel Testcontainers für Integrationstests.

Wie sich das Arbeiten mit Java-Objekten und -Klassen aus Clojure heraus gestaltet, zeigt sich in *Listing 6*. Bei Aufrufen von statischen Methoden werden der Klassenname und der Methodenname mit einem Schrägstrich getrennt als erste Form in einem Ausdruck angegeben. Eventuelle Parameter folgen, wie bei normalen Clojure Expressions. Möchte man ein neues Objekt erzeugen, kann der Konstruktoraufruf entweder über die Funktion `new` mit dem Typ als erstem Parameter erfolgen oder über die Kurzschreibweise – dem Klassennamen gefolgt von einem Punkt. Bei Methodenaufrufen an Java-Objekten wird der Methodenname mit einem vorangestellten Punkt als Funktion übergeben, das betroffene Objekt als erster Parameter. Die restlichen Parameter sind die der Java-Methode. Mit diesen Hilfsmitteln lässt sich recht flüssig mit Java-Typen und -Bibliotheken interagieren – auch, wenn die unterschiedlichen Paradigmen der Sprache aufeinanderstoßen und Java-Objekte, die in Clojure-Datenstrukturen verwendet werden, selbstverständlich nicht automatisch zu unveränderlichen Datentypen werden.

Abschluss

Unter den alternativen JVM-Sprachen sticht Clojure nicht nur optisch heraus. Es bringt ein für Java-Entwickler komplett neues Paradigma und andere Herangehensweisen mit. Viele Entwickler schätzen an Clojure die reduzierte Syntax, die datennahe Arbeit, den Fokus auf unveränderliche Datenstrukturen und das Schreiben von Funktionen, die möglichst wenig Seiteneffekte verursachen.

```
;; Zugriff auf statische Methode
(java.time.LocalDateTime/of 2020 12 01 23 59)

;; Konstruktoraufruf mit Parameter
(new String "Hello")

;; Kurzschreibweise für Konstruktoraufruf
(String. "Hello")

;; Methodenaufruf mit dot-Syntax und Parameter
(def now (java.time.LocalDateTime/now))
(.minusDays now 10)
```

Listing 6: Interaktion mit Java-Klassen und -Objekten

Tatsächlich lassen sich diese Eigenschaften und Paradigmen auch auf Java-Code übertragen, nachdem man sich einmal an sie gewöhnt hat. Nach einer kurzen Eingewöhnungsphase berichten viele Entwickler davon, dass sie mit Clojure mit vergleichsweise wenig Code schnell zu Ergebnissen kommen (und manche davon, wie schwer es ihnen fällt, wieder zu Java zurück zu wechseln). Wer sich für die Sprache interessiert und einen Einstieg sucht, kann sich auf eine überaus freundliche und inklusive Community freuen, die beim Einstieg gerne helfen wird.

Quellen

[1] *https://youtu.be/ZyTH8uCzil4?t=2896*

[2] *https://www.youtube.com/watch?v=VSdnJDO-xdg*

Tim Zöller

lambdaschmiede GmbH

tim.zoeller@lambdaschmiede.com

Tim Zöller ist Gründer und Berater bei der Firma lambdaschmiede GmbH in Freiburg im Breisgau. Er entwickelt seit 12 Jahren Software für die JVM, davon die meiste Zeit in Java. Er beschäftigt sich intensiv mit Prozessautomatisierung, Architektur, funktionaler Programmierung und insbesondere Clojure. Er hat die Java Usergroup Mainz mitgegründet und ist als Sprecher auf Java- und Clojure-Konferenzen anzutreffen.

Concurrency und State in Clojure

Nicolai Mainiero, sidion GmbH

Bei funktionalen Programmiersprachen wird immer wieder hervorgehoben, dass sich nebenläufige (concurrent) Programme einfacher realisieren lassen. In Clojure wurde neben den effizienten unveränderlichen (immutable) Datenstrukturen transaktionaler Speicher (software transactional memory) implementiert, um den Zustand zu kontrollieren. Damit hat der Programmierer ein mächtiges Werkzeug, um koordiniert und synchron den Zustand der Anwendung zu ändern.

Wenn in einer Anwendung Nebenläufigkeit verwendet wird, ist es immer aufwendig, den Zustand während der Ausführung zu kontrollieren, um Fehler wie race conditions oder lost updates zu verhindern. Wenn die Anwendung auch noch auf einem System mit mehreren Prozessorkernen ausgeführt wird, vergrößert sich die Wahrscheinlichkeit, dass bei einer fehlerhaften Implementierung subtile Fehler auftreten, die schwer bis unmöglich zu debuggen sind. Threads, die Low-Level-Schnittstelle, um nebenläufige Programme zu modellieren, werden auch in Clojure genutzt, um nebenläufige Programme zu schreiben. Die Integration geht sogar so weit, dass jede Funktion das `java.util.concurrent.Callable`-Interface implementiert. Dadurch kann jede Clojure-Funktion mit der High-Level-Abstraktion von Java, den Executors, zusammen verwendet werden.

Identitäten und Werte

Clojure unterscheidet klar zwischen Identitäten (identity) und Werten (value), zum Beispiel könnte die deutsche Nationalmannschaft jede Spielerin in ihrem Kader tauschen und würde dennoch die deutsche Nationalmannschaft bleiben. Die Identität bleibt unverändert, ihr Wert ändert sich jedoch im Laufe der Zeit. Das bedeutet, wenn wir eine Identität – die deutsche Nationalmannschaft – ändern, definieren wir einen neuen Wert für einen bestimmten Zeitpunkt beziehungsweise Zeitraum. Es ist aber möglich, zum Kader von 2019 zurückzukehren, um herauszufinden, wer damals der Nationalmannschaft angehört hat.

In Clojure sind alle Datenstrukturen unveränderlich und persistent *[3]*. Dadurch kann man sie wie Werte behandeln und das macht sie automatisch Thread-safe, das bedeutet, sie können von verschiedenen Threads bearbeitet werden, ohne sich gegenseitig zu stören.

Für Situationen, in denen dieses Modell nicht geeignet ist, bietet Clojure folgende verschiedene Referenztypen (Identitäten):

- **Atoms** für unkoordinierte synchrone Änderungen an geteiltem Zustand
- **Refs** für koordinierte synchrone Änderungen an geteiltem Zustand
- **Agents** für asynchrone Änderungen an geteiltem Zustand
- **Vars**, um Thread-local Zustand zu handhaben

Diese Unterscheidung zwischen Wert und Identität wird in anderen Sprachen oft vermischt. Das erfordert dann von der Programmiererin, dass sie beim Schreiben von nebenläufigem Code besonders aufpassen muss, um keine Fehler zu machen. Diese Sprachen greifen oft auf Locks, Mutexe, Semaphoren oder das defensive Kopieren zurück, um eine gleichzeitige Veränderung einer Variablen zu verhindern. Das kann allerdings mit zunehmender Komplexität der Anwendung schnell zu Problemen wie Verklemmungen (deadlocks) oder race conditions führen. Clojure vereinfacht dies, da der überwiegende Teil der Codebasis funktional ist beziehungsweise frei von Seiteneffekten realisiert werden kann und die wenigen Teile, die von der Veränderbarkeit profitieren, klar ersichtlich sind, da sie einen der vier Referenztypen verwenden.

```
(def current-series (atom "Lucifer")) ;1
-> #'user/current-series
(deref current-series) ;2
-> "Lucifer"
@current-series ;3
-> "Lucifer"
(reset! current-series "The Expanse") ;4
-> "The Expanse"
```

Listing 1: Erzeugen und Ändern eines Atoms

Unkoordinierte synchrone Änderungen mit Atomen

Atome bieten den einfachsten Mechanismus, um Zugriff auf eine Datenstruktur zu steuern. Ein Atom wird durch folgende Funktion definiert: `(atom initial-state options?)`. In *Listing 1* sieht man, wie ein Atom erzeugt, gelesen (dereferenziert) und geändert werden kann.

In Zeile 1 wird unter dem Namen `current-series` ein neues Atom mit dem Wert „Lucifer" erzeugt. Ein Atom beinhaltet also immer einen Wert. Mit der `(deref ref)`- Funktion beziehungsweise mit dem @ Reader-Makro können das Atom dereferenziert und der aktuelle Wert gelesen werden. Mit `(reset! atom newval)` in Zeile 4 kann dem Atom ein neuer Wert zugewiesen werden. Dabei wird der alte Wert unkontrolliert überschrieben. Mithilfe der `(swap! atom f)`-Funktion wird der aktuelle Wert des Atoms durch die übergebene Funktion `f` aktualisiert. Das hat den Vorteil, dass die gewünschte Änderung auch mit dem tatsächlichen Wert des Atoms durchgeführt wird, auch wenn mehrere Threads gleichzeitig den Wert ändern wollen. Die Funktion `f` sollte keine Seiteneffekte haben, da sie eventuell mehrfach aufgerufen wird, bis die Aktualisierung erfolgreich war.

In Atomen kann man auch komplexere Datenstrukturen, wie zum Beispiel eine Map, speichern. Dadurch ist es möglich, zusammenhängende Daten atomar zu ändern, wie es in *Listing 2* demonstriert wird. Hier wurden nun der Titel der Serie und die Streaming-Plattform in einer Map zusammengefasst. Wenn man nun die Serie wechselt, so kann dies mithilfe von `swap!` und einer entsprechenden Funktion (`(assoc map key val)`) realisiert werden.

Refs und *transaktionaler Speicher (STM)*

Atome steuern den synchronisierten unkoordinierten Zugriff, es ist also nicht möglich, Änderungen über mehrere Atome koordiniert durchzuführen. Um dies zu ermöglichen, wurde in Clojure transaktionaler Speicher implementiert. Dazu wird eine Ref analog zu einem Atom erstellt, wie es in *Listing 3* zu sehen ist. Die Erzeugung und der Zugriff erfolgen auf Refs genauso wie auf Atome.

Das Ändern einer Ref unterscheidet sich allerdings von dem eines Atoms, wie in *Listing 4* zu sehen ist. Das STM sorgt dafür, dass eine Ref nicht ohne umschließende Transaktion geändert werden kann. Diese Transaktion kann mit der `(dosync & exps)`-Funktion erzeugt werden.

```
(def current-series (atom {:title "Lucifer" :platform "Prime"}))
(swap! current-series assoc :title "Star Trek: Lower Decks")
-> {:title " Star Trek: Lower Decks ", :platform "Prime"}
```

Listing 2: Ändern einer komplexen Datenstruktur

```
(def current-series (ref "The Expanse"))
-> #'user/current-series
(deref current-series)
-> "The Expanse"
@current-series  ; Alternative zur deref Funktion
-> "The Expanse"
```

Listing 3: Deklaration und Dereferenzierung einer Ref

Innerhalb des `dosync`-Blocks kann `ref-set` beliebig oft aufgerufen werden. Clojure garantiert, dass für solche Transaktionen folgende Eigenschaften gelten:

- atomar – wenn mehr als eine Ref in einer Transaktion geändert wird, wird die Änderung für einen Außenstehenden bei allen Refs gleichzeitig sichtbar
- konsistent – Refs können Validierungsfunktionen definieren, die, wenn eine davon fehlschlägt, dazu führen, dass die ganze Transaktion fehlschlägt
- isoliert – Transaktionen sehen keine Teilergebnisse von anderen Transaktionen

Da die Transaktionen bei Clojure im Speicher ablaufen, wird keine Dauerhaftigkeit wie bei einer Datenbank garantiert. *Listing 5* zeigt, wie mehrere Refs in einer Transaktion koordiniert geändert werden können.

In dem Beispiel wird der bestehende Wert einfach überschrieben, wie wir es auch schon bei den Atomen gesehen haben. Das nächste Beispiel in *Listing 6* beschreibt eine vereinfachte Chatanwendung, die aber ausreichend komplex ist, um zwei weitere Funktionen zum Aktualisieren von Refs zu demonstrieren.

Zunächst wird ein Record *[1]*, ein eigener Datentyp, der sich wie eine Map verhält, mit den Feldern `:sender` und `:text` definiert. Um alle bereits gesendeten Nachrichten zu speichern, wird eine Liste verwendet (Zeile 3). Das erste Element in dieser Liste ist dadurch automatisch die zuletzt gesendete Nachricht. Zeile 4 enthält eine naive Implementierung der `add-message`-Funktion. Erst wird die Ref dereferenziert, anschließend die neue Nachricht vorangestellt, um dann mit `ref-set` die Ref zu aktualisieren. Das ist sehr umständlich und nicht besonders funktional.

```
(ref-set current-series "Lucifer")
-> java.lang.IllegalStateException: No transaction running
(dosync (ref-set current-series "Lucifer"))
-> "Lucifer"
```

Listing 4: Änderung des Werts einer Ref ohne Transaktion

Wie bei den Atomen gibt es auch für Refs eine Funktion, um eine Datenstruktur innerhalb einer Ref zu ändern. Diese nennt sich passenderweise `(alter ref update-fn & args...)`. Dieser Funktion übergibt man eine Ref, eine Update-Funktion und gegebenenfalls Argumente für die Update-Funktion. In den meisten Fällen wird die `alter`-Funktion zum Ändern von Refs verwendet. In einigen Spezialfällen, wenn die Update-Funktion kommutativ (zum Beispiel a + b = b + a) ist, kann auf eine optimierte Funktion `(commute ref update-fn & args...)` zurückgegriffen werden. Diese erlaubt es Clojure, die Reihenfolge der Änderungen zu verändern und damit schneller auszuführen.

Um die Konsistenz zu gewährleisten, kann bei der Deklaration einer Ref eine Validierungsfunktion angegeben werden. *Listing 7* erweitert die Chatanwendung und fügt solch eine Validierungsfunktion hinzu, damit sichergestellt ist, dass jede Nachricht gültige Werte für `:sender` und `:text` hat.

Agents für asynchrone Änderungen

Für den Fall, dass eine Änderung am Zustand der Anwendung aufwendig ist oder auch erst später erfolgen kann, stellt Clojure mit Agents eine asynchrone Lösung bereit. Die Verwendung von Agents unterscheidet sich nicht groß von Atomen oder Refs. *Listing 8* zeigt, wie ein Agent erzeugt wird, wie er geändert wird und wie das Ergebnis abgeholt werden kann.

```
(def current-series (ref "Lucifer"))
-> #'user/current-series
(def current-stream-provider (ref "Prime"))
-> #'user/current-stream-provider
(dosync
  (ref-set current-series "Das Damengambit")
  (ref-set current-stream-provider "Netflix"))
-> "Netflix"
```

Listing 5: Koordinierte Änderungen mehrerer Refs

```
(defrecord Message [sender text]) ;1
-> user.Message
(->Message "Murtaugh" Ich bin zu alt für diesen Mist.") ;2
-> #user.Message{:sender "Murtaugh", :text "Ich bin zu alt für diesen Mist."}
(def messages (ref ())) ;3
-> #'user/messages
; bad idea
(defn naive-add-message [msg]
    (dosync (ref-set messages (cons msg @messages)))) ;4
-> #'user/naive-add-message
(defn add-message [msg]
    (dosync (alter messages conj msg))) ;5
-> #'user/add-message
(add-message (->Message "user 1" "hello")) ;6
-> (#user.Message{:sender "user 1", :text "hello"})
(add-message (->Message "user 2" "howdy")) ;7
-> (#user.Message{:sender "user 2", :text "howdy"}
    #user.Message{:sender "user 1", :text "hello"})
```

Listing 6: Einfache Chatanwendung

```
(defn valid-message? [msg]
(and (:sender msg) (:text msg)))
-> #'user/valid-message?
(def validate-message-list #(every? valid-message? %))
-> #'user/validate-message-list
(def messages (ref () :validator validate-message-list))
-> #'user/messages
(add-message "not a valid message")
-> java.lang.IllegalStateException: Invalid reference state
(add-message (->Message "Riggs" "Du bist der beste Cop den ich kenne."))
-> (#:user.Message{:sender " Riggs ", :text " Du bist der beste Cop den ich kenne."})
```

Listing 7: Validierung bei Ref-Änderungen

```
(def counter (agent 0))
-> #'user/counter
(send counter inc)
-> #object[clojure.lang.Agent 0x7be3c47d {:status :ready, :val 1}]
@counter
-> 1
```

Listing 8: Asynchrone Änderungen mit Agents

Der Funktionsaufruf `send` liefert im Gegensatz zu `swap!` oder `alter` nicht den aktualisierten Wert zurück, sondern eine Referenz auf den Agent. Die Dereferenzierung erfolgt jedoch genauso wie bei den Refs oder Atoms mit `deref` oder dem @ Reader-Makro.

Auch bei Agents kann, analog zu den Refs, eine Validierungsfunktion angegeben werden. Im Unterschied zu Refs, die synchron verarbeitet werden und bei denen ein Fehler bei der Validierung sofort bemerkt wird, muss bei der Verwendung von Agents überprüft werden, ob die Änderung erfolgreich war. *Listing 9* zeigt, wie man dabei vorgehen kann.

In Zeile 1 erzeugen wir einen Agent mit einer Validierungsfunktion, die prüft, ob der neue Wert, den wir im Agent speichern wollen, auch eine Zahl ist. Dann senden wir eine Updatefunktion an den Agent, die immer einen String zurückliefert, die Validierung also absichtlich verletzt (Zeile 2). Wenn diese Funktion schlussendlich ausgeführt wird, führt es zu einer Ausnahme im Agent und je nach Fehlermodus `:fail` oder `:continue` verhält sich der Agent unterschiedlich. Wurde beim Erzeugen des Agent kein `:error-handler` angegeben, befindet er sich im `:fail`-Modus. Jede Ausnahme führt dazu, dass der Agent in einen Ausnahmezustand wechselt. Diesen Zustand kann der Agent nur verlassen, wenn er neu gestartet wird (Zeile 4).

```
(def counter (agent 0 :validator number?)) ;1
-> #'user/counter (send counter inc)
(send counter (fn [_] "boo")) ;2
-> #object[clojure.lang.Agent 0x9ee5b57c {:status :ready, :val 0}]
(agent-error counter) ;3
-> #error {
 :cause "Invalid reference state"
 :via
 [{:type java.lang.IllegalStateException
   :message "Invalid reference state"
   :at [clojure.lang.ARef validate "ARef.java" 33]}]
 :trace
 […]}
(restart-agent counter 0) ;4
-> 0
@counter
-> 0
(defn handler [agent err]
  (println "ERR!" (.getMessage err))) ;5
-> #'user/handler
(def counter2
  (agent 0 :validator number? :error-handler handler)) ;6
-> #'user/counter2
(send counter2 (fn [_] "boo")) ;7
-> #object[clojure.lang.Agent 0xbae653d9 {:status :ready, :val 0}]
user=> ERR! Invalid reference state
(send counter2 inc) ;8
-> #object[clojure.lang.Agent 0xbae653d9 {:status :ready, :val 0}]
@counter2 ; 9
-> 1
```

Listing 9: Validierung bei Agents

```
(def backup-agent (agent "output/messages-backup.clj")) ;1
-> #'user/backup-agent
(defn add-message-with-backup [msg]
  (dosync
    (let [snapshot (commute messages conj msg)]
    (send-off backup-agent (fn [filename]
                             (spit filename snapshot)
                              filename))
snapshot))) ;2
-> #'user/add-message-with-backup
(add-message-with-backup (->Message "John" "Message One")) ;3
-> (#user.Message{:sender "John", :text "Message One"})
```

Listing 10: Kombination von Transaktionen und Agents

Wurde beim Erstellen des Agent ein `:error-handler` angegeben, befindet sich der Agent im `:continue`-Modus. In diesem Modus wird im Fehlerfall der übergebene `error-handler` aufgerufen (Zeile 7). Der Agent bleibt aber ganz normal benutzbar (Zeile 8 und 9).

Agents und Transaktionen

Die Funktionen zum Ändern von Atomen oder Refs sollten frei von Seiteneffekten sein, da Clojure eventuell versucht, die Aktualisierungsfunktion mehrfach aufzurufen, bis die Aktualisierung erfolgreich war. Allerdings kann es vorkommen, dass bei einer erfolgreichen Transaktion ein Seiteneffekt ausgelöst werden soll. Hier kann auf Agents zurückgegriffen werden. Wenn einem Agent innerhalb einer Transaktion eine Anweisung geschickt wird, wird diese genau einmal und auch nur bei Erfolg der Transaktion ausgeführt. Ergänzen wir die Chatanwendung um ein Backup, das die Nachrichten in eine Datei schreibt *(siehe Listing 10)*.

Die neue Funktion `add-message-with-backup` in Zeile 2 erweitert die alte Funktion und speichert zunächst die aktualisierte Liste der Nachrichten in der Variablen `snapshot`, um dann mit `send-off` an den Agent die Anweisung zur Aktualisierung der Datei zu senden. Im Unterschied zu `send`, das wir bereits kennengelernt haben, wird hier `send-off` verwendet. Diese Funktion wird immer dann verwendet, wenn es sich um einen blockierenden Vorgang, wie das Schreiben in einer Datei, handelt. Es ist verlockend, diese Konstruktion als Datenbank zu verwenden, Clojure STM liefert ACI und das Schreiben in die Datei trägt das D von ACID bei. Allerdings garantiert Clojure nicht, dass der Agent innerhalb der ACI-Transaktion ausgeführt wird. Das bedeutet, die Transaktion kann erfolgreich gewesen sein und der Agent versucht nun, auf das Dateisystem zuzugreifen. Wenn das jetzt fehlschlägt, geht die Anwendung davon aus, dass die Daten persistiert worden sind, auch wenn das gar nicht der Fall ist.

Einheitliches Aktualisierungs-Modell

Clojure legt sehr viel Wert auf konsistente APIs. Das bemerkt man auch beim Umgang mit Atomen, Refs und Agents. Sie werden alle ähnlich erzeugt und gelesen und können auf die gleiche Art und Weise geändert werden. Hier nochmals zusammengefasst, welche Methode wofür verwendet wird *(siehe Tabelle 1)*.

Fazit

Clojure bietet sehr flexible und ausgefeilte Möglichkeiten, um den Zustand einer Anwendung über mehrere Threads hinweg zu managen. Besonders hervorzuheben ist das einheitliche Modell, um mit Atomen, Refs oder Agents zu interagieren. Durch die unveränderlichen Datenstrukturen und STM ermöglicht es, Aktualisierungen über Threads hinweg zu koordinieren. Und wenn die Threads unabhängig voneinander sind, muss man sich keine Gedanken machen, ob ein Dritter die eigenen Daten gerade geändert hat.

	Atom	Ref	Agent
Änderungsfunktion	`alter`	`swap!`	`send-off`
Änderungsfunktion (kommutativ)	`commute`	-	-
Änderungsfunktion (non-blocking)	-	-	`send`
Setter	`reset!`	`ref-set`	-

Tabelle 1

Quellen

[1] Alex Miller (2018): Programming Clojure. The Pragmatic Bookshelf, Raleigh, North Carolina

[2] Michael Sperber (2015): Zusammengesetzte Daten in Clojure, *https://funktionale-programmierung.de/2015/04/27/clojure-records.html*

[3] Clojure Data Structures, *https://clojure.org/reference/data_structures*

Nicolai Mainiero
sidion GmbH
nicolai.mainiero@sidion.de

Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der sidion GmbH. Er entwickelt seit über 13 Jahren Geschäftsanwendungen in Java, Kotlin und Clojure für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen.

Java und Clojure: Das Beste aus zwei Welten in einer VM

Manuel Herzog und Dominik Galler, esentri AG

Entwickelt man große Anwendungen mit Java in der JVM, die beispielsweise große Mengen an Objekten im Speicher halten müssen, so wird man hin und wieder unliebsame Speicherkapazitätsengpässe feststellen. In diesem Artikel soll nachvollzogen werden, wie ein solcher Fall konstruiert werden kann. Dazu wird ein anschauliches Szenario anhand von großen Excel-Dateien aufgebaut und anschließend eine mögliche Lösung des Speicherengpasses durch Auslagern der entsprechenden Funktionalitäten nach Clojure erörtert. Zuletzt wird beschrieben, wie durch die Interoperabilität zwischen Clojure und Java die erarbeitete Lösung wieder zurück in das ursprüngliche Programm fließen kann. Um den bereitgestellten Code auszuführen, sind ein Git-Client, Maven 3.6, Java 11 sowie Leiningen 2.9.5 [1] nötig.

Das Ausgangsszenario

Stellen Sie sich vor, Sie sind ein Bürofachhändler. Ihr größter Kunde – ein Konzern – soll einmal jährlich zu Controlling-Zwecken von Ihnen Excel-Dateien bekommen, die die Bestellungen zusammenfassen. Um die Bestellungen für die einzelnen Abteilungen des Konzerns so einfach wie möglich zu gestalten, können diese direkt bei Ihnen bestellen und tragen ihre Bestellung dann in eine Excel-Datei des Konzerns ein. Diese große Excel-Datei geht am Jahresende bei Ihnen – als Bürofachhändler – ein und Sie müssen diese nun für die Controlling-Auswertung anreichern. Sie müssen also die Excel-Datei mit 200.000 bis 300.000 Zeilen einlesen und auswerten. Einen beispielhaften Auszug finden Sie in *Tabelle 1*.

Wie Sie erkennen können, haben die Tochterfirmen des Konzerns die Einträge einfach fortlaufend aufgelistet. Dabei werden die Tochterfirma des Konzerns, die Abteilung, der Artikeltyp und dessen Menge angegeben. Auf dieser Basis wollen Sie nun die Auswertung berechnen. Zunächst müssen Sie dazu jede Bestellung um den Einzelstückpreis sowie den Gesamtpreis pro Bestellung anreichern und als Vergleichsdatei ablegen. Für das Beispiel aus *Tabelle 1* ergibt sich also die unter *Tabelle 2* aufgelistete Auswertungs-Excel-Datei.

Auf dieser Basis aggregieren Sie die Tabelleneinträge nach Tochterfirma und Artikel, um die Gesamtkosten des Jahres nach Tochterfirma und Artikel aufzuführen und eine Gesamtmenge neben einer Gesamtsumme für die Artikelbestellungen aufzuführen. Ein Ausschnitt ist in *Tabelle 3* gegeben.

Dieser Service wird von Ihrem Unternehmen angeboten, um die Arbeit mit dem Konzern für diesen so leicht wie möglich zu gestalten. Außerdem haben Sie festgestellt, dass die Abteilungen mehr bestellen, wenn die Kosten nur einmal im Jahr angezeigt werden, mit allen daraus folgenden Implikationen.

Um in diesem schlanken und fiktiven Szenario nun mit relativ wenig Aufwand an die Excel-Datei zu kommen, haben wir Ihnen einen Excel-Generator zur Seite gestellt. Sie können ihn im GitHub Repository *[2]* finden. Dazu können Sie das Repository mit Git auschecken und mit dem Befehl `mvn clean package` eine Jar-Datei erstellen. Führen Sie diese nun ohne Argumente mit `java -jar target/order-excel-generator-1.0-jar-with-dependencies.jar` aus und es wird eine Datei „order-example.xlsx" im selben Verzeichnis für Sie erstellt. Mit dieser Datei können Sie nun die Generierung der Controlling-Dateien angehen.

Firma	Abteilung	Artikel	Menge
...	...	...	...
Donton	IT-Abteilung	Stift	8
Tecicon	Produktmanagement	Stift	8
TumTum	Controlling	Toner	2
Bernd	Personal	Stift	7
...	...	...	...

Tabelle 1

Firma	Abteilung	Artikel	Menge	Einzelpreis	Gesamtpreis
...	...	...	...	...	...
Donton	IT-Abteilung	Stift	8	3,92	31,36
Tecicon	Produktmanagement	Stift	8	3,92	31,36
TumTum	Controlling	Toner	2	134,79	134,79
Bernd	Personal	Stift	7	3,92	27,44
...	...	...	...	...	...

Tabelle 2

Firma	Artikel	Gesamtmenge	Gesamtpreis
...	...	...	...
Donton	Büroklammern	35.542	44.072,08
Donton	Stift	36.520	143.158,40
Donton	Druckerpapier	338.287	1.532.440,00
...	...	...	...

Tabelle 3

Arbeiten mit Excel unter Java

Für das Arbeiten mit Excel ist in Java Apache POI *[3]* eine weit verbreitete und gute Option. Mit Apache POI lassen sich einfach Excel-Dateien einlesen und bearbeiten. Dieser Artikel soll nicht zum Fokus haben, Apache POI vollumfänglich zu erläutern. Hierfür sei auf den wirklich sehr guten und hilfreichen Quick-Guide für und von Apache POI *[4]* als erste Anlaufstelle verwiesen.

Um später eine Nachvollziehbarkeit des Codes zu ermöglichen und auch den groben Ablauf zu schildern, soll *Abbildung 1* dienen. Sie zeigt, wie das Programm zur Erstellung der Excel-Übersichtsdatei ausgeführt wird. Der Prozess ist linear und wenig kompliziert. Die Excel-Datei soll von dem Programm eingelesen werden und anschließend werden die Zeilen der Excel-Datei zu entsprechenden Objekten im Programm gelesen. Diese werden um den Einzelpreis angereichert. Um auch hier die Komplexität zu reduzieren, sind die Preise fest im Programmcode hinterlegt. In einer realen Anwendung würden Sie diese Preise vermutlich eher aus einem anderen System, etwa einer Datenbank oder einem ERP-System, beziehen. Mit diesen Informationen kann letztendlich auch die Zusammenfassung erzeugt werden. Schließlich wird die Excel-Datei mit den beiden Arbeitsblättern für die Zusammenfassung und die Übersicht in das Dateisystem geschrieben.

Den Java-Code für das Einlesen der Dateien finden Sie unter folgendem GitHub Repository *[6]*. Sie können das Programm wieder mit Git auschecken und mit folgendem Maven-Befehl eine ausführbare Jar erzeugen: `mvn clean package`. Diese Jar können Sie nun mit einem Dateipfad (beispielsweise zu der oben generierten „order-example.xlsx") ausführen: `java -jar target/order-excel-reader-java-1.0-jar-with-dependencies.jar -f <PFAD_ZUR_DATEI>/order_overview.xlsx`. Nach erfolgreicher Ausführung wird eine Log-Ausgabe mit der Laufzeit in Nanosekunden erzeugt. Daneben finden Sie die bereits beschriebene Excel-Datei mit den zwei Arbeitsblättern für das Controlling.

Die Vorteile einer Ausführung mit Clojure

Will man große Dateien im Speicher halten oder generell viele Objekte mit Java kurz hintereinander bearbeiten, so stößt man mit Java schnell an seine Grenzen. Bei der Arbeit mit diesen Dateien muss Java alle Objekte für die Verarbeitung im Speicher halten und schließlich auch durch den Garbage Collector aufräumen. Die Speicherverwaltung unter Clojure funktioniert hingegen anders. Diese Funktionsweise kann man sich gerade beim Umgang mit großen Objektlisten zunutze machen. Durch die Immutabilität des Speichers können unter Clojure Objekte nach Verlassen des Scopes sofort wieder in die Verwendung zurückfließen, denn sie sind unveränderbar. Die Speicherverwaltung zwischen Objektsprachen und funktionalen Skriptsprachen lässt sich hier nicht vergleichen und ist bei letzteren häufig deutlich im Vorteil.

Da es sich bei Clojure um eine Java-VM-Sprache handelt, die eine überragende Interoperabilität zu Java vorweist und auch die Gegenrichtung für Java bereitstellt, eignet es sich durchaus, beide Sprachen zu kombinieren, um das Beste aus beiden Welten zu vereinen. Dies soll im Folgenden auch in beide Richtungen genutzt werden. Denn zunächst soll im Clojure-Code auch nicht sämtliche Arbeit zum Einlesen einer Excel-Tabelle per Hand gemacht werden, sondern ebenfalls die Java-Bibliothek Apache POI verwendet werden. Den Clojure-Code finden Sie ebenfalls in einem Repository auf GitHub *[7]*. Nachdem Sie das Repository ausgecheckt haben, können Sie mit `lein run` den Code ausführen. Achten Sie darauf, dass Sie hierfür zuvor Leinigen *[1]* installiert haben müssen. `lein run` führt einen JIT(„Just-in-Time")-Compiler aus. Dieser wandelt den Clojure-Code direkt in Bytecode um und führt ihn aus. Durch den JIT-Compiler ist der Prozess der Ausführung hier deutlich langsamer, als wenn Sie eine kompilierte, ausführbare Datei erzeugen. Diese können Sie mit `lein uberjar` generieren. In der Jar-Datei sind alle Abhängigkeiten enthalten und Sie können die Datei mittels `java -jar target/uberjar/poi-test-0.1.0-SNAPSHOT-standalone.jar` ausführen.

Die Arbeitsweise des Clojure-Codes ist dabei ähnlich dem unter *Abbildung 1* aufgeführten Ablaufdiagramm für die Abarbeitung unter Java. Zunächst wird die Excel-Datei eingelesen und in einen POI XSSFReader (ein Reader-Objekt für xlsx-Dateien) umgewandelt. Dieser Reader wird anschließend um eine Handler-Funktion angereichert, die beim Einlesen der Excel-Zellen aufgerufen wird und die verschiedenen Excel-Datentypen behandelt. Die Daten aus jeder Zeile werden anschließend als vollständige Bestellung an einen Callback übergeben, der die Bestellung mit Preisen anreichert und bereits in eine neue Tabelle ausgibt. Zusätzlich werden die Bestellungen gesammelt und für die beiden Zusammenfassungen aufbereitet. Am Ende wird ebenfalls eine Excel-Datei erzeugt, die über zwei Arbeitsblätter verfügt mit den identischen Ergebnissen wie im Java-Code-Beispiel. Ebenfalls analog zum Java-Teil wurden die Preislisten fest im Code des Demo-Codes hinterlegt.

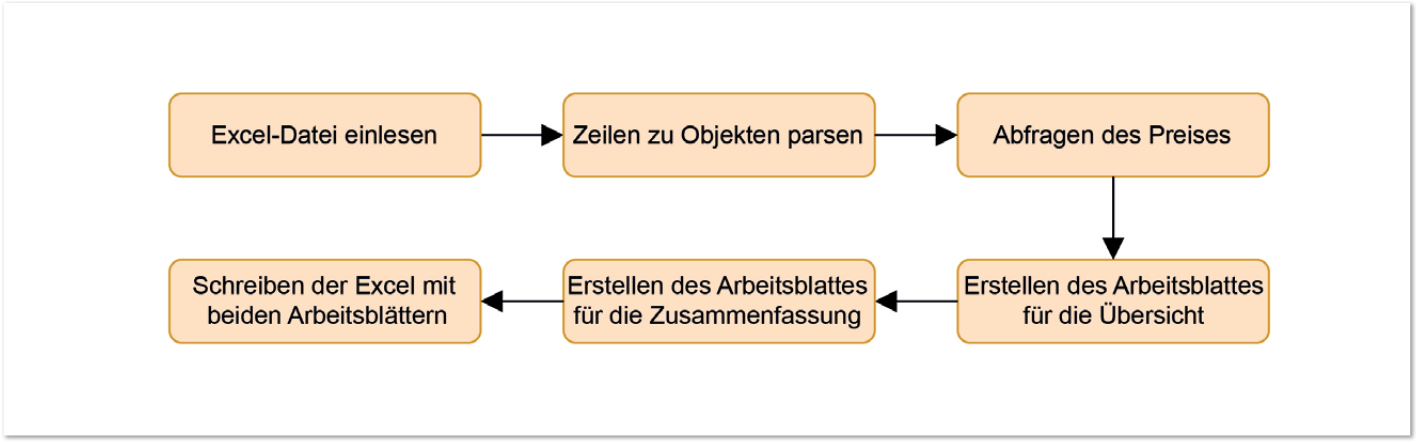


Abbildung 1: Programmfluss zum Einlesen und Generieren der Excel-Datei (© Dominik Galler)

Um innerhalb von Clojure auf die Java-Bibliothek Apache POI zuzugreifen, muss der Handler (vgl. *Listing 1*, Zeile 34-66) als Java-Objekt mit Vererbung erzeugt werden. Hierbei kann mittels Proxy *[8]* eine Implementierung zu Java-Klassen oder -Interfaces erfolgen (vgl. *Listing 1*, Zeile 42-66). Ein Proxy in Clojure ist vereinfacht mit einer anonymen Klasse in Java vergleichbar.

Der größte Vorteil in der Verarbeitung der Excel-Datei mit Clojure ist durch die Nebenläufigkeit gegeben. Dadurch, dass Variablen in Clojure unveränderbar sind, ist die Arbeit mit diesen auch frei von Seiteneffekten. Allerdings hat man in der Nebenläufigkeit keinen generischen Kontext beim Eventing. Um das Fehlen des generischen Kontextes zu kompensieren, werden sogenannte Atome *[9]* verwendet.

Diese Atome sind ein performanter Weg, um sich Datenobjekte in unterschiedlichen Threads zu teilen. Eine Änderung an Atomen ist immer frei von Race Conditions. In *Listing 1* werden in Zeile 36-38 drei Atome für die Informationen aus den Zeilen zwischengespeichert. Das Value-Atom aus Zeile 36 in *Listing 1* speichert den Wert zwischen zwei XML-Tags (Excel-Dateien im XLSX-Format liegt ja auch XML als Definitionsformat zugrunde). Hierzu wird `DefaultHandler.characters(...)` aufgerufen (vgl. *Listing 1*, Zeile 44). Das Value-Atom wird dabei ausgetauscht. Im Gegensatz zum – beispielsweise von Java – gewohnten Umgang mit global gültigen Variablen wird dabei aber nicht die Variable selbst geändert, sondern dem Atom wird eine Manipulationsfunktion übergeben, um die Anforderungen an seiteneffektfreie Nebenläufigkeit zu erfüllen *[9]*.

```
34  (defn sheet-handler
35    [^SharedStringsTable sst rowfn]
36    (let [value (atom "")
37          cell-info (atom nil)
38          row (atom {})
39          get-sst-fn (fn [_value] (.getString (.getItemAt       sst (Integer/parseInt _value))))
40          get-sst (memoize get-sst-fn)
41          ]
42  (proxy [DefaultHandler]
43        []
44        (characters [chs start length]
45          (swap! value str (String. chs start length)))
46        (startElement [^String uri ^String localName ^String name ^Attributes attributes]
47          (case name
48            "c"
49              (case (.getValue attributes "t")
50                "s" (reset! cell-info [:s (get-column-as-keyword (.getValue attributes "r"))])
51                "n" (reset! cell-info [:n (get-column-as-keyword (.getValue attributes "r"))])
52                nil)
53            "v" (reset! value nil)
54            nil
55          ))
56        (endElement [uri localName name]
57          (case name
58            "v" (let [[type position] @cell-info]
59                  (case type
60                    :s (swap! row assoc position (get-sst @value))
61                    :n (swap! row assoc position (Double/parseDouble @value))
62                    nil))
63            "row" (do (rowfn @row)
64                      (reset! row {}))
65            nil
66            )))))
```

Listing 1

```
1  (ns com.esentri.clojure.order.excel
2    (:gen-class
3      :name com.esentri.clojure.order.excel.reader
4      :prefix "cls-"
5      :main false
6      :methods [^:static [execute [String String] void]]))
```

Listing 2

```
126  (defn cls-execute
127    [^String filename ^String out-filename]
128    (load-wb filename out-filename)
129    nil)
```

Listing 3

Durch den Einsatz von Memoize *[10]* kann ein weiterer Performancegewinn bei der Arbeit mit großen Datenströmen erzielt werden. Vereinfacht ausgedrückt erweitert Memoize eine Funktion um einen Caching-Mechanismus (vgl. *Listing 1*, Zeile 40). Hier wird eine Funktion genutzt, die anhand einer ID den entsprechenden Wert aus der Excel Shared Strings Table ausliest, die im XML-Format vorliegt. Um hier die Umwandlung von String zu Integer und darauffolgend dem Auslesen des passenden Strings zu optimieren, werden die Ergebnisse gecacht.

Muss man jetzt ein ganzes Clojure-Programm schreiben?

Es ist nicht notwendig, die ganze Anwendung in Clojure zu schreiben. Die Interoperabilität kann, wie bereits erwähnt, auf beiden Seiten ausgenutzt werden. Mit dem `gen-class`-Parameter (vgl. *Listing 2*, Zeile 2-6) wird eine Java-Klasse mit Funktionen erzeugt. Da für den hier beschriebenen Anwendungsfall eine eigene Klasse nicht notwendig ist, kann eine statische Funktion im Java-Kontext verwendet werden. Dies wird dadurch ermöglicht, dass Clojure das Klassenkonzept von Java über Funktionen innerhalb des Namespace emuliert (vgl. *Listing 2*, Zeile 6).

Weiterhin ist es beim Einbinden von Clojure in Java gängige Praxis, eine eigene Funktion in Clojure zu schreiben, die über Java aufgerufen werden kann. Diese Funktion dient als Wrapper für die Clojure-Funktion `load-wb`. Diese Funktion wurde in Teilen bereits in *Listing 1* erläutert. Der Klassengenerierung wird dafür das Präfix `cls-` (vgl. *Listing 2*, Zeile 4) vorangestellt. Zusammen mit dem vergebenen Funktionsnamen `execute` (vgl. *Listing 2*, Zeile 6) ergibt sich die Funktion `cls-execute`, die schließlich als Wrapper definiert wird *(siehe Listing 3)*. Durch `nil` (*Listing 3*, Zeile 129) wird ein eventueller Rückgabewert ausgeschlossen, damit kein Parsing-Fehler auftreten kann, der bei Java für unerwartetes Verhalten sorgen könnte.

In der normalen Ausführung wird Clojure per JIT-Compiler in Bytecode übersetzt. Daher ist es zuletzt noch notwendig, den JIT-Compiler durch einen AOT(„Ahead Of Time")-Compiler zu ersetzen. Dadurch kann der Clojure-Code direkt und ohne weitere Hilfsmittel mit Java aufgerufen werden, da er dann direkt in Bytecode vorliegt *[11]*. Dies geschieht durch Aktivieren von AOT in der Datei project.clj, die als Pendant unter Leiningen zur POM unter Maven verstanden werden kann. Nun können Sie die vorbereitete Klasse durch `lein install` generieren und in Ihrem lokalen Maven-Repository speichern. Dies ist notwendig, da die so generierte Jar nun in dem Java-Projekt zum Lesen der Excel-Dateien eingebunden werden soll. Dafür wird zunächst die Jar in die POM des Projekts integriert. Im Branch „clojure-integration" des Repository für den Excel-Reader unter Java (den direkten Link finden Sie unter *[12]*) finden Sie bereits die entsprechenden Anpassungen. Die Hauptmethode der Applikation wurde dabei um einen zweiten Aufruf erweitert, um eine bessere Vergleichbarkeit der beiden Ausführungen – Java und Clojure – zu ermöglichen. Wie in *Listing 4* gezeigt, kann die vorher generierte Klasse aus Clojure wie eine herkömmliche Java-Klasse (in diesem Fall mit statischer Funktion) aufgerufen werden (*Listing 4*, Zeile 6 und 30).

```
6  import com.esentri.clojure.order.excel.reader;
7  public final class OrderExcelReaderApplication {
…
12     public static void main(final String... args) {
…
30         reader.execute(filePath, "output-clojure.xlsx");
…  }
```

Listing 4

Nun können Sie, wie bereits im Java-Teil dieses Artikels beschrieben, die von diesem Branch gebaute Jar ausführen und die Ausführungszeiten anhand der Log-Ausgaben auf Ihrem Computer vergleichen.

Fazit

Bei der Ausführung auf unseren Computern haben wir eine Laufzeit des Java-Programms von rund 22 Sekunden und 14 Sekunden des Clojure-Programms festgestellt. Trotz der eingeschränkten Funktionalität für dieses Szenario und der künstlich generierten Testdaten mit wenigen Produkten ist dies ein durchaus signifikanter Unterschied. Der Java-Code hat dabei rund 160 Prozent der Laufzeit des Clojure-Codes benötigt. Bei realen Daten und einer komplexeren Verarbeitung dürfte dieser Performance-Unterschied noch deutlich größer ausfallen. Weiterhin fällt bei deutlich mehr zu verarbeitenden Dateien auch schon dieser kleine Unterschied deutlicher ins Gewicht.

Neben dem Vorteil im Hinblick auf Performance konnte das vorgestellte Demoprojekt allerdings auch die hervorragende Integration von Java und Clojure aufzeigen. Durch diese nahtlose Integration ist es einfach möglich, die besser geeignete Sprache für ein gegebenes Problem zu wählen. Damit kann man selbst in fest definierten Java-VM-Umgebungen polyglotte Systeme konzipieren. Im Gegensatz zu anderen Java-VM-Sprachen kann einerseits der Clojure-Code problemlos aus Java aufgerufen werden, andererseits kann vorhandener Java-Code eines Projektes oder einer Bibliothek genauso in Clojure verwendet werden. Durch diese Eigenschaften lässt sich nach und nach eine Balance aus funktionaler und imperativer Programmierung finden. Dies kann eine erhöhte Entwicklungsgeschwindigkeit, bessere Problemlösung und mehr Flexibilität bei der Umsetzung im Entwicklerteam ermöglichen.

Quellen

[1] *https://leiningen.org/*, abgerufen am 11.02.2021
[2] *https://github.com/esentri/order-excel-generator*
[3] *https://poi.apache.org/*, abgerufen am 11.02.2021
[4] *https://poi.apache.org/components/spreadsheet/quick-guide.html*, abgerufen am 11.02.2021
[6] *https://github.com/esentri/order-excel-reader-java*
[7] *https://github.com/esentri/order-excel-reader-clojure*
[8] *https://clojuredocs.org/clojure.core/proxy*
[9] *https://clojure.org/reference/atoms*
[10] *https://clojuredocs.org/clojure.core/memoize*
[11] *https://clojure.org/reference/compilation*
[12] *https://github.com/esentri/order-excel-reader-java/tree/clojure-integration*

Manuel Herzog
esentri AG
manuel.herzog@esentri.com

Manuel Herzog ist als Architekt und Berater bei der esentri AG tätig. Das Kennenlernen neuer Technologien und deren Aufbereitung für den Enterprise-Einsatz bereiten ihm große Freude. Aufgrund der langjährigen Erfahrung in der Administration betrachtet er die Architektur auch aus Betriebssicht und pflegt damit die DevOps-Kultur.

Dominik Galler
esentri AG
dominik.galler@esentri.com

Dominik Galler ist als Architekt und Berater bei der esentri AG in der Entwicklung und Konzeption verteilter Anwendungen tätig. Er probiert gerne neue Techniken aus und bewertet sie für den Enterprise-Einsatz. Mit seinem breiten Wissen über Problemlösungsstrategien und Architektur setzt er Anforderungen der Systemlandschaft unter Berücksichtigung der Unternehmenskultur und -struktur um.

Kotlin Koroutinen

Thomas Künneth, MATHEMA GmbH

Nebenläufigkeit und Asynchronität sind feste Bestandteile moderner Anwendungen. Java unterstützt seit der ersten Version das Verteilen der Programmlogik auf unterschiedliche Threads. Zug um Zug wurde die Standardklassenbibliothek um darauf aufbauende, moderne Konzepte erweitert. Das einst als JVM-Sprache gestartete Kotlin ist mittlerweile auf mehreren Plattformen zuhause. Allerdings gehen diese beim (Multi-)Threading natürlich unterschiedliche Wege. Koroutinen sind Kotlins Ansatz, über Plattformgrenzen hinweg asynchron zu programmieren.

Das Erzeugen und Ausführen eines Thread in der JVM ist mit wenigen Zeilen Quelltext erledigt *(siehe Listing 1)*. Alle Beispiele dieses Artikels stehen auf GitHub *[1]* zur Verfügung. Die Umsetzung als Koroutine zeigt *Listing 2*. Damit das Beispiel funktioniert, muss die Bibliothek `org.jetbrains.kotlinx:kotlinx-coroutines-core` verfügbar gemacht werden. Besonders einfach geht das mit Maven oder Gradle. `runBlocking()` startet eine neue Koroutine und blockiert den aktuellen Thread bis zu ihrer Beendigung. Die Funktion bildet eine Brücke zwischen regulärem blockierendem und unterbrechbarem Code. Sie sollte nicht innerhalb von Koroutinen verwendet werden. Zum einen, um ein Vermischen beider Konzepte zu vermeiden. Zum anderen, um nicht ungewollt asynchrone Konstrukte eben doch zu synchronisieren. `delay()` verzögert die Ausführung um die übergebene Anzahl an Millisekunden.

```
import kotlin.concurrent.thread

fun main() {
  thread {
    println("Waiting 5 seconds")
    Thread.sleep(5000)
  }.join()
  println("Ready")
}
```

Listing 1: Erzeugen und Starten eines Thread in Kotlin

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking

fun main() = runBlocking {
  println("Waiting 5 seconds")
  delay(5000)
  println("Ready")
}
```

Listing 2: Erzeugen und Starten einer Koroutine

Unter Kotlin/JS steht `runBlocking()` allerdings nicht zur Verfügung. Eine Alternative könnte folgendermaßen aussehen: `suspend fun main() = coroutineScope { ...`

Es gibt eine ganze Reihe weiterer sogenannter **Coroutine Builder**. `launch()` *(siehe Listing 3)* startet eine Koroutine, ohne auf deren Beendigung zu warten. Die Funktion liefert ein `Job`-Objekt, mit dem der Zustand der neuen Koroutine (oder besser: des Jobs) abgefragt werden kann. `join()` hält die Ausführung der aktuellen Koroutine an, bis ein Job beendet wurde. Der Aufruf von normalem, blockierendem Code innerhalb einer Koroutine ist nicht möglich. Er muss in einer Koroutine oder einer anderen unterbrechbaren Funktion erfolgen. Deshalb ist `join()` in `runBlocking()` eingebettet. Was aber macht eine Funktion unterbrechbar?

Unterbrechbare Funktionen

Praktisch die gesamte Funktionalität von Koroutinen wird durch Bibliotheken und den Kotlin-Compiler umgesetzt. Auf Sprachebene findet sich kaum Unterstützung. Mit einer entscheidenden Ausnahme. Das Schlüsselwort `suspend` kennzeichnet eine Funktion als unterbrechbar. In *Listing 4* wird kurzerhand die `main`-Funktion entsprechend markiert. Damit kann `join()` ohne eine weitere (eigentlich unnötige) Koroutine als Klammer aufgerufen werden. In produktivem Code ist das aber nicht üblich. Denn `main()` ist ja nur der Einstieg in das Programm. Das Starten von Koroutinen wird vermutlich in eigenen, später aufgerufenen Funktionen erfolgen. Übrigens macht `suspend` eine Funktion nicht nebenläufig. Das Schlüsselwort sagt dem Compiler aber, dass sie unterbrochen werden kann. Und das ist die Basis für Koroutinen. Die Bedeutung für den Programmfluss ist in *Abbildung 1* zu sehen.

```
import kotlinx.coroutines.*

fun main() {
  val job: Job = GlobalScope.launch {
    println("Waiting 5 seconds")
    delay(5000)
  }
  runBlocking {
    job.join()
  }
  println("Ready")
}
```

Listing 3: Eine Koroutine starten und auf ihre Beendigung warten

```
import kotlinx.coroutines.*

suspend fun main() {
  val job = GlobalScope.launch {
    println("Waiting 5 seconds")
    delay(5000)
  }
  job.join()
  println("Ready")
}
```

Listing 4: Verwendung des Schlüsselworts `suspend`

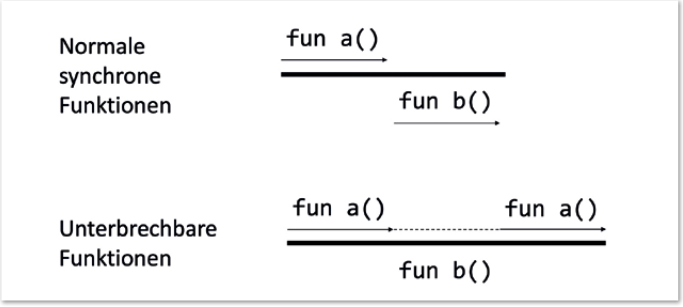


Abbildung 1: Unterschied zwischen blockierenden und unterbrechbaren Funktionen (© Thomas Künneth)

`launch()` ist sehr gut für klassische Fire-and-Forget-Szenarien geeignet. Um auf das Ergebnis einer asynchronen Operation zu warten, wird der Coroutine Builder `async()` verwendet. Er liefert ein Objekt des Typs `Deferred`, eine nicht blockierende, abbrechbare Future; wie `Job`, aber mit einem Ergebnis. `await()` wartet, bis dieses vorliegt. *Listing 5* zeigt, wie man andere Dinge erledigt (einmal pro Sekunde einen Asterisk ausgeben), während man auf das Ende einer asynchronen Operation (drei Sekunden pausieren und danach einen Text zurückgeben) wartet. Da `await()` ein synchroner Aufruf ist, werden er und die Ausgabe des gelieferten Strings in eine eigene Koroutine ausgelagert.

```
import kotlinx.coroutines.*

fun main(): Unit = runBlocking {
  val deferred = async { returnHello() }
  launch {
    println(deferred.await())
  }
  while (deferred.isActive) {
    println("*")
    delay(1000)
  }
}

suspend fun returnHello(): String {
  delay(3000)
  return "Hello, world!"
}
```

Listing 5: Asynchrone Operationen mit `async`

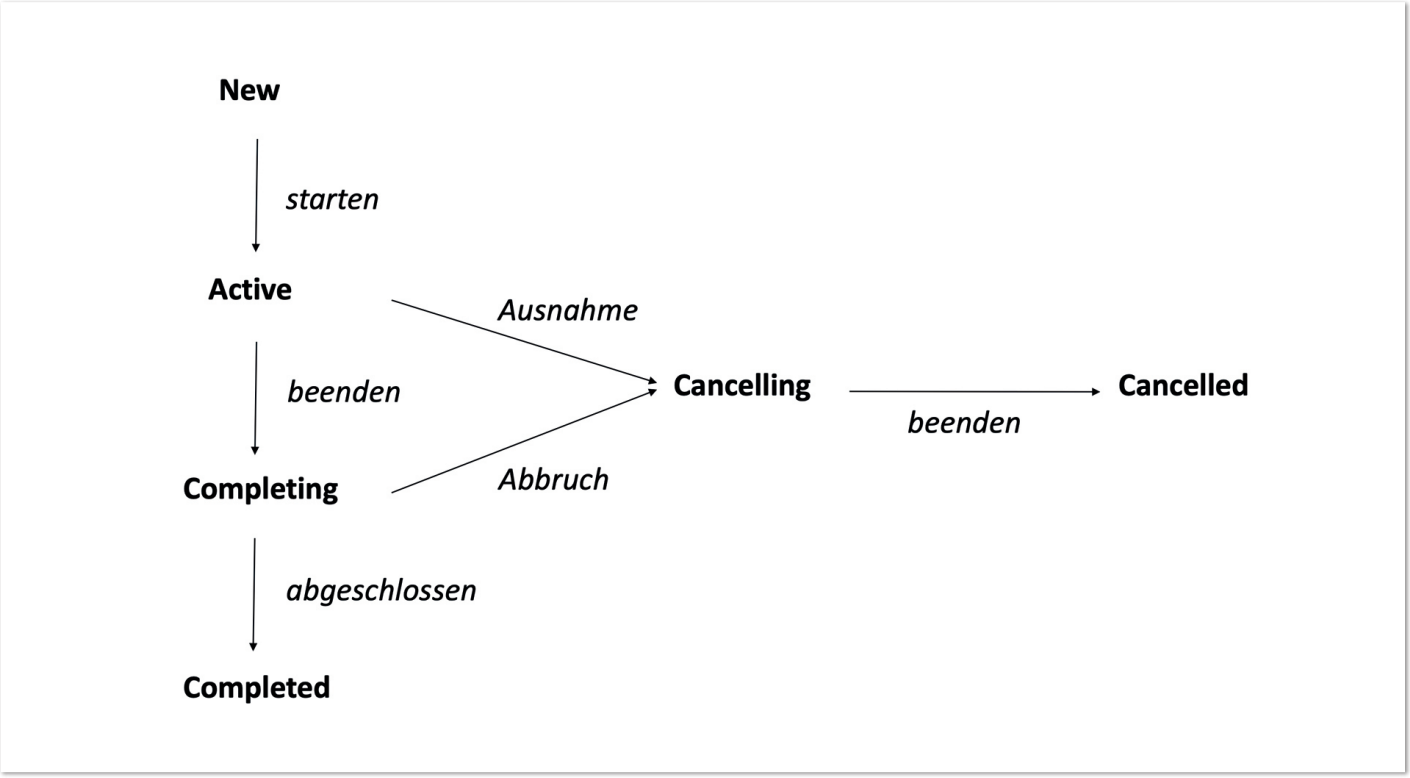


Abbildung 2: Der Lebenszyklus von Job und Deferred (© Thomas Künneth)

`Job` und das davon abgeleitete `Deferred` haben einen Lebenszyklus. Dieser ist in *Abbildung 2* zu sehen. Der aktuelle Zustand lässt sich über die Eigenschaften `isActive`, `isCancelled` und `isCompleted` abfragen. Das sollte man auch regelmäßig tun. Weshalb, zeigen *Listings 6* und *7*. Ersteres startet einen klassischen Thread und unterbricht ihn nach drei Sekunden. `measureTimeMillis()` misst, wie lange die Ausführung des übergebenen Blocks dauert. *Listing 7* zeigt eine vergleichbare Umsetzung mit Koroutinen. `cancelAndJoin()` bricht einen Job ab und hält die aufrufende Koroutine so lange an, bis der abgebrochene Job fertig ist. Das Problem: *Listing 7* hält nicht an.

Koroutinen müssen sich kooperativ verhalten und in regelmäßigen Abständen prüfen, ob sie noch ausgeführt werden sollen. Was nach einer archaischen Einschränkung klingt, ist im praktischen Alltag leicht umzusetzen. Es reicht nämlich, beispielsweise in Schleifen `isActive` abzufragen. Auch das Aufrufen von `ensureActive()`, `yield()` und `delay()` führt bei Bedarf zum Abbruch der Koroutine.

Bausteine von Koroutinen

Was genau sind Koroutinen eigentlich? Beim Überfliegen der Beispiele dieses Artikels fallen vor allem die Lambda-Ausdrücke auf, die an die Coroutine Builder `launch()`, `runBlocking()` und `async()` übergeben werden. JetBrains definiert Koroutinen als Instanzen einer unterbrechbaren Berechnung (englisches Original siehe *[2]*). Konzeptionell ähneln sie Threads:

- es wird ein übergebener Codeblock quasi parallel ausgeführt
- sie haben einen Zustand (erzeugt, gestartet, ...)

Allerdings sind Koroutinen nicht an einen Thread gebunden. Sie können auf einem unterbrochen und später auf einem anderen fortgesetzt werden. Threads beziehungsweise Thread-Pools (genauer: ihre plattformspezifischen Umsetzungen) sind sozusagen das Vehikel für die Ausführung. Wie Futures oder Promises können Koroutinen ein Ergebnis liefern, entweder einen Wert oder eine Ausnahme. Die Bausteine von Koroutinen sind in *Abbildung 3* zu sehen.

```
import kotlin.concurrent.thread
import kotlin.system.measureTimeMillis

fun main() {
  println(measureTimeMillis {
    val t = thread {
      var a = 0L
      while (true) a += 1
    }
    Thread.sleep(3000)
    t.interrupt()
    println("Interrupted")
  })
}
```

Listing 6: Einen Thread starten und nach drei Sekunden unterbrechen

```
import kotlinx.coroutines.*
import kotlin.system.measureTimeMillis

fun main() = runBlocking {
  println(measureTimeMillis {
    val t = launch(Dispatchers.IO) {
      var a = 0L
      while (true)
        a += 1
    }
    delay(3000)
    t.cancelAndJoin()
    println("Cancelled")
  })
}
```

Listing 7: Eine Koroutine starten und nach drei Sekunden abbrechen

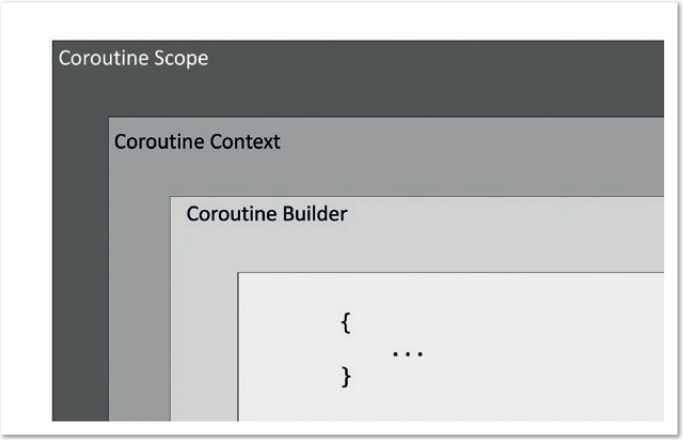


Abbildung 3: Bausteine von Koroutinen (© Thomas Künneth)

`CoroutineScope`-Instanzen steuern und verwalten eine oder mehrere Koroutinen. Sie können diese starten und abbrechen. Sie werden bei Abbrüchen und Fehlern benachrichtigt. Scopes definieren, wie lange Koroutinen existieren. Mit dem Ende eines Scope enden auch dessen Koroutinen. Die Lebensdauer eines `CoroutineScope` ist deshalb üblicherweise an den Lebenszyklus einer Komponente gebunden. Beispielsweise gibt es unter Android die Eigenschaft `viewModelScope` als Bestandteil von *ViewModel KTX*. Sie wird wie in *Listing 8* gezeigt verwendet.

Alle mittels `viewModelScope.launch` gestarteten Koroutinen sind an den Lebenszyklus des ViewModel gebunden. Warum das so praktisch ist, wird deutlich, wenn man sich eine Umsetzung für Swing ansieht *(siehe Listing 9)*. Die App ist sehr einfach aufgebaut. Sie erzeugt ein Fenster und startet drei Koroutinen. Beim Schließen des Fensters werden diese automatisch abgebrochen.

```
class MainViewModel : ViewModel() {
  private fun makeNetworkRequest() {
    viewModelScope.launch {
      ...
    }
  }
}
```

Listing 8: Der CoroutineScope viewModelScope unter Android

`GlobalScope` ist übrigens der am längsten zur Verfügung stehende Scope. Er endet mit der Anwendung. Was genau das bedeutet, ist von der Plattform abhängig. Jede Koroutine ist einem `CoroutineContext` zugeordnet. Er wird über die Eigenschaft `CoroutineScope.coroutineContext` zur Verfügung gestellt. Technisch umgesetzt ist er als eine Index-basierte Menge von Elementen. Üblich sind `CoroutineDispatcher`, `Job`, `CoroutineExceptionHandler` und `CoroutineName`. **Coroutine Dispatcher** legen fest, auf welchen Threads Koroutinen ausgeführt werden können. `Dispatchers.IO` wird üblicherweise für Datei- und Netzwerkzugriffe verwendet. `Dispatchers.Main` nutzt den Thread, der vom jeweiligen UI-Framework für die Aktualisierung der Benutzeroberfläche verwendet wird.

```
import kotlin.coroutines.*
import kotlinx.coroutines.*
import java.awt.event.*
import javax.swing.*

fun main() {
  SwingUtilities.invokeLater {
    val jframe = MyJFrame()
    jframe.setSize(200, 100)
    jframe.setLocationRelativeTo(null)
    for (i in (1..3)) {
      jframe.launch(CoroutineName(i.toString())) {
        var bool = false
        while (isActive) bool = !bool
        println("${coroutineContext[CoroutineName.Key]?.name} cancelled")
      }
    }
    jframe.isVisible = true
  }
}

class MyJFrame : JFrame("CoroutineScopeDemo"), CoroutineScope {

  private val job = SupervisorJob()
  override val coroutineContext: CoroutineContext
    get() = job + Dispatchers.IO

  init {
    defaultCloseOperation = DO_NOTHING_ON_CLOSE
    addWindowListener(object : WindowAdapter() {
      override fun windowClosing(e: WindowEvent?) {
        job.cancel()
        dispose()
      }
    })
  }
}
```

Listing 9: Koroutinen mit der Lebensdauer eines JFrame

Coroutine Builder sind Erweiterungsfunktionen von `CoroutineScope`. Sie starten Koroutinen. Der auszuführende Code wird den Buildern als Lambda übergeben. Der Parameter `start` (`DEFAULT`, `ATOMIC`, `LAZY` oder `UNDISPATCHED`) legt fest, wann die Verarbeitung beginnt. Was im Detail geschieht, ist vom verwendeten Builder abhängig. `launch` beispielsweise packt den Block als unterbrechbare (mit dem Schlüsselwort `suspend` versehene) `CoroutineScope`-Erweiterungsfunktion dem Startmodus entsprechend in ein `LazyStandaloneCoroutine-` oder `StandaloneCoroutine`-Objekt und ruft dessen Funktion `start()` auf. Der Code der Koroutine wird durch den Kotlin-Compiler in Teile zerlegt. Nur an den Übergängen zwischen diesen Teilen ist eine Unterbrechung möglich. JetBrains prägt hier die Begriffe **Suspension Point** und **Continuation** *[1]*. Syntaktisch ist ein Suspension Point der Aufruf einer unterbrechbaren Funktion. Die Continuation ist der Zustand zum Zeitpunkt der Unterbrechung. Stürzen mit `launch` gestartete Koroutinen ab, wird auch die Elternkoroutine abgebrochen. Ist das nicht gewünscht, kann, wie in *Listing 9* gezeigt, ein `SupervisorJob` verwendet werden. Um auf nicht gefangene Ausnahmen zu reagieren, wird `launch` ein `CoroutineExceptionHandler` übergeben. Als Alternative steht der Coroutine Builder `runCatching` zur Verfügung,

Fazit

Kotlins Koroutinen bieten weit mehr, als diese Einführung zeigen kann. Insbesondere die funktionalen Aspekte der Programmiersprache erschließen sich mit Koroutinen weitaus besser als mit klassischen Threads. Bei Experimenten sei aber das regelmäßige Zurateziehen der Dokumentation empfohlen. Nicht immer erschließen sich Standardparameter sofort. Oft hilft es, einen Blick auf die Implementierung einer Funktion zu werfen. Ohne Frage sind Koroutinen eine willkommene Bereicherung des Kotlin-Ökosystems. Für Android-Entwickler sind sie mittlerweile fast ein Muss. Aber auch auf anderen Plattformen erleichtern sie das Programmiererleben deutlich. Eine gewisse Einarbeitungszeit ist aber vonnöten, um sie sicher verwenden zu können.

Quellen

[1] *https://github.com/tkuenneth/mind_the_thread/tree/master/Kotlin/src/java_aktuell*

[2] *https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md#terminology*

Thomas Künneth

MATHEMA GmbH

thomas.kuenneth@mathema.de

Thomas Künneth ist Principal Consultant und Head of Mobile bei der MATHEMA GmbH. Seine Schwerpunkte sind Native und Cross Platform Apps sowie das Mobile Enterprise. Seit über 20 Jahren beschäftigt er sich intensiv mit Java. Außerdem ist Thomas Künneth Android-Entwickler der ersten Stunde und Autor zahlreicher Artikel und Bücher zu Java, Eclipse, Android und Mobile Computing. Er spricht regelmäßig auf Konferenzen und Community Events.

Quarkus macht's vor, WildFly zieht nach

Bernd Müller, Ostfalia

Das aktuelle Zeitgeschehen im Hinblick auf Unternehmensanwendungen im Java-Kontext ist von einer hohen Agilität geprägt. Zu einigen der zuletzt häufiger gehörten Stichwörter oder Namen gehören sicher Jakarta EE, MicroProfile, GraalVM und Quarkus. Um die altehrwürdigen Application-Server scheint es hingegen eher ruhiger zu werden. Wir wollen in diesem Artikel zeigen, dass diese publizistische Ruhe für WildFly zwar auch zutrifft, aber nicht der Realität der Weiterentwicklung entspricht. Beispielhaft wollen wir zwei attraktive Features von Quarkus und deren entsprechende Umsetzung in und mit WildFly vorstellen; das Deployment als JAR sowie ein Entwicklermodus mit Hot Deployment bei Änderungen an Code-Artefakten.

Ein wenig Vorgeschichte

WildFly ist der Nachfolgername des JBoss AS. Nach der Version JBoss AS 7 erschien WildFly 8.0 als nächste Version des Application-Servers. Wie alt der JBoss AS wirklich ist, konnten wir leider nicht in Erfahrung bringen. Die älteste von uns gefundene Version 3.0.0 *[1]* datiert von Mai 2002. Man kann also getrost von über 20 Jahren ausgehen – und in der IT den Application-Server damit mit Fug und Recht ein Urgestein nennen. Nachdem Java EE in der Version 5 viele Altlasten der in Ungnade gefallenen Vorversionen über Bord warf, hielt sich trotzdem standhaft der Mythos der großen und schwerfälligen Application-Server.

Im Juni 2016 stellte JBoss daher WildFly Swarm unter dem Motto „Just enough app-server for microservice-type applications" vor. Die grundlegende Idee ist das Verpacken der tatsächlich genutzten Teile von Java EE mitsamt der Anwendung selbst in ein JAR; quasi „Application-Server to go". Die Anwendung nimmt die verwendeten Teile des Application-Servers mit sich und bildet eine Einheit, in dem Fall ein JAR. Da *Swarm* ein relativ überladener Begriff war, entschied sich JBoss bereits Mitte 2018 zur Umbenennung. Aus *WildFly Swarm* wurde *Thorntail*. Aber auch dies hatte nicht lange Bestand. Nach der großen Aufmerksamkeit, um nicht zu sagen Hype – die Quarkus auf sich gezogen hatte, entschied sich JBoss, Thorntail nicht mehr weiterzuentwickeln. Die Überschneidungen mit Quarkus waren einfach zu groß. Das Angebot von JBoss besteht nun aus Quarkus und WildFly sowie der kommerziellen WildFly-Variante EAP.

WildFly, Galleon und Maven-Plug-ins

Die Weiterentwicklung des WildFly-Application-Servers ist von einer hohen Release-Frequenz gekennzeichnet. Die Version 12 erschien im Februar 2018, die Version 22 im Januar 2021, im Schnitt also alle drei bis vier Monate ein neues Release. Es scheint jedoch so, dass sich dies zukünftig zu Gunsten von Quarkus ändern wird. Zumindest lässt die Analyse der Commits der beiden Projekte auf GitHub diesen Schluss zu. Stuart Douglas, der Hauptentwickler von WildFly, wurde zum Hauptentwickler von Quarkus.

Eine der letzten großen Änderungen innerhalb des WildFly war die interne Modularisierung auf Basis von Galleon-Layers. Galleon *[2]* ist ein Provisionierungswerkzeug, um einen WildFly-Application-Server zu erstellen, der nur die tatsächlich verwendeten Bestandteile von Java EE verwendet. Im Cloud-Zeitalter ist die Minimierung des Application-Servers ein attraktives Ziel. Ein mit Galleon erzeugter WildFly mit JAX-RS und CDI hat eine Größe von 72 MB, der komplette Server eine Größe von 244 MB.

Das altbekannte WildFly-Maven-Plug-in (`org.wildfly.plugins:wildfly-maven-plugin`) kann Anwendungen deployen, redeployen und undeployen. Außerdem kann es den Application-Server herunterladen und starten, um beispielsweise Integrations- oder Systemtests mit Selenium durchzuführen.

Das relativ neue WildFly-Jar-Maven-Plug-in (`org.wildfly.plugins:wildfly-jar-maven-plugin`) erlaubt es, die benötigten Galleon-Layer und die Anwendung in ein JAR zu packen, so wie man das von Thorntail, Quarkus oder auch Spring Boot kennt.

Anwendung und Server verpacken

Der Application-Server besteht nun also aus Galleon-Layern, weiß aber selbst gar nichts von seinem Glück, dass er auch in Teilen distributiert werden kann. Dafür sorgt das WildFly-Jar-Maven-Plug-in. Um einen Eindruck von der Verwendung zu bekommen, zeigt *Listing 1* die Konfiguration des Plug-ins innerhalb des Build-Elements eines POM.

Man erkennt die einzelnen Galleon-Layer in den `<layers>/<layer>`-Elementen: CDI, JSF, EJB, JPA, JAX-RS, BV. Zusätzlich soll die Java EE Default Datasource von WildFly verwendet werden, sodass der H2-JDBC-Treiber sowie die Definition der Datasource ebenfalls zu packen sind. Der Management-Layer wird in der Regel ebenfalls benötigt. Der JAX-RS-Layer hat als Abhängigkeit den Web-Server-Layer, der wiederum den Deployment-Scanner als Abhängigkeit transitiv packt. Soll er wieder entfernt werden, so kann das Element `<excluded-layer>` wie im Beispiel verwendet werden. Wird mit diesem Plug-in gepackt, so entsteht ein ausführbares JAR, das die entsprechenden Galleon-Layer inklusive Abhängigkeiten sowie die Anwendung enthält. Die deployte Anwendung ist unter dem Root-Kontext verfügbar. Alternativ kann ein sogenanntes Hollow-Jar erzeugt werden, bei dem die Galleon-Layer in ein JAR, die Anwendung in ein WAR gepackt werden.

```
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-jar-maven-plugin</artifactId>
  <version>${wildfly-jar-maven-plugin.version}</version>
  <configuration>
    <feature-pack-location>...</feature-pack-location>
    <layers>
      <layer>cdi</layer>
      <layer>jsf</layer>
      <layer>ejb</layer>
      <layer>jpa</layer>
      <layer>jaxrs</layer>
      <layer>bean-validation</layer>
      <layer>h2-driver</layer>
      <layer>h2-default-datasource</layer>
      <layer>management</layer>
    </layers>
    <excluded-layers>
      <layer>deployment-scanner</layer>
    </excluded-layers>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>package</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Listing 1

Das zweite interessante Feature des Plug-ins ist die Möglichkeit des automatischen Bauens und Redeployens bei Code-Änderungen, häufig unter *Hot Deployment* oder *Hot Swapping* bekannt. Durch den Aufruf von `mvn wildfly-jar:dev-watch` wird das Plug-in angewiesen, dies durchzuführen. Alle Änderungen unter `src/main/java`, `src/main/webapp` und `src/main/resources` werden erkannt und sind sofort verfügbar, da die Anwendung als sogenanntes *Exploded Deployment* deployt wird. Das bedeutet, dass zum Beispiel Änderungen an Java-Klassen, JSF-Seiten oder Deployment-Deskriptoren erkannt werden und zum Redeployment führen. Das Entwickeln einer Anwendung bekommt neuen Schwung, da der Ändern-Redeploy-Zyklus deutlich schneller ist. Wem der Aufwand des Selbstausprobierens zu hoch ist, findet unter *[3]* ein kurzes Demonstrationsvideo.

Zusammenfassung

Durch die Modularisierung des WildFly-Application-Servers in einzelne Galleon-Layer ist es mit dem WildFly-Jar-Maven-Plug-in möglich, die von einer Anwendung benötigten Implementierungen verschiedener Java-EE-Teilspezifikationen und die Anwendung selbst in ein ausführbares JAR zu packen. Der Bedarf eines eigenständigen Application-Servers entfällt, was vor allem im Container-Umfeld Prozesse vereinfacht und für kleinere Images sorgt. Zusätzlich stellt das Plug-in ein Goal für die Entwicklung bereit, das bei Änderungen verschiedenster Projektdateien ein automatisches Redeployment durchführt. Eine von Entwicklern gern genutzte Vereinfachung.

Referenzen

[1] *https://jbossas.jboss.org/downloads*
[2] *https://docs.wildfly.org/galleon/*
[3] *https://youtu.be/OHXID3XZuOQ*

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.

Flexibilität auf Mikroarchitekturebene in Java: OSGi und Modulsystem

Holger Tiemeyer, AUSY Technologies Germany AG

Wenn von Flexibilität oder auch Anpassungsfähigkeit gesprochen wird, ist im eigentlichen Sinne die Fähigkeit zur Veränderung oder Selbstorganisation gemeint. Durch sie kann ein System auf sich verändernde, äußere Umstände reagieren. Flexible Systeme haben dabei den Vorteil, dass sie sich ohne großen Aufwand anpassen und anhand unterschiedlicher Qualitätsziele optimieren lassen. Dieser Artikel beschreibt unterschiedliche Möglichkeiten, eine erhöhte Flexibilität auf Mikroarchitekturebene in Java zu erreichen. Dies wird anhand des Modulsystems in Java 9 sowie der erweiterten Möglichkeiten mit OSGi vorgestellt.

Die Grundprinzipien der Modularisierung ermöglichen Flexibilität auf den drei gängigen Architekturebenen

Module und Services unterstützen uns dabei, geschlossene, austauschbare und voneinander entkoppelte Funktionseinheiten innerhalb eines Gesamtsystems zu bilden. Bereits 1972 wurden in einer Veröffentlichung von D.L. Parnas Kriterien für die Zerlegung eines Systems in Module vorgeschlagen:

> *„The effectiveness of a 'modularization' is dependent upon the criteria used in dividing the system into modules." [1]*

Parnas schlägt vor, mit einer Liste von Entwurfsentscheidungen zu beginnen, die schwierigen, dynamischen und zahlreichen Änderungen unterliegen. Jedes Modul, das entworfen wird, sollte eine dieser Entwurfsentscheidung jeweils vor den anderen verstecken („kapseln"). Hierdurch würden zwei Prinzipien erfüllt werden:

1. Ein Modul sollte so wenig wie möglich über ein anderes Modul wissen und
2. ein Modul sollte neu kompiliert und ersetzt werden können, ohne dass das gesamte System neu kompiliert werden muss.

Diese Grundprinzipien finden sich heutzutage auf jeder der drei gängigen Architekturebenen in unterschiedlichen Ausprägungen wieder – nämlich auf der Domänen-, der Makro- und der Mikroarchitektur. Die Dekompositionskriterien für Module lassen sich hierbei anhand der ISO-Norm-25010 *[2]* festlegen. Die durch die Norm definierten Qualitätsmerkmale für Software-Systeme erlauben die Ableitung von Qualitätszielen und somit die Bestimmung von Modulgrenzen.

Im folgenden Beispiel ist ein Interface gegeben, das eine `process`-Methode für Eingabedaten definiert. Für verschiedene Zielmärkte, Europa, Amerika und Asien, soll jeweils ein regionsspezifischer Algorithmus durch die `process`-Methode implementiert werden *(siehe Abbildung 1)*.

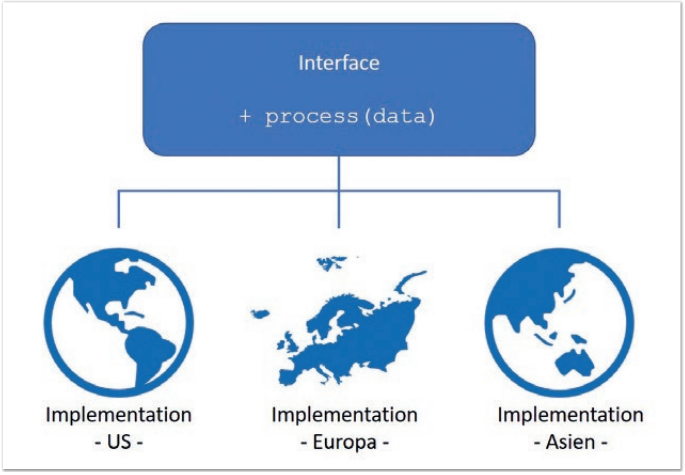


Abbildung 1: Regionsabhängige Implementierung eines Interface (Quelle: Holger Tiemeyer)

Es ist hierbei naheliegend, die Implementierung für die verschiedenen Zielmärkte voneinander zu entkoppeln und als jeweiliges Modul zur Verfügung zu stellen.

Die Entkopplung von Modulen erhöht die Flexibilität des Systems

Die Entkopplung von Modulen ist ein wichtiger Aspekt in der Softwareentwicklung. Denn eine zu starke Kopplung von Klassen und Methoden führt häufig zu einem „degenerierten Design" oder einem „Big Ball of Mud" *[3]*. Insbesondere erhöht eine Entkopplung die Flexibilität des (Software-)Systems. Zur Entkopplung von Modulen gibt es zwei etablierte Strukturierungsmöglichkeiten:

1. Interfaces: Durch die Verwendung von Interfaces lässt sich durch vertraglich zugesicherte Vorgaben sowohl in Richtung der Implementierung als auch in Richtung der Konsumenten des Moduls eine Entkopplung herstellen.
2. Entwurfsmuster: Entwurfsmuster sind Lösungsschablonen für wiederkehrende Probleme, die sich in Erzeugungsmuster, Strukturmuster und Verhaltensmuster untergliedern und zu einer Entkopplung von Modulen und Komponenten beitragen *[4]*.

Eine direkte Abhängigkeitsbeziehung einzelner Module untereinander lässt sich wie folgt auflösen: Die beiden Prinzipien werden kombiniert. Zudem werden ein Interface sowie beispielsweise das Fabrik-Entwurfsmuster verwendet und diese in ein jeweiliges Modul gekapselt.

Listing 1 zeigt die Interface-Definition. *Listing 2* zeigt eine Fabrik-Implementierung, die regionsabhängige Implementierungsklassen erzeugt.

Abbildung 2 zeigt die modulare Zerlegung. Insgesamt werden fünf Module bereitgestellt: Interface-, Fabrik-, und jeweilige Implementierungsmodul(e). Wie in *Abbildung 2* dargestellt, existiert eine Abhängigkeit des Factory-Moduls zu den anderen Modulen. Kommt beispielsweise ein neues Modul für eine weitere Region, müsste das Factory-Modul neu kompiliert werden, was keinen direkten Gewinn im Sinne einer Flexibilität der vorhandenen Module darstellt. Eine weitere Flexibilisierung kann durch die Definition von Services durch das Projekt Jigsaw erreicht werden, was im Folgenden kurz vorgestellt wird.

Modulgrenzen und Kapselung im Java-Modulsystem

Das „Projekt Jigsaw" (JSR 376), das mit Java 9 eingeführt wurde, ermöglicht die Kapselung und eine erweiterte Festlegung der Außensicht auf ein Modul sowie die Definition von Services. Das Hinzufü-

```
public interface RegionAlogrithm {

    void process(InputStream someData);

}
```

Listing 1: Interface-Definition

```
public class RegionAlgorithmFactory {

    public static RegionAlogrithm getRegionAlogrithm(Region region) {
        switch(region) {
            case US: return new UsRegionAlgorithm();
            case EUROPE: return new EuropeRegionAlogrithm();
            case ASIA: return new AsiaRegionAlgorithm();
            default: throw new IllegalArgumentException(
     "Region not supported.");
        }
    }
}
```

Listing 2: Fabrik-Klasse zur Erzeugung der jeweiligen Regions-Implementierungen

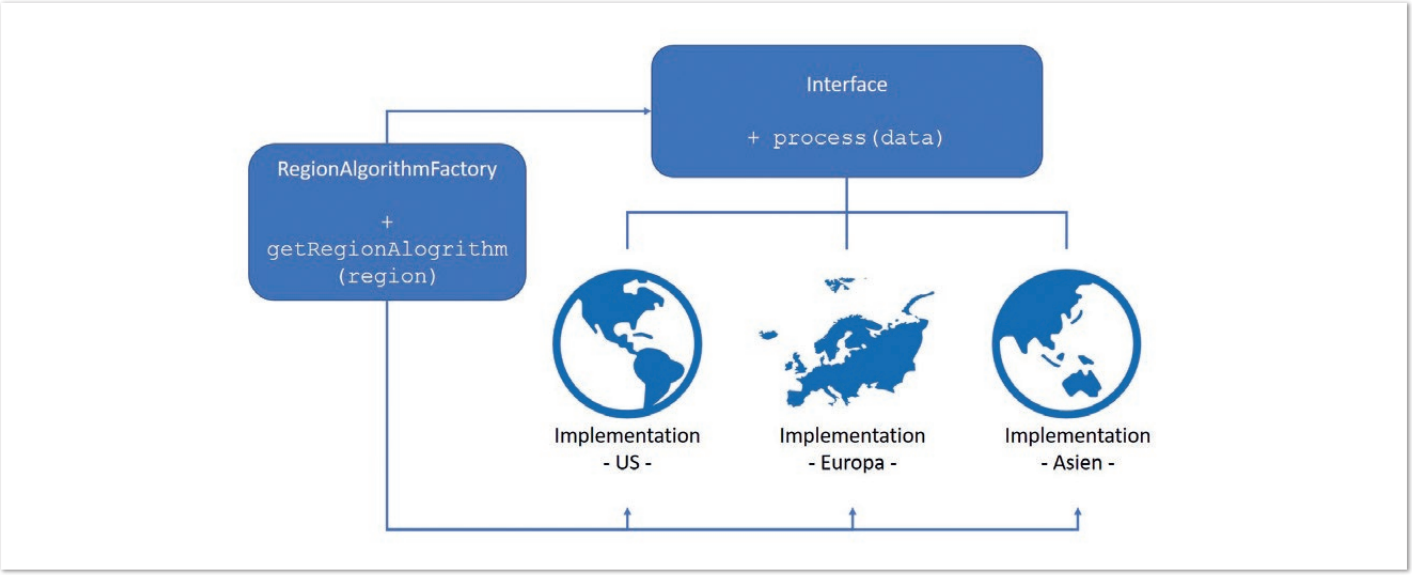


Abbildung 2: Factory zur Erzeugung der Implementierungsklassen (Quelle: Holger Tiemeyer)

```
module region.algorithm.api {
    exports com.example.api.RegionAlgorithm;

    requires ...; // e.g. package-dependencies from external modules
}
```

Listing 3: Abgekürzter, beispielhafter Inhalt der module-info.java-Datei

gen einer `module-info.java`-Datei in dem Wurzelverzeichnis einer Package-Hierarchie definiert die modularen Aspekte für ein Modul in Java.

In dieser Datei sind zunächst die `requires`-Anweisungen für Pakete und Klassen abhängiger Systemmodule sowie die Export-Anweisung des Pakets enthalten. In diesem befindet sich die Interface-Klasse `RegionAlgorithm.java` für die Sichtbarkeit in anderen Modulen *(siehe Listing 3)*.

Neben diesen beiden Strukturierungsmöglichkeiten existiert in Java zudem die Möglichkeit der weiteren Entkopplung und Erweiterung von Modulen über Services. Diese werden im Folgenden vorgestellt.

(Java-)Services zur Entkopplung von Modulen

Ein Service ist eine autarke Einheit, die über eine klar definierte Schnittstelle Funktionalitäten nach außen *anbietet (engl. provides)*. Diese können von Service-Konsumenten *benutzt (engl. uses)* werden.

Mit der Einführung des Modul-Systems in Java wurde der seit der Version 6 im JDK vorhandene `ServiceLoader` überarbeitet. Starre Modulabhängigkeiten, die mittels der `exports`- und `requires`-Anweisungen entstehen, lassen sich in dem neu eingeführten Konzept durch angebotene und konsumierende Services auflösen.

In den regionalen Implementierungs-Modulen werden dazu zunächst die angebotenen Services innerhalb der jeweiligen `module-info.java` deklariert *(siehe Listing 4)*.

```
module region.us {
    requires region.algorithm.api;

    provides com.example.api.RegionAlgorithm with
            com.example.region.us.UsRegionAlgorithm;
}
```

Listing 4: provides-Deklaration des UsRegionAlgorithm-Service innerhalb des Moduls region.us

Anstatt der `exports`-Anweisung wird ein `UsRegionAlgorithm`-Service, der das Interface `RegionAlgorithm` implementiert, von dem Modul `region.us` angeboten *(siehe Listing 4)*. Innerhalb der Klasse `UsRegionAlgorithm` kann nun eine statische `provider()`-Methode implementiert werden, die eine Instanz der Klasse erzeugt *(siehe Listing 5)*.

Durch eine Implementierung des `ServiceLoader` innerhalb einer statischen Methode in das `RegionAlgorithm`-Interface (seit dem JDK 8 möglich, *siehe Listing 6*), wird das `region.factory`-Modul obsolet.

Eine direkte Kopplung zu den Implementierungsklassen ist durch dieses Vorgehen ebenfalls nicht gegeben. Daher kann das `region.factory`-Modul entfernt werden: Damit dieses Vorgehen funktioniert, muss das Interface selbst in der `module-info.java` des `region.algorithm.api` -Moduls als Konsument deklariert werden *(siehe Listing 7)*.

Hierdurch sind die Module weitestgehend entkoppelt. Ein neues Region-Modul lässt sich nun einfach hinzufügen, ohne andere Module neu zu kompilieren.

Modularisierung mit OSGi

Eine erweiterte Möglichkeit, Module auch zur Laufzeit aufzufinden und für die Ausführung zur Verfügung zu stellen, ist die Modularisierung mit OSGi. Um die beschriebene Komplexität der flexiblen Austauschbarkeit von Java-Modulen und Anwendungslogiken beherrschbar zu machen, existieren zwei grundlegende Konzepte: Modularisierung und Abstraktion. Auf diesen basiert das Komponentenmodell OSGi. Die von der OSGi Alliance bereitgestellte Spezifikation *[5]* definiert eine dynamische, Java-basierte Softwareplattform, deren Kern einzelne Services bilden.

Bereitgestellt werden die Services dabei in speziell aufbereiteten JAR-Dateien, sogenannten Bundles. Diese können sich aus Java-Klassen und weiteren Ressourcen zusammensetzen (Konfigurationsdateien, abhängige Jar-Dateien usw.) Notwendige OSGi-spezifische Meta-Informationen werden in einer Manifest-Datei angegeben. Dazu gehören der Bundle-Name, die Version des Bundles, importierte und exportierte Packages sowie von dem jeweiligen Bundle benötigte Abhängigkeiten (Requirements) und angebotene Definitionen (Capabilities). Auf der JVM aufsetzend lässt sich das OSGi-Framework dabei in mehrere Schichten gliedern, wie in *Abbildung 3* ersichtlich.

Während die Security-Schicht die Java-eigenen Sicherheitsmechanismen ergänzt, wird über die Modul-Schicht die Modularisierung der Anwendung ermöglicht. Dabei muss jedes Bundle explizit angeben, welche Bundles es importiert und welche Packages es selbst exportiert. Da jedes Bundle einen eigenen Class-Loader besitzt, können so mehrere Services unabhängig voneinander in einer Java Virtual Machine (JVM) laufen.

Die Lifecycle-Schicht stellt dagegen ein API bereit, das die Installation und Deinstallation ermöglicht sowie das manuelle, programmatische oder automatisierte Starten und Stoppen der Bundles. Besonderer Vorteil hierbei ist, dass OSGi-Frameworks das sogenannte „Hot Deployment" unterstützen: Bundles können jederzeit

```java
public class UsRegionAlgorithm implements RegionAlgorithm {

    public static UsRegionAlgorithm provider() {
        return new UsRegionAlgorithm();
    }
}
```

Listing 5: Statische provider-Methode

```java
public interface RegionAlgorithm {

    static Iterable<RegionAlgorithm> getRegionAlgorithms() {
        return ServiceLoader.load(RegionAlgorithm.class);
    }
}
```

Listing 6: Statische Methode zum Laden von Services

```
module region.algorithm.api {
    exports com.example.api.region.algorithm.api;

    uses com.example.api.RegionAlgorithm;
}
```

Listing 7: Definition des RegionAlgorithm als Konsument innerhalb des Moduls region.algorithm.api

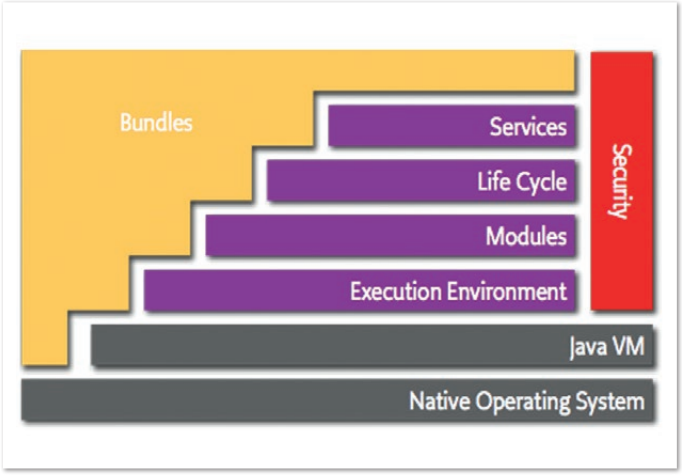


Abbildung 3: OSGi-Schichten- und Komponentenmodell (aus [5]).

zur Laufzeit während des Betriebs des Gerätes installiert, gestartet, beendet oder ausgetauscht werden. So lassen sich Änderungen an den einzelnen Anwendungen vornehmen – ohne dass dies mit einem kurzzeitigen Ausfall des Gerätes oder der laufenden Software verbunden ist.

Die Service-Schicht ist dagegen für die Kommunikation der einzelnen Bundles untereinander zuständig sowie für die Entkopplung der Schnittstellen der Services von ihrer Implementierung. Hierfür kann in einem Bundle ein einfaches Java-Interface bei einer zentralen Service-Registry als Schnittstelle registriert werden, beispielsweise in der `start()`-Methode der `BundleActivator`-Klasse. Diese wird ausgeführt, sobald ein Bundle startet. Auf diese Weise werden die einzelnen Services abstrahiert, wodurch unter anderem die Komplexität sinkt.

Aus diesem Aufbau ergeben sich zusätzliche Vorteile für die Verwendung in flexiblen Umgebungen: So sieht die OSGi-Spezifikation vor, ein stabiles API für ein Bundle zu erzeugen und die Service-Implementierung gegen dieses vorzunehmen. Die Idee dabei ist, dass das API anderen Lebenszyklen unterliegt als die Service-Implementierungen. Auf diese Weise lassen sich für jedes API mehrere Service-Implementierungen zur Verfügung stellen, die sich in ihrer Version und Funktionalität unterscheiden.

OSGi ermöglicht die Trennung von API- und Service-Implementierung

Ein Vorteil in der Verwendung von OSGi für flexible Mikroarchitekturen ergibt sich aus der strikten Trennung von API- und Service-Implementierungen. Dies geschieht durch die Bereitstellung von Bundles, die zum einen das API bereitstellen und zum anderen die entsprechende Implementierung des API. Insbesondere in der Kommunikation der Bundles untereinander entfaltet OSGi anhand der Trennung der API- von der Service-Implementierung sowie einer zentralen Service-Registrierung sein Potenzial: Bundles können dynamisch zur Laufzeit durch die Service-Registry anhand ihrer mitgelieferten META-Informationen zueinander in Beziehung gebracht, registriert, de-registriert, gestartet und gestoppt werden. Besonders hervorzuheben ist dabei, dass sich hinter einem gegebenen API mehrere Service-Implementierungen dynamisch zur Laufzeit laden lassen – unter anderem auch in unterschiedlichen Versionen und Ausprägungen.

Durch die zuletzt genannte Besonderheit unterschiedlicher Bundle-Versionen kann ein Bundle für zu importierenden Artefakte die benötigten Versionen explizit angeben; dies beispielsweise auch als Intervall mit Höchst- beziehungsweise Mindestwert. Dank des OSGi-eigenen Versionierungssystems können so mehrere Versionen eines Bundles gleichzeitig nebeneinander laufen. Sollen nun beispielsweise für die EU-Region verschiedene Algorithmen hinzugefügt werden, können diese somit parallel in Betrieb genommen werden. Durch diese Trennung ergeben sich folgende Vorteile:

1. Hinter der stabilen API-Definition können unterschiedliche Ziel-Implementierungen stehen (im Anwendungsbeispiel wäre dies eine weitere Europa-Algorithmus-Bundle-Implementierung oder auch eine US-Implementierung). Dabei entscheidet das OSGi-Service-Framework durch die Auswertung der MANIFEST-Dateien über die Service-Registry zur Laufzeit, welche Implementierung gerade benötigt wird.
2. Es können für jede Implementierung eine Vielzahl von Versionen existieren. So kann beispielsweise die Europa-Implementierung in den Versionen 1.0.0 und 1.5.0 vorliegen. Benötigt ein Service nun die Version 1.0.0, so kann dies in der Requirements-Definition innerhalb des entsprechenden Bundles definiert werden. Ein anderer Service kann dagegen beispielsweise die Version 1.5.0 anfordern. Die Definitionen werden zur Laufzeit ausgewertet und passende Bundles zueinander über das stabile API in Beziehung gesetzt.

OSGi eignet sich demnach hervorragend für die Reduktion der Komplexität und damit für den Einsatz in flexiblen Microservices-Umgebungen. Dies liegt zum einen an dem dynamischen Komponentenmodell. In diesem lassen sich Bundles jederzeit austauschen, hinzufügen oder auch parallel in verschiedenen Versionen betreiben. Und andererseits daran, dass die strikte Trennung der Serviceimplementierungen von ihren Schnittstellen eine einfache Abstraktion ermöglicht.

Fazit: Modularisierungsansätze in Java ermöglichen flexible Architekturen

Durch eine geschickte Verwendung von Modularisierungsansätzen in Java lassen sich flexible Umsetzungsmöglichkeiten auf der Java-Mikroarchitekturebene erreichen. Insbesondere das dynamische Komponentenmodell OSGi scheint sehr gut geeignet, der einfachen Abstraktion und Implementierung der Services einen sehr hohen Grad an Flexibilität zur Verfügung zu stellen. Dennoch benötigt der OSGi-Ansatz eine entsprechende Laufzeitumgebung, die auf der JVM aufsetzt *[6]*.

Ein besonderer Vorteil ergibt sich in der Verwendung von OSGi in IoT-Anwendungen, weil die Referenzimplementierungen der OSGi-Spezifikation bereits Implementierungen für unterschiedlichste Sensoren und Devices mitbringen. Dies spiegelt sich auch in einer stetig steigenden Anzahl an auf OSGi-aufbauenden IoT-Frameworks wider, die zum großen Teil auch als Open-Source-Projekte entwickelt werden. Interessant ist die Kombination des Java-9-Modulsystems mit OSGi, um die Vorteile der Sichtbarkeit auf innere Klassen von Bundles aktiv auszunutzen.

Quellen

[1] D.L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communication of the ACM, Volume 15, Number 12, December 1972

[2] *https://iso25000.com/index.php/en/iso-25000-standards/iso-25010*

[3] *https://de.wikipedia.org/wiki/Big_Ball_of_Mud*

[4] *https://de.wikipedia.org/wiki/Entwurfsmuster*

[5] OSGi Compendium, OSGi.cmpn-7.0.0.pdf, Seite 264

[6] *http://felix.apache.org/*

Holger Tiemeyer
AUSY Technologies Germany AG
Holger.tiemeyer@ausy-technologies.de

Holger Tiemeyer hat an der Universität Hamburg Informatik mit Nebenfach Psychologie studiert. Er realisiert in der Rolle als Senior-Software-Architekt unterschiedliche Projekte in verschiedenen Enterprise-Kontexten bei der AUSY Technologies Germany AG. Seine Hauptaufgabe liegt in dem Entwurf und der Umsetzung von komplexen Softwarearchitekturen. Dabei vermittelt er fundiertes, fachliches Wissen um die Materie der Softwarearchitektur. Er ist Mitglied im iSAQB e.V., leitet dort die Arbeitsgruppe Hochschulen und gibt Modulschulungen des iSAQB als akkreditierter Trainer.

Sicherheitslücken durch Open-Source-Abhängigkeiten in Java-basierten, Cloud-nativen Applikationen finden und beheben

Mathias Conradt, Snyk

Dieser Artikel zeigt auf, wie sich mithilfe von Tools wie Snyk Sicherheitslücken durch Open-Source-Abhängigkeiten in Java-basierten, Cloud-nativen Applikationen finden und beheben lassen, bevor diese produktiv gestellt werden.

In modernen Softwareprojekten verlassen wir uns sehr auf Frameworks und Bibliotheken von Drittanbietern. Nur 10 bis 20 Prozent der gesamten Codebasis ist eigens geschriebener Code, die restlichen 80 bis 90 Prozent entstammen Open-Source-Abhängigkeiten durch verwendete Frameworks oder Support-Libraries. Hinzu kommt, dass neben dem eigentlichen Applikationscode und

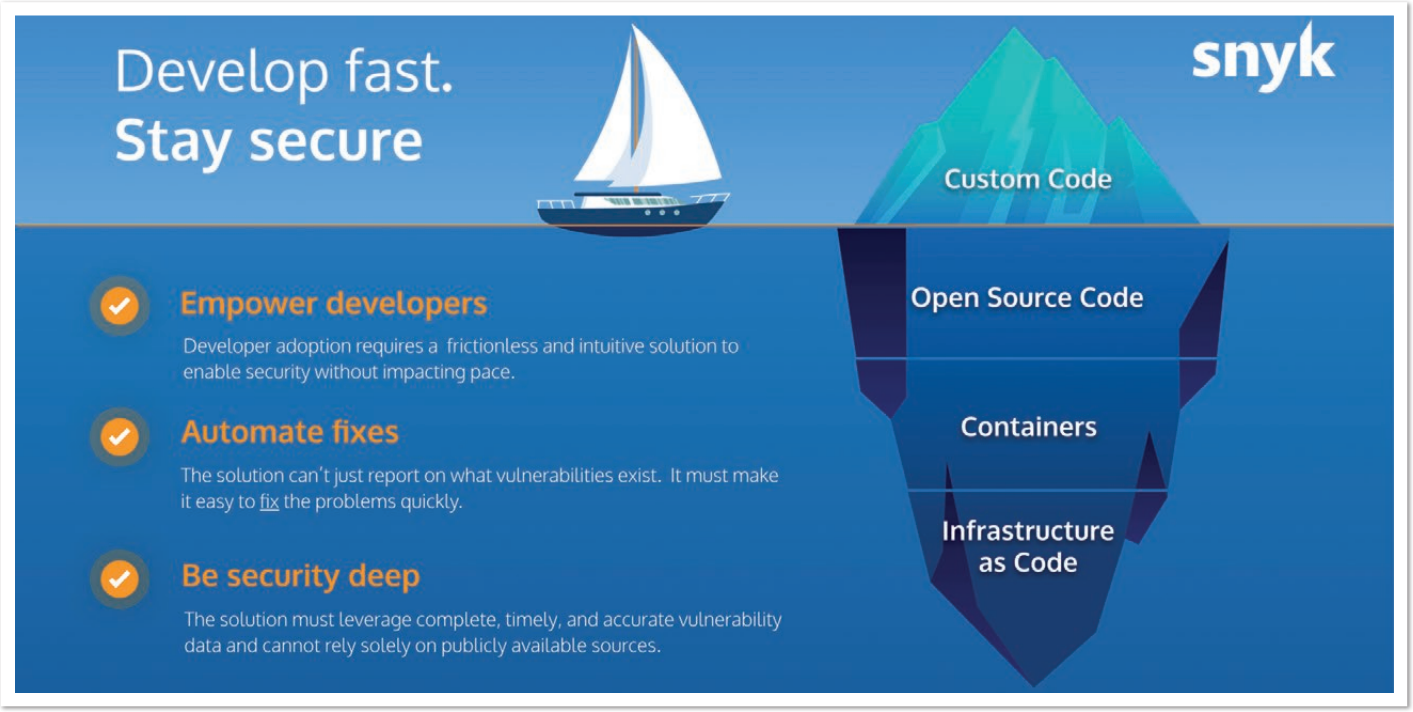


Abbildung 1: Die moderne Software-Supply-Chain (© Snyk)

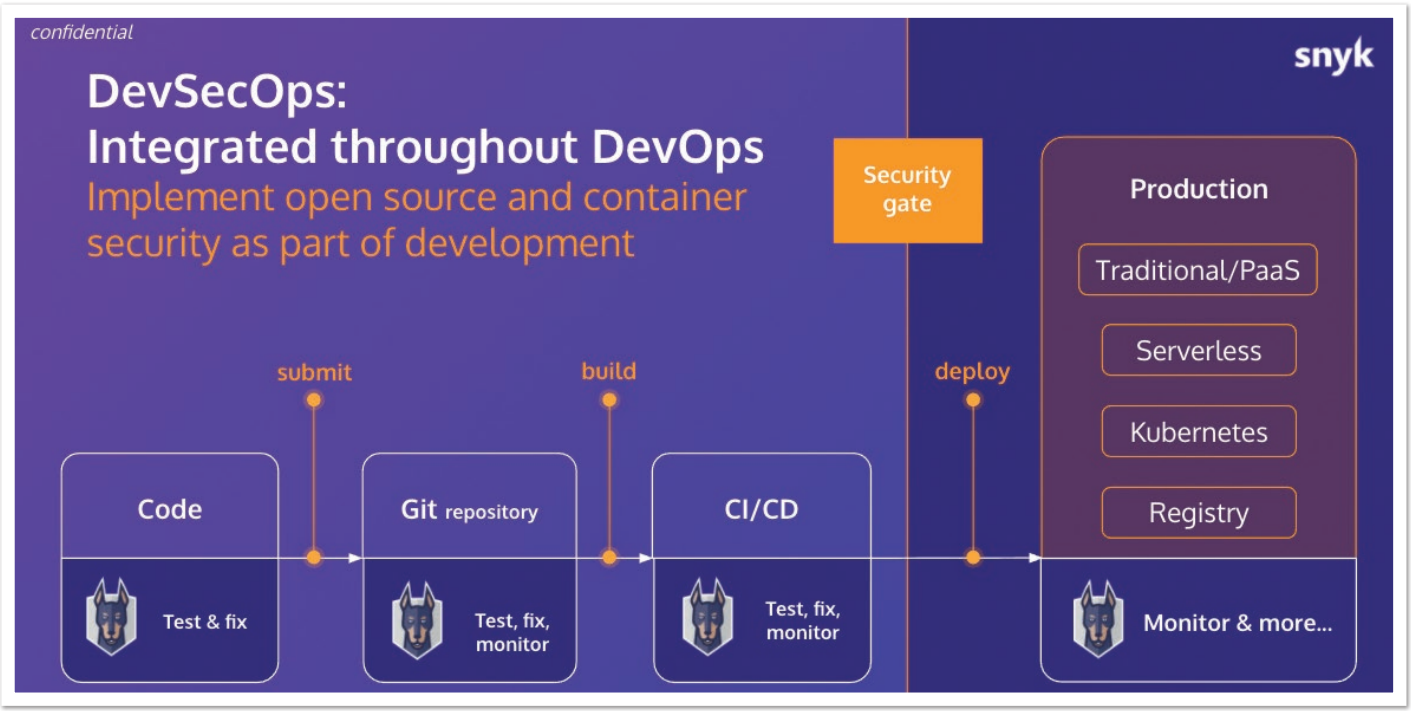


Abbildung 2: Snyk unterstützt entlang des gesamten SDLC (© Snyk)

dessen Abhängigkeiten bei Cloud-nativen Anwendungen auch noch die Container-Definition sowie Infrastruktur als Code vorliegt *(siehe Abbildung 1)*. Diese liegen etwa in Form von Dockerfiles, Kubernetes-Deployment- oder Terraform-Files vor. Dies alles bildet entsprechende Angriffsfläche, die – zumindest teilweise – im Verantwortungsbereich des Application Developer liegt.

Während ich mich als Entwickler eigentlich auf die Business-Logik konzentrieren möchte und nicht unbedingt ein Security-Experte bin, möchte ich trotzdem sicherstellen, dass meine Applikation vor Angriffen sicher ist.

Schwachstellen zu finden und zu beheben, ist dann am wenigsten kosten- und zeitintensiv, je früher ich damit links in meinem SDLC anfange („Shift-Left"), bestenfalls gleich im lokalen Terminal oder in der IDE, oder im Git Repository *(siehe Abbildung 2)*.

Snyk

Snyk ist ein Freemium-Tool, das Open-Source-Abhängigkeiten in Manifest-Files (pom.xml, build.gradle) ausliest, einen Dependency-Tree generiert und diesen dann mit seiner Schwachstellen-Datenbank vergleicht. Die Herausforderung für Entwickler, ihre Applikation in Bezug auf Schwachstellen abzusichern, ist zweigeteilt:

Abbildung 3: Generierter Dependency-Tree, generiert basierend auf der pom.xml (© Snyk)

Zum einen befinden sich 80 Prozent aller Schwachstellen in indirekten Abhängigkeiten, also jenen, denen ein Entwickler sich nicht unbedingt bewusst ist, da er diese nicht explizit in seinem pom.xml angefragt hat. Diese ganzen indirekten Abhängigkeiten teilweise in zweiter, dritter oder gar vierter Ebene zu identifizieren, ist ohne ein entsprechendes Tool mühsame Handarbeit.

Zum anderen ist das Wissen über eine Schwachstelle einer bestimmten Version einer Abhängigkeit nur die halbe Miete, wenn ich nicht weiß, wie ich diese beheben kann. Ein entsprechendes Tool gibt hierüber ebenfalls sofort Auskunft *(siehe Abbildung 3)*.

Testen in der CLI

Um ein Tool wie Snyk zu nutzen, legt man sich einen kostenfreien Account unter snyk.io an und installiert dann das zugehörige CLI *[1]* mittels `npm`, `scoop` oder `brew`, je nach System *(siehe Listing 1)*. Im Blog *[2]* ist ein CLI-Cheat-Sheet verfügbar, das einen guten Überblick über alle verfügbaren Aktionen und Optionen bietet. Um mein Projekt, in meinem Beispiel ein Maven-basiertes Projekt, zu testen, reicht der einfache, in *Listing 2* gezeigte Befehl.

```
# Installation mittels npm
npm install -g snyk
#
# alternativ: Installation mittels Brew
brew install snyk
#
# Authentifizierung
snyk auth
```

Listing 1: Snyk-CLI-Installation und -Authentifizierung

```
mvn install
snyk test
```

Listing 2: Snyk-Test via CLI

```
snyk monitor
```

Listing 3: Snyk-Monitoring via CLI

Das Ergebnis ist ein entsprechender Report mit gefundenen Schwachstellen sowie der Information, ob diese jeweils mittels Upgrade oder Patch beseitigt werden können.

Beispielsweise können alle in *Abbildung 4* gezeigten Schwachstellen (wie XSS, DoS und Remote-Code-Execution-Probleme, jeweils gelb oder rot markiert), die durch die direkte Abhängigkeit `struts2-core@2.3.20` eingeführt wurden, durch ein einfaches Upgrade auf `struts2-core@2.5.26` behoben werden. Gelb markierte Schwachstellen stehen für einen mittleren Schweregrad, wohingegen rot markierte Schwachstellen für einen hohen Schweregrad stehen.

Bei einer Upgrade-Empfehlung wird dabei versucht, den minimal möglichen Sprung auf die nächste, nicht-betroffene Version zu machen, um „Breaking Changes" zu vermeiden. Major Upgrades werden also – soweit möglich – vermieden.

Um das Projekt langfristig zu monitoren – oder gar den Snapshot, der sich in der Produktivumgebung befindet – und es der Snyk Admin UI hinzuzufügen, um somit regelmäßig auf Schwachstellen hin zu prüfen, auch wenn der Entwickler den Code selbst eine Zeit lang gar nicht anfasst, kann der folgende Befehl ausgeführt werden *(siehe Listing 3)*.

Testen in der IDE

Wer das Testen lieber in der IDE statt IntelliJ vornimmt, der kann auf das Snyk-Plug-in *[3]* für IntelliJ oder Eclipse zurückgreifen, die im jeweiligen Marketplace der IDE zu finden sind. Hier integriert sich dieser Test direkt in die IDE, die Information ist die gleiche, lediglich die Darstellung des Reports ändert sich etwas *(siehe Abbildung 5)*.

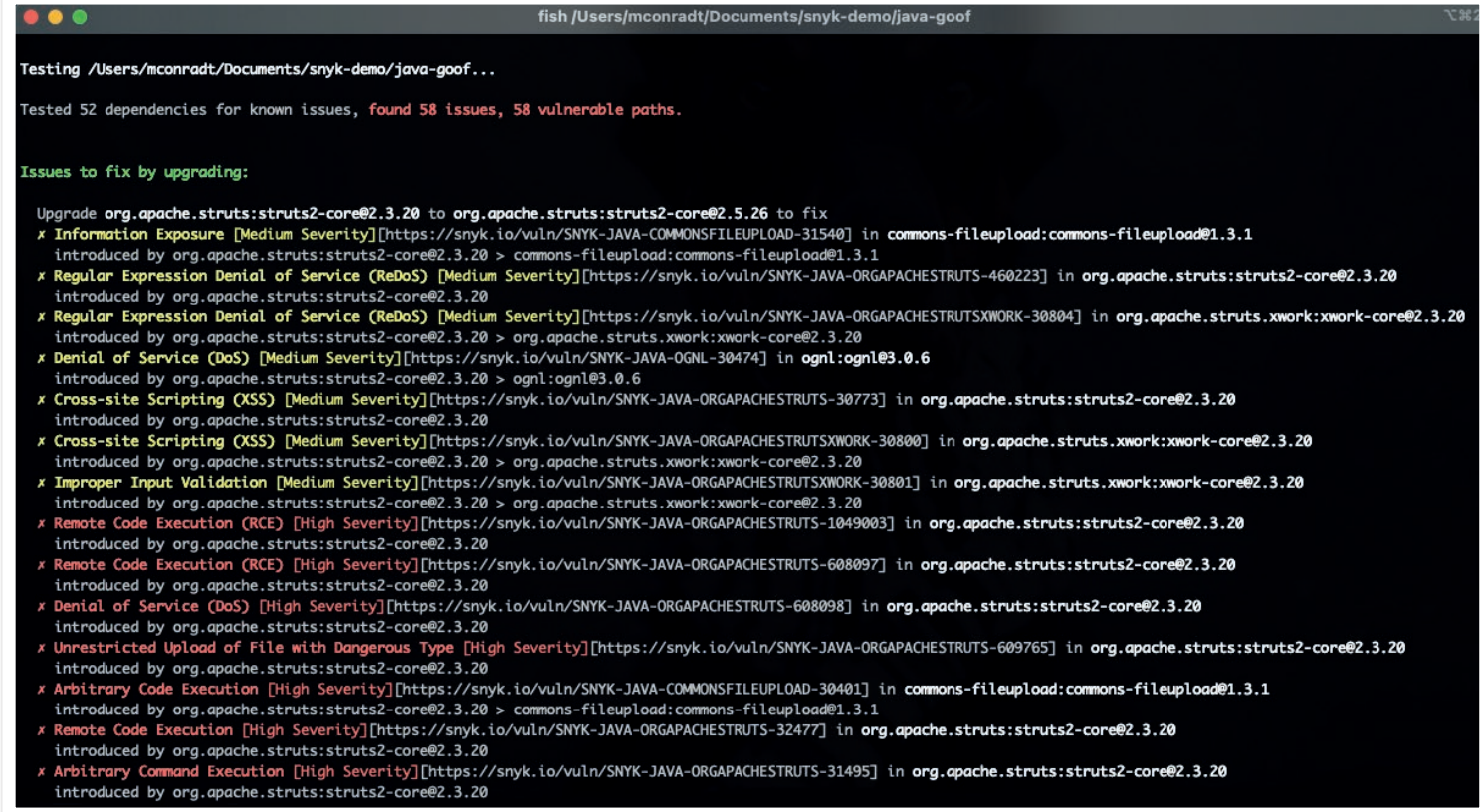


Abbildung 4: Snyk-Test im CLI (© Snyk)

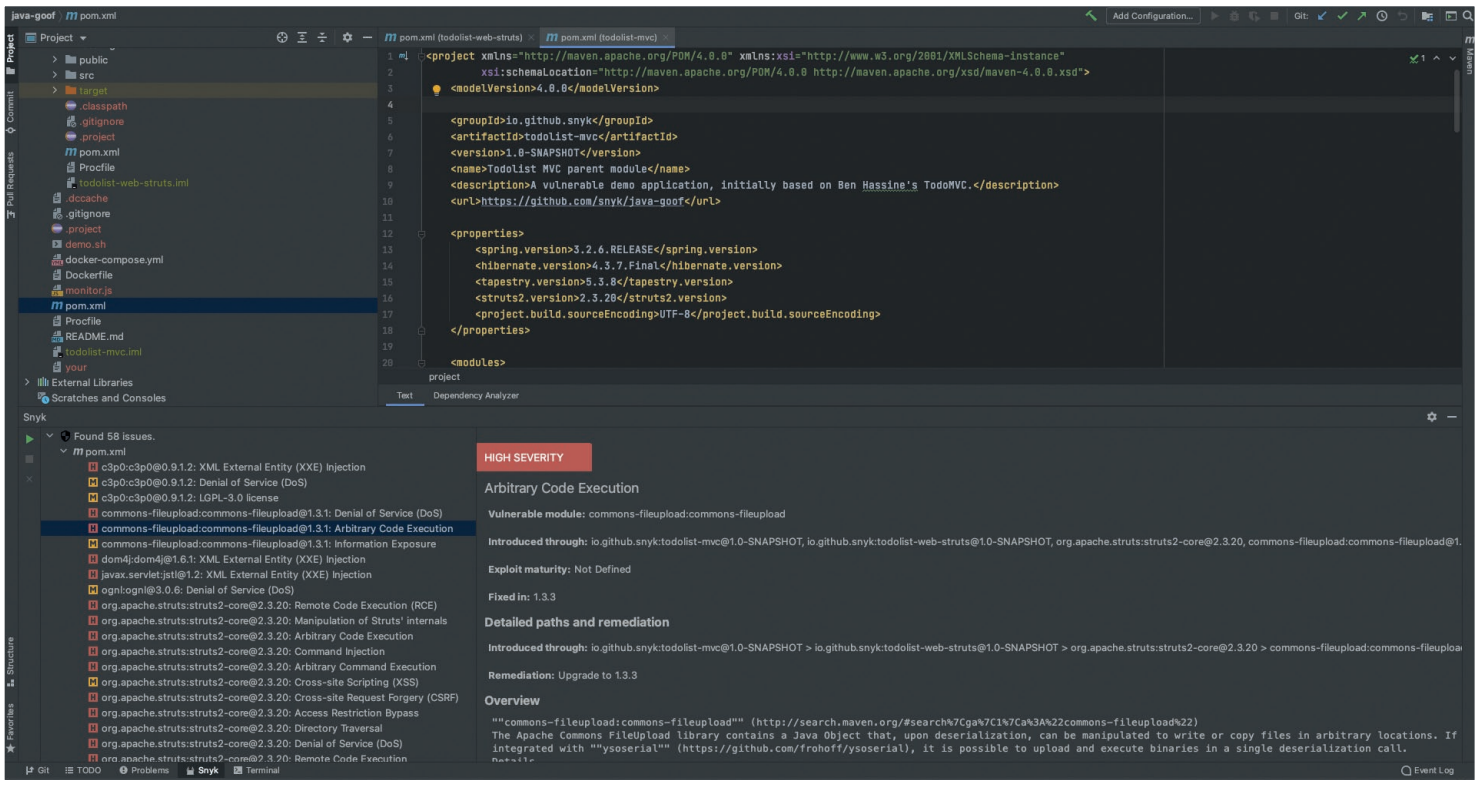


Abbildung 5: Snyk-Test in der IDE (© Snyk)

Testen des Git Repository

Snyk bietet Integrationen in GitHub, GitLab, Bitbucket, Azure Repos. Dies erlaubt das Testen, Beheben sowie das regelmäßige Monitoring von Projekten. Beispielhaft wird dies im Folgenden anhand der GitHub-Integration gezeigt. Es lassen sich alle oder einzelne Projekte einer GitHub-Organisation zu Snyk hinzufügen. Snyk identifiziert entsprechende Manifest-Dateien (pom.xml, build.gradle) und führt einen ersten Test durch.

Das Ergebnis wird daraufhin in der Weboberfläche angezeigt *(siehe Abbildung 6)*. Die Projektliste zeigt die Anzahl der gefundenen Schwachstellen mit ihrem jeweiligen Schweregrad (High/Medium/Low) an. Besteht ein Projekt aus mehreren Modulen oder Unterprojekten, werden alle Manifest-Dateien angezeigt. Mit Klick auf eine dieser Dateien gelangt man in die Report-Ansicht *(siehe Abbildung 7)*.

In dem gezeigten Beispielprojekt (pom.xml) wurden 57 Schwachstellen in 49 (direkten als auch indirekten) Abhängigkeiten gefunden.

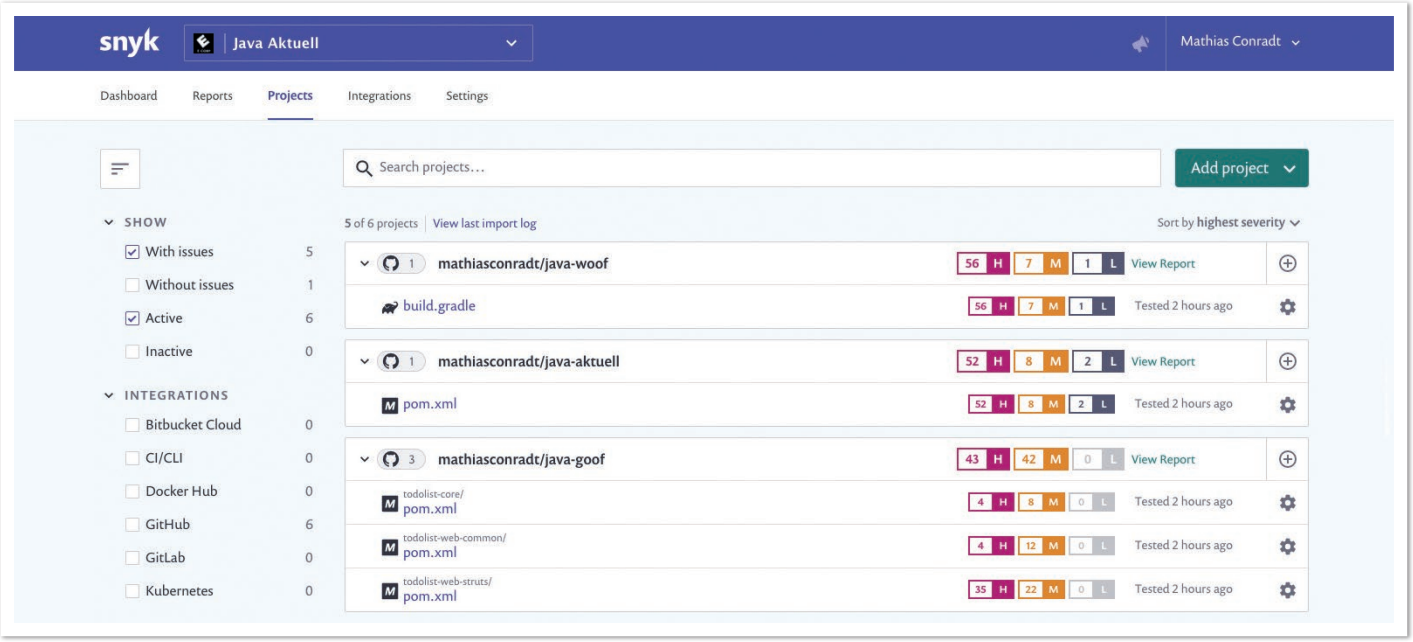


Abbildung 6: Snyk-Test mittels Git-Integration und Web UI (© Snyk)

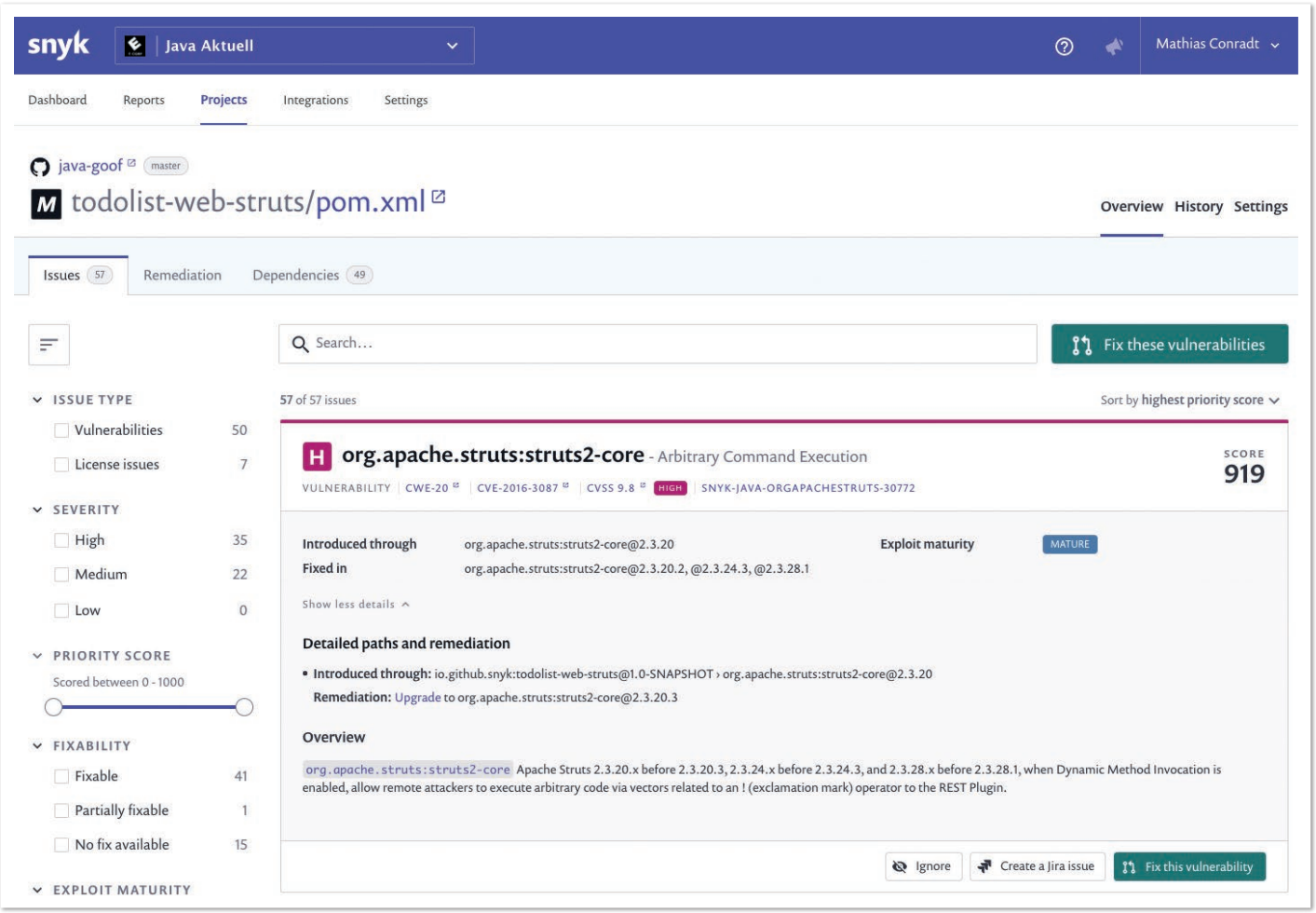


Abbildung 7: Report-Ansicht eines Projektes in der Snyk Web UI (© Snyk)

Komplexe Projekte können teilweise Hunderte von Schwachstellen aufweisen. Um den Entwickler nicht zu überfrachten und zu überfordern, ist es wichtig, eine entsprechende **Priorisierung** vorzunehmen.

Hierzu werden gefundene Schwachstellen zunächst grob nach Schweregrad High/Medium/Low gruppiert, darüber hinaus ist jeder Schwachstelle ein sogenannter Priority-Score zugewiesen. Dieser wird von Snyk berechnet und kalkuliert verschiedene Faktoren mit ein:

- CVSS Score
- Verfügbarkeit eines Fix
- „Exploit Maturity" – diese kann in drei Stufen vorliegen:
  - „Mature": Es ist ein veröffentlichter Code-Exploit verfügbar, der leicht für diese Sicherheitslücke verwendet werden kann.
  - „Proof of Concept": Ein veröffentlichtes, theoretisches Proof-of-Concept oder eine detaillierte Erklärung sind verfügbar, die zeigen, wie diese Schwachstelle ausgenutzt werden kann.
  - „No known Exploit": Für diese Sicherheitslücke wurde weder ein Proof-of-Concept-Code noch ein Exploit gefunden, beziehungsweise sind keine öffentlich verfügbar.

Der Report listet die Schwachstellen absteigend nach Priority-Score auf und gibt Details über gefundene Schwachstellen. Die Details beinhalten Informationen zum Namen und zur Kategorie der Schwachstelle, zu betroffenen Modulen und Versionen, in denen die Schwachstelle auftritt, ob und in welcher Version diese behoben ist, den Abhängigkeitspfad sowie Hintergrundinformation. Ein Link zur Snyk-Schwachstellen-Datenbank liefert optional weitere Hintergrundinformationen.

Von diesem Bildschirm aus hat der Entwickler nun die Möglichkeit, sofern ein Fix vorhanden ist, direkte einen Pull-Request zu erstellen. Mit Klick auf „Fix this Vulnerability" erstellt Snyk automatisch für die ausgewählten Schwachstellen einen Fix-Branch und Pull-Request *(siehe Abbildung 8)*.

Der Pull-Request beinhaltet das entsprechende Upgrade der direkten Abhängigkeit, durch die die Schwachstelle in das Projekt eingeführt wurde *(siehe Abbildung 9)*.

Das gezeigte Beispiel zeigt den Workflow eines **Pull-Request**, der manuell durch den Entwickler im Snyk Web UI angestoßen wurde.

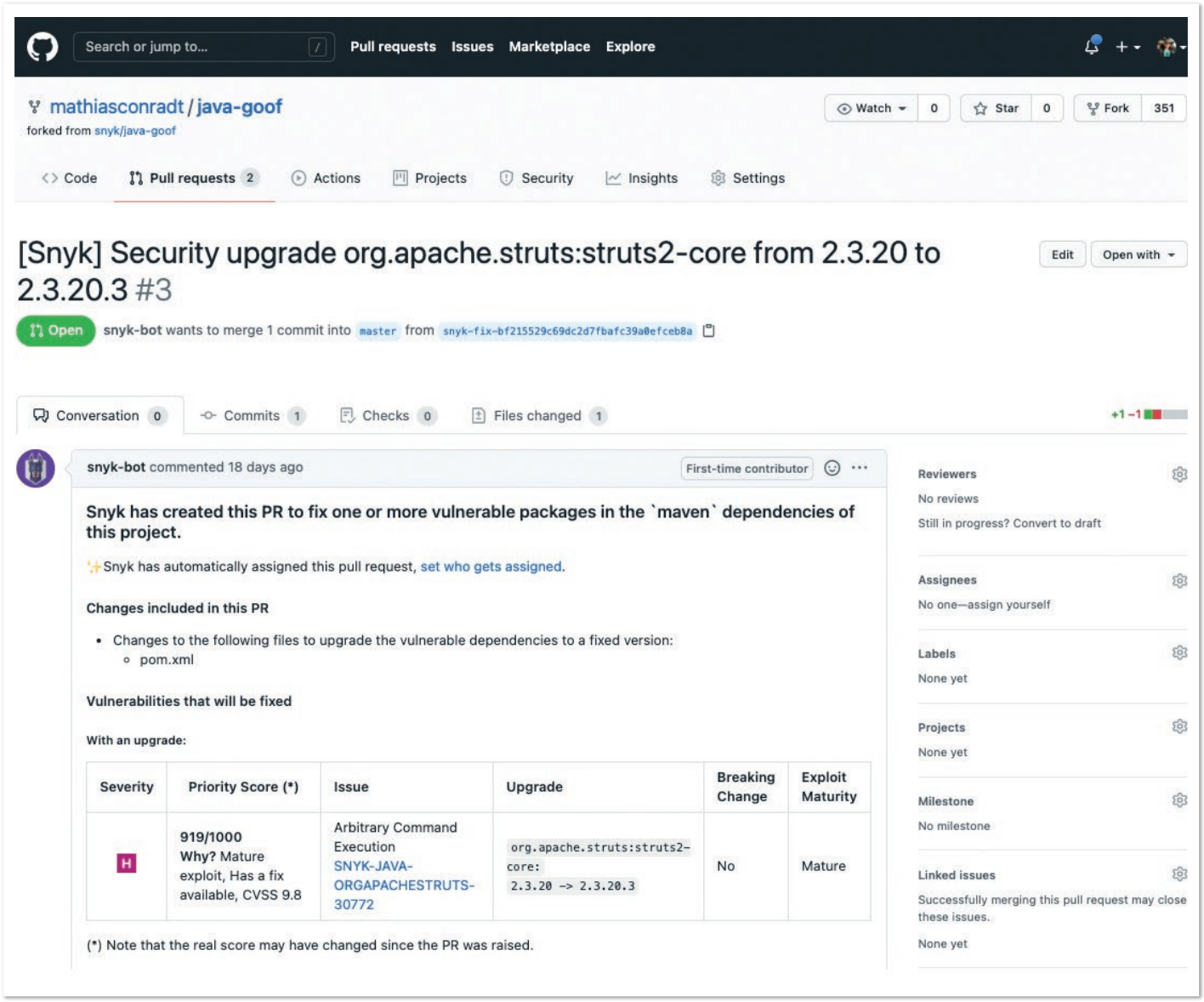


Abbildung 8: Automatisch generierter Pull-Request in GitHub (© Snyk)

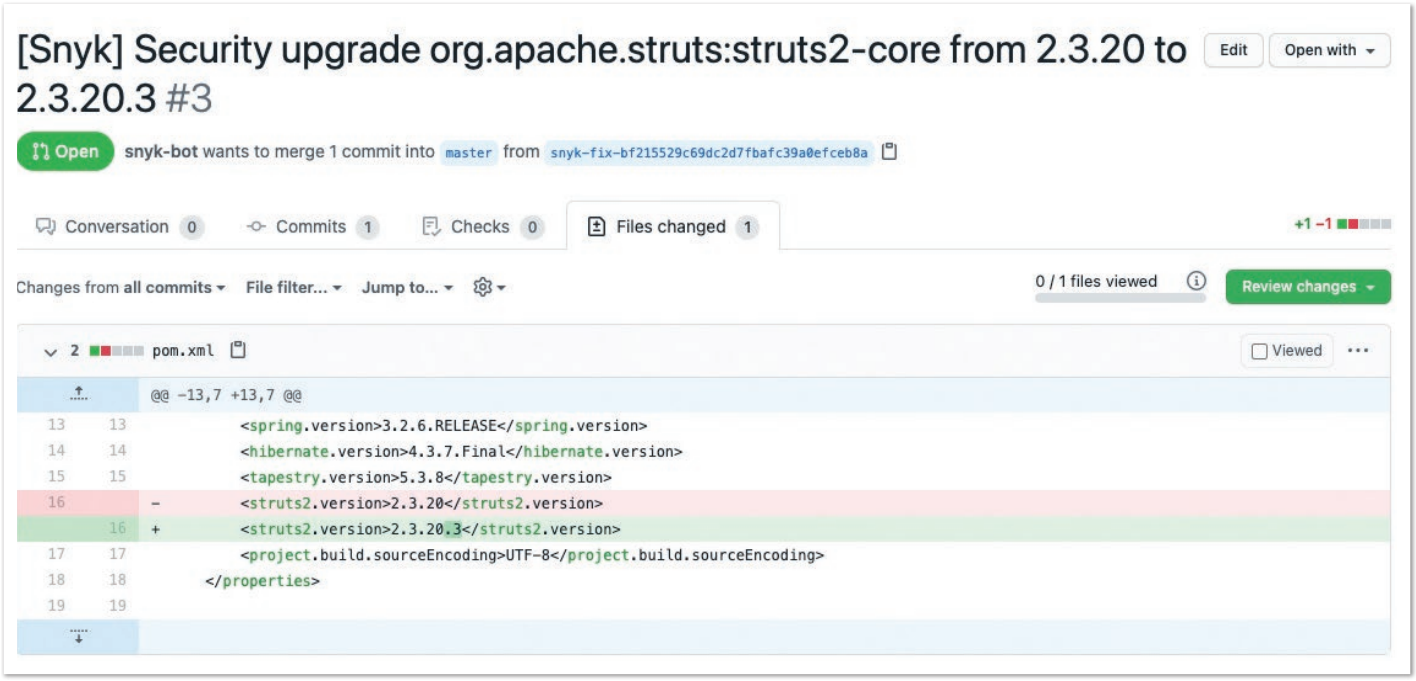


Abbildung 9: Upgrade der direkten Abhängigkeit in der pom.xml (© Snyk)

Natürlich lässt sich dies auch **automatisieren**. Es ist möglich, Snyk so zu konfigurieren, dass das Git Repository automatisch – täglich oder wöchentlich – gemonitort wird und Pull-Requests automatisch erstellt werden.

Die Datenbasis

Ein Security Tool ist nur so gut wie die dahinter liegende Datenbasis. Viele Tools sind ungenau in dem Sinne, dass sie viele „False Positives" zurückliefern, das heißt angebliche Schwachstellen, die tatsächlich keine sind. Im Falle von Snyk werden Schwachstellen aus den unterschiedlichsten Quellen immer nochmal manuell durch ein Security Research Team verifiziert, bevor sie letztlich in die eigene Datenbank Einzug halten.

Auf welcher Datenbasis basieren die gefundenen Schwachstellen? Die Datenbasis, auf der die Snyk-Schwachstellendatenbank *[4]* basiert, ist auf fünf Säulen aufgebaut:

1. Angereicherte Daten aus zahlreichen Schwachstellen-Datenbanken wie der CVE, NVD und mehr. Die aus diesen Ressourcen gewonnenen Daten werden analysiert, getestet und angereichert, bevor sie in die Snyk-Datenbank aufgenommen werden.
2. Proprietäre Forschung nach neuen Schwachstellen: Das Snyk Research Team arbeitet daran, schwerwiegende Schwachstellen in Schlüsselkomponenten aufzudecken – so wurden 2019 beispielsweise 54 Zero-Day-Schwachstellen entdeckt.
3. Threat-Intelligence-Systeme: Diese scannen Security-Bulletins, Jira-Boards und GitHub-Commits, um automatisch Schwachstellen zu identifizieren, die noch nicht gemeldet wurden.
4. Beziehung zur Community: Snyk arbeitet mit der Community zusammen und betreibt Bug Bounties.
5. Zusammenarbeit mit der akademischen Welt: Das Team arbeitet mit akademischen Einrichtungen wie Berkeley, Virginia Tech und Waterloo zusammen, um Tools, Methoden und Daten auszutauschen. Die Ergebnisse werden dann exklusiv von Snyk veröffentlicht.

Die hier gezeigten Projektbeispiele verwenden Java und Maven. Snyk unterstützt darüber hinaus viele weitere Sprachen (Java, Node, Python, .NET, Go, Ruby, Kotlin, PHP) beziehungsweise deren Package Manager.

Container Security

Das Verpacken einer Java-Anwendung in einen Container ermöglicht es, die komplette Anwendung, einschließlich der JRE, Konfigurationseinstellungen sowie Abhängigkeiten auf Betriebssystemebene und die Build-Artefakte in eigenständigen, bereitstellbaren Artefakten, den sogenannten Container-Images, zu definieren. Diese Images werden in Form von Code (zum Beispiel Dockerfiles) definiert, was eine vollständige Wiederholbarkeit bei ihrer Erstellung ermöglicht und Entwicklern die Möglichkeit gibt, dieselbe Plattform in allen Umgebungen auszuführen. Schließlich können Entwickler mit Containern einfacher mit neuen Plattformversionen oder anderen Änderungen direkt auf ihren Desktops experimentieren, ohne dass spezielle Berechtigungen erforderlich sind.

Wer seine Applikation im Docker-Container ausliefert, definiert diesen im Dockerfile und mit entsprechender Auswahl eines geeigneten Base-Image. Nur, welches Base-Image ist am geeignetsten und sichersten?

Das richtige Docker-Image wählen

Wenn wir ein Docker-Image erstellen, erstellen wir dieses Image auf der Grundlage eines Image, das wir aus Docker Hub ziehen. Dies nennen wir das Base-Image. Das Base-Image ist die Grundlage für das neue Image, das wir für unsere Java-Anwendung erstellen wollen. Das Basis-Image, das wir wählen, ist wichtig, weil es uns erlaubt, alles zu nutzen, was in diesem Image verfügbar ist. Dies hat jedoch seinen Preis – wenn ein Basis-Image eine Sicherheitslücke aufweist, erben wir diese in unserem neu erstellten Image.

Bei der Erstellung eines Docker-Image sollten wir nur die nötigsten Ressourcen zuweisen, sodass die Applikation korrekt funktioniert.

Das bedeutet, dass wir zunächst eine geeignete Java-Laufzeitumgebung (JRE) für unser Produktions-Image verwenden sollten und nicht das komplette Java Development Kit (JDK). Darüber hinaus sollte unser Produktions-Image kein Build-System wie Maven oder Gradle enthalten. Das Produkt eines Builds, zum Beispiel eine jar-Datei, sollte ausreichen.

Auch wenn wir unsere Anwendung innerhalb eines Docker-Containers bauen möchten, können wir unser Build-Image mithilfe eines mehrstufigen Builds leicht von unserem Produktions-Image trennen.

Ein Beispiel: Ich möchte ein Docker-Java-Image für meine Java-Anwendung erstellen. Es handelt sich um eine Spring-Boot-basierte Anwendung, die mit Maven gebaut wurde und Java Version 8 benötigt. Der naive Weg, dieses Docker-Java-Image zu erstellen, wäre, das Basis-Image `maven:3-openjdk-8` zu wählen.

Mithilfe von Snyk können Schwachstellen in Docker-Images mittels CLI-Befehl *(siehe Listing 4)* einfach gefunden werden. In diesem Beispiel wollen wir das `maven:3-openjdk-8`-Image testen.

Der Scan *(siehe Abbildung 10)* zeigt uns, dass das besagte Image insgesamt 118 Schwachstellen beinhaltet, die durch 179 Abhängigkeiten Einzug gehalten haben. Der Report zeigt, welche System-Library in welcher Version von einer Schwachstelle betroffen ist, und im Falle eines möglichen Fix die jeweilige Version, auf die die Library upgegradet werden sollte.

Wenn wir hingegen auf ein schmaleres Alpine-Image wechseln, ohne Maven, und auch nur auf die JRE statt auf das JDK setzen, sehen wir, dass dieses Image lediglich 17 Abhängigkeiten beinhaltet und keine bekannten Schwachstellen enthält *(siehe Listing 5 und Abbildung 11)*.

```
$ snyk container test maven:3-openjdk-8
```

Listing 4: Snyk-Test des Docker-Image OpenJDK 8

```
snyk container test adoptopenjdk/openjdk8:alpine-jre
```

Listing 5: Snyk-Test des Docker-Image `adoptopenjdk/openjdk8:alpine-jre`

```
✗ High severity vulnerability found in gcc-8/libstdc++6
  Description: Information Exposure
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-GCC8-347558
  Introduced through: gcc-8/libstdc++6@8.3.0-6, apt@1.8.2.2, meta-common-packages@meta
  From: gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > apt/libapt-pkg5.0@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  and 2 more...

✗ High severity vulnerability found in gcc-8/libstdc++6
  Description: Insufficient Entropy
  Info: https://snyk.io/vuln/SNYK-DEBIAN10-GCC8-469413
  Introduced through: gcc-8/libstdc++6@8.3.0-6, apt@1.8.2.2, meta-common-packages@meta
  From: gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  From: apt@1.8.2.2 > apt/libapt-pkg5.0@1.8.2.2 > gcc-8/libstdc++6@8.3.0-6
  and 2 more...


Organization:      mathias.conradt-axp
Package manager:   deb
Project name:      docker-image|maven
Docker image:      maven:3-openjdk-8
Platform:          linux/amd64
Licenses:          enabled

Tested 179 dependencies for known issues, found 100 issues.
```

Abbildung 10: Ergebnisse des Image-Scans von `maven:3-openjdk-8` (© Snyk)

```
└─$ snyk container test adoptopenjdk/openjdk8:alpine-jre

Testing adoptopenjdk/openjdk8:alpine-jre...

Organization:      mathias.conradt-axp
Package manager:   apk
Project name:      docker-image|adoptopenjdk/openjdk8
Docker image:      adoptopenjdk/openjdk8:alpine-jre
Platform:          linux/amd64
Licenses:          enabled

✓ Tested 17 dependencies for known issues, no vulnerable paths found.
```

Abbildung 11: Ergebnisse des Image-Scans von `adoptopenjdk/openjdk8:alpine-jre` (© Snyk)

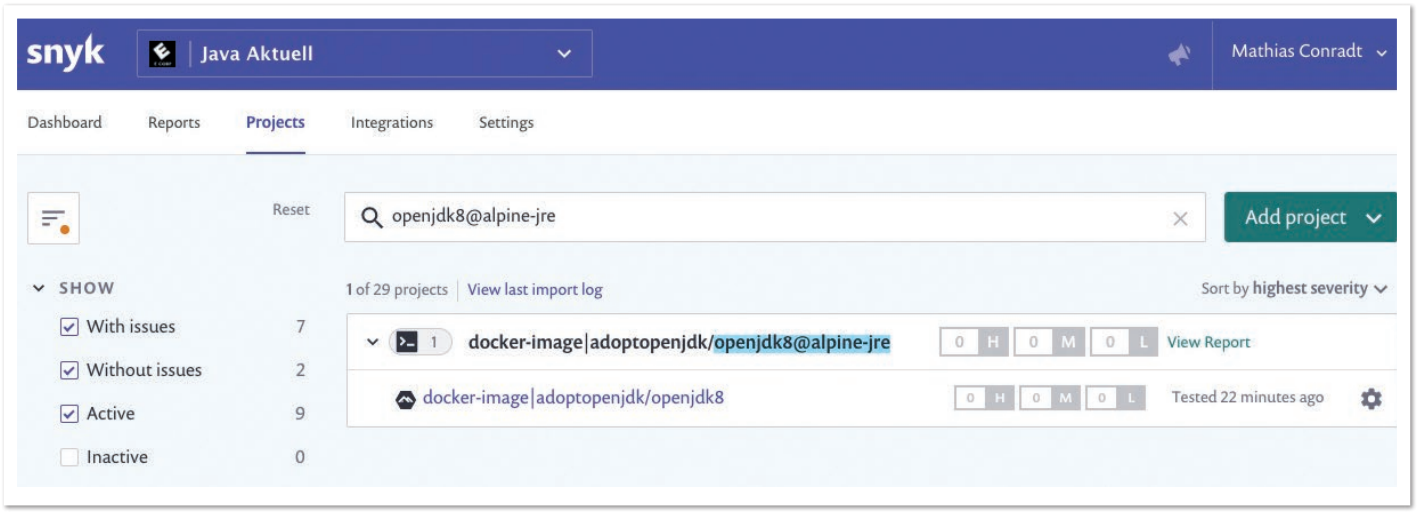


Abbildung 12: Snyk-Monitoring des Docker-Image `adoptopenjdk/openjdk8:alpine-jre` (© Snyk)

```
snyk container monitor adoptopenjdk/openjdk8:alpine-jre
```

Listing 6: Snyk-Monitoring des Docker-Image `adoptopenjdk/openjdk8:alpine-jre`

Das Docker-Image auch langfristig über die Snyk WebUI zu monitoren und bei neu gefundenen Schwachstellen informiert zu werden, ist mit dem `monitor`-Befehl in der Snyk-CLI *(siehe Listing 6 und Abbildung 12)* möglich.

Zusammenfassung

Bei Cloud-nativen Applikationen kommt der Entwickler auf mehreren Ebenen mit Code- und Open-Source-Abhängigkeiten in Berührung, sowohl bei der Applikation und deren direkten Open-Source-Dependencies selbst als auch bei den Open-Source-basierten System-Libraries im verwendeten Container-Image, in dem sie ausgeliefert wird. Diese beinhalten oftmals viele Sicherheits-schwachstellen, vor allem in älteren Versionen.

Snyk bietet hier eine kostenlose, entwicklerfreundliche Möglichkeit, all diese Komponenten entlang des gesamten DevOps-Prozesses auf Sicherheitslücken zu prüfen und diese möglichst früh zu beheben.

Referenzen

[1] *https://support.snyk.io/hc/en-us/articles/360003812538-Install-the-Snyk-CLI*
[2] *https://snyk.io/blog/snyk-cli-cheat-sheet/*
[3] *https://support.snyk.io/hc/en-us/sections/360001138118-IDE-tools*
[4] *https://snyk.io/vuln*

Mathias Conradt
Snyk
mathias.conradt@snyk.io

Mathias Conradt ist Sr. Solutions Engineer bei Snyk, einer „Cloud-native Application Security"-Plattform. Er verfügt über mehr als 20 Jahre Berufserfahrung im Bereich Software-Engineering und IT-Projektmanagement und besitzt diverse Zertifizierungen. Sein derzeitiger Schwerpunkt liegt auf Cybersecurity im Allgemeinen, wobei er sich auf Open Source Security spezialisiert hat. Mathias ist regelmäßiger Speaker bei diversen Security-bezogenen Branchenveranstaltungen.

Bin ich eine Pfeife, wenn ich kein Experte bin?

Anika Zohren

Nach fast 20 Jahren in der Softwareentwicklung fragte ich mich, warum manche Softwareentwickler Experten werden und andere wiederum nicht. Und was einen Experten eigentlich ausmacht. Und was man ist, wenn kein Experte aus einem wird. Manchmal hilft ja fragen. Doch in diesem Fall konnte mir kein Softwareentwickler so recht eine Antwort darauf geben.

Zu dem finalen Entschluss, mich kritisch mit dem Thema „Experten" auseinanderzusetzen, brachte mich dann ein Arbeitskollege, den ich sehr schätze und der mich schon oft mit seinem tiefen Fachwissen unterstützt hat. Er sagte etwas Ähnliches wie: „Ich bin hier der Experte zu dem Thema. Ich muss nicht nett zu den Leuten sein. Die kommen eh immer wieder, wenn sie was brauchen."

Wann ist man ein Experte?

Zu Beginn beschäftigte mich die Frage, was einen Experten eigentlich ausmacht. Und an wen denke ich, wenn ich an Experten denke? Natürlich irgendwie an Leonard, Sheldon, Rajesh, Howard, Bernadette und Amy von The Big Bang Theory *[1]*. Ihre nerdig verschrobene Art in zwölf Staffeln hat ein Millionenpublikum unterhalten. Gerne denke ich an die allererste Folge zurück, in der Leonard einer Dame an der Rezeption das gesamte Kreuzworträtsel vorsagt, in der Hoffnung, dann eher an die Reihe zu kommen.

Für mich ist diese Szene eine gute Definition von Expertentum. Ein Mensch, der ein komplexes Problem mit Wissen löst, das er nebenbei aus dem Ärmel schüttelt, wofür andere Stunden, Tage oder Monate brauchen oder gar ganz daran scheitern. Experten sind Menschen, die mehr als nur ein tiefes Verständnis für ein Thema haben und sogar bis an die Randgebiete vordringen. Die den Kern eines Problems ergründen können, um dieses vollständig theoretisch zu lösen, ohne ausprobieren oder testen zu müssen. Menschen, die genau und schnell, mit einem hohen Maß an Selbstdisziplin und mit niedriger Fehlerquote arbeiten. Experten haben keine fachlichen Schwächen.

Ein Experte in der Softwareentwicklung ist für mich jemand, der zum Beispiel ein ganzes Computerprogramm ohne Aufzeichnung theoretisch entwerfen kann, natürlich direkt mit den optimalen Lösungsansätzen, und der dieses Programm dann in einem Stück herunterschreibt mit dem Effekt, dass es beim ersten Test direkt fehlerfrei läuft. Und auch SonarQube *[2]* mit seinem Sonar Way Quality Profile würde keinen einzigen Fehler anzeigen.

Solche Experten und jene, die danach streben, sind mir im Laufe der Zeit begegnet. Doch meiner Wahrnehmung nach entspricht eine Mehrheit nicht dieser Definition und strebt auch nicht danach.

Was gibt es also noch? Pfeifen

Der Begriff „Pfeife" erregte bei mir durch die auf YouTube verfügbaren Videos von Profilerin Suzanne *[3]* Aufmerksamkeit. Ihrer Definition nach handelt es sich bei Pfeifen um Personen, die mehr Energie darauf verwenden, einen Eindruck zu vermitteln, als tatsächlich Werte oder Ergebnisse zu schaffen. Diese Eindrucks-Manager stellen Behauptungen auf und betreiben hauptsächlich Fachwort-Dropping, ohne ein tieferes Verständnis zu einem Thema zu besitzen.

Mich erinnerte die Beschreibung einer Pfeife von Suzanne Grieger-Langer ein bisschen an die Wikipedia-Definition der Phase der „unbewussten Inkompetenz" aus dem Modell der Kompetenzstufenentwicklung *[4]*.

Wikipedia, den 21.02.2021: *Unbewusste Inkompetenz: Mangels Anreizen versteht das Individuum nicht, worum es geht, oder weiß nicht, wie etwas bewirkt werden soll; ebenso erkennt es seine eigenen Defizite nicht oder hat ein Problem, sie zu erkennen. Für die Tendenz, sich trotz Unkenntnis als kompetent einzuschätzen, hat sich populärwissenschaftlich der Begriff Dunning-Kruger-Effekt eingebürgert. Personen mit unbewusster Inkompetenz handeln auch intuitiv falsch.*

So wenig ich diese Definition im Allgemeinen schätze, so sehr passt sie meiner Ansicht nach als Beschreibung für die „Pfeife". In dem Modell der Kompetenzstufenentwicklung ist diese Stufe die erste von vier, die man angeblich auf dem Weg zur Kompetenz immer durchläuft. Die Pfeife scheint jedoch in dieser ersten Phase dauerhaft zu verweilen und sich völlig darin zu verweigern, in die nächste Phase, die der „bewussten Inkompetenz", zu wechseln. Grieger-Langer beschreibt dies als Verweilen der Pfeife in einem „Ganzkörper-Kritik-Kondom".

In den letzten zwei Jahrzehnten musste ich oft mit ansehen, was Grieger-Langer auf den Punkt bringt: dass Pfeifen karrieretechnisch heutzutage sehr gut vorankommen, obwohl sie kein tiefes Wissen haben und viele fachliche Schwächen aufweisen. Sie kratzen nur an der Oberfläche, solange sich die anderen von schönen, schlauen Worten überzeugen lassen und nicht genau hinschauen. Wenn irgendwann auffällt, dass die Substanz fehlt, ist die Pfeife längst befördert worden oder kann die Verantwortung auf jemand anderen abschieben.

Doch meiner Erfahrung nach sind nicht alle Softwareentwickler gleich Pfeifen, nur weil sie keine Experten werden wollen. Das Pfeifentum scheint mir immer noch eine Randerscheinung zu sein. Es kommt einem nur oft größer vor, weil eine einzige Pfeife so viele Menschen in ihrer Umgebung unglücklich macht.

Ist ein Team aus Experten eine gute Idee?

Das vorhin erwähnte Modell der Kompetenzstufenentwicklung sieht das Expertentum scheinbar als Ziel der Entwicklung zur Kompetenz – die sogenannte „unbewusste Kompetenz". Die Beschreibung dieser Phase bei Wikipedia scheint mir zutreffend, jedoch wenig erstrebenswert.

Wikipedia 21.02.2021: *Das Individuum hat so viel praktische Erfahrung mit seinen Fähigkeiten, dass sie ihm in Fleisch und Blut übergehen und jederzeit abgerufen werden können, oftmals ohne höhere Konzentration in Anspruch nehmen zu müssen. Diese Person kann ihre Fähigkeiten, da sie sich ihrer nicht bewusst ist, nicht mehr problemlos weitervermitteln. Mit unbewusster Kompetenz handeln die Menschen zwar intuitiv richtig, können ihr Handeln aber nicht mehr analysieren.*

In einem WDR5-Radiobeitrag *[5]* hörte ich, dass die NASA sich fragt, ob eine Gruppe aus Experten ein gutes Team bilden kann. Es wird dort von einem Projekt berichtet, das die Berechenbarkeit sozialer Interaktion innerhalb einer Gruppe über einen längeren Zeitraum erforscht. Natürlich war der Hintergrund die anstehende Reise zum Mars. Immerhin muss dabei ein Team aus Experten ein Jahr lang auf engstem Raum zu einem fremden Planeten reisen, dort die geplanten Arbeiten und Experimente durchführen und danach wieder ein Jahr lang zurückfliegen. „Und was die Crewmitglieder in der ersten Woche noch unterhaltsam und charmant finden, zum Beispiel wenn jemand ein Musikinstrument spielt, finden sie in der letzten Woche vielleicht so unerträglich, dass sie denjenigen am liebsten ermorden würden", wird aus dem Team um Professor Noshir Contractor *[6]* als Begründung für die Notwendigkeit solch eines Projekts angegeben.

Das ist schon nachvollziehbar, denn Astronauten werden nicht für eine Mars-Mission vorgeschlagen, weil sie gute Teamplayer, sondern weil sie gute Kommandanten, Piloten, Chemiker, Biologen, Raumfahrt-Techniker oder Ingenieure sind.

Die Softwareentwicklung ist zwar nur schwer mit einer Reise zum Mars zu vergleichen, jedoch verbringt man auch hier sehr viel Zeit mit einem Team. Manchmal über viele Jahre hinweg.

Auf heise online wurde 2019 ein Artikel von Justin Vaughan-Brown *[7]* veröffentlicht, der sich mit der Frage beschäftigt, ob sich DevOps mit der deutschen Arbeitskultur verträgt. Auf Platz eins seiner Liste der „potenziellen Reibungspunkte" steht der Hang zum Expertentum. Er führt weiter aus, dass sich Experten in agilen Teams wenig zu Hause fühlen.

Wenn ich wieder an The Big Bang Theory denke, dann erweckt es den Eindruck, dass sich Experten in keiner Art von Team richtig wohl fühlen. In beinahe jeder Folge gibt es eine Situation, in der jemand – meist Sheldon – nicht mit den anderen kooperieren möchte und auf seinen Experten-Status besteht. Natürlich ist dies überspitzt dargestellt. Doch ich glaube, es trifft auf einen wahren Kern.

Auch bei Jeff Sutherland *[8]* hörte ich so etwas heraus, denn er schreibt in seinem Buch mit dem Titel *„Scrum: The Art of Doing Twice the Work in Half the Time" [9]*, von mir frei übersetzt: „Keine Heldentaten. Wenn du einen Helden benötigst, um Dinge erledigt zu bekommen, hast du ein Problem. Außergewöhnlich heldenhafter Einsatz sollte als Versagen der Planung betrachtet werden."

Um erfolgreich als Softwareentwickler in einem agilen Team zu arbeiten, scheint es also nicht erstrebenswert zu sein, sich zu einem Experten zu entwickeln. Doch zu was dann?

Die bewussten Performer

Für Suzanne Grieger-Langer ist das Gegenteil einer Pfeife nicht der Experte. Sie verwendet den Begriff „Performer", ein positiv belegtes Wort im Sinne von „Leistender" oder „Durchführender". Also all diejenigen Menschen, die Leistung erbringen beziehungsweise etwas erschaffen oder mitgestalten, das eine gewisse Substanz aufweist.

Nach dem bereits erwähnten Modell der Kompetenzentwicklung (über das ich mich an anderer Stelle auch noch einmal ausführlich auslasse *[10]*) liegen zwischen der unbewussten Inkompetenz (Pfeifen) und der unbewussten Kompetenz (Experten) die Phasen der bewussten Inkompetenz und der bewussten Kompetenz – die „bewussten" Phasen.

Bezogen auf die Softwareentwicklung sah ich darin die ganz normalen Junior- und Senior-Softwareentwickler. Entwickler, die er-

klären können, was sie wissen und was sie nicht wissen. Die ihre Tätigkeiten dokumentieren (könnten, wenn sie wollten), Wissen weitergeben beziehungsweise Fragen zu fehlendem Wissen formulieren können.

Ich nannte die Gruppe der Nicht-Experten und Nicht-Pfeifen ab diesem Zeitpunkt also bewusste Performer und meiner Beobachtung nach sind die meisten Entwickler, die ich kennengelernt habe, mit diesem Begriff passend beschrieben. Es scheint also erstrebenswert, als Softwareentwickler ein solcher bewusster Performer zu sein. Diese haben zwar in Randbereichen oder Detail-Themen fachliche Schwächen, können sie jedoch erkennen und bei Bedarf füllen.

Stereotype innerhalb der bewussten Performer

Vera F. Birkenbihl *[11]* hat jahrzehntelang menschliche Verhaltensweisen analysiert, um dann den Menschen ihre eigenen unbewussten Verhaltensweisen zu erklären, damit sie diese besser und bewusster zu ihrem eigenen Vorteil anwenden können. Ein Beispiel ist das intuitive Aufteilen von unüberschaubar großen Gruppen in kleine, besser überschaubare Gruppen – eine natürliche, unbewusste Handlung. Wenn dem Menschen das jedoch bewusst ist, kann es beispielsweise beim Lernen eingesetzt werden.

Also versuchte ich, mir meine unbewusst erstellten Gruppen ins Bewusstsein zu rufen und zu benennen. Interessanterweise hatte ich Softwareentwickler nach auffälligen Stärken und Schwächen gruppiert. Natürlich passte in diese Stereotype nicht jeder hinein, der mir je begegnete. Oft traf ich auch auf Menschen, die mehrere Stereotype vereinten. Der erste Versuch einer bewussten Formulierung ergab die nachfolgenden Gruppen.

Der Perfektionist

Der Perfektionist macht das Thema, an dem er arbeitet, zur persönlichen Herausforderung. Er geht nicht nur einen, sondern drei Extraschritte, um sein Ziel zu erreichen. Er vergisst sogar, seine Überstunden aufzuschreiben, weil es ihm ein persönliches Anliegen ist, ein perfektes Ergebnis abzuliefern.

Ich meine damit nicht die Entwickler, die mehr als die oft mit Managern diskutierten 80 Prozent des Pareto-Prinzips *[12]* erreichen wollen. Ich meine genau diejenigen, die auch das letzte ein Prozent noch fertig bekommen möchten.

Leider ist der Perfektionist permanent unzufrieden. Denn es findet sich immer etwas, das nicht perfekt ist (und seien es Leerzeichen statt Tabs in der Formatierung). Und auch so etwas wie Coding Conventions *[13]* basiert meistens auf einem Kompromiss unter allen Entwicklern, sodass der Perfektionist nicht immer so programmieren kann, wie er gerade will.

Der motivierte Beamte

Der motivierte Beamte arbeitet sehr effizient und kann Probleme auch pragmatisch lösen, bevor sie zu lange dauern. Er verschwendet keine Zeit. Er versucht, jede Aufgabe so schnell wie möglich zu erledigen. Wenn man ihn lässt, fängt er um 6:00 Uhr morgens an und arbeitet die Mittagspause durch.

Jedoch bestehen Dokumentationen bei ihm üblicherweise aus einem einleitenden Satz und dann kopiertem Quelltext. Er organisiert seine Arbeit so, dass er keine Minute zu spät in den Feierabend geht. Er meidet Konfrontationen und einen Extraschritt zu machen kostet ihn Überwindung.

Der Macher

Der Macher ist hoch motiviert und sehr ergebnisorientiert. Er treibt nicht nur sich selbst, sondern auch die anderen zu Höchstleistungen an. Er zelebriert das Erreichen von Zielen.

Jedoch ist der Macher nur ein Liebling all derer, die nicht so genau auf die Details schauen, denn viel mehr als die 80 Prozent des Pareto Prinzips macht er nicht. Er will immer gewinnen und hält sich mit Unit-Tests, Dokumentationen, Erklärungen oder Hilfe für andere nur auf, wenn er muss. Er geht genauso wenig rücksichtsvoll mit sich selbst wie mit anderen um.

Der Unterhaltungskünstler

Der Unterhaltungskünstler ist sich für keine Blödelei zu schade. Gerade bei Stresssituationen im Team schafft er es, wie von Zauberhand wieder eine gute Stimmung herzustellen. Mit einem Unterhaltungskünstler im Team läuft niemand Gefahr, sich auf Dauer zu ernst zu nehmen, denn er schafft es, Kritik unter dem Deckmantel des Humors gezielt anzubringen, ohne jemanden vor den Kopf zu stoßen.

Leider übertreibt er aber auch manchmal und hält die anderen längere Zeit von der Arbeit ab. Ein guter Witz ist ihm wichtiger als seine Aufgaben, weshalb er gerne auch mal länger benötigt, um diese zu erledigen.

Der Organisator

Der Organisator arbeitet aktiv am Teamzusammenhalt, indem er gemeinsame Mittagessen oder Team-Events vorschlägt und dann gleich auch organisiert. Er kümmert sich meistens um die übergreifenden Themen, macht Vorschläge zur Branching-Strategie *[14]* oder Definition of Done *[15]*.

Jedoch ist er oftmals so sehr mit diesen Aufgaben beschäftigt, dass seine Entwicklungsaufgaben zu kurz kommen oder nicht fertig werden. Er lässt sich auch sehr von negativer Stimmung im Team beeinflussen.

Stereotype und -tussis

Egal wie groß die Firma war, in der ich arbeitete, es fanden sich selten viele Entwicklerinnen dort. Eine bis zwei, wenn überhaupt. Da man bei Stereotypen viele Beispiele benötigt, um klar hervorstechende Ähnlichkeiten zu finden, konnte ich bisher keinen „typischen weiblichen" Stereotype finden. Die oben erwähnten Stereotype trafen auch immer auf die Softwareentwicklerinnen zu – aber gerne mit einem weiblichen Augenzwinkern.

Das schönste Auftreten eines Unterhaltungskünstler-Stereotyps erlebte ich bei einer Softwareentwicklerin, die viele Jahre lang die einzige Frau im Team war. Die Stimmung war gerade recht schlecht und jeder murrte an seinem Platz vor sich hin, da der Projektleiter eine für Entwickler schwer nachvollziehbare Entscheidung getroffen hatte, die sich nun auf ihre Arbeit auswirke. Sie stand auf, setzte sich auf einen der gemütlichen Sessel und sagte: „So, ihr Lieben, Mutti erzählt euch jetzt ein Märchen". Alle standen auf und setzten sich

wie kleine Kinder vor sie auf den Boden und schauten sie erwartungsvoll an. Sie erzählte aus dem letzten Meeting mit diesem Kunden und dem Projektleiter und wie es zu der Entscheidung gekommen war. Jedoch in Märchenform mit „Es war einmal…" Es wurde viel gelacht und als alle wieder zurück an ihre Plätze gingen, war die Stimmung wieder gut.

In vielen Köpfen sehe ich gerade in der Softwareentwicklung immer noch das Bild verankert, dass Frauen und Männer in allem gleich sein müssen. Ich sehe das eher wie Birkenbihl, die sagt „Frauen und Männer sind gleichwertig, aber sicherlich nicht gleich" *[16]*. Und mir scheint, es ist langsam angekommen, dass Teams, die aus Männern und Frauen bestehen, wirkungsvoller arbeiten. Sei es in Kindergärten oder in der Softwareentwicklung.

Unterschiede sind Bereicherungen

Es ist also gar nicht schlimm, sich nicht zu einem Experten zu entwickeln und individueller Teil einer Menge bewusster Performer zu sein. Und die meisten Softwareentwickler scheinen dies unbewusst auch schon zu wissen. Doch woher kommt dann dieser Impuls zum Expertentum? Zur permanenten Selbstoptimierung und Vertuschen von Fehlern oder fehlendem Fachwissen?

Denn es gibt auch keinen „perfekten" bewussten Performer. Denn der Macher bremst den Perfektionisten und sorgt dafür, dass etwas abgeliefert wird. Der Unterhaltungskünstler setzt sich dafür ein, dass sich die daraus resultierenden Spannungen wieder legen. Der motivierte Beamte arbeitet die Routine-Fälle ab und holt die Zeit wieder rein, die der Organisator für die Planung des nächsten Team-Events vertrödelt hat. Die Frauen im Team sorgen dafür, dass die männlichen Kollegen nicht tagelang allein über einem Problem brüten, bevor sie darüber sprechen. Und die Männer im Team tun so, als würden sie den Frauen zuhören, während diese beim Beschreiben des Problems selbst auf die Lösung kommen.

Es kommt nicht darauf an, dass jeder nur Stärken und keine Schwächen besitzt, sondern dass sich ein Team gut in seinen Stärken und Schwächen ergänzt. Das Streben nach Expertentum sollte den Bereichen vorbehalten bleiben, in denen weder Teamarbeit noch zwischenmenschliche Fähigkeiten benötigt werden. Warum so viele Firmen heutzutage immer noch Softwareentwicklungs-Experten für die Arbeit in Scrum-Teams suchen, ist mir ein großes Rätsel. Warum sie sich von Zertifikaten und Expertenwissen beeindrucken lassen, um dann eine rollenunabhängige, themenflexible Tätigkeit im Arbeitsalltag zu verlangen, löst bei mir Unverständnis aus.

Ein Team ist nicht einfach die Summe seiner Mitglieder in einer Excel-Tabelle ihrer Kenntnisse. Das Betrachten von Menschen als Ressourcen und das Ausklammern der menschlichen Besonderheiten jedes Einzelnen führt lediglich zu Teams, die nicht das gewünschte Ergebnis liefern und gerade unter Stress menschlich extrem reagieren könnten.

Die NASA hat dies in der Aufarbeitung des Mission-Skylab-4-Zwischenfalls von 1973 vielleicht schon geahnt und bereitet die Mars-Mission und die Auswahl des Teams auch unter Berücksichtigung zwischenmenschlicher Verhaltensweisen vor. Warum sollten wir in der Softwareentwicklung dies nicht auch bei der Zusammenstellung der Teams tun?

Referenzen

[1] *https://de.wikipedia.org/wiki/The_Big_Bang_Theory*
[2] *https://www.sonarqube.org*
[3] *https://profilersuzanne.com/*
[4] *https://de.wikipedia.org/wiki/Kompetenzstufenentwicklung*
[5] *https://www1.wdr.de/mediathek/audio/wdr5/quarks/topthemen-aus-der-wissenschaft/audio-teamsuche-fuer-den-mars-100.html*
[6] *https://sonic.northwestern.edu/home/people/noshir-contractor/*
[7] *https://www.heise.de/developer/artikel/Vertraegt-sich-DevOps-mit-der-deutschen-Arbeitskultur-4320139.html*
[8] *https://de.wikipedia.org/wiki/Jeff_Sutherland*
[9] ISBN-13 : 978-1847941107
[10] *https://cyberland.ijug.eu/sessions/session-24.html/*
[11] *https://de.wikipedia.org/wiki/Vera_F._Birkenbihl*
[12] *https://de.wikipedia.org/wiki/Paretoprinzip*
[13] *https://en.wikipedia.org/wiki/Coding_conventions*
[14] *https://en.wikipedia.org/wiki/Branching_(version_control)*
[15] *https://en.wikipedia.org/wiki/Scrum_(software_development)#Definition_of_done_(DoD)*
[16] *https://www.youtube.com/watch?v=xnqtvLTrItw*

Anika Zohren

StayStrange42@gmail.com

Anika Zohren arbeitet seit fast 20 Jahren als Software-Entwicklerin (meist mit Java und Java-nahen Technologien). Ihr Steckenpferd ist die statische Quelltext-Analyse, sowohl manuelle als auch automatisierte (mit SonarQube). Sie hält unter dem Nickname StayStrange42 Vorträge und schreibt über ihre Erfahrungen und Beobachtungen in der Softwareentwicklung. Kontakt gerne über StayStrange42@gmail.com oder öffentlich über Twitter @StayStrange42.

Digital Leadership – Leistungserbringung und Motivation im virtuellen Team sicherstellen

Dr. Dominic Lindner, Agile Unternehmen

Trotz räumlicher Distanz die Zusammenarbeit mithilfe von Technologie zu ermöglichen, ist nicht erst seit der Corona-Krise ein zentrales Thema. Doch vor allem seit dem ersten Lockdown aufgrund von COVID-19 nahm die virtuelle Arbeit deutlich zu. Es liegt nun an Führungskräften, mithilfe von neuen Technologien und Arbeitsmethoden Teams virtuell effizient zu führen. Doch wie gelingt für Führungskräfte die Sicherstellung der Leistungserbringung und Motivation im virtuellen Team?

Unter einem virtuellen Team wird im Allgemeinen die Zusammenarbeit einer Gruppe von Personen auf Distanz mithilfe von Technologie verstanden (Lindner 2020). Im Rahmen der Sicherheitsmaßnahmen bedingt durch die COVID-19-Pandemie wurden Unternehmen gebeten, Mitarbeitende (virtuell) von zu Hause arbeiten zu lassen. Laut einer Studie des Instituts für Arbeitsmarkt und Berufsforschung *[1]* arbeiteten im April 2020 knapp 25 Prozent der deutschen Arbeitnehmer aufgrund von COVID-19 ausschließlich im Homeoffice und waren somit Teil eines virtuellen Teams. Vorher waren dies laut Schätzungen lediglich fünf Prozent.

Eine besondere Herausforderung ergibt sich in dieser neuen virtuellen Arbeit vor allem für Führungskräfte. Führung unterliegt seit jeher dem

Status	ID	Betreff	Besitzer	Datum	Letzter Kommentar
Neu	#54	Neue Datenbank	System	02.04.2020	neues Ticket
	#55	Neuer Server	System	03.04.2020	neues Ticket
In Bearbeitung	#50	100 neue User	Clara	27.03.2020	50/100 angelegt
	#53	SQL Dump ziehen	Michael	01.04.2020	Dump wird gezogen
Waiting	#51	Datenbank reset	Clara	15.03.2020	Freigabe von Kunden fehlt
	#50	Server 404	Anna	17.03.2020	warte auf Logs vom Kunden
Fertig	#52	Server Update	Michael	28.03.2020	erledigt
	#49	ownCloud upgrade	Clara	30.03.2020	erledigt

Abbildung 1: Abbildung eines Ticketsystems mit regelmäßigen Statusupdates (© Lindner 2020)

Umgang mit wechselnden Herausforderungen und sieht sich nun damit konfrontiert, virtuelle Teams mithilfe von Technologie effizient zu führen. Generell hat sich die Definition von Führung nicht verändert und wird weiterhin als die Sicherstellung von Leistungserbringung und Motivation von Mitarbeitenden verstanden, lediglich die Umsetzung hat sich verändert. Die Führung von Teams mithilfe von Technologie wird unter dem Begriff Digital Leadership in der Praxis diskutiert.

Digital Leadership ist eine Querschnittskompetenz sowie ein Sammelbegriff für verschiedene Methoden, Theorien und Werkzeuge, die die Führung und insbesondere die Führungskompetenz im digitalen Zeitalter beschreiben (Lindner 2019).

Schon die aktuelle Berichterstattung zu COVID-19 und den Konsequenzen für die Arbeitswelt zeigt, dass wenige Unternehmen wirklich auf virtuelle Arbeit vorbereitet sind (vgl. Handelsblatt 2020a *[2]*), viele Mitarbeitende keine Erfahrung haben, wie die Arbeit von zu Hause organisiert werden kann und fehlende Methodenkenntnis das Arbeiten von zu Hause erschwert (vgl. Handelsblatt 2020b *[3]*). Die Auswirkungen in der Praxis liegen auf der Hand: Führungskräfte und Mitarbeitende hetzen von Videokonferenz zu Videokonferenz und dokumentieren Arbeit in Softwaretools, die sie oftmals nur schwer bedienen können oder die umständlich sind. Angesichts dessen erscheint es kaum noch möglich, die Leistungserbringung und Motivation der Mitarbeitenden sicherzustellen.

Im Folgenden sollen Methoden und Organisation einer digitalen Führung (Digital Leadership) zur Sicherstellung der Leistungserbringung und anschließend Empfehlungen zur Sicherstellung der Motivation von Mitarbeitenden gegeben werden.

Digital Leadership – Sicherstellung der Leistungserbringung

Virtuelle Teams müssen auf Distanz mithilfe von Software geführt werden. Besonders die virtuelle Führung verlangt eine Abkehr von der Hands-on-Mentalität hin zu einer methodischen Führung. Es gilt, agile oder klassische Methoden sinnvoll im Team umzusetzen und als Führungskraft zu moderieren. Ich möchte im Folgenden dazu zwei Beispiele geben.

Klassische Methoden zur Steuerung virtueller Teams

Oftmals werden klassische Methoden in virtuellen Teams durch Ticketsysteme umgesetzt. Beispiele dafür sind Zendesk, OTRS oder OSTicket. Besonders eignet sich diese Art, wenn Aufgaben nicht proaktiv geplant werden, sondern auf bestimmte Ereignisse, wie beispielsweise eine Kundenanfrage, reagiert wird.

Es gilt, im virtuellen Team einen Überblick darüber zu behalten, welche Aufgaben aktuell durchgeführt werden und welche/r Mitarbeitende woran arbeitet. Sie können sich dies wie folgt vorstellen *(siehe Abbildung 1)*: Eine Aufgabe wird durch ein Ereignis ausgelöst und einem Systemnutzer/einer Systemnutzerin als Ticket zugewiesen. Anschließend verteilen Sie als Teamleiter/in die neuen Tickets nach Spezifikation an die Mitarbeitenden und überwachen die Ausführung. Ihre Aufgabe ist es, Hindernisse aus dem Weg zu räumen und die Dokumentation der Arbeit zu überwachen.

Meine Empfehlung ist, dass Sie sich als Teamleiter/in vor allem um neue Tickets kümmern und Hindernisse, die im Status „Blocker/warten" sind, übernehmen. Der Grund dafür ist, dass solche Tickets den Arbeitsfluss des Teams verlangsamen. Sie bemerken sicherlich einen Unterschied zur Präsensarbeit: Der zentrale Informationsknoten sind nicht mehr Sie, sondern das Software-Tool.

Agile Methoden zur Steuerung virtueller Teams

Agilität wird als die Fähigkeit eines Unternehmens definiert, schnell und flexibel möglichst in Echtzeit auf sich verändernde Marktbedingungen zu reagieren *[4]*. Agile Methoden sind unter anderem konkrete Abläufe und Meetingformate, die Unternehmen nach der genannten Definition agiler machen können.

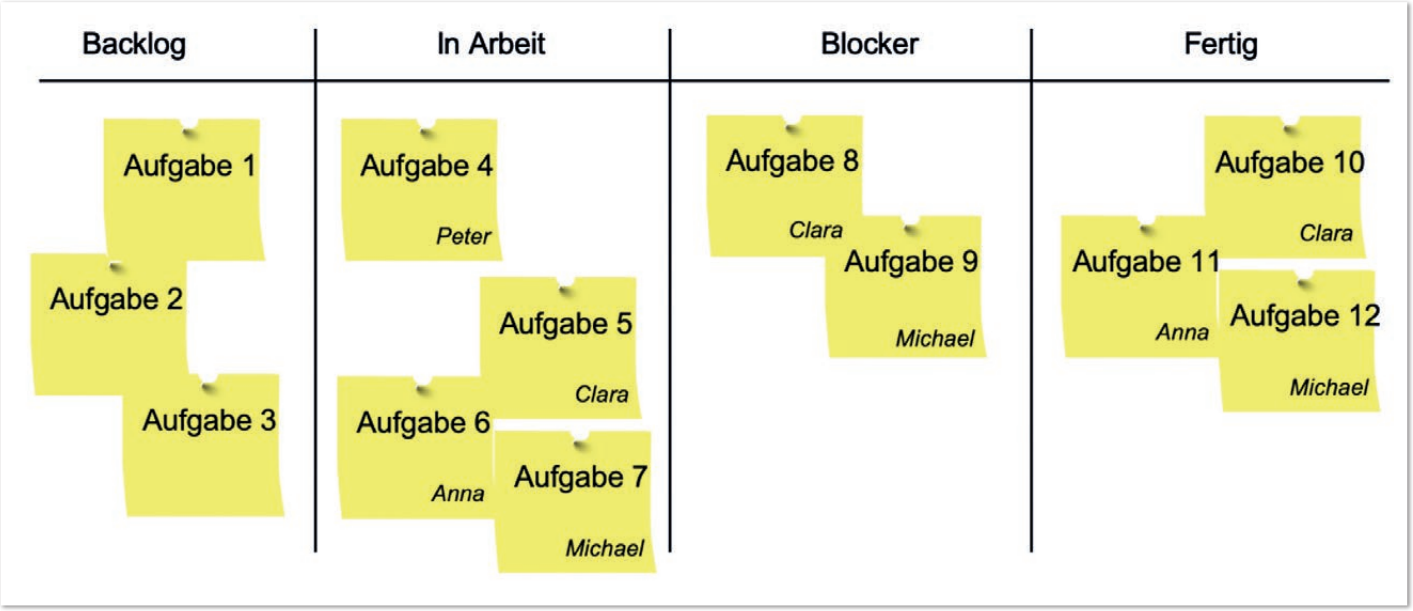


Abbildung 2: Kanban-Board für ein virtuelles Team

Agile Methoden haben viele Facetten und Möglichkeiten zur Umsetzung. Sicherlich sind nicht alle davon für jedes Unternehmen sinnvoll und ich gebe zu, dass ich einige Varianten sogar sehr esoterisch finde, aber die Grundidee von agilen Methoden ist wahrscheinlich in 99,9 Prozent aller Unternehmen sinnvoll. Diese Grundidee ist die Visualisierung komplexer Wissensarbeit und die Aufteilung der Arbeit an das Team.

Zur Visualisierung komplexer Wissensarbeit werden bei agilen Methoden wie Scrum und Kanban sogenannte Boards *(siehe Abbildung 2)* verwendet. Diese Grundidee sollten Sie für die Organisation von virtuellen Teams ebenfalls nutzen und die einzelnen Aufgaben inklusive Bearbeitende auf dem Board abbilden. Sie merken schnell: Das Board dient als Informationsknoten.

Sie finden auf dem Board alle Aufgaben, die das Team erledigen sollte. Es gilt, die Aufgaben sinnvoll zu beschreiben, eine Zielerreichung (Definition of Done) anzufertigen und die Aufgaben zu priorisieren. Dies kann mithilfe zahlreicher Softwaretools wie Jira, Kanboard, Redmine und MS Teams Agile Modul umgesetzt werden. Wie auch in klassischen Methoden sollte jede Aufgabe eine/n Bearbeitende/n haben und mit Status und Kommentar zum aktuellen Stand versehen sein.

Als Führungskraft sollten Sie am Board die Aufgabenverteilung und Erledigung moderieren. Hierzu führen Sie Meetings durch, um das Board zu organisieren und Informationen transparent im Team zu verteilen. Sie bemerken, dass schon ein kleiner Teil der Umsetzung agiler Methoden hilft, deutlich mehr Transparenz in ein Team zu bringen. Es liegt an Ihnen als Führungskraft zu identifizieren: Was bringt mir aus dem agilen Methodenkoffer einen Mehrwert und wie kann ich es in klaren, aber flexiblen Prozessen abbilden?

Digital Leadership – Sicherstellung der Motivation

Die Organisation von Arbeit mithilfe von Software stellt eine große Herausforderung für Führungskräfte dar. Die aktuelle Arbeitsweise muss oft vollkommen neu gedacht werden und Führungskräfte werden vermehrt zu Administratoren/innen von Software. Vor allem die Leistung der Mitarbeitenden wird nicht in persönlichen Gesprächen, sondern über ein anonymes Softwaretool geprüft. Auch Meetings finden nicht mehr bilateral statt, sondern dienen dazu, gemeinsam das Softwaretool zu organisieren. Es stellt sich die Frage, wo bei der vielen Software eigentlich der Mensch bleibt und wie die Motivation der Mitarbeitenden abseits der Softwaretools sichergestellt werden kann.

Sicherlich haben Sie schon viel über Führung gelesen. Aktuelle Management-Literatur redet von einem Wechsel des Führungsstils von Kommando und Kontrolle zu einer menschlichen und authentischen Führung. In der Präsenzarbeit wurde dieser Wechsel auch zunehmend in Unternehmen umgesetzt. Es veränderte sich viel in deutschen Chefetagen. Beispielsweise sorgte Ex-Daimler-Vorstand Dieter Zetsche für einen Skandal, als er die Krawatte bei Daimler abschaffte und bei einer Präsentation in einem T-Shirt mit der Aufschrift „Do Epic Shit" auftrat. Weiterhin stellen Manager wie Elon Musk (Tesla) und Valentin Stalf (N26) den Antityp zum grauhaarigen älteren Mann mit Krawatte und Anzug dar und brechen mit dem klassischen Managerbild.

Gute Führung bleibt auch im virtuellen Raum eine „Verhaltensweise" oder eine „persönliche Einstellung" (Lindner 2020). Führungskräfte sollten also in virtuellen Teams wie auch in der Präsenzarbeit an der eigenen Persönlichkeit arbeiten. Zusätzlich stellen bestimmte agile Arbeitsmethoden eine Brücke zwischen Distanzarbeit und menschlicher Nähe her. Die aktuellen Charaktereigenschaften und Methoden von Digital Leadership wurden von Tobias Greff und mir 2019 in einer Studie mit über 60 Führungskräften virtueller Teams genauer untersucht, in der diese wichtigsten Ergebnisse zu finden sind *[5]*:

- **Konkrete Ziele:** Es wird empfohlen, virtuelle Teams über Ziele zu steuern. Regelmäßige Statusmeldungen sind als Instrument zur Zielkorrektur sinnvoll.
- **Klare Rollenverteilung:** In einem virtuellen Team können eine stabile Rollenverteilung sowie das Arbeiten nach Pull-Prinzip eine Führungskraft entlasten.

- **Mitarbeitenden vertrauen:** Virtuelle Führung basiert laut den Teilnehmenden der Studie auf Vertrauen. Es kann davon ausgegangen werden, dass die delegierten Arbeitspakete mit steigendem Vertrauen umfangreicher werden.
- **Authentisches Auftreten:** Es gibt das einheitliche Verständnis, dass eine Führungskraft nicht perfekt sein muss, Fehler zugeben darf oder einen „schlechten Tag" haben kann.
- **Primärcharakteristika** sind Führung auf Augenhöhe, Vertrauen in Mitarbeitende, Experimente, Echtzeitinformationen und Inspiration fördern.
- **Sekundärcharakteristika** sind Echtzeitfeedback, Agilität vorleben, Partizipation und zur Veränderung motivieren.
- **Neue Kompetenzen** sind neue Trends antizipieren, agile Methoden und neue Arbeitskonzepte einsetzen, die Motivation der Mitarbeitenden sowie Technologien evaluieren.
- **Learning by Doing:** Da Learning by Doing für Führungskräfte die am weitesten verbreitete Lernmethode darstellt, wird geraten, das Führungsleitbild direkt an den Arbeitsalltag anzupassen. Experimentierfreude ist gefragt!

Wenn Mitarbeitende keine Ziele haben, können sie sich niemals verbessern. Es gilt deswegen, Mitarbeitenden besonders in virtuellen Teams klare Ziele zu setzen. Sie zeigen ihnen stets Perspektiven und geben Sicherheit.

Ein weiterer Punkt, der die Sicherheit von Mitarbeitenden verstärkt, ist eine klare Rollenverteilung. Vermitteln Sie jedem Teammitglied, dass es wichtig für das Team ist und respektieren Sie die Rollen. Verweisen Sie bei wichtigen Entscheidungen auf die jeweilige Rolle und beziehen Sie diese mit ein. Fördern Sie jeden Tag aktiv die Identifikation mit der Rolle.

Es ist fast unmöglich, alle einzelnen Mitarbeitenden zu steuern, weshalb die Steuerung über Gruppenziele und über die Software (Taskboard) zu bevorzugen ist. Lernen Sie, Ihren Mitarbeitenden zu vertrauen. Weiterhin ist es wichtig, am eigenen Charakter zu arbeiten. Lernen Sie, authentisch zu sein und auch Fehler zuzugeben oder mal einen „schlechten Tag" zu haben. Auch Zuhören ist wichtig, um das Vertrauen von Mitarbeitenden zu gewinnen. Machen Sie sich immer bewusst, welche Bedeutung die One-on-One-Gespräche haben. Diese sind bei virtuellen Mitarbeitenden besonders wichtig *[6]*. Vernachlässigen und verschieben Sie diese Gespräche in keinem Fall und führen Sie sie wöchentlich durch.

Und zu guter Letzt: Fördern Sie Experimente und Veränderungen. Beziehen Sie Mitarbeitende aktiv ein und werden Sie zum Coach, Treiber und Anwalt der Veränderung.

Digital Leadership – Fazit

Seit mittlerweile fast einem Jahr arbeitet eine große Zahl der deutschen Arbeitnehmer/innen im Homeoffice und damit auf Distanz in virtuellen Teams. Die Etablierung und Steuerung solcher Teams ist eine neue Herausforderung für Führungskräfte und wird als Digital Leadership in der Praxis diskutiert. Um die Herausforderungen der virtuellen Teamarbeit zu lösen und eine zielgerichtete Zusammenarbeit von virtuellen Teams zu erreichen, werden neue Arbeitsmethoden und die Nutzung von Technologie empfohlen. Ziel ist es, die Leistungsbringung und Motivation durch Führungskräfte sicherzustellen.

Es zeigt sich, dass die Führung und das Verhalten in virtuellen Teams nicht mit denen in Präsenzteams gleichgesetzt werden können. Die Empfehlungen, um die Leistungserbringung sicherzustellen, sind mehrheitlich, dass Technologie genutzt wird, um die Arbeitsleistung von Mitarbeitenden zu überwachen und Meetings zur Organisation der Softwaretools abzuhalten.

Weiterhin stellt sich die Frage, wie bei der vielen Nutzung von Technologie noch die Motivation von Mitarbeitenden sichergestellt werden kann. Natürlich wird die Arbeit mit virtuellen Teams aktuell nicht wie bekannte Präsenzarbeit sein, aber durch regelmäßige One-on-One-Gespräche, klare Rollen, aktives Zuhören und die Begleitung von Mitarbeitenden durch den Veränderungsprozess kann trotz Distanz eine Kultur des Vertrauens und der Zusammenarbeit geschaffen werden.

Die Arbeit in virtuellen Teams erfordert neue Methoden und eine Kultur der Offenheit und gegenseitiger Hilfe. Es gilt, gemeinsam daran zu arbeiten, diese neue Art der Arbeit gemeinsam in Unternehmen umzusetzen. Packen wir es an!

Quellen

[1] Bundesverband digitale Wirtschaft (2020): *Deutschland geht ins Homeoffice,* https://de.statista.com/infografik/21121/umfrage-zum-arbeiten-im-home-office-wegen-des-coronavirus/

[2] Handelsblatt (2020a): *Deutsche Wirtschaft rüstet sich gegen Coronavirus-Ausbruch in Europa,* https://tinyurl.com/yb5wycy2

[3] Handelsblatt (2020b): *Konferieren ohne Corona-Angst – wie Homeoffice gelingen kann,* https://tinyurl.com/y7kajx8x

[4] Agile-Unternehmen.de (2020): *Was ist agil?* https://agile-unternehmen.de/was-ist-agil-definition/

[5] Lindner, D. (2019): *KMU im digitalen Wandel: Ergebnisse empirischer Studien.* Wiesbaden: Springer Gabler.

[6] Lindner, D. (2020): *Virtuelle Teams und Homeoffice – Empfehlungen zu Technologien, Arbeitsmethoden und Führung.* Wiesbaden: Springer Gabler.

Dr. Dominic Lindner
Agile Unternehmen
dominic.lindner@agile-unternehmen.de

Dr. Dominic Lindner hat an der FAU Erlangen-Nürnberg am Lehrstuhl für IT-Management zum Thema virtuelle Teamarbeit promoviert und ist selbst Führungskraft in einem KMU. Sein Forschungsinteresse liegt im Bereich Arbeit 4.0, Agilität und Digital Leadership. Er ist Autor mehrerer Bücher zu diesen Themen. Beruflich beschäftigt er sich mit dem Thema Cloud und Organisationsentwicklung. Dominic Lindner bloggt aktiv auf agile-unternehmen.de.

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: Christian Luda
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Marcus Fihlon, 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 © Sylverarts | *https://stock.adobe.com*
S. 10: Bild © ink drop | *https://stock.adobe.com*
S. 14: Bild © chaliya | *https://de.123rf.com*
S. 16+17: Bild © DOAG | *https://www.doag.org*
S. 18+19: Bild © Robert Kneschke | *https://stock.adobe.com*
S. 23: Bild © fisher4 | *https://de.123rf.com*
S. 28+29: Bild © ipopba | *https://stock.adobe.com*
S. 35: Bild © prabowo | *https://stock.adobe.com*
S. 39: Bild © fisher4 | *https://de.123rf.com*
S. 40: Bild © Ihor | *https://stock.adobe.com*
S. 42+43: Bild © Maksim Kabakou | *https://de.123rf.com*
S. 48: Bild © your123 | *https://stock.adobe.com*
S. 57: Bild © naum | *https://stock.adobe.com*
S. 59: Bild © gmast3r | *https://de.123rf.com*
S. 62: Bild © pickup | *https://stock.adobe.com*

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

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

Druck:
WIRmachenDRUCK GmbH,
www.wir-machen-druck.de

Inserentenverzeichnis

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

JavaLand 2022

Save the Date

15. bis 17. März 2022

im Phantasialand bei Köln

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu