Java aktuell

iJUG Verbund
www.ijug.eu

Cloud
Mit DDD in die Cloud, Hetzner Cloud

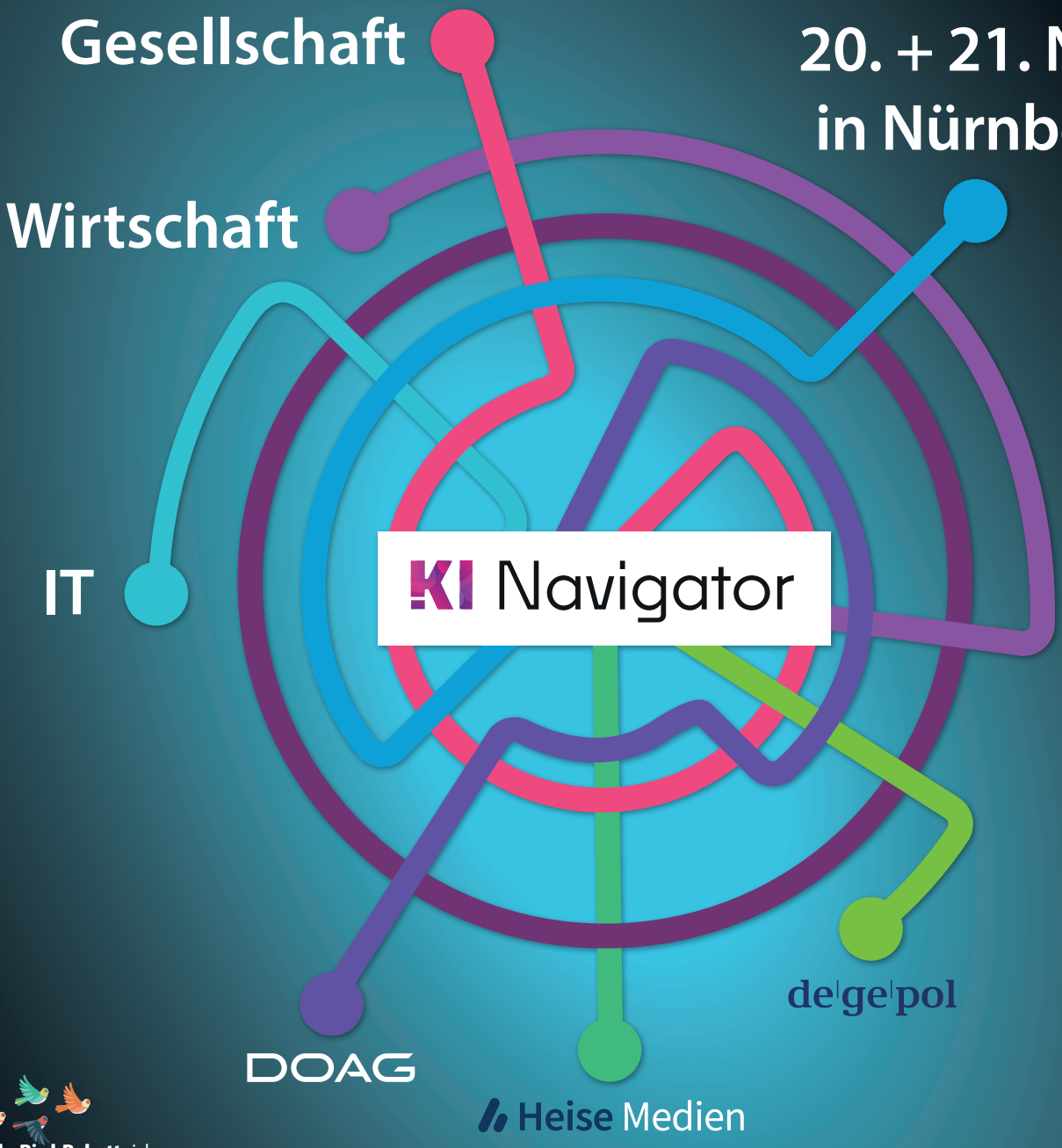
YouDebug
Debugging mit dem Groovy-basierten Tool

Künstliche Intelligenz
Generierte Unittests, KI im öffentlichen Sektor

Ausgabe 05/2024
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwicklerinnen und Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 05

KI Navigator 2024

Dein Kompass in der Welt der KI



20. + 21. Nov.
in Nürnberg

Early-Bird-Rabatt sichern

ki-navigator.doag.org

Programm
jetzt online

Liebe Leserinnen und Leser,

in dieser Ausgabe der Java aktuell widmen wir uns wieder einmal der Cloud. Doch eins nach dem anderen: Auf den ersten Seiten dieser Ausgabe bringen euch das Java-Tagebuch und die vorerst letzte Ausgabe der Eclipse Corner (mehr dazu erfahrt ihr in der Kolumne selbst) auf den neuesten Stand rund um aktuelle Geschehnisse innerhalb der Java-Community. Die goldenen Regeln dürfen selbstverständlich ebenfalls nicht fehlen. Was genau sind eigentlich Java User Groups und was machen die eigentlich? Dies beantwortet euch Daniel van Ross in seinem Bericht zum mobilen iJUG Café auf der JavaLand 2024, in dem er außerdem einen Einblick in die Arbeit der Java User Groups in Deutschland und der ganzen Welt gibt.

Ab Seite 16 tauchen wir in die Schwerpunktthematik ein. Dr. Sönke Magnussen präsentiert, wie mithilfe von Domain-Driven-Transformation die richtige Cloudarchitektur entwickelt und unterstützt werden kann. Er stellt diese Methodik vor und verdeutlicht sie anhand eines Beispiels. In seinem Beitrag "Wie Cloud, nur anders!" beschreibt Thomas Michael, wie man eine native Cloud-Applikation mithilfe von Microservices auf einem embedded Kubernetes statt in der Cloud erstellen kann. Amazon, Google und Microsoft sind die wohl bekanntesten Cloud-Anbieter. Doch wie sieht es mit deutschen Anbietern aus? Wie eine Anwendung automatisiert in der Hetzner Cloud publiziert und gehostet werden kann, zeigt uns Marcus Fihlon.

Ab Seite 38 zeigt Markus Karg, wie er das I/O-Subsystem in OpenJDK modifiziert hat, um eine Java-Anwendung zu beschleunigen. Im Anschluss stellt Wolfgang Schell uns das Groovy-basierte Debugging Tool "YouDebug" näher vor. Er legt anhand von Beispielen die Anwendungsfälle und die Funktionsweise dar. Welche Vorteile unveränderliche ("immutable") Objekte für Entwicklerinnen und Entwickler mit sich bringen, widmet sich Yves Schubert in seinem Artikel ab Seite 52. Darauf folgt Teil 2 der Reihe "Strukturzementierende Tests und wie man sie vermeidet" von Richard Gross. Darin stellt er die Konzepte der im ersten Teil implementierten TestDsl vor und zeigt, wie man mithilfe dieser Tests schreiben kann, die durch Änderung von lediglich einer Codezeile von Integration- zu Unit-Tests werden. Anschließend widmet sich Jan-Hendrik Telke ab Seite 70 dem Thema KI-generierter Unittests. In seinem Beitrag zeigt er die Möglichekiten und bewertet sowohl Schwachstellen als auch Vorteile generierter Tests. Wir bleiben beim Thema KI und runden diese Ausgabe mit dem Artikel von Eldar Sultanow, Thomas Heimann und Matthias Seßler ab. Darin geht es um KI-Sprachmodelle aus dem Open-Source-Bereich im öffentlichen Sektor.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

16

Mit Domain-Driven-Design in die Cloud

30

Hetzner Cloud: Anwendungen automatisiert publizieren und hosten

Die Vorteile von Immutability

KI im öffentlichen Sektor

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

22. April 2024

Jakarta EE 11

Das nächste Jakarta-EE-Release soll im Juni beziehungsweise eher im Juli fertig werden. Einige Spezifikationen wie CDI 4.1 oder Interceptors 2.2 haben bereits den Review-Prozess und die abschließende Abstimmung („Ballot") durchlaufen, andere sind noch mittendrin oder benötigen noch ein bisschen Zeit (zum Beispiel Authentication 3.1 und Security 4.0). Es scheint aber aktuell keine größeren Probleme oder Diskussionen mehr zu geben, die das Release verzögern könnten. Zum größten Feature – der neuen Data-1.0-Spezifikation, die ein höheres Abstraktionslevel für Datenzugriff über unterschiedliche Datenzugriffsformen (SQL, NoSQL, Webservices …) bietet, – gibt es einen guten, ausführlichen Artikel von Gavin King *[1]*.

26. April 2024

Eclipse Temurin wächst

Eclipse Adoptium hat gerade Updates der eigenen OpenJDK-Variante Temurin für Java 8, 11, 17, 21 und 22 herausgegeben. Soweit nichts Neues. Nach eigener Aussage ist es aber mit 54 unterschiedlichen Kombinationen aus Java-Version und -Plattform das bisher größte Release – schon beachtlich. Als Plattform hinzugekommen sind zuletzt die RISC-V Prozessor-Architekturen. Die Linux-Versionen ab 21 werden (noch mit Ausnahme von riscv64) ab jetzt mit einem Dev-Kit gebaut, um sichere, reproduzierbare Builds zu gewährleisten.

30. April 2024

Java 17 liegt vorne

Nach Statistiken von New Relic *[2]*, basierend auf den von ihrer Software überwachten Applikationen, hat Java 17 inzwischen Java 11 überholt: 35 % dieser Applikationen nutzen Java 17 und nur noch 33 % das vorherige Long-Term-Support-Release. Aber auch Java 8 ist immer noch mit 29 % vertreten, während das aktuelle LTS-Release 21 nach einem halben Jahr „nur" auf 1,4 % kommt. Allerdings kam Java 17 damals nach einem halben Jahr erst auf 0,4 % „Adoption", weshalb die Zahl vergleichsweise gut ist. Dazu kommt: Es sind Zahlen aus der Produktion, im Entwicklungs- und Teststadium sind die neueren Releases vermutlich deutlich weiter vorne.

Weitere Zahlen: Das Oracle JDK verliert weiter Anteile und liegt im New-Relic-Bericht mit 21 % zu 18 % nur noch knapp vor Eclipse Adoptium [sic!] (gemeint ist Eclipse Temurin, das vom Adoptium-Projekt herausgegeben wird). Der „Star" aus 2023, Amazon Corretto, verliert sogar noch mehr Anteile und liegt ganz knapp hinter Temurin nur noch auf Rang drei. Insgesamt gibt es aber eine gesunde Vielfalt an verbreiteten Varianten, wozu sicher nicht nur Oracle mit seiner Lizenzpolitik, sondern auch das Adoptium-Projekt mit seinem offenen Markplatz beigetragen hat.

14. Mai 2024

11 bis '32

Das könnte selbst für die langsamsten Organisationen reichen: Oracle dehnt den Support für Java 11 (auch auf Solaris) bis Januar 2032 aus und verzichtet dabei auf die „extended support fees" *[5]*. Aber es gibt auch andere Anbieter mit ähnlich langen Laufzeiten für den Support, zum Beispiel Azul *[6]*. Java 11 wird dann bei beiden (zumindest Stand heute) noch länger unterstützt als die nachfolgende LTS-Version 17. Muss ich das verstehen? Vielleicht nicht, ist halt „based on customer feedback".

15. Mai 2024

Jakarta EE Starter: Einblicke in die Nutzung

Auch der neue Jakarta EE Starter hilft nicht nur Entwicklerinnen und Entwicklern, sondern ermöglicht ganz nebenbei Einblicke in das, was „da draußen" an Versionen genutzt wird – allerdings zur Entwicklungszeit. Die Nutzungsstatistiken seit dem „Go-Live" der Seite letztes Jahr zeigen, dass Java EE 10, Java SE 17 und das Full Profile in ihren Auswahlkategorien jeweils weit vorne liegen – so wie auch die Kombination aus den Dreien, die die am häufigsten gewählte ist. Nur knapp 30 % nutzen das Web Profile, das Core Profile rangiert mit 5 % unter „ferner liefen" *[3]*.

20. Mai 2024

MicroProfile: Wird GraphQL entstaubt?

Die MicroProfile-GraphQL-Spezifikation hat ihr letztes Update im März 2022 erfahren und benötigt dringend ein wenig Aufmerksamkeit. Kito Mann, der in der MicroProfile Working Group unter anderem von Kompatibilitätsproblemen mit JDK 21 berichtet hat, will nun die Reaktivierung des Projekts starten und sammelt dazu Unterstützung in der entsprechenden Google-Gruppe *[4]*.

24. Mai 2024

Spring Boot 3.3

Spring Boot 3.3 bringt einige neue Features *[7]*: zum Beispiel direkte Virtual-Threads-Unterstützung für Websockets und „Class Data Sharing" (CDS), das Startup-Zeiten und Speicherbedarf reduzieren soll. Auch im Bereich Observability hat sich einiges getan, etwa mit dem Upgrade auf den seit September verfügbaren Prometheus Client 1.x. Und die Actuators bieten einen neuen Endpunkt „/sbom", mit dem die „Software Bill of Material" der Applikation nun auch „online" per HTTP abgerufen werden kann – wenn es denn nötig ist; die Actuator-Endpoints aber immer schön vor Zugriffen von außen schützen, gell…?

29. Mai 2024

Quarkus mit Observability-Unterstützung für die Entwicklung

Quarkus 3.11 bringt einen neuen „Dev Service" mit, genau genommen sogar mehrere: die „Observability Dev Services". Diese umfassen Grafana, OpenTelementry Collectors und was so dazugehört, und sie sind im neuen Konzept der „Dev Resource" zusammengefasst. Über Testcontainers wird im Hintergrund einfach das docker-otel-lgtm-Image *[8]* eingebunden und schon ist die gesamte Observability-Umgebung in Miniaturausgabe für Testzwecke da.

30. Mai 2024

Jakarta/MicroProfile AI

Jetzt habe ich schon wochenlang nichts über KI geschrieben – mal schauen, welche Umwälzungen in der Zwischenzeit im Java-Ökosystem stattgefunden haben. Die AI-Diskussion ist jedenfalls auch in der Jakarta-EE-Working-Group angekommen; aber vernünftigerweise schnell damit beendet worden, dass sich das „Schwesterprojekt" MicroProfile bereits der Sache annimmt, und Jakarta vielleicht doch eher den Fokus auf Standardisierung setzen sollte – wofür es noch etwas früh ist.

Also weiter: Spring AI macht Fortschritte und hat gerade Version „1.0.0 Milestone 1" freigegeben. Es bietet auf den ersten Blick nichts, was langchain4j nicht schon seit einiger Zeit bereitstellen würde, bis auf die nahtlose Integration der Konzepte und APIs in die Spring-Welt, was natürlich ein Wert an sich sein kann.

Auch MicroProfile wird im ersten Wurf sicher hauptsächlich von den existierenden Projekten abschauen, aber langfristig spornt der Wettbewerb hoffentlich die Kreativität an (sonst müssen wir die großen LLMs fragen, welche Features sie sich wünschen).

Quarkus geht den Weg der Integration von langchain4j in das eigene Projekt. Links und rechts von den generischen Frameworks, die ihre Zehen ins AI-Becken tauchen, entstehen aber kontinuierlich weitere reine AI-Projekte. So zum Beispiel JLama *[8]*, das sich ausschließlich mit der Nutzung von LLMs („Inferenz") aus Java-Applikationen beschäftigt und für einfache Anwendungen eine eigene CLI und simple Web-UI mitbringt.

31. Mai 2024

JEPs für Java 23

Die Feature-Liste für das Java-23-Release Mitte September ist noch ordentlich angewachsen. Der „Rampdown Phase One" („Feature-Annahmeschluss") wird erst Anfang Juni stattfinden, in den letzten Wochen sind aber noch einige interessante „Java Enhancement Proposals" hinzugekommen wie beispielsweise JEP-476 „Module Import Declarations" (als Preview). Damit soll neuer Schwung in das Java-Modulsystem gebracht werden, indem alle Top-Level-Klassen und -Interfaces eines Moduls mit einem „import module X"-Statement auf einmal importiert werden können.

Für alle Tuning-Fans, JEP-474: Der ZGC soll in 23 per Default im „generational" Modus arbeiten, während der „non-generational" Modus als Auslaufmodell markiert (deprecated) und in einem späteren Release verschwinden soll. Das gleiche Schicksal wird (JEP-471) die Memory-Access Methoden in sun.misc.Unsafe ereilen, für die es ja inzwischen offizielle Alternativen gibt (die VarHandle API aus JDK 9 und die „Foreign Function & Memory API" aus 22). Die gesamte Liste ist online zu finden *[9]*.

Verweise

[1] *https://in.relation.to/2024/04/01/jakarta-data-1/*
[2] *https://newrelic.com/sites/default/files/2024-04/new-relic-state-of-the-java-ecosystem-report-2024-04-30.pdf*
[3] *https://www.eclipse.org/lists/jakarta.ee-community/msg03183.html*
[4] *https://groups.google.com/g/microprofile/c/e4uMmvxXdZ4/m/pmcpI_F4AgAJ*
[5] *https://blogs.oracle.com/java/post/java-se-spring-2024-road-map-update*
[6] *https://www.azul.com/de/azul-support-roadmap/*
[7] *https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.3-Release-Notes*
[8] *https://github.com/tjake/Jlama*
[9] *https://openjdk.org/projects/jdk/23/*

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

Andreas Badelt ist seit 2001 ehrenamtlich im DOAG e.V. aktiv und hat dort inzwischen seine Heimat in der Cloud Native Community gefunden, wobei ihn das Java-Ökosystem bis heute fasziniert. Beruflich hat er von Ende des vorigen Jahrtausends an als Entwickler und später auch Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet. Seit 2016 ist er als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

MARKUS' ECLIPSE™-Corner

Nachdem ich es mir in der letzten Ausgabe vermutlich schon ganz gehörig mit euch verscherzt habe, lege ich diesmal noch eins drauf und schreibe nicht mal ansatzweise über Jakarta EE! Der Grund ist simpel: Es gibt einfach fortwährend nichts Neues in dieser Hinsicht zu berichten, weil es mit diesem Projekt nicht vorwärts geht! Die viel zu wenigen Entwickler, die überhaupt noch dediziert an der Jakarta-EE-API arbeiten, bekommen viel zu wenig Zeitbudget von ihren jeweiligen Arbeitgebern – wohlgemeint meist Branchengrößen mit Milliardenumsätzen – für den Fortschritt der Unternehmens-Java-API bezahlt. Entsprechend kommt halt auch nix rum!

Dieses Leid habe ich euch bekanntlich seit Jahren geklagt und vermutlich seid ihr längst satt davon, ebenso wie von den steten Verbalattacken gegen all jene, die gerne ihr Business mit fremder Open Source machen, jedoch nichts Substanzielles zu deren langfristigem Fortbestehen beitragen. Weiterhin sinn- und inhaltslos eine Kolumne damit zu füllen, macht aus meiner Sicht irgendwann einfach keinen Sinn mehr und verkommt somit zum reinen Selbstzweck. Das habt ihr nicht verdient, und ich auch nicht, und auch die Java aktuell als Zeitschrift ebenfalls nicht. Daher verspreche ich euch, dass ihr es nun endlich hinter euch habt: Ich gebe diese Kolumne hiermit auf!

Natürlich kehre ich der Zeitschrift nicht endgültig den Rücken. Hinter den Kulissen gibt es viele Ideen, wie ich euch auf anderen Themengebieten mit beißenden Kommentaren und unerfreulichen Wortergüssen zum Thema Java auch weiterhin den Tag vermiesen kann. So einfach kommt ihr mir also nicht davon!

In den Sinn kommt mir da beispielsweise der iJUG selbst mit seinen diversen Infrastruktur-Projekten, über die mein Wahlschweizer-Kollege Marcus Fihlon bereits zu berichten begonnen hat. Außerdem investiert der iJUG neben Jakarta EE auch noch in weitere Stützen des Java-Universums, allen voran MicroProfile und Adoptium. Und unter den durch den iJUG vertretenen Java-Entwicklern sind ja auch nicht gerade wenige, die das weitere Java-Umfeld am Laufen halten – zum Beispiel durch Mitarbeit an Bibliotheken und Tools, wie beispielsweise OpenJDK und Maven. Dort tut sich – im Gegensatz zu Jakarta EE – durchaus einiges! Wenn es dann endlich mal ein neues Jakarta EE Release gibt, will ich gar nicht ausschließen, auch darüber ein paar Sätze zu verlieren.

Insofern schließe ich heute mit einem lachenden und einem weinenden Auge. Auch wenn es die letzte Kolumne zum Thema Jakarta EE war (und das ja der eigentliche Grund, aus dem heraus Wolfgang Taschner und ich vor vielen Jahren diese Kolumne ursprünglich gegründet hatten), will ich euch wissen lassen: „Heute ist nicht alle Tage; ich komm' wieder, keine Frage!"

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.

Die goldenen Regeln: Teil 3

Andreas Monschau, Haeger Consulting

Stetig bin ich auf der abenteuerlichen Suche nach ihnen – den goldenen Regeln, mit denen man Neulingen, Junioren oder Quereinsteigern den Start möglichst vermiesen kann. Softwareentwicklung ist hart und unfair. So soll es auch bleiben. Du hast gelitten, alle sollen leiden.

Diese Regeln entspringen nicht meiner bunten Fantasie, sie finden sich noch in vielen Projekten, Unternehmen und Organisationen wieder – mit dieser Sammlung möchte ich zum einen erheitern, und zum anderen mahnen. Mahnen, diese Regeln nicht mehr anzuwenden, und Menschen, die darunter leiden ermutigen, sich zu wehren. Hast du weitere Beispiele für Regeln und möchtest sie mit mir und anderen teilen? Dann kontaktiere mich gerne!

Deine konkrete Rolle ist da egal: Ob du nun Projektmanager, Teamleiter, Entwickler, PO oder auch Tester bist – solltest du derjenige sein, der den Neuling begrüßt, lade ich dich herzlich ein, die folgenden Hinweise, Tipps und Tricks zu beherzigen. Nicht jeder Tipp wird passen, such dir einfach das für dich Beste raus!

Regel 3: Dokumentiere „richtig"

„Lies' dich doch erstmal ein!" – wer diesen Satz nicht schon mal gesagt hat, möge den ersten Stein werfen. Dokumentation ist wichtig, das wollen wir nicht vergessen, und gerade für Neulinge stellt eine angemessene Dokumentation den perfekten Einstieg in den neuen Wirkungsbereich dar. Aber du weißt auch: Dokumentation ist nervig und kostet Zeit sowie Geld! Und außerdem: Es geht ja gar nicht darum, dass sich der Neuling wohl und gut informiert fühlt, wir sind ja schließlich bei den goldenen Regeln, die ihnen das Leben erschweren. Wie sollte also eine entsprechende Dokumentation in deinem Projekt, deiner Organisation oder deinem Unternehmen am besten aussehen, um dieses Ziel zu erreichen? Da gebe ich dir gerne drei kleine Tipps:

Veraltete Dokumentation

Was ist besser als alte Dokumentation? Genau, richtig alte Dokumentation! Ein Neuling wird zweimal hinschauen, wenn er Links zu einem Tool bekommt, und die wichtigen, dort hinterlegten Informationen schon mehrere Jahre lang nicht mehr aktualisiert worden sind. Was sollte sich schon beispielsweise über mehrere Jahre hinweg an einem Sicherheitskonzept ändern? Oder an einer Prozessbeschreibung? Und sollte sich doch was geändert haben – mit dem

Neuling hat man jemanden an der Hand, der es dann doch direkt aktualisieren kann, während er die Dokumentation per Try-and-Error durchspielt. Ein absoluter Gewinn!

Dokumentation vortäuschen

Einfach mal in deinem Dokumentationstool eine Seite hinzufügen und als einzigen Inhalt „To-do" einfügen – irgendjemand wird es sicher irgendwann zu Ende führen. Irgendwann. Vielleicht. Um denen Zustand einer Dokumentation zu verschleiern, gehört auch dies dazu: Lasse niemanden wissen, ob du mit der Doku fertig bist, mittendrin steckst oder gar erst damit angefangen hast. Auf Nachfragen erst verspätet reagieren. Das wird für viel Frust beim Leser sorgen, versprochen!

Dokumentation im Sourcecode

Auch hier gibt es einige Aspekte, die du sicherlich schon kennst, aber es schadet nicht, sie noch mal in Erinnerung zu rufen. Ein „To-do" kann man ebenso gut auch in den Javadoc-Block oberhalb einer Methode packen *(siehe Abbildung 1)*. Hier gilt ebenfalls – irgendjemand wird den Kommentar schon irgendwann mal vervollständigen.

```
/**
 *
 * todo
 *
 * @return
 */
@Override
public ArrayList<Client> importDataFromDB() {
    return datenbank.readClientList();
}
```

Abbildung 1: „To-do" im Javadoc-Block

Die Antithese hierzu wäre, übertrieben viel zu kommentieren – ob es nun Sinn macht oder nicht. Dafür gibt es klassische Beispiele, von denen ich gerne an dieser Stelle eins aufgreife *(siehe Abbildung 2)*.

```
/**
 * incremets i
 */
i = i + 1;
```

Abbildung 2: Kommentare sind wichtig, selbst mit Schreibfehlern

Wie du siehst, schon mit sehr einfach umzusetzenden Maßnahmen kannst du für schlechte Dokumentation sorgen. Der Neuling wird es dir nicht danken, aber du kannst dir sicher sein, für Frustration zu sorgen. Und noch besser: Der Neuling wird sich daran erinnern und künftig seine Dokumentation ebenfalls auf diese Art und Weise erstellen. Die Saat ist gelegt.

Im nächsten Teil dieser Reihe berichte ich über etwas, dass sicher jeder kann, der dieses Magazin liest: schlechten Code schreiben!

Andreas Monschau
Haeger Consulting
amonschau@haeger-consulting

Andreas Monschau ist seit über 10 Jahren als Senior IT-Consultant mit den Schwerpunkten Softwarearchitektur- und Entwicklung sowie Teamleitung bei Haeger Consulting in Bonn tätig und aktuell als Solution Designer im Kundenprojekt unterwegs. Neben seinen Projekten leitet er das umfangreiche Traineeprogramm des Unternehmens und ist als Sprecher und Autor unterwegs.

Das Java User Group Café auf der JavaLand

Daniel van Ross

In der deutschsprachigen Java Community spielen User Groups eine entscheidende Rolle. Was diese User Groups zu bieten haben, wie sie zusammenarbeiten und welchen Einfluss sie innerhalb der Java-Community haben, wird in diesem Artikel im Detail beleuchtet.

Eine Java User Group (JUG) ist eine lokale oder regionale Gruppe von Java-Entwicklerinnen und Entwicklern sowie -Enthusiastinnen und -Enthusiasten, die sich regelmäßig treffen, um ihr Wissen über Java-Technologien zu teilen, Erfahrungen auszutauschen, neue Techniken zu diskutieren oder auch gemeinsam an Java-Projekten arbeiten. Diese Gruppen organisieren oft Treffen, Vorträge, Workshops, Hackathons und andere Veranstaltungen rund um Java-Programmierung und -Entwicklung. Allein im deutschsprachigen Raum gibt es über 40 JUGs. Die Organisationsformen sind dabei so unterschiedlich wie die Themen bei den Treffen und reichen von eingetragen Vereinen über größere, von Firmen gesponsorte Gruppen bis hin zu einer Hand voll Gleichgesinnter, die sich einmal im Monat zum Bier oder Kaffee treffen.

Viele der heute noch aktiven Java User Groups gab es bereits zu den Zeiten, in denen die Weiterentwicklung von Java noch unter der Ägide von Sun Microsystems lag. Sun hat damals das Konzept der User Groups aktiv unterstützt, unter java.net wurde unter anderem ein weltweites Verzeichnis von User Groups gepflegt. Insbesondere in Deutschland bestand die User-Group-Landschaft aus sehr vielen kleinen, voneinander unabhängigen Gruppen, die sich untereinander allenfalls dann austauschten, wenn sich die Organisatoren oder Organisatorinnen persönlich kannten. Im Gegensatz zu großen Gruppen in anderen Ländern, wie beispielsweise der brasilianischen SouJava oder der britischen London Java Community, war der Einfluss einzelner deutscher User Groups auf das Java-Ökosystem aufgrund der kleinen Gruppengröße relativ gering.

Die Übernahme Suns durch Oracle im Jahr 2009 brachte Unruhe in die gesamte Java-Community und einige der Java User Groups in Deutschland schlossen sich gemeinsam mit der Deutschen Oracle Anwendergruppe (DOAG) zum Interessenverbund der Java User Groups e. V. (iJUG) zusammen. Der iJUG verlieh den darin organisierten Gruppen die Möglichkeit, gegenüber Herstellern und Unternehmen geschlossen aufzutreten und ihren Interessen größeres Gehör zu verschaffen, ohne dabei ihre Eigenständigkeit zu verlieren. Inzwischen sind im iJUG über 40 Gruppen aus dem deutschsprachigen Raum organisiert. Aus der Zusammenarbeit der Vertreter und Vertreterinnen der einzelnen JUGs im iJUG ist in den letzten 15 Jahren vieles entstanden, unter anderem zum Beispiel ein enger Kontakt zur Eclipse Foundation und die Mitarbeit in den Working Groups von Adoptium und MicroProfile. Auch die jährlich stattfindende JavaLand-Konferenz ist aus dem iJUG heraus entstanden und bis heute haben die User Groups einen maßgeblichen Einfluss auf die Gestaltung der Konferenz.

Um den einzelnen User Groups auf der Konferenz einen Raum und eine Plattform zu bieten, auf der sie sich selber präsentieren können, gibt es seit der ersten JavaLand das JUG Café. Ein Ziel des Cafés ist es, den Konferenzbesucherinnen und -besuchern die Möglichkeit zu geben, die Menschen hinter ihrer lokalen JUG zu treffen und sich über deren Treffen und Veranstaltungen informieren zu können. Ein weiteres Ziel ist es, den Austausch der User Groups untereinander zu fördern und neue Kontakte zu knüpfen. Auf der vergangen JavaLand war das Café im DOAG Eventbus zu finden, der bei vielen Teilnehmerinnen und Teilnehmern die Neugier weckte. Daher lag der Fokus des JUG Cafés in diesem Jahr darauf, die Konferenzbesucherinnen und -besucher anzusprechen, die das Konzept von User Groups und deren Arbeit noch nicht kannten. Es ergaben sich viele interessante Gespräche und einige waren überrascht, zu erfahren, dass es auch in ihrer Nähe eine Java User Group gibt.

Das JUG Café auf der JavaLand dient in erster Linie dem Networking und der Bekanntmachung der vielen Java User Groups in Deutschland, Österreich und der Schweiz, die von Freiwilligen organisiert werden und ohne die die Java Community nicht so eine lebendige und offene Gemeinschaft wäre.

Dabei ist es egal, wie groß die User Group ist oder wie ihre Veranstaltungen aussehen. Einige treffen sich einfach regelmäßig für einen lockeren Austausch, andere koordinieren Vorträge innerhalb der Gruppe über Meetup, wieder andere laden teils international bekannte Speaker ein und manche organisieren große Konferenzen mit mehreren hundert Teilnehmenden. Aber alle freuen sich, neue Menschen in ihrem Kreis aufnehmen zu können, egal ob als Besucherinnen und Besucher oder als neue aktive Mitglieder.

Wer jetzt sein Interesse geweckt sieht, findet einen ersten Überblick über die im iJUG organisierten Gruppen in *Abbildung 1*, auf den Webseiten des iJUG *[1]* oder im aggregierten Terminkalender auf den Seiten des Java User Group Deutschland e. V. *[2]*. Wer keine Java User Group in seiner Nähe findet, sollte darüber nachdenken, eine neue

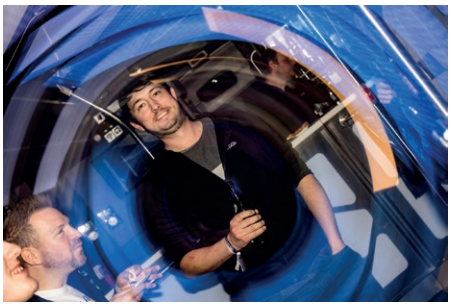
Abbildung 1

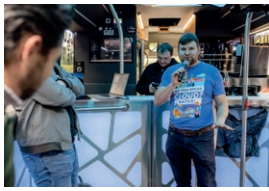
zu gründen – hierbei sind alle User-Group-Organisatorinnen und -Organisatoren sowie der iJUG sehr gerne behilflich.

Quellen

[1] Interessenverbund der Java User Groups e.V. - Der Verein: *https://www.ijug.eu/de/verein/*

[2] Java User Group Deutschland e.V. - User Group Treffen: *https://java.de/user-group-treffen*





Daniel van Ross

daniel@vanross.de

Daniel van Ross ist Softwareentwickler bei FIZ Karlsruhe – Leibniz-Institut für Informationsinfrastruktur und beschäftigt sich dort hauptsächlich mit Backend-Themen im Java Bereich. Außerdem ist er seit Jahren Vorstandsmitglied des Java User Group Deutschland e. V. und beteiligt sich an der Organisation des Java Forum Nord in Hannover. Auf der JavaLand ist er Co-Host des JUG Cafés und seit kurzem ist er Mitglied des Java-Programmkomitees für die Open Community Experience.

Mit Domain-Driven-Design beim Redesign von Legacy-Systemen die richtige Cloudarchitektur finden

Dr. Sönke J. Magnussen, WPS Workplace Solutions Hamburg

Beim Redesign von Legacy-Systemen ist Cloudfähigkeit häufig ein wichtiges Ziel, damit die Vorteile moderner Technologien für die neuen Anforderungen an die Software genutzt werden können. So ein Redesign kann mit Domain-Driven-Design (DDD) und Domain Storytelling (ein Workshop-Format, in dem Fachseite und Entwicklungsteam gemeinsam Geschäftsprozesse als konkrete Geschichten modellieren) geplant und unterstützt werden. Die verwendete Methodik heißt Domain-Driven-Transformation (DDT) – eine Methodik, die DDD verwendet, um die Softwarearchitektur schrittweise zu modernisieren. Die geeignete Cloudarchitektur, in der das Softwaresystem betrieben werden soll, kann ebenso mit Hilfe von DDD und Domain Storytelling entwickelt werden. Dabei orientiert sich die Cloudarchitektur am strategischen Design von DDD und die genaue Ausgestaltung hängt von den Qualitätsanforderungen der Software ab. Diese Qualitätsanforderungen können durch Qualitätsszenarien beschrieben werden. Durch die Identifizierung der betroffenen Bereiche in den Domain Stories, die von den Qualitätsszenarien betroffen sind, können detaillierte Anforderungen an die Cloudarchitektur für die Module aus dem strategischen Design von DDD identifiziert werden. Die so gewonnenen Informationen helfen beim Entwurf der Cloudarchitektur.

Im Folgenden werden wir die Methodik kurz vorstellen und mit einem Beispiel verdeutlichen. Wir beginnen mit einer allgemeinen kurzen Einführung in den Begriff Domain-Driven-Transformation. Für einen tieferen Einstieg in das Thema empfehlen wir das Buch „Domain-Driven-Transformation" [3] von Dr. Carola Lilienthal und Henning Schwentner.

Domain-Driven-Transformation

In den letzten Jahrzehnten wurden viele Softwaresysteme entwickelt, die nun modernisiert und zukunftsfähig gemacht werden müssen. Dafür gibt es hauptsächlich zwei Gründe. Erstens passen die Qualitätsanforderungen, die bei der ursprünglichen Entwicklung des Systems angenommen wurden, nicht mehr zum heutigen Umfeld der Systeme. Zu den geänderten Anforderungen gehören typischerweise Plattformunabhängigkeit, Benutzerfreundlichkeit, Wartbarkeit, Erweiterbarkeit, Performanz/Effizienz, Zuverlässigkeit und Sicherheit. Jedes dieser Qualitätskriterien kann heutzutage eine andere Qualität als bei der ursprünglichen Entwicklung erfordern, wie aus zahlreichen Gartner-Studien hervorgeht (vgl. Gartners Top Strategic Technology Trends 2023 *[9]*). Zweitens haben sich die inneren Qualitätsmerkmale wie Wartbarkeit und Verständlichkeit aufgrund von Architekturerosion (vgl. *[4]*) verschlechtert. Daher sind Anpassungen an die geänderten Qualitätsmerkmale und das volatile Geschäftsumfeld sowie notwendige funktionale Änderungen und Erweiterungen kaum möglich.

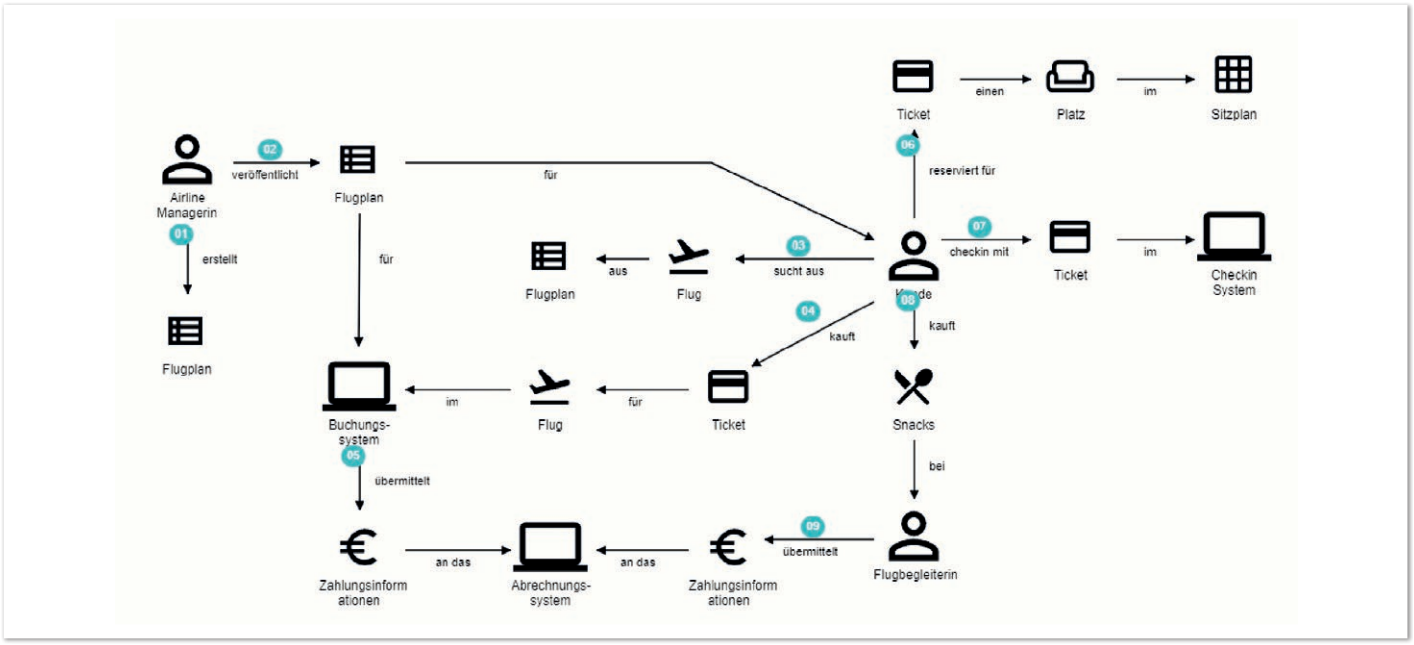


Abbildung 1: Domain Story für den Gesamtprozess zum Verkauf von Tickets (© Sönke Magnussen)

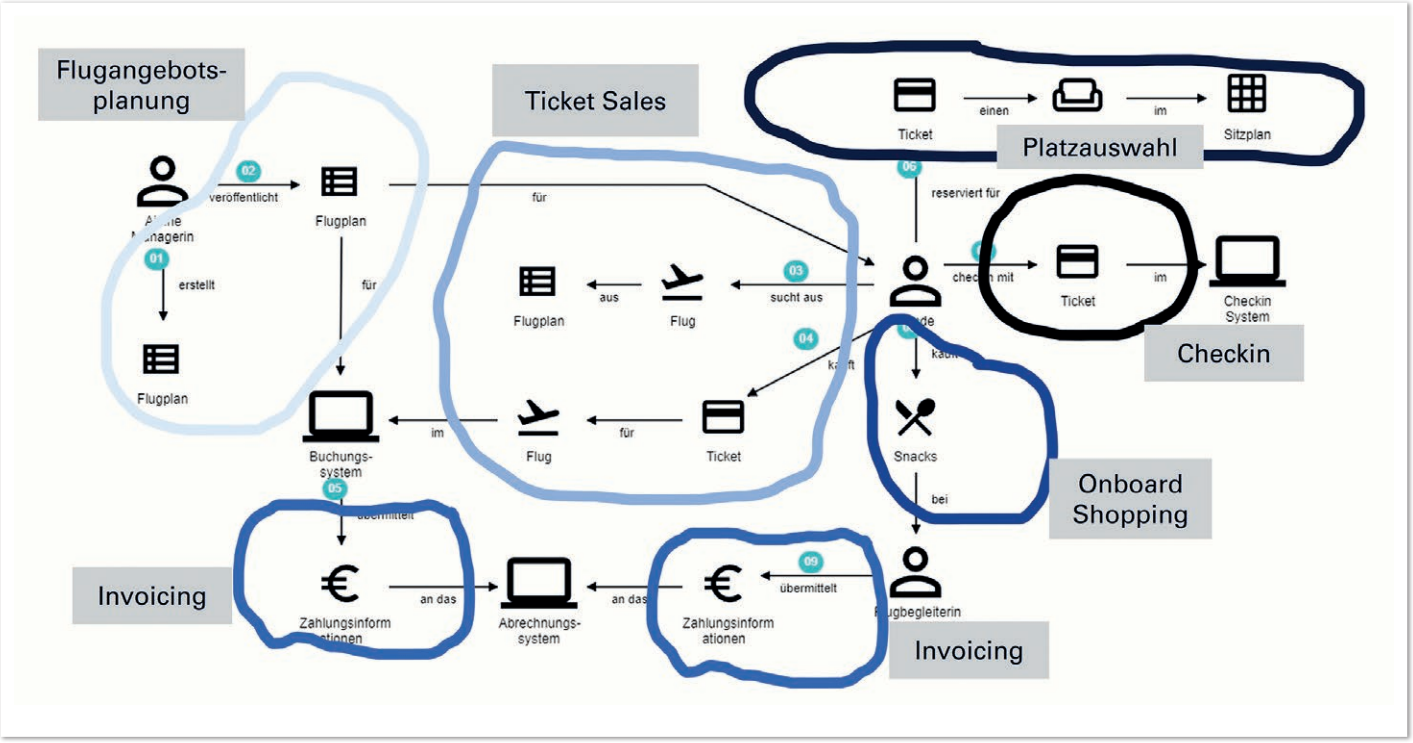


Abbildung 2: Domain Story mit Subdomänen (© Sönke Magnussen)

Im Kontext der Legacy-Systeme erweist sich Domain-Driven Design (DDD) als vielversprechende Methode, um große Systeme in Microservices zu zerlegen oder wartbarere Monolithen aus wohlstrukturierten Modulen zu bauen.

Dabei wird der Fokus auf den fachlichen Kern des Softwaresystems gelegt. Auf der Makro-Architektur-Ebene wird das System in unabhängige DDD-Module unterteilt, und auf der Mikro-Architektur-Ebene werden DDD-Muster verwendet, um den Code besser zu strukturieren und damit wartbarer zu halten. Der Prozess, wie aus einem Legacy-System und den durch DDD gewonnenen fachlichen Strukturen eine wartbare und dem Umfeld angemessene Architektur entsteht, nennen wir Domain-Driven-Transformation (DDT) *[3]*.

Dabei wird zunächst über kollaboratives Modellieren die Fachlichkeit neu aufgenommen und strukturiert. Das kann beispielsweise über Domain Storytelling *[1]* geschehen. Basis für die strategischen Transformationsschritte ist die aus dem DDD gewonnene Kontext-Map, die die fachlich geschnittenen Module und deren soziotechnischen Beziehungen untereinander als eine Soll-Architektur darstellt. Die Abweichungen der Ist-Architektur zur Soll-Architektur müssen analysiert und priorisiert werden, um daraufhin Maßnahmen zu entwerfen, die in Projekten umgesetzt werden. Die Ist-Architektur des Legacy-Systems kann so schrittweise in Richtung der Soll-Architektur umgeformt werden.

Ein kleines Beispiel (eine Airline)

Um das Thema besser veranschaulichen zu können, stellen wir im Folgenden eine Beispielsoftware einer kleinen Fluggesellschaft vor. Die Ist-Situation der Software ist, dass diese in nicht Cloud-fähigen proprietären Technologien entwickelt wurde und die Geschäftsprozesse für die Erstellung des Flugplans, für den Verkauf von Tickets und für die Sitzplanauswahl nur rudimentär unterstützt. Das System besteht bisher aus einem schwer wartbaren Monolithen. Web-Oberflächen für die Kunden gibt es nicht, der Verkauf wird größtenteils über Reisebüros und deren Buchungssysteme abgewickelt. Der Soll-Prozess ist in der Domain Story *[1]* in *Abbildung 1* grobgranular dargestellt.

Der Prozess beginnt mit einer Flugangebotsplanung, in der ein Flugplan erstellt wird. Sobald der Flugplan veröffentlicht ist, können Kunden ein Ticket über eine Web-Oberfläche erwerben. Die Web-Oberfläche nutzt ein zentrales Buchungssystem, das auch den Flugplan erhalten hat. Nach Kauf und Buchung des Tickets kommt es zur Abrechnung des Kaufs. Die Zahlungsinformationen werden an das Abrechnungssystem übermittelt und von dort wird die Zahlungsabwicklung angestoßen.

Wie in der Airline-Branche üblich, können die Plätze erst danach in einer Platzauswahl-Anwendung gewählt werden. In der Regel geschieht dies direkt vor dem Check-in-Prozess ebenfalls über eine Web-Oberfläche. Während des Fluges können On-Board-Verkäufe getätigt werden, in denen Zahlungsinformationen aufgenommen werden, die später für die Abrechnung genutzt werden. Die Informationen aus dem On-Board-Verkauf werden in das Abrechnungssystem übernommen, das die weitere Verarbeitung bezüglich der Abrechnungsprozesse abwickelt. Die Backendsysteme für Buchung, Abrechnung und Check-in werden in dieser Domain Story als gegeben angenommen. Diese Systeme sollen nicht neu entworfen werden. Alle anderen Schritte der Domain Story sollen von der transformierten Software unterstützt werden.

Mit Hilfe des strategischen Designs von DDD *[6]* können wir Subdomänen bestimmen *(siehe Abbildung 2)*.

Für die erkannten Subdomänen sollen Softwaremodule entwickelt werden. Die Airline hat strategisch entschieden, die entstehenden Applikationen größtenteils in der Public Cloud zu betreiben, damit

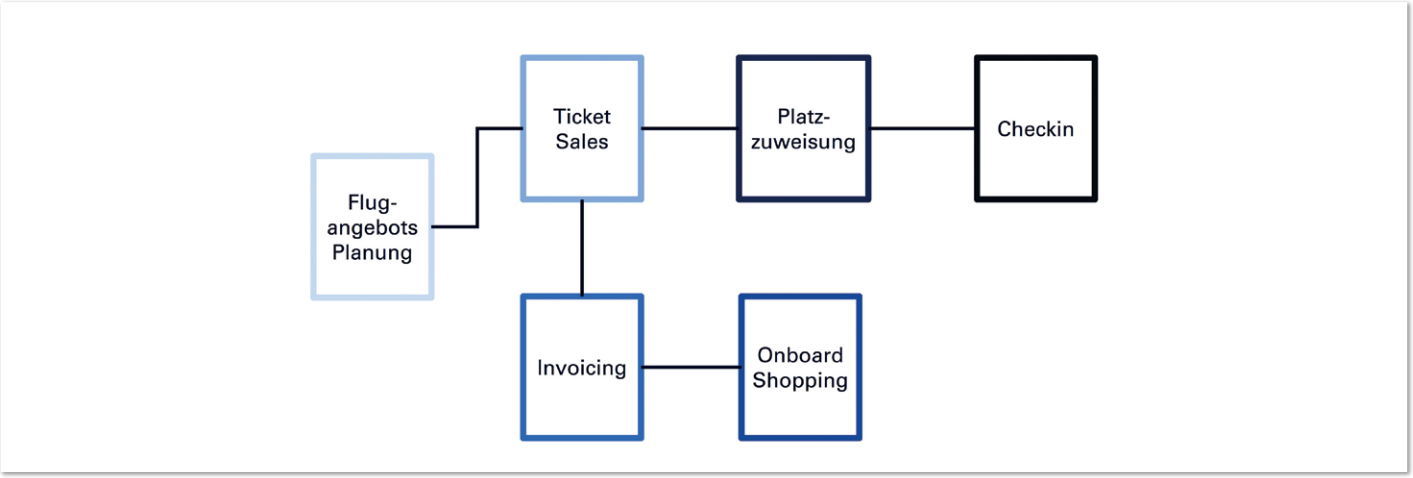


Abbildung 3: Kontext-Map für das Airline-System (© Sönke Magnussen)

Kunden von jedem Ort mit einem Browser darauf zugreifen können. Da die bisherige Software technologisch nicht in der Cloud betrieben werden kann und die Software durch Architektur-Erosion nur schlecht wartbar ist, soll es ein Redesign-Projekt geben. Das Ziel ist, die Software mit DDT so zu überarbeiten, dass sie den neuen Anforderungen gerecht wird, einfach zu warten ist und in der Cloud laufen kann.

Aus den Subdomänen, die allein aus dem Problemraum stammen und auch ohne Software einfach vorhanden sind, entwerfen wir Bounded Kontexte *[6]*, die in einzelnen Modulen oder Microservices implementiert werden. Aus diesen Bounded Kontexten und den Beziehungen zwischen ihnen entwerfen wir eine Kontext-Map, die den Ausgangspunkt für die Domain-Driven-Transformation *[3]* darstellt. Die Kontext-Map für die Airline-Anwendung würde schematisch wie in *Abbildung 3* dargestellt aussehen.

Wie kommen wir jetzt zur Cloud-Architektur?

Die Abbildung eines Softwaresystems auf eine Infrastruktur, auf der das Softwaresystem läuft, bietet in der Cloud-Technologie wesentlich mehr Freiheitsgrade als in klassischen Rechenzentren. Der Hauptgrund dafür liegt in der Kostenstruktur und der leichten Verfügbarkeit der Cloud-Services. Die meisten Anforderungen an den Betrieb von IT-Systemen könnten auch in einem traditionellen Rechenzentrum umgesetzt werden. Allerdings wären der manuelle Aufwand und die Zeitspanne für den Aufbau in einem Rechenzentrum deutlich höher.

Mit Cloud-Technologie werden beispielsweise die folgenden Freiheitsgrade für eine Verteilungssicht schnell und kostengünstig nutzbar:

- Virtuelle Maschinen & Netzwerke
- Containerisierung & Serverless-Umgebungen
- Geographische Verteilung
- Automatisierungsoptionen wie DevOps und Infrastructure as Code
- Deploymentstrategien wie Rolling Updates und horizontale Skalierung
- Nutzung von gemanagten Services für Datenbanken, Middleware & Monitoring

Die Cloudarchitektur befasst sich mit der Kombination und Interaktion verschiedener Cloud-Technologiekomponenten wie beispielsweise virtuellen Infrastrukturressourcen, virtuellen Plattformservices, Softwarefunktionen und virtuellen Netzwerksysteme, um Cloud-Umgebungen für den Betrieb eines Softwaresystems zu konzipieren. Sie dient als strategisches Modell, das definiert, wie Ressourcen optimal kombiniert werden, um eine Cloud-Umgebung für die spezifischen Qualitätsanforderungen des Softwaresystems zu schaffen *[7]*.

Eine Microservice-Architektur mit Cloud-native-Eigenschaften *[2]* könnte wahrscheinlich viele der Qualitätsanforderungen von IT-Systemen erfüllen. Die Entwicklung und Betreuung einer solchen Anwendungslandschaft kommt aber mit dem Preis von erhöhtem Entwicklungs- und Umbauaufwand der Legacy-Software einher. Darüber hinaus ist nicht klar, ob die Teams, die die Systemlandschaft in vollem Umfang betreuen sollen, das nötige Know-how haben, dies umzusetzen. Auch vor dem Hintergrund einer möglichst schnellen Umsetzung der Anforderungen und dem Erreichen der Cloudfähigkeit müssen Prioritäten gesetzt werden, sodass eine Cloud-native Anwendung nicht unbedingt die erste und beste Wahl ist.

Um für die angepasste Legacy-Software eine gute Cloud-Architektur zu finden, widmen wir uns nun den Qualitätsanforderungen.

Die Renaissance der Qualitätsanforderungen

Die Qualitätsanforderungen, die früher auch als „nicht-funktionale" Anforderungen bezeichnet wurden, spielten bei der Infrastrukturplanung schon vor der Nutzung von Cloud-Technologien eine ent-

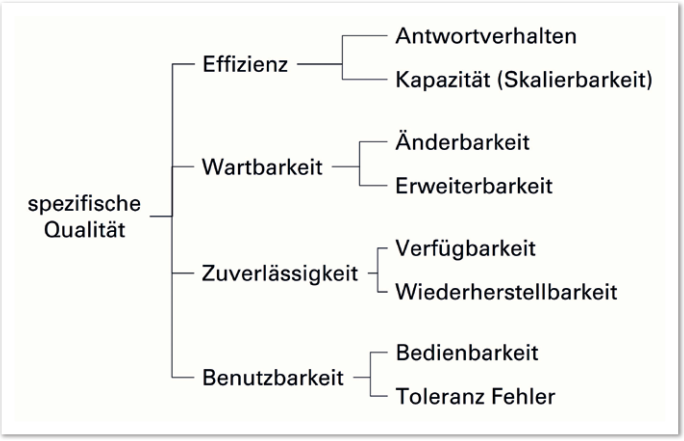


Abbildung 4: Beispiel eines Qualitätsbaums (© Sönke Magnussen)

scheidende Rolle. Der Begriff „nicht-funktional" suggerierte jedoch, dass diese Anforderungen eher von nachrangiger Bedeutung seien. Mit den neuen Möglichkeiten von Cloud-Technologien kann jetzt mit Hilfe der Infrastruktur zielgenauer und kostengünstiger auf die Qualitätsanforderungen reagiert werden. Daher kommt den Qualitätsanforderungen für Cloud-Architekturen eine immer wichtigere Rolle zu.

Für die Beschreibung von Qualitätsanforderungen haben sich in den letzten Jahren immer mehr Qualitätsbäume und Qualitätsszenarien *[5]* etabliert.

Ein **Qualitätsbaum** *(siehe Abbildung 4)* ist ein hierarchisches Modell, das das abstrakte Konzept der „Qualität" in greifbarere und handhabbarere Komponenten zerlegt. Die Wurzel des Baums stellt die Gesamtqualität der Software dar, die sich dann in verschiedene Qualitätsattribute *[8]* wie Zuverlässigkeit, Benutzerfreundlichkeit, Leistung oder Wartbarkeit verzweigt. Jedes Attribut wird weiter in spezifischere Kriterien zerlegt, die gemessen und bewertet werden können.

Wie die Qualitätsattribute gemessen werden können, steht in den **Qualitätsszenarien** zu den jeweiligen Attributen. Qualitätsszenarien sind detaillierte Erzählungen, die eine bestimmte Situation beschreiben, in der sich die Qualität der Software zeigt. Sie umreißen den Kontext, einschließlich der Quelle des Stimulus, der Umgebung, in der er auftritt, des betroffenen Systemteils und der Reaktionen des Systems auf den Stimulus. Ein Qualitätsszenario könnte zum Beispiel beschreiben, wie das System auf eine große Anzahl von Benutzeranfragen innerhalb eines bestimmten Zeitrahmens reagieren soll, um die Einhaltung der Leistungsstandards zu gewährleisten. Diese Szenarios bieten einen praktischen Rahmen, um zu beurteilen, ob die Software die Qualitätsanforderungen erfüllt.

Zum Beispiel des obigen Qualitätsbaums könnten für unser Airline-System folgende Qualitätsszenarien definiert werden, die Qualitätsmerkmale der Software beschreiben:

- **Szenario 1 (Antwortverhalten)**: *„Jede Antwort auf Kundeninteraktionen in der Customer Journey muss im Regelbetrieb bei 98 von 100 Anfragen in weniger als 2 Sekunden angezeigt werden."*
- **Szenario 2 (Änderbarkeit)**: *„Neue fachliche Anforderungen zu Bedingungen für die Kooperationsangebote im Flugplan (aus Codeshare Agreements) können innerhalb von 4 Wochen live gesetzt werden."*
- **Szenario 3 (Verfügbarkeit)**: *„Der Ticketverkauf soll für den Kunden in 99,9 % der Zeit verfügbar sein, sofern der Kunde eine geeignete Internetverbindung hat. Auch geplante Downzeiten zählen zur Offline-Zeit."*
- **Szenario 4 (Fehlertoleranz)**: *„Beim Board-Verkauf soll eine Zahlungstransaktion mit Kreditkarte und NFC-Chip auch dann abgeschlossen werden können, wenn keine Verbindung zum Invoicing besteht."*

Wie wird aus den nicht-funktionalen Anforderungen eine Cloud-Architektur?

Wir werden kurz überblicksartig in drei Schritten vorstellen, wie sich daraus eine Cloud-Architektur entwickeln lässt. Grundsätzlich ist das Vorgehen wie folgt:

1. Wir markieren in den Domain Stories (in der Praxis werden es mehrere sein) diejenigen Bereiche, in denen die Qualitätsszenarien stattfinden würden.
2. Wir können jetzt feststellen, welche Bounded Kontexte von den Qualitätsszenarien betroffen sind. Dazu identifizieren wir diejenigen Bounded Kontexte in den Domain Stories, die von den Qualitätsszenarien ganz oder teilweise überlappt werden. Diese Kontexte sind von den Qualitätsszenarien und damit von den jeweiligen Qualitätsanforderungen betroffen.
3. Nun haben wir alle Qualitätsmerkmale, die für einen Bounded Kontext gelten müssen, zusammengetragen. Zu diesen können nun die richtigen Cloud-Architekturen gefunden werden. Da die verschiedenen Bounded Kontexte unterschiedliche Deployment-Einheiten darstellen können, die auf verschiedenen Knoten laufen (eigene VMs oder eigene Container), kann relativ fein gesteuert werden, welche Qualitätsmerkmale für welche Infrastruktur-Knoten gelten sollen.

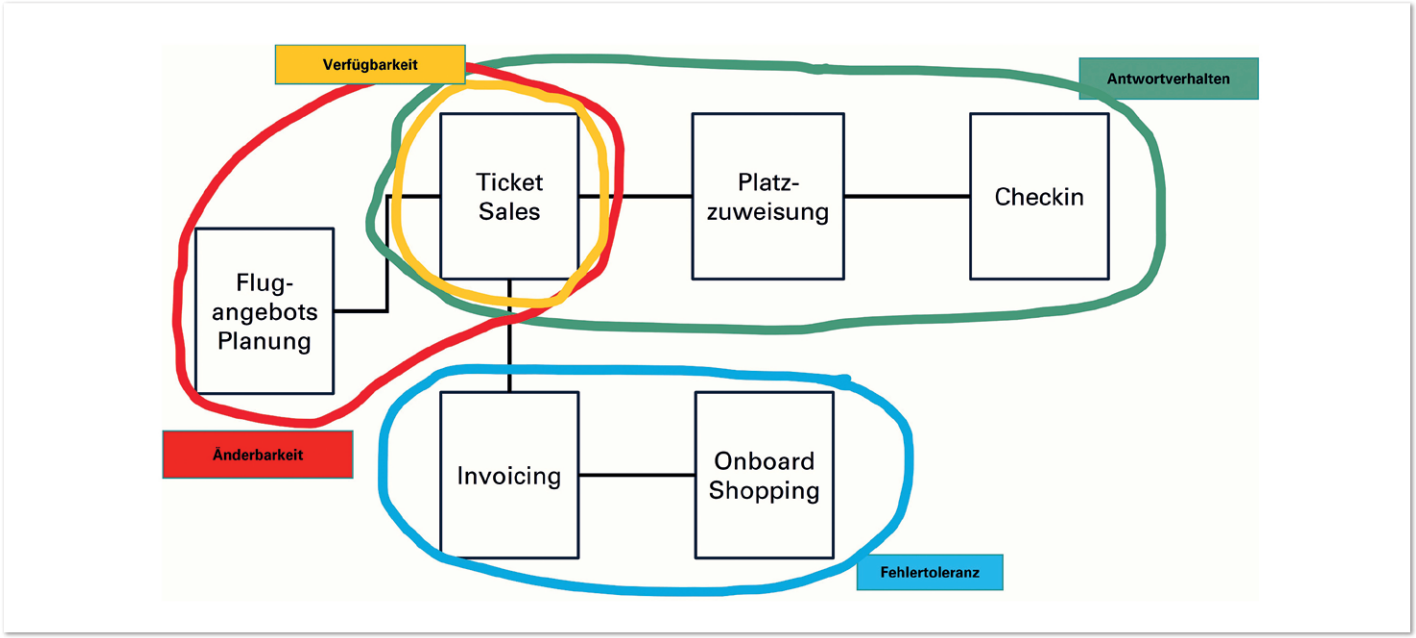


Abbildung 5: Kontext-Map mit Qualitätsszenarien (© Sönke Magnussen)

In Domain Stories betroffene Bereiche von Qualitätsszenarien erkennen

Um den ersten Schritt durchzuführen, müssen wir die Domain Stories detailliert durchgehen und die betroffenen Bereiche eines Qualitätsszenarios markieren. Es gibt einige Heuristiken, mit denen die betroffenen Bereiche von Qualitätsszenarien in Domain Stories identifiziert werden können. Beispielsweise können das sein:

- Antwortverhalten
  - Wo kommen die Akteure/Rollen in den Domain Stories vor, die im Qualitätsszenario genannt werden?
  - Welche Aktivitäten sind davon betroffen (Customer Journey)?
  - Bis wohin muss die Domain Story laufen, damit das System eine Antwort geben kann?
- Änderbarkeit
  - In welchen Schritten und an welchen Fachobjekten genau wird es mögliche Änderungen geben?
  - Wo werden gleiche Fachbegriffe verwendet?
  - Wie volatil sind welche Bereiche der Abläufe in den Domain Stories?
  - Welche Upstream-/Downstream-Abhängigkeiten gibt es in der Kontext-Map?

Weitere Heuristiken lassen sich leicht erschließen. Die so gefundene detaillierte Zuordnung von Qualitätsanforderungen zu Bounded Kontexten bietet eine gute Grundlage für den Entwurf einer Cloud-Architektur *(siehe Abbildung 5)*.

Mit Hilfe von Qualitätsmerkmalen die Cloud-Architektur entwerfen

Nicht erst seit der Entwicklung von Cloud-Architekturen wissen wir, welche Infrastruktur-Varianten gewählt werden können, um verschiedene Qualitätsanforderungen abzudecken. Sicherlich lassen sich einige Qualitätsanforderungen auch ergänzend oder alternativ durch geeignete Architekturentscheidungen in der Software abdecken oder ergänzen (beispielsweise Resilienz-Muster oder effiziente Programmierung). Cloud-Technologien vereinfachen allerdings die Abdeckung über Infrastrukturen. Die folgende Aufstellung von Qualitätskriterien zu Infrastrukturvarianten gibt einen Auszug von Heuristiken wieder und zeigt, wie auf die spezifischen Qualitätsanforderungen für einzelne Bounded Kontexte reagiert werden kann:

- Antwortverhalten → Geographische Verteilung, Skalierbarkeit
- Änderbarkeit → Rolling Updates, CI/CD, Containerization, Serverless, Automation
- Verfügbarkeit → Replikation, Rolling Updates, Monitoring, Cloud Automation, multiple DUs
- Fehlertoleranz → Managed Services, Middleware, geographische Verteilung, Redundanz

Für unsere Beispielapplikation könnte die in *Abbildung 6* dargestellte Cloud-Architektur entstehen.

Die *Flugangebotsplanung* läuft auf einer eigenen virtuellen Maschine. Es ist weder eine erhöhte Verfügbarkeit noch ein besonderes Antwortverhalten wichtig, da dieses System nur von internen Mitarbeitern verwendet wird und es auch lokal im eigenen Rechenzentrum laufen könnte, kommen wir hier mit einer einfachen virtuellen Maschine aus.

Anders ist es beim Ticketverkauf. Hier benötigen wir eine sehr hohe Verfügbarkeit sowie ein garantiertes Antwortverhalten. Daher wird für diesen Bounded Kontext eine eigene Deployment-Einheit auf horizontal skalierbaren Containern geplant. Unter Umständen ist hier auch eine geographische Verteilung sinnvoll. Beide Kontexte werden unabhängig von den anderen Bounded Kontexten deployt, da zu erwarten ist, dass sie häufig geändert werden müssen.

Die beiden Bounded Kontexte *Platzzuweisung* und *Check-in* besitzen in etwa die gleichen Anforderungen bezüglich Antwortverhalten. Da auch die zugrundeliegenden Subdomänen nahe beieinander liegen und zu erwarten ist, dass sie zu gleichen Zeiten skaliert und geändert werden müssen, wurde hier entschieden, die beiden Kontexte in einer Deployment-Einheit zu verwalten.

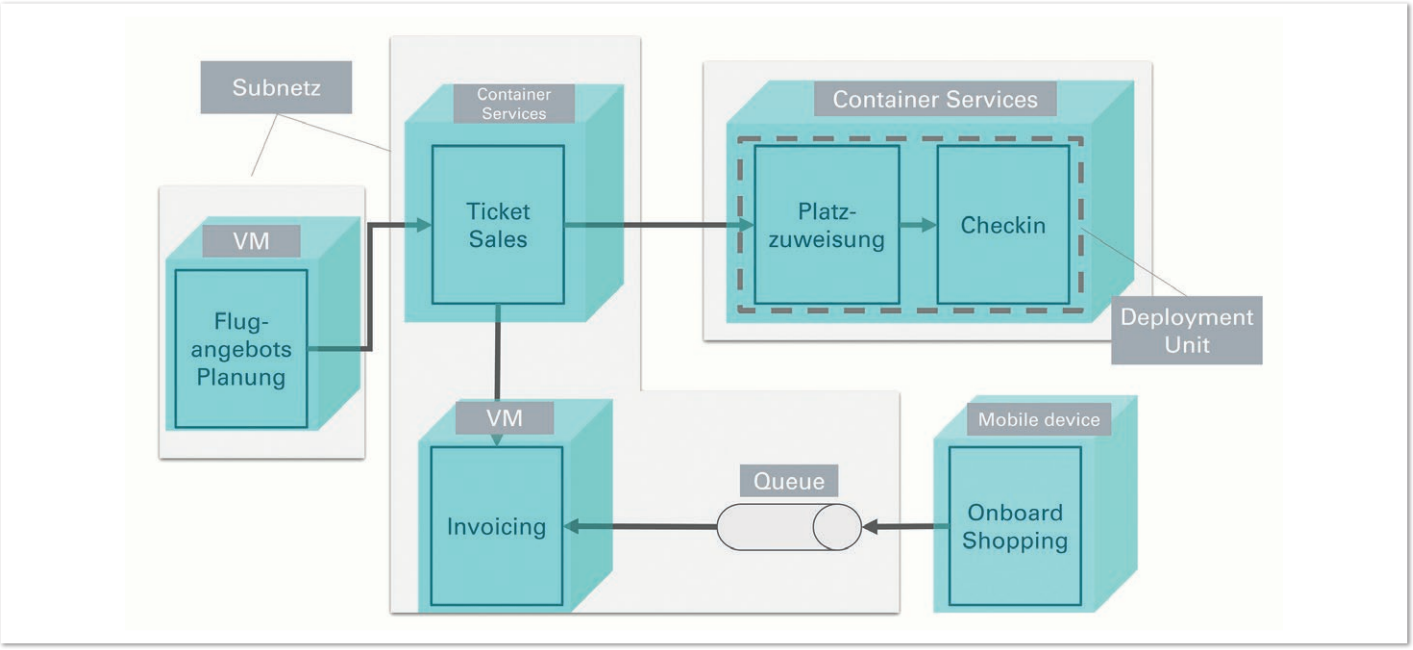


Abbildung 6: Mögliche Cloud-Architektur des Airline-Systems (© Sönke Magnussen)

Aufgrund ähnlicher Datensicherheitsanforderungen wurden die Bounded Kontexte zu *Ticketsales* und *Invoicing* in ein eigenes Subnetz gelegt.

Ausblick

Die Zukunft der Cloud für den Betrieb von Softwaresystemen bietet vielfältige, innovative Möglichkeiten, die den Betrieb und die Entwicklung von Softwarelösungen revolutionieren könnten. Daher wird eine Einbeziehung der Cloudinfrastrukturen in den Entwicklungsprozess immer wichtiger. Beispiele für die Möglichkeiten einer Revolutionierung des Betriebs sind: erhöhte Skalierbarkeit, verbesserte Sicherheit und Datenschutz, Künstliche Intelligenz, serverlose Architektur, Multi-Cloud-Strategien, verbesserte DevOps-Praktiken oder eine Kombination dieser Möglichkeiten. Mit den stetigen Fortschritten in Technologie und Infrastruktur wird die Cloud eine zentrale Rolle in der digitalen Transformation und für neue Innovation spielen. Die Methoden, wie sie in diesem Artikel beschrieben sind, werden unserer festen Überzeugung nach zu unverzichtbaren Werkzeugen für die Transformation und Weiterentwicklung von Softwaresystemen. Sie bieten die notwendigen Mittel, um Software- und Cloudarchitekturen effizient an sich verändernde Rahmenbedingungen anzupassen und weiterzuentwickeln.

Quellen

[1] Stefan Hofer, Henning Schwentner 2023: Domain Storytelling, gemeinschaftlich, visuell und agil zu fachlich wertvoller Software. dpunkt.verlag Heidelberg.

[2] Nane Kratzke 2021: Cloud-native Computing Software Engineering von Diensten und Applikationen in der Cloud, 1. Auflage. Hanser Verlag, München.

[3] Carola Lilienthal, Henning Schwentner 2023: Domain-Driven-Transformation, Monolithen und Microservices zukunftsfähig machen. dpunkt.verlag, Heidelberg.

[4] Carola Lilienthal 2024: Langlebige Softwarearchitekturen, Technische Schulden analysieren, begrenzen und abbauen. dpunkt.verlag, Heidelberg.

[5] Gernot Starke 2020: effektive Softwarearchitekturen, ein praktische Leitfaden, 9. Überarbeitete Auflage. Hanser Verlag, München.

[6] Vaughn Vernon 2017: Domain-Driven-Design kompakt (aus dem Englischen übersetzt von Carola Lilienthal und Henning Schwentner). dpunkt.verlag, Heidelberg.

[7] *https://cloud.google.com/learn/what-is-cloud-architecture?hl=de#section-2* am 24.5.2024.

[8] ISO 25010 Software Product Quality: *https://iso25000.com/index.php/en/iso-25000-standards/iso-25010* am 29.5.2024.

[9] *https://www.gartner.com/en/articles/gartner-top-10-strategic-technology-trends-for-2023* am 29.05.2024.

Dr. Sönke J. Magnussen

soenke.magnussen@wps.de

Dr. Sönke Magnussen hat an der Christian-Albrechts-Universität in Kiel Diplom-Informatik studiert und promovierte im Jahr 2003 an der Universität Lübeck im Bereich Software Engineering und Programmiersprachen. Nach der Promotion arbeitete er in verschiedenen Rollen (Softwareentwickler, IT-Architekt, Manager mit Führungsverantwortung) an der Wartung und Weiterentwicklung, an Architektur-Projekten sowie Betriebsthemen von größeren Softwaresystemen und Applikationslandschaften. Dr. Magnussen fokussiert seine Arbeit nun auf die Transformation von größeren Systemen und Organisationen mit dem Ziel, diese zukunftsfähig für die Digitalisierung oder andere große Herausforderungen aufzustellen. Meistens bedeutet dies eine Transformation hin zu modularen IT-Architekturen, Cloudtechnologie und DevOps. Dabei bildet häufig Domain-Driven-Design (DDD) das Fundament für die Erarbeitung einer angemessenen IT-Architektur sowie den richtigen Aufbau der IT-Organisation.

Wie Cloud, nur anders!

Thomas Michael, Ares Consulting GmbH

In diesem Artikel geht es um eine native Cloud-Applikation, die allerdings nicht in der Cloud, sondern auf einem embedded Kubernetes mit limitierter Hardware läuft. Erschwerend kommt hinzu, dass die Applikation dezentral eingesetzt wird. Dadurch ergeben sich aber andere Lösungen für bereits gelöste Probleme wie Rollout, Support, Datenspeicherung oder Backups.

Die Cloud. Unendliche Wei… Ressourcen. Wir schreiben das Jahr 2024 und können Java-Programme schreiben wie nie zuvor.

Wenn man heutzutage eine neue Java-Anwendung für die Cloud schreiben möchte, dann geht die Entscheidung oft in Richtung Microservices-Architektur. Man wählt zwischen SpringBoot *[1]*, Quarkus *[2]* oder einem ähnlichen Framework als Unterbau. Über CI/CD wie Jenkins *[3]* oder Bamboo *[4]* werden die Microservices gebaut, getestet und automatisch deployt.

Benötigte Funktionen werden in den Microservices einfach als Bibliotheken per Maven *[5]* oder Gradle *[6]* eingebunden. Warum das Rad neu erfinden, wenn es eine Bibliothek für mein Problem gibt? Allerdings war es auch schon vor dem Log4j-Desaster *[7]* im November 2021 oder der aktuellen XZ-Attacke *[8]* mehr als sinnvoll, die genutzten Bibliotheken zu kennen. Und deren Bibliotheken und die Bibliotheken der Bibliotheken… Ob am Ende ein Docker Container 300 MB oder 600 MB groß ist, spielt keine Rolle, denn: Wir sind ja in der Cloud! Alles straight forward!

What if

Was wäre, wenn wir nicht unendliche Ressourcen hätten? Was wäre, wenn unsere Java-Anwendung auf einer dedizierten Box mit endlichen Ressourcen laufen muss?

Warum sollte das jemand wollen?

Gegenfrage: Hast du zuhause Internet? Hast du zuhause eine rote, weiße oder schwarze Box, nennen wir sie die Freddy-Box, in der du die Verbindungsdaten deines Internetanbieters hinterlegt hast? Auf dieser Freddy-Box läuft mindestens eine Anwendung. Sie speichert deine Daten und sorgt für eine sichere Internetverbindung über deinen Internetanbieter. Eine weitere Anwendung kann beispielsweise WLAN erzeugen und so weiter. Der Punkt ist: Diese Freddy-Box hat nur begrenzte Ressourcen und damit zurück zu unserer Anwendung.

Ein international aufgestellter Konzern hat auf der ganzen Welt seine Händler. Die Händler müssen auf einem sicheren Weg mit der Konzernzentrale kommunizieren *(siehe Abbildung 1)*, um zum Beispiel

- Bestellungen aufzugeben,
- angebotene Services des Konzerns zu nutzen,
- Garantie und Reparaturen abzuwickeln,
- Mitarbeiter in Produkten zu schulen.

Die bereitgestellten Konzern-Services sind über verschiede Protokolle wie REST, SOAP oder einem anderen speziellen XML-Format, das ich vorher noch nie sah und auch hoffentlich hinterher nie wieder sehen werde, verfügbar.

Um eine sichere Kommunikation zu gewährleisten, bekommen die Händler eine eigene Konzernbox, ähnlich der Freddy-Box, um die verschiedenen Protokolle nutzen zu können. Auf dieser Freddy-Box läuft ein embedded Kubernetes, ein K3s *[9]*, als Plattform für unsere Anwendung.

Auswahl der Programmiersprache

Mit unserer Anwendung soll der Händler mit der Konzernzentrale kommunizieren. Für jedes Protokoll (REST/SOAP/XML) wird ein eigener Microservice geplant. Das Client-API besteht für jedes Protokoll aus generiertem Code sowie einigem Anpassungscode und lässt sich dadurch sehr gut separieren. Weiterhin ist ein Microservice für

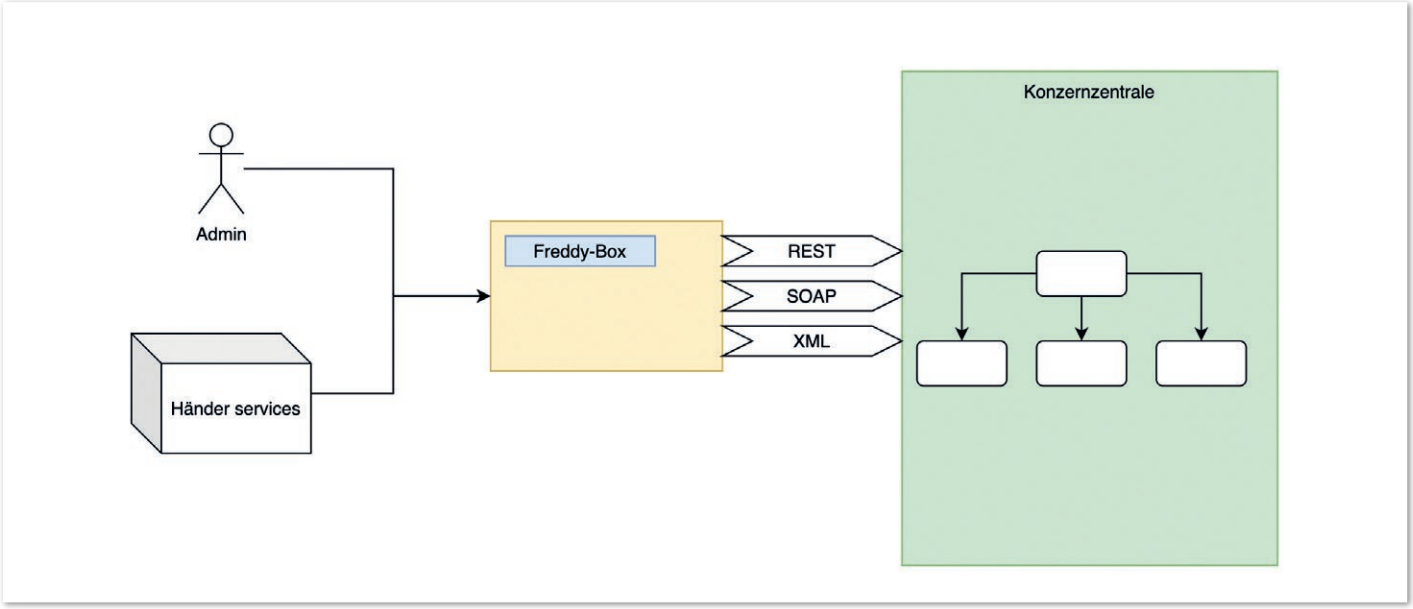


Abbildung 1: Übersicht der Kommunikation (© Thomas Michael)

die zentrale Datenspeicherung geplant, der mittels API die Daten für die anderen Microservices zur Verfügung stellt, und natürlich ein Microservice für das UI.

Also starten wir mit fünf Microservices, respektive fünf Docker-Containern, *(siehe Abbildung 2)* – aber mit welcher Programmiersprache?

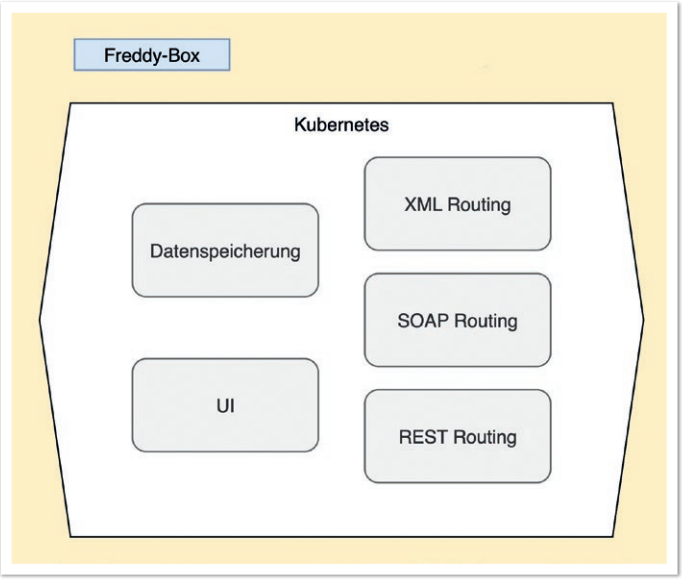


Abbildung 2: Fünf Microservices im Container (© Thomas Michael)

Abhängig von den limitierten Ressourcen ist eine Wahl gegen Java sinnvoll, da die JVM speicherhungriger ist als beispielsweise eine reine Go- oder Python-Anwendung. Allerdings besteht das Team aus Java-Entwicklern und es gibt ein Vorgängerprodukt, das auch in Java geschrieben ist. Kurz: Java ist gesetzt!

Daher bleibt nur die Wahl zwischen Spring Boot und Quarkus, was aber kein Problem darstellt, da beide Java-Frameworks ausreichend Bibliotheken und Unterstützung für unser Problem bieten. Mit beiden Frameworks sind native Docker-Container möglich und damit sind die Unterschiede beim Ressourcenverbrauch marginal. Somit ist die Entscheidung für oder gegen ein Framework unabhängig von den Ressourcen. Auch ein Mix wäre möglich. Ein guter Vergleich ist online zu finden *[10]*.

Welches Framework soll für das UI verwendet werden? Moderne Frameworks wie Angular oder React benötigen ein NodeJS und damit wieder mehr Ressourcen als eine einfache HTML-Seite. Jeweils ein guter Kompromiss sind das Web-MVC-Framework *[11]* von Spring oder das Quinoa-Framework *[12]* von Quarkus, da beide ein HTML-Frontend haben, das embedded mit einem Java-Microservice ausgeliefert werden kann.

Da das Frontend in der Hauptsache ein UI für die Datenspeicherung ist, kann es auch mit diesem Microservice zusammen ausgeliefert werden. Mit dieser Entscheidung reduziert sich die Gesamtzahl der Microservices auf vier *(siehe Abbildung 3)*.

Auswahl der Datenbank

Damit unsere Freddy-Box die unterschiedlichen Endpunkte der Konzernzentrale mit den unterschiedlichen Protokollen für REST, SOAP und XML nutzen kann, müssen – vereinfacht dargestellt – unter anderem die Zugangsdaten gespeichert werden. Also Username und Passwort und andere Key/Value-Paare.

Welche Datenbank beziehungsweise welcher Datenbanktyp ist sinnvoll? SQL oder NoSQL? MongoDB ist sehr beliebt und für Key/Value auch sinnvoll. Da wir embedded unterwegs sind, haben wir uns entschieden, die Key/Value-Paare, und später auch JSON-Strukturen, direkt als Datei in einem angebundenen Laufwerk zu speichern *(siehe Abbildung 4)*. Warum?

1. Dies erspart den Ressourcen-Overhead von weiteren Containern für eine Datenbank und
2. architekturbedingt hat nur ein Microservice Schreib-/ und Leseberechtigung.

Embedded bedeutet aber, dass wir das Laufwerk selbst mitbringen und konfigurieren müssen. Wir können nicht ohne weiteres einfach auf die Festplatte der Freddy-Box zugreifen.

Für unsere K3s-Distribution *[13]* ist Longhorn *[14]* empfohlen, um ein Laufwerk bereit zu stellen.

Rollout und Update

Der Bamboo baut die Microservices, also die Docker-Images und im Erfolgsfall wird sie direkt deployt. Wie geht das jedoch bei einer dezentralen Anwendung, weltweit verteilt? Ich behaupte: gar nicht. Was aber geht:

- Bereitstellung der Anwendung als Helm-Charts *[15]*
- Bereitstellung der nötigen Docker-Images
- Einrichtung eines RSS-Feeds oder eines Mailverteilers für Updates

Über den RSS-Feed oder den Mailverteiler erfährt der Händler von der neuen Version der Anwendung und kann selbst die Aktualisierung vornehmen.

Support

Wie supported man eine verteilte Anwendung? Man ist auf die Logs angewiesen! Aus Architektursicht speichert man die Logs zentral

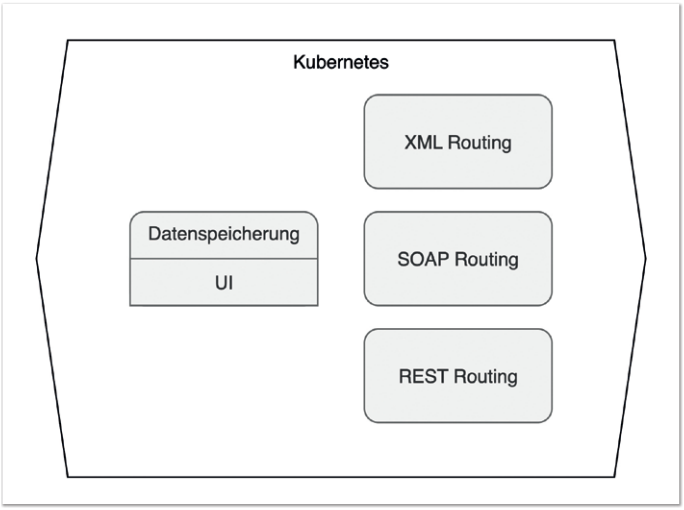


Abbildung 3: Clusterübersicht mit nur noch vier Containern (© Thomas Michael)

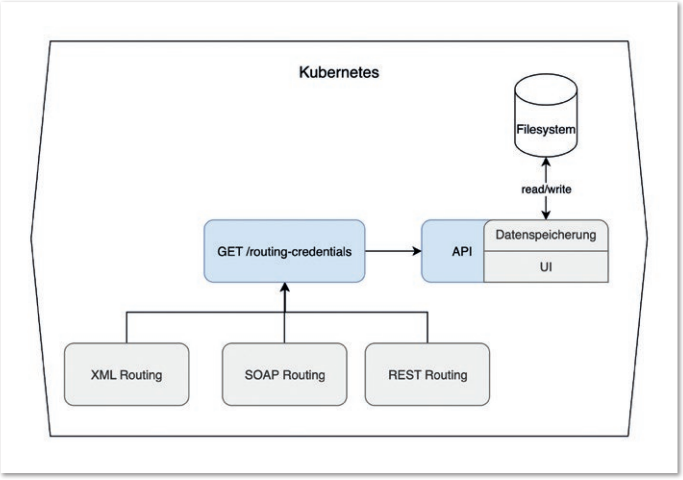


Abbildung 4: Zugriff der Routings-MS auf den Datenspeicherungs-MS per GET (© Thomas Michael)

auf einem gemeinsamen Laufwerk. Im Fehlerfall werden die Logdateien gezippt und dem Support-Team bereitgestellt. Ein Tool wie Logstash *[16]* oder FluentD *[17]* kann bei der Umsetzung helfen, um die Logs zu zentralisieren.

Backups

Hier gilt es, vorab zu klären, was Bestandteil eines Backups ist und was nicht. Da die Anwendung jederzeit mittels Helm-Charts neu aufgesetzt werden kann, müssen nur die Daten vom Longhorn-Laufwerk gesichert werden. Aber da die Anwendung embedded läuft, reicht es nicht, auf Longhorns eigene Backup- und Restore-Strategy *[18]* zu vertrauen. Wenn die Freddy-Box zerstört wird, ist alles weg. Daher muss das Backup extern erfolgen.

Aus architektonischer Sicht müssen nur die Dateien aus dem Longhorn-Laufwerk auf einer externen Festplatte oder einem Remote-Laufwerk gespeichert werden. Hier bin ich sehr an einem Erfahrungsaustausch mit den Lesern interessiert.

Skalierung

Ein Vorteil von Microservices? Bei Bedarf kann horizontal skaliert werden, also können weitere Pods mit Docker-Containern des Microservices starten. Nur, wo kommen die Ressourcen her? Jeder Microservices sollte von Anfang an so dimensioniert sein, dass ein Maximum an möglichen Ressourcen genutzt wird. Skalierung ist, meiner Meinung nach, im embedded Kontext keine gute Idee.

Fazit

Wie viele Microservices sind wirklich nötig, wenn die Ressourcen knapp sind? In unserem Beispiel könnten die drei Microservices für das Routing zu einem zusammengefasst werden, um ressourcenschonend zu arbeiten. Bei jeder eingebundenen Bibliothek sollte zweimal überlegt werden, ob sie notwendig ist, damit die Größe des Docker-Container gering bleibt. Rollout und Support sind zufriedenstellend gelöst. Für automatisierte Backups wird noch eine geeignete Lösung gesucht.

Quellen

[1] *https://spring.io/projects/spring-boot*
[2] *https://quarkus.io*
[3] *https://www.jenkins.io*
[4] *https://www.atlassian.com/de/software/bamboo*
[5] *https://maven.apache.org*
[6] *https://gradle.org*
[7] *https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Empfehlungen-nach-Angriffszielen/Webanwendungen/log4j/log4j_node.html*
[8] *https://www.bsi.bund.de/SharedDocs/Cybersicherheitswarnungen/DE/2024/2024-223608-1032.html*
[9] *https://k3s.io*
[10] *https://www.baeldung.com/spring-boot-vs-quarkus*
[11] *https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html*
[12] *https://quarkus.io/blog/quinoa-modern-ui-with-no-hassle/*
[13] *https://docs.k3s.io/storage*
[14] *https://longhorn.io*
[15] *https://helm.sh/de/*
[16] *https://www.elastic.co/de/logstash*
[17] *https://www.fluentd.org*
[18] *https://longhorn.io/docs/1.6.1/snapshots-and-backups/*

Thomas Michael
thomas.michael@ares-consulting.de
Twitter: @Kroneker_Delta

Thomas Michael ist Softwareentwickler aus Leidenschaft. Nach dem Informatikstudium widmete er sich ganz der Softwareentwicklung und hat damit sein Hobby zum Beruf gemacht. Nach dem ersten Kontakt mit der Cloud lassen ihn die Möglichkeiten nicht mehr los. Als Cloud Expert Consult arbeitet er bei der Ares Consulting GmbH in Braunschweig an der erfolgreichen Umsetzung von Projekten mit Cloud-Technologien für interne und externe Kunden. Privat arbeitet er an kleineren Spring-Boot- oder Angular-Anwendungen, die immer öfter durch serverless Lambda ersetzt werden.

iJUG Verbund

NEWSLETTER

Anmeldung

Java aktuell: der iJUG Newsletter informiert dich alle vier Wochen über das aktuelle Geschehen in der Java-Welt und im iJUG.

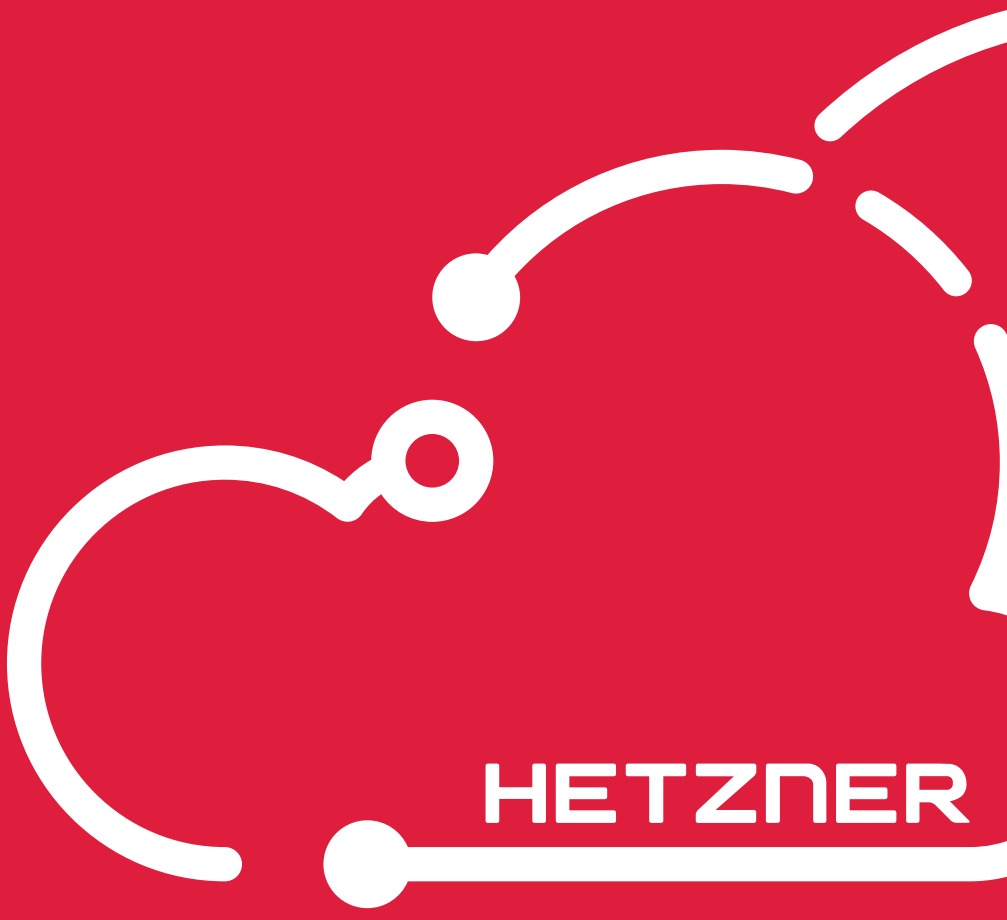
Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!

https://shop.doag.org/newsletter/

oder nach dem Login mit euren Zugangsdaten direkt über euer Profil abonnieren.

Automatisierte Nutzung der Hetzner Cloud

Marcus Fihlon, Adcubum AG



Schon seit vielen Jahren ist die Cloud in aller Munde. Wobei „Cloud" gefühlt ein Synonym für die Amazon Cloud, Google Cloud und Azure Cloud darstellt. Doch was ist mit deutschen Clouds? Auch die gibt es, zum Beispiel von Hetzner. In diesem Artikel möchte ich dir zeigen, wie einfach es ist, Anwendungen automatisiert in der Hetzner Cloud zu publizieren und DSGVO-konform in Deutschland zu hosten.

Wir, die Infrastruktur-Gruppe des iJUG, setzen dazu auf das Kommandozeilen-Werkzeug *hcloud* *[1]*. Es handelt sich dabei um ein Open-Source-Werkzeug für die Kommandozeile zum Zugriff auf die Hetzner Cloud. Die entsprechenden Befehle fassen wir in Shell-Skripten zusammen.

Auf den folgenden Seiten werde ich Schritt für Schritt das automatisierte Bereitstellen von Diensten in der Hetzner Cloud mit Skripten an der Kommandozeile aufzeigen. Als Beispiel und Platzhalter für eine Webanwendung verwende ich der Einfachheit halber direkt den NGINX-Webserver *[2]*, der in einer Docker-Umgebung laufen soll.

Vorbereitungen

Es sind auch Vorbereitungen im Webinterface der Hetzner Cloud nötig. Natürlich benötigst du einen Account für Hetzner Cloud. In diesem Account muss eine Kreditkarte hinterlegt sein, denn die Nutzung der Hetzner Cloud ist kostenpflichtig. Im Web-Interface der Hetzner Cloud müssen drei Vorbereitungsaufgaben erledigt werden.

1. Du benötigst ein Projekt. Das kann ein eigenes Projekt sein, das du selbst angelegt hast, oder ein existierendes Projekt, für das dir jemand Zugriffsrechte eingeräumt hat. Ich habe mir ein neues Projekt angelegt und es „demo-project" genannt.
2. In diesem Projekt muss nun noch im Bereich „Sicherheit" auf dem Tab „SSH Keys" der öffentliche Teil des SSH-Schlüssels hinterlegt werden, mit dem man sich später auf dem Server per SSH anmelden möchte. Der hier vergebene Name wird später noch benötigt. Ich habe ihn „demo-ssh-key" genannt.
3. Ebenfalls im Bereich „Sicherheit" aber im Tab „API-Tokens" muss nun noch ein API-Token hinzugefügt werden, der anschließend vom hcloud-Befehl zur Authentifizierung verwendet werden kann. Dieser Token benötigt die Berechtigungen „Lesen & Schreiben". Achtung: Der Token wird nur einmal angezeigt! Notiere ihn dir daher an einem sicheren Ort, zum Beispiel deinem Passwort-Manager.

Die Vorbereitungen im Web-Interface der Hetzner Cloud sind hiermit abgeschlossen. Jetzt musst du nur noch das Werkzeug *hcloud* lokal auf deinem Rechner installieren. Die Installation ist am einfachsten mittels Homebrew *[3]* (macOS und Linux) oder Scoop *[4]* (Windows) durchzuführen. Details dazu findest du im GitHub Repo *[1]* von *hcloud*.

REST ohne REST

Die Hetzner Cloud bietet eine REST-API, die ideal für Automatisierungsaufgaben ist. Das hcloud-CLI-Tool macht nichts anderes, als die REST-API hinter deutlich einfacheren sowie leichter zu merkenden Befehlen zu verstecken und sich zusätzlich um die Authentifizierung zu kümmern. Mit der integrierten Hilfe kommt man dabei sehr schnell ans Ziel, ohne die umfangreiche Dokumentation der REST-API studieren zu müssen. Denn sind wir mal ehrlich: Wer liest schon gerne die Dokumentation, wenn es sich vermeiden lässt?

Projekte und Kontexte

Das, was in der Hetzner Cloud die Projekte sind, wird beim hcloud-Kommando „Kontext" genannt. Projekte und Kontexte sind nichts anderes als zwei Namen für das Gleiche. Um an der Kommandozeile Zugriff auf das Projekt zu erhalten, muss man dieses dem hcloud-Werkzeug als Kontext bekannt machen. Dazu wird der zuvor angelegte API-Token benötigt. Etwas ungewöhnlich lautet das Kommando nicht „login", sondern „create": `hcloud context create demo-project`

Nach der Eingabe des API-Token sollte eine Bestätigung erscheinen und ab sofort kann mit diesem Projekt an der Kommandozeile gearbeitet werden. Der Wechsel zwischen Projekten beziehungsweise Kontexten ist ganz einfach mit `hcloud context use demo-project` möglich. Du kannst dir natürlich auch alle Kontexte anzeigen lassen, für die du dich lokal authorisiert hast: `hcloud context list`.

Firewall einrichten

Uns allen sollte klar sein, dass jeder Server, den wir betreiben, mit einer Firewall geschützt werden sollte. Die Hetzner Cloud bietet uns eine Firewall an, die wir ganz einfach erstellen und konfigurieren können. Das geht über eine einfache Textdatei im JSON-Format. In unserem Beispiel möchte ich den Zugriff per SSH erlauben und natürlich muss auch der von uns angebotene Dienst über das Web erreichbar sein. Insofern müssen die Ports 22 für SSH, 80 für HTTP und 443 für HTTPS geöffnet werden.

```
[
  {
    "description":"ssh",
    "destination_ips":[],
    "direction":"in",
    "port":"22",
    "protocol":"tcp",
    "source_ips":["0.0.0.0/0","::/0"]
  },
  {
    "description":"web",
    "destination_ips":[],
    "direction":"in",
    "port":"80",
    "protocol":"tcp",
    "source_ips":["0.0.0.0/0","::/0"]
  },
  {
    "description":"websecure",
    "destination_ips":[],
    "direction":"in",
    "port":"443",
    "protocol":"tcp",
    "source_ips":["0.0.0.0/0","::/0"]
  }
]
```

Listing 1: firewall-config.json: Die Konfiguration der Firewall.

Wie das aussieht, wird in *Listing 1* gezeigt. Jeder Abschnitt definiert eine Portfreigabe. Die Beschreibung ist frei wählbar. Die Ziel-IP können wir leer lassen, da wir die Firewall später direkt mit dem Server verknüpfen und so eine 1:1-Beziehung entsteht. Die Angabe des Ports ist klar, ebenso das Protokoll (in unserem Fall `tcp`). Über die Angabe der Quell-IPs kann eingeschränkt werden, wer unseren Dienst in Anspruch nehmen darf. Wir möchten hier keine Einschränkung, daher erlaubt die Angabe von `0.0.0.0/0` einen Zugriff von allen Systemen mit IPv4-Adresse sowie `::/0` von allen Systemen mit IPv6-Adresse.

Permanente IP einrichten

Würden wir nun direkt einen neuen Server erstellen, bekommt dieser eine zufällige IP zugewiesen, die nach dem Löschen des Servers wieder freigegeben wird. Für unsere Tests wäre das ok, jedoch nicht für ein produktives System. Denn wir möchten das gegebenenfalls ins DNS eintragen, TLS-Zertifikate erstellen, und vieles mehr. Daher sollten die IP-Adressen von den Servern entkoppelt werden.

Das ist in der Hetzner Cloud recht einfach möglich. Wir müssen nur die IP-Adressen einzeln erstellen und können diese später dem Server zuweisen. Einzeln erstellte IP-Adressen werden nicht wieder freigegeben, wenn der Server gelöscht wird. Das muss man explizit veranlassen.

IPv4-Adressen sind mittlerweile ein rares Gut geworden, da der Adresspool ausgeschöpft ist. Daher kosten diese eine monatliche Gebühr. IPv6-Adressen dagegen sind mehr verfügbar als es Sandkörner am Strand gibt. Daher sind diese in der Regel gratis. Auf jeden Fall sollte jeder Server per IPv6 erreichbar sein. IPv4 wird immer weniger genutzt und ist nicht mehr in jedem Fall nötig. Ich zeige hier dennoch beides und statte unseren Demo-Server sowohl mit einer IPv4- als auch einer IPv6-Adresse aus.

IP-Adressen sind auf Netzwerke aufgeteilt. Die Netzwerke wiederum liegen in einem Rechenzentrum. Hetzner betreibt verschiedene Rechenzentren in Deutschland sowie zusätzlich noch welche in Finnland und mittlerweile auch in den USA. Wir möchten unsere Server in Deutschland hosten. Hier können wir zwischen den Rechenzentren in Falkenstein und Nürnberg wählen. Jedem Rechenzentrum ist ein Kürzel zugeordnet. Für Nürnberg beispielsweise `nbg1-dc3`. Das Rechenzentrum muss beim Einrichten der IP mit angegeben werden und es muss das gleiche Rechenzentrum sein, in dem wir im nächsten Schritt den Server einrichten möchten.

Damit man später einfach auf die IP-Adresse referenzieren kann, sollte man ihr einen Namen geben. Ich verwende dazu `demo-ipv4` und `demo-ipv6`. Ebenfalls ist es empfehlenswert, die IP-Adresse gegen versehentliches Löschen im Webinterface zu schützen. Die auszuführenden Befehle lauten

`hcloud primary-ip create --datacenter nbg1-dc3 --enable-protection delete --type ipv4 --name demo-ipv4` für eine IPv4-Adresse

sowie

`hcloud primary-ip create --datacenter nbg1-dc3 --enable-protection delete --type ipv6 --name demo-ipv6` für eine IPv6-Adresse.

Die entsprechenden Befehle findest du auch im Demo-Repository *[5]* in der Datei `server-init.sh`.

Server erstellen

Wir haben eine Firewall-Konfiguration und IPs. Jetzt können wir uns endlich um den eigentlichen Server kümmern. Dazu gehören drei Schritte: Firewall einrichten, Server einrichten und den Server gegen versehentliches Löschen schützen.

Beginnen wir mit der Firewall. Wir weisen *hcloud* an, eine neue Firewall einzurichten und dabei die Konfiguration aus der von uns angelegten Datei (`firewall-config.json`) zu lesen. Der Firewall geben wir einen Namen (`demo-firewall`), um leichter auf diese referenzieren zu können. Der komplette Befehl dazu lautet: `hcloud firewall create --name demo-firewall --rules-file firewall-config.json`.

Jetzt kommt der Server an die Reihe. Wir weisen ihm die von uns vorgängig erstellten IPs zu und geben das gleiche Rechenzentrum an. Unser Server soll auf Debian 12 basieren und bekommt den Namen `demo-server` (das ist nicht der DNS-Name). Dann geben wir noch unseren SSH-Schlüssel an, damit wir uns später auch per SSH auf den Server verbinden können. In der Hetzner Cloud gibt es verschieden ausgestattete Server zur Auswahl. Wir wählen den Typ `cx11` (1 vCPU, 1 GB RAM, 20 GB SSD). Abschließend folgt noch die Angabe der Firewall, die wir gerade eben erstellt haben. Das ergibt den folgenden, recht langen Befehl: `hcloud server create --primary-ipv4 demo-ipv4 --primary-ipv6 demo-ipv6 --datacenter nbg1-dc3 --image debian-12 --name demo-server --ssh-key demo-ssh-key --type cx11 --firewall demo-firewall`.

Damit der eben erstellte Server nicht versehentlich, zum Beispiel mit einem falschen Klick im Webinterface, gelöscht wird, können wir ihn noch davor schützen: `hcloud server enable-protection demo-server delete rebuild`.

Achtung: Die Bereitstellung des Servers verläuft asynchron! Dass der Befehl erfolgreich durchgelaufen ist, bedeutet nicht, dass der Server sofort einsatzbereit ist. Es dauert normalerweise einige Sekunden, kann aber manchmal auch einige Minuten in Anspruch nehmen. Unser Server ist noch nicht weiter konfiguriert, einen Dienst anzubieten. Dass er läuft, sehen wir im Webinterface der Hetzner Cloud. Und wir können uns bereits per SSH mit dem Server verbinden.

Die Befehle aller drei Schritte findest du auch im Demo-Repository *[5]* in der Datei `server-create.sh`.

Alles wieder abräumen

Was wir bisher an der Kommandozeile erstellt haben, können wir auf gleichem Wege natürlich auch wieder bereinigen. Dazu deaktivieren wir den Löschschutz unseres Servers:

```
hcloud server disable-protection demo-server delete rebuild
```

Dann können wir unseren Server herunterfahren:

```
hcloud server shutdown demo-server
```

Nun lässt er sich auch löschen:

```
hcloud server delete demo-server
```

Die Firewall dürfen wir auch nicht vergessen:
`hcloud firewall delete demo-firewall`
Das Einzige, was wir nicht löschen, sind die IP-Adressen. Damit wir genau die gleichen IP-Adressen beim erneuten Erzeugen des Servers wiederverwenden können und sich diese somit nicht ändern.

Diese Befehle zum Aufräumen findest du auch im Demo-Repository *[5]* in der Datei `server-delete.sh`.

Software installieren

Unser Server ist bisher mit einem Standard-Image auf Basis von Debian 12 ausgerüstet. Über eine Konfigurationsdatei im YAML-Format *[6]* können wir weitere Pakete angeben, die beim Installieren des Servers automatisch per APT (dem Repository Manager von Debian) hinzugefügt werden sollen. In meinem Beispiel nenne ich diese Datei `cloud-config.yaml`. Wie du in *Listing 2* sehen kannst, wird Hetzner darin angewiesen, die Paketlisten zu aktualisieren (`apt_update: true`), aber keine Updates zu installieren (`apt_upgrade: false`). Dann folgt eine Auflistung von APT-Paketnamen, die installiert werden sollen.

```
apt_update: true
apt_upgrade: false
packages:
- apparmor
- binutils
- ca-certificates
- curl
- fail2ban
- gnupg
- lsb-release
- python3-systemd
```

Listing 2: cloud-config.yaml: Zusätzliche Software installieren.

Diese Datei können wir dem hcloud-Befehl zum Erzeugen eines neuen Servers über den zusätzlichen Parameter `--user-data-from-file cloud-config.yaml` mitgeben. Der komplette Befehl lautet nun: `hcloud server create --primary-ipv4 demo-ipv4 --primary-ipv6 demo-ipv6 --datacenter nbg1-dc3 --image debian-12 --name demo-server --ssh-key demo-ssh-key --type cpx11 --firewall demo-firewall --user-data-from-file cloud-config.yaml`.

Dateien erstellen

Wie du vielleicht in *Listing 2* gesehen hast, installieren wir fail2ban. Das ist ein *Intrusion Prevention System [7]* und benötigt eine Konfigurationsdatei. Diese können wir ebenfalls über unsere schon angelegte Datei `cloud-config.yaml` beim Erstellen des Servers automatisiert erzeugen lassen. Den in *Listing 3* gezeigten Inhalt fügen wir einfach am Ende hinzu.

```
write_files:
- content: |
      [DEFAULT]
      backend = systemd
      bantime = 1d
      maxretry = 2
      findtime = 1h
      [sshd]
      enabled = true
  path: /etc/fail2ban/jail.local
- content: |
      {
         "ipv6": true,
         "fixed-cidr-v6": "fd00:ffff::/80"
      }
  path: /etc/docker/daemon.json
```

Listing 3: cloud-config.yaml: Dateien erzeugen.

Der neue Abschnitt `write_files:` ist relativ selbsterklärend. Jede Datei wird mit `- content: |` eingeleitet, dann folgt der Dateiinhalt. Den Abschluss bildet die Angabe des Pfads und des Dateinamens: `path: /etc/fail2ban/jail.local`.

Zu beachten ist dabei die korrekte Einrückung in der YAML-Datei. In unserem Beispiel sind das vier Leerzeichen vor jeder Zeile des Dateiinhalts, die *nicht* in die Zieldatei übernommen werden, sowie zwei Leerzeichen vor der Pfadangabe. Dies ist der YAML-Syntax geschuldet und etwas gewöhnungsbedürftig.

Listing 3 enthält noch eine zweite Datei, die angelegt wird und Docker für IPv6 fit macht. Aber Docker ist noch nicht installiert! Denn wir sollten Docker selbst aus den originalen Docker-Repositories installieren und nicht das Debian-Paket verwenden, da das leider hoffnungslos veraltet ist.

Befehle ausführen

Für die Installation von Docker aus den originalen Docker-Repositories müssen wir zuerst das Repository selbst und auch den GnuPG-Schlüssel zur Paketverifikation unserer Debian-Installation hinzufügen. Die dazu nötigen Befehle können wir der entsprechenden Dokumentation von Docker entnehmen und 1:1 in unsere existierende Datei `cloud-config.yaml` einfügen. Dazu erstellen wir einfach einen Abschnitt `runcmd:` und führen jeden Befehl einzeln als Liste auf *(siehe Listing 4)*, wie wir ihn auch selbst an der Kommandozeile eingeben würden.

```
runcmd:
- mkdir -p /etc/apt/keyrings
- curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /etc/apt/keyrings/docker.gpg
- chmod a+r /etc/apt/keyrings/docker.gpg
- echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/
linux/debian $(lsb_release -cs) stable" | tee /etc/apt/sources.list.d/docker.list > /dev/null
- apt -y install docker-ce docker-ce-cli containerd.io docker-compose-plugin
- reboot
```

Listing 4: cloud-config.yaml: Befehle ausführen.

Mit `mkdir` erstellen wir ein Verzeichnis für die GnuPG-Schlüssel, mit denen der Paketmanager-APT die Unversehrtheit der Pakete verifiziert. In der zweiten Zeile laden wir die GnuPG-Schlüssel per `curl` und speichern sie in das eben erstellte Verzeichnis. Mittels `chmod` sichern wir korrekte Zugriffsrechte auf die Schlüsseldatei.

Mittels einer Kombination aus echo und `tee` fügen wir eine Referenz auf das originale Docker-Repository zu APT hinzu. Dann werden alle nötigen Pakete per `apt` installiert und nach einem abschließenden `reboot` läuft unser Server mit installiertem aktuellem Docker.

Es ist Zeit – oder auch nicht?

Leider gibt es häufig ein Problem mit der Zeit. Genauer gesagt der Zeitzone. Diese ist nicht immer – oder eher fast nie – korrekt eingestellt. Das sollten wir korrigieren, damit es mit den Services auf unserem Server und gegebenenfalls später genutzten SSL-Zertifikaten keine Probleme gibt. Dazu fügen wir einfach am Anfang unserer Datei `cloud-config.yaml` eine Zeile mit dem Inhalt `timezone: Europe/Berlin` hinzu.

Je nach gewähltem Betriebssystem-Abbild reicht das jedoch noch nicht. Daher empfiehlt es sich immer, im Abschnitt `runcmd:` der `cloud-config.yaml` zusätzlich den entsprechenden Linux-Befehl zum Setzen der Zeitzone einzubauen: `timedatectl set-timezone Europe/Berlin`.

Endlich: Unser Service

Nun wird es Zeit, dem Server auch etwas zu tun zu geben. Als Platzhalter für eine Anwendung deiner Wahl habe ich mich in diesem Beispiel für *nginx* entschieden. Doch ich möchte nicht das Debian-Paket nehmen und installieren, sondern das Docker-Image. Dieses möchte ich in einer Docker-Compose-Datei wrappen und beim Hochfahren des Servers automatisch als Dienst starten lassen. So hast du eine funktionierende Vorlage für deine eigene Anwendung, sofern du diese als Docker-Image publizierst.

Zuerst legen wir die Datei `docker-compose.yaml` an. Natürlich nicht per Hand, sondern automatisiert über einen weiteren Abschnitt in der Datei `cloud-config.yaml`. Auf den Inhalt der Docker-Compose-Datei gehe ich in diesem Artikel nicht näher ein. Du findest ihn in *Listing 5*. Ich habe mich entschieden, die Datei `docker-compose.yaml` in einem eigenen Unterverzeichnis `demo` des `root`-Users anzulegen.

Damit Linux unser Docker-Image beim Hochfahren startet, müssen wir noch einen `systemd`-Service anlegen. Das ist nichts anderes als

```
- content: |
    services:
      nginx:
       restart: unless-stopped
         image: nginx
         ports:
           - "80:80/tcp"
           - "80:80/udp"
           - "443:443/tcp"
           - "443:443/udp"
  path: /root/demo/docker-compose.yaml
```

Listing 5: cloud-config.yaml: Docker-Compose-Datei anlegen.

```
- content: |
    [Unit]
    Description=Docker Compose Demo Service
    Requires=docker.service
    After=docker.service

    [Service]
    Type=oneshot
    RemainAfterExit=yes
    WorkingDirectory=/root/demo
    ExecStart=/usr/bin/docker compose up -d
    ExecStop=/usr/bin/docker compose down
    TimeoutStartSec=0

    [Install]
    WantedBy=multi-user.target
  path: /etc/systemd/system/demo-compose.service
```

Listing 6: cloud-config.yaml: Docker-Service-Datei anlegen.

eine Textdatei, in der der Service beschrieben wird, und lässt sich ganz wunderbar ebenfalls über eine weitere Ergänzung der `cloud-config.yaml` bewerkstelligen, wie *Listing 6* zeigt.

Der Service muss nur noch aktiviert werden. Dazu ergänzen wir, wieder in der `cloud-config.yaml`, fast ganz am Ende vor dem `reboot` den entsprechenden Befehl: `systemctl enable demo-compose`.

Selbstverständlich findest du die vollständige Datei `cloud-config.yaml` wie alles andere ebenfalls im Demo-Repository *[5]*.

Damit ist alles Nötige erledigt. Über Kommandozeilen-Befehle können neue Server-Instanzen in wenigen Sekunden angelegt werden, die alle nötige Software selbstständig installieren und konfigurieren. Ebenso schnell können diese Instanzen restlos entfernt werden.

Aller guten Dinge sind drei

Um es für uns noch einfacher zu machen, gruppieren wir alle bisher gelernten hcloud-Befehle in drei Shell-Skripte:

1. `server-create.sh`: Erstellen der Firewall sowie des Servers und Schutz des Servers gegen versehentliches Löschen *(siehe Listing 7)*.
2. `server-delete.sh`: Aufheben des Server-Schutzes, Herunterfahren und Löschen des Servers sowie der Firewall *(siehe Listing 8)*.
3. `server-reset.sh`: Baut alles neu auf, indem zuerst mittels `server-delete.sh` alles gelöscht und anschließend mittels `server-create.sh` neu erzeugt wird *(siehe Listing 9)*.

Alle drei Listings beginnen mit `set -e` um sicherzustellen, dass das Skript bei einem Fehler sofort abbricht und nicht noch andere Befehle in der Datei aufruft. Der erste Befehl ist stets `hcloud context use demo-project`, um alle folgenden Befehle im richtigen Kontext abzusetzen.

Fazit

Wir haben unsere Infrastruktur mittels des hcloud-Werkzeugs und einfachen Skripten aufgebaut, die nur aus wenigen und leicht zu verstehenden Befehlen bestehen. Durch den einfachen Aufbau können neue Mitglieder, die bei uns in der Administration mitarbeiten

```
set -e
hcloud context use demo-project
hcloud firewall create --name demo-firewall --rules-file firewall-config.json
hcloud server create --primary-ipv4 demo-ipv4 --primary-ipv6 demo-ipv6 --datacenter nbg1-dc3 --image debian-12
--name demo-server --ssh-key demo-ssh-key --type cpx11 --firewall demo-firewall --user-data-from-file cloud-config.
yaml
hcloud server enable-protection demo-server delete rebuild
```

Listing 7: server-create.sh: Anlegen von Firewall und Server.

```
set -e
hcloud context use demo-project
hcloud server disable-protection demo-server delete rebuild
hcloud server shutdown demo-server
hcloud server delete demo-server
hcloud firewall delete demo-firewall
```

Listing 8: server-delete.sh: Löschen von Server und Firewall.

```
set -e
./server-delete.sh
./server-create.sh
```

Listing 9: server-reset.sh: Löschen und neu anlegen.

möchten, recht schnell einsteigen. Zusätzliche Dienste sind rasch verfügbar gemacht, sofern es entsprechende Docker-Images gibt.

Bei Problemen mit einem Server können wir von unseren lokalen Systemen aus jederzeit den Server neu aufsetzen lassen, was meist nur wenige Augenblicke dauert. Auch die Skalierung des Servers bezüglich der verfügbaren vCores und des Arbeitsspeichers stellt auf diese Weise kein Problem dar, wir passen einfach den Typ im Skript `server-create.sh` an und führen das Skript `server-reset.sh` aus. Ein oder zwei Minuten später läuft der „größere" Server anstelle des alten.

Selbst bei Updates ist es unter Umständen einfacher, das Skript `server-reset.sh` aufzurufen und den Server neu installieren zu lassen – mit allen zum jetzigen Zeitpunkt aktuellen Versionen der Docker-Images unserer Services.

Zugegeben, das ist nicht vollständig unterbrechungsfrei. In unserem Fall ist das aber auch nicht nötig, denn einen Reset führen wir nur sehr selten durch.

Den gesamten Quelltext aus diesem Artikel findest du in einem eigenen GitHub-Repository *[5]*. Du kannst das Repo klonen, im Shell-Skript den Kontext anpassen und laufen lassen. Benutze es gerne als Basis für deine eigenen Experimente.

Ausblick

Um dieses Setup sicher produktiv nutzen zu können, fehlen noch drei wichtige Punkte:

1. Dienste verschlüsselt anbieten: SSL-Zertifikate beispielsweise von *Let's Encrypt [8]* automatisiert beantragen und einbinden.
2. Daten persistent speichern: Mit einem Hetzner Volume die Daten des Service so ablegen, dass diese das Löschen und Neuerstellen eines Servers überleben.
3. Verschlüsseltes Backup: Mittels einer Hetzner Storage Box und Borg Backup *[9]* automatisierte und sicher verschlüsselte Datensicherungen durchführen.

Wenn du mehr darüber erfahren möchtest, wie wir *Let's Encrypt* eingebunden haben, Daten sicher speichern oder verschlüsselte Datensicherungen durchführen, melde dich gerne per E-Mail bei *redaktion@ijug.eu* und bei größerem Interesse wird einer aus unserer Gruppe gerne einen weiteren Artikel erfassen.

Die Infrastruktur-Gruppe des iJUG

Am Anfang des Artikels habe ich die *Infrastruktur-Gruppe des iJUG* erwähnt. Der iJUG *[10]* ist der *Interessenverbund der Java User Groups e. V.* und umfasst aktuell 42 Mitglieder aus Deutschland, Österreich und der Schweiz. Eine Liste findest du auf der letzten Seite dieses Magazins, denn das wird ebenfalls vom iJUG herausgegeben.

Wir von der iJUG-Infrastrukturgruppe stellen allen unseren Mitgliedern kostenfrei verschiedene Services zur Verfügung:

- Matrix inklusive Slack Bridge
- eine eigene Mastodon-Instanz
- exim für ausgehende E-Mails
- Social Media Wall mit Konferenz-Agenda
- Nextcloud mit diversen Plug-ins
- Jitsi Meet Videokonferenzen

Die meisten der hier genannten Services nutzen dabei die in diesem Artikel beschriebenen Techniken. Aktuell besteht unsere Infrastrukturgruppe aus vier Personen:

- Markus Karg
- Tobias Frech
- Stefan Koospal
- Marcus Fihlon

Wenn auch du Lust hast, mit uns zusammen verschiedenste Dienste für die Java User Groups zu betreiben, laden wir dich herzlich zum Mitmachen ein! Kontaktiere uns zum Beispiel über Matrix, bei Mastodon oder schicke eine E-Mail an: *itservice@ijug.eu*

Quellen

[1] hcloud-GitHub-Repository: *https://github.com/hetznercloud/cli*
[2] nginx-Website: *https://nginx.org/*
[3] Homebrew-Website: *https://brew.sh/*
[4] Scoop-Website: *https://scoop.sh/*
[5] Demo-Code-Repository: *https://github.com/McPringle/java-aktuell-hcloud-demo*
[6] YAML-Erklärung: *https://de.wikipedia.org/wiki/YAML*
[7] fail2ban-Erklärung: *https://de.wikipedia.org/wiki/Fail2ban*
[8] Let's-Encrypt-Website: *https://letsencrypt.org/*
[9] Borg-Backup-Website: *https://www.borgbackup.org/*
[10] iJUG-Website: *https://www.ijug.eu/*

Marcus Fihlon
marcus@fihlon.swiss

Marcus Fihlon beschäftigt sich schon fast sein ganzes Leben lang mit Software-Entwicklung, vorzugsweise im Open-Source-Umfeld und mit Java. Er engagiert sich in der Planung und Durchführung von Konferenzen in der Schweiz und Deutschland. Marcus spricht bei internationalen Konferenzen, schreibt Artikel für verschiedene Fachmagazine und organisiert den monatlichen Hackergarten in Luzern. Hauptberuflich arbeitet er bei der Adcubum AG als Scrum Master und ist leidenschaftlicher Langstrecken-Radler.

Hacking OpenJDK – Hurra, ich habe Java schneller gemacht!

Markus Karg, Head Crashing Informatics

Machmal muss man ungewöhnliche Wege gehen, um seine Anwendung zu beschleunigen. In meinem Fall war es der Umbau des I/O-Subsystems in OpenJDK.

Es war einmal ein lahmer Build…

Vor einigen Jahren bemerkte ich, dass mein Maven-Build sehr langsam war – und das störte mein Team sehr. Schnell war als Ursache die Menge und Größe der von uns verwendeten Dateien ausgemacht, denn unser Projekt beinhaltete sehr viele und vor allem sehr große Dateien. Seltsamerweise schien Maven sie in den Hauptspeicher zu laden, obwohl sie keines Processings bedurften (beispielsweise beim „Download" aus dem lokalen Repository). Eine Prüfung des Quellcodes von Maven (genau genommen von dessen I/O-Bibliothek aus dem Plexus-Projekt) zeigte, dass Maven zum Kopieren eine Schleife über ein Paar an *InputStream* und *OutputStream* benutzte. Eine Schleife, wie wir sie alle in den zurückliegenden Dekaden schon mehrfach geschrieben hatten.

Als versierter Open-Source-Contributor sandte ich einen Pull Request ein, der die Schleife durch den seit Java 9 existierenden Befehl `InputStream.transferTo(OutputStream)` ersetzte. Meine Hoffnung war, dass OpenJDK die Methode nicht nur als „Syntactic Sugar" anbieten, sondern sie in irgendeiner Art und Weise performancetechnisch optimieren würde.

Wie ein auf Wunsch des Maven-Teams durchgeführter JMH-Benchmark *[1]* aber zeigte, war dem leider nicht so. Maven war demnach genau so langsam wie vorher! Also grub ich noch ein paar Schichten tiefer und fand zu meinem Erschrecken im Quellcode von OpenJDK die gleiche, simple Schleife, die ich kurz vorher aus Maven ausgebaut hatte *(siehe Listing 1)*.

Das schnellste Java ist: gar kein Java

Um die Performance zu steigern, musste ich also wohl oder übel einen Pull Request an OpenJDK senden, der der Java-Laufzeitbibliothek abgewöhnt, die Datei *grundlos* in Häppchen der Größe `DEFAULT_BUFFER_SIZE` (damals gerade mal 8K) durch den Java-Heap zu pumpen. Die Vergrößerung des `DEFAULT_BUFFER_SIZE` bringt zwar auch schon was (und wurde von mir tatsächlich in OpenJDK vorgenommen) – aber das wäre ja keinen Artikel in der Java aktuell wert. Was mir vorschwebte, war eher, die JVM gar nicht erst mit der Aufgabe zu belasten, denn ein Betriebssystem ist für Aufgaben der Art „Daten von hier nach da schieben" bestens gerüstet. Es gab keinerlei Grund, diese Aufgabe überhaupt in Java lösen zu wollen.

Hierzu muss man wissen, dass der Kern des Betriebssystems (egal, ob Linux oder Windows) diese Aufgabe *sehr schnell* erledigen kann. Je nach OS und Hardware kann er diese Aufgabe sogar direkt an die Hardware (zum Beispiel ein SAN-Device) delegieren, der Transfer durch die JVM läuft jedoch um Größenordnungen langsamer ab, da jedes Byte den ganzen Weg in das Java-Heap und von dort wieder zurück zur Hardware laufen muss. Vereinfacht gesagt: je länger der Weg, desto langsamer der Transfer; im schlimmsten Fall liegen die Dateien in einem SAN oder gar Cloud-Laufwerk und müssten zweimal LAN oder WAN passieren. Aber selbst im besten Fall (Transfers zwischen lokalen Dateien) wäre der Umweg über die JVM und deren Heap so, als müsste man zum Briefkasten um die Ecke erst einmal über die Autobahn fahren!

Was nun aber einfach klingt, war in OpenJDK gar nicht so einfach umzusetzen. In letzter Konsequenz waren mehrere, aufeinander aufbauende Pull Requests nötig, um in jedem Fall die optimale Performance zu erzielen. Dies begründet sich unter anderem darin, dass `transferTo()` nicht auf Dateien beschränkt ist (dann wäre die Lösung trivial gewesen), und dass die diversen Betriebssysteme unterschiedliche APIs für unterschiedliche Arten von Datenquellen und -Senken besitzen. So gibt es beispielsweise Befehle, die einen direkten Datei-zu-Datei-Transfer beschleunigen, aber andere, die das gleiche mit anderen Ressourcen können. Was aber, wenn von einem Socket empfangen und an eine Datei gesendet werden soll? Meine Optimierung von `transferTo()` sollte ja alle Nutzungsarten beschleunigen. Außerdem ist OpenJDK intern sehr komplex und gerade der NIO-Code hat einige Tücken, die einem in letzter Konsequenz erst klar werden, wenn man sich intensiv mit dem internen Aufbau der einzelnen Klassen beschäftigt.

Frage: Wer wusste, dass es ein explizit zu setzendes Lock gibt, das verhindert, dass man den Betriebsmodus eines Channels ändert? Ich will Hände oben sehen!

Die von mir entwickelte und über Monate hinweg gemeinsam mit dem Java-I/O-Kernteam um Alan Bateman, Brian Burkhalter und Lance Andersen immer weiter verbesserte Lösung gestaltete sich entsprechend aufwendig, da weder OpenJDK noch zwingend *jedes* Betriebssystem einen einheitlichen Zugriffsweg für unterschiedliche

```
public long transferTo(OutputStream out) throws IOException {
      Objects.requireNonNull(out, "out");
      long transferred = 0;
      byte[] buffer = new byte[DEFAULT_BUFFER_SIZE];
      int read;
      while ((read = this.read(buffer, 0, DEFAULT_BUFFER_SIZE)) >= 0) {
            out.write(buffer, 0, read);
            transferred += read;
      }
      return transferred;
}
```

Listing 1: Nicht zur Nachahmung empfohlen: Sehr langsamer Original-Code aus JDK 9

```
public long transferTo(OutputStream out) throws IOException {
     if (source is a file and target is a file, socket or other hardware-device)
           return offloadToOperatingSystem(file, target);

     if (target is a file and source is a file, socket or other hardware-device)
           return offloadToOperatingSystem(source, file);

     return performOriginalLoop(out);
}
```

Listing 2: Performance durch Offloading (Pseudocode)

```
var inputStream = new FileInputStream(name); // weg damit!
var inputStream = Files.newInputStream(name); // bitte nur noch so!
```

Listing 3: Schneidet alte Zöpfe ab!

Hardware kennt. Weiterhin bietet OpenJDK intern einige Fallstricke und es gibt keine *einheitliche* API innerhalb der internen OpenJDK-Klassen für den Zugriff auf *jegliche* Hardware. Entsprechend muss der Code einige Fallunterscheidungen treffen und wird dadurch schwer lesbar (und noch schwieriger zu testen: allein die umfangreichen Tests haben mich Monate an Freizeit gekostet). Aus Gründen der Lesbarkeit wird daher nur der Pseudocode *(siehe Listing 2)* abgedruckt; der vollständige Quellcode ist im GitHub-Repository *[2]* von OpenJDK zu finden und ist auf jeden Fall einen Blick unter Java's Motorhaube wert!

Wie am Pseudocode zu sehen ist, werden im Prinzip drei Wege unterschieden: Die Quelle ist ein File, das Ziel ist ein File, oder keines von beiden ist ein File. Dies begründet sich in den API-Befehlen, die die Betriebssysteme zum Offloading, also zur vollständigen Auslagerung an das Betriebssystem, unterstützen. Tatsächlich hardwarebeschleunigt sind also zum aktuellen Entwicklungsstand nur jene Transfers, bei denen Quelle oder Ziel eine Datei ist. Zukünftig sind weitere Optimierungen denkbar, wie ich im Folgenden noch ausführen werde. Weitere Beschleunigung erledigt OpenJDK durch andere Tricks, wie aufwendig ermittelte Puffergrößen (für moderne Speichertechnologie relative optimale 16K), das Wiederverwenden von Off-Heap-Puffern (vermeidet Reallokationskosten und reduziert GC pressure) und so weiter.

Wer sich den tatsächlichen Quellcode der Optimierung auf GitHub anschaut, wird sicherlich einige Fragen haben. Nicht alle sind im Rahmen dieses Zeitschriftenartikels zu beantworten; hierzu sei daher auf meine Live-Events zum Thema verwiesen *[3]* – dort können wir wirklich in die Details der Lösung gehen.

Erfolg oder Misserfolg, das ist hier die Frage!

Nach nunmehr Jahren, die von der ersten Idee bis zur finalen Umsetzung ins Land gingen, fragt man sich: Hat es sich gelohnt? Die Antwort ist eindeutig ja – und zwar aus mehreren Gründen.

Zum einen beschleunigt die vorgenommene Änderung, die etwas umfangreicher war als das, war ich im Rahmen dieses Artikels beleuchten konnte, nicht nur meinen eigenen Code beziehungsweise den von Maven, sondern *jede* Anwendung, die `transferTo()` verwendet – und das sind vermutlich tausende.

Zum anderen wird durch das Offloading, also die Delegation an das Betriebssystem beziehungsweise die Hardware, der Transfer *effizienter*. Das bedeutet, es wird weniger Strom verbraucht, die CPUs stehen für andere Dinge zur Verfügung, und in letzter Konsequenz entsteht vermutlich auch eine ganze Menge CO_2 weniger (zumindest, wenn man bedenkt, dass Filetransfers eine ziemlich große Rolle bei vielen Anwendungen spielen, multipliziert mit tausenden an Diensten, die in Java geschrieben sind).

Und zu guter Letzt läuft mein Build tatsächlich etwas schneller – aber eben nur etwas, denn mehr als einen ein- bis zweistellige Prozentsatz bringt das Offloading in der Realität leider nicht, denn dort gibt es zu viele weitere Einflussfaktoren, beispielsweise was die Hardware sonst gerade noch zu erledigen hat, oder wie gut der Cache gefüllt ist. Mehr bringt der Einsatz von `Files.copy()` zur Kopie ganzer Dateien (was ich, das habt ihr sicher schon vermutet, ebenfalls in Maven respektive Plexus eingebaut habe). Spaß hat es aber auf jeden Fall gemacht. Ich habe sehr viel über die Interna von OpenJDK gelernt und arbeite seither mit Freude an weiteren Optimierungen von OpenJDK und anderen Open-Source-Projekten, die `transferTo()` dank meiner Änderung nun einsetzen (aktuell: Jersey, der Kern von GlassFish).

An dieser Stelle eine Bitte an euch alle: Nutzt ein modernes JDK (zum Beispiel 21-LTS), ersetzt die alten File-Streams durch solche, die per `Files.new*Stream` erzeugt wurden, und werft die handgemachten Schleifen weg. Wo immer es geht, setzt bitte `transferTo()` und `Files.copy()` ein *(siehe Listing 3)*. Eure Stromrechnung und unsere Umwelt danken es euch!

Und wie geht's nun weiter?

Natürlich ist an dieser Stelle längst nicht Schluss. OpenJDK bietet noch viel Luft nach oben für weitere Optimierungen! Ich möchte die Effizienz zukünftig noch weiter erhöhen und habe mir schon eine Liste gemacht:

- Ein PR für die Beschleunigung weiterer Quelle-/Ziel-Paare liegt bereits vor.
- Es wäre sinnvoll, die Beschleunigung auch dann zu erhalten, wenn die *alte* I/O-API (`new FileInputStream()`) benutzt wird.
- Der Code sollte definitiv besser lesbar sein.
- Nach `InputStream` wartet die Klasse Reader auf mich, die potenziell das gleiche Problem hat: Sie macht eine Schleife über einen Puffer im Heap.

Wie ihr seht, ist es also gar nicht so schwer (und schon gar nicht unmöglich) als „Außenstehender" an OpenJDK mitzuarbeiten. Ich würde mich freuen, wenn ihr diesen Artikel als Inspiration nehmt, um eigene Optimierungen an der Java-Laufzeitbibliothek vorzunehmen. Wenn ihr dazu Fragen habt, wendet euch gerne an mich. Weitere Informationen dazu findet ihr auch auf meinem YouTube-Kanal *[4]*.

Quellen

[1] Java Microbenchmark Harness: *https://github.com/openjdk/jmh/*

[2] Quellcode der finalen Lösung: *https://github.com/openjdk/jdk22/blob/9f0469b94a97886e4ac0ee6cb870763430a1e487/src/java.base/share/classes/sun/nio/ch/ChannelInputStream.java#L224*

[3] Live-Mitschnitt eines Online-Events: *https://youtu.be/fga97dXb9G8*

[4] Videos zu meinen Open-Source-Contributions: *https://www.youtube.com/@headcrashing*

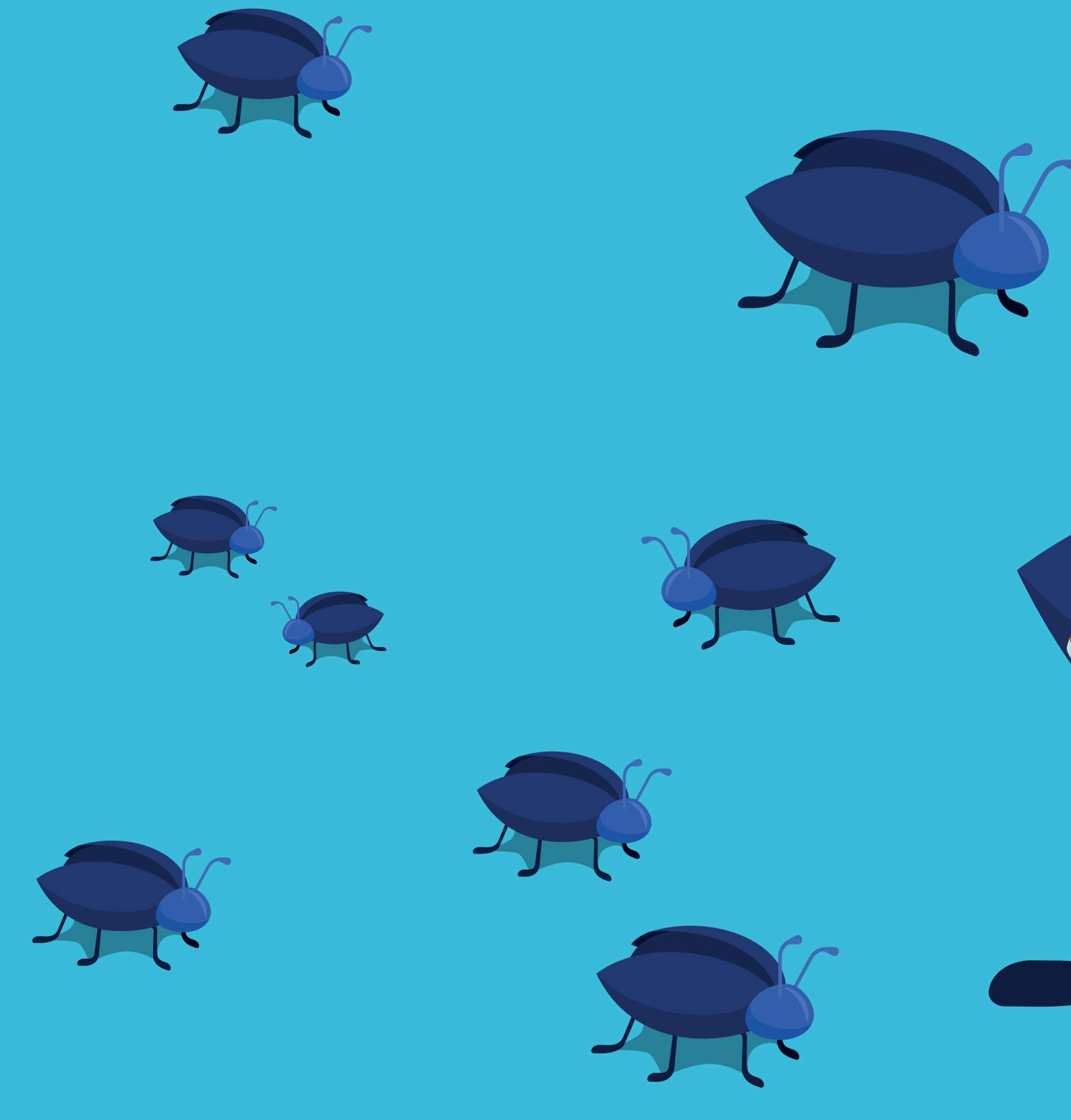
Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. Der passionierte Open-Source-Contributor ist Co-Autor der JAX-RS-Spezifikation und optimiert seit einigen Jahren leidenschaftlich die Performance der Java-Laufzeitbibliothek.

YouDebug – Scriptable JVM Debugger

Wolfgang Schell, metaphacts GmbH



Debugging gehört zum Alltag in der Softwareentwicklung und die gängigen Entwicklungsumgebungen bieten gute Unterstützung dafür. Aber wie sieht das in Produktionsumgebungen aus, in denen die Installation einer IDE nicht möglich oder erlaubt ist? Was ist, wenn ein Problem nur sporadisch auftritt? Oder auf vielen Maschinen in einem Container oder einer Cloud-Umgebung? Und wie untersucht man nur gelegentlich auftretende Probleme? Behebt man sie mit einer magischen Mischung aus JVM und Groovy?

Wer programmiert, gießt nicht nur Funktionen in Code, sondern baut mit Sicherheit auch den einen oder anderen Fehler ein. Diese Fehler können offensichtlich oder auch subtil sein und werden oft nicht gleich erkannt. Besonders komplexe Fehler verstecken sich in nebenläufigem Code oder bei der Kommunikation mit Drittsystemen, die – auch (oder besonders) bei der Verwendung standardisierter Protokolle – unterschiedliche oder unerwartete Eingaben schicken können.

Ist ein Fehler identifiziert, gilt es zunächst herauszufinden, was genau ihn auslöst: fehlerhafte Daten, unvollständige Bedingungen oder Sicherheitsabfragen, Race Conditions in nebenläufigen Algorithmen oder eine der vielen anderen Fehlerquellen.

Es gibt viele Möglichkeiten, um Fehler in einem Programm zu finden und jede:r Entwickler:in verwendet eine oder mehrere dieser Möglichkeiten:

- Debuggen mit grafischer Unterstützung der Entwicklungsumgebung
- Debuggen auf der Kommandozeile, zum Beispiel mithilfe des in der OpenJDK-Distribution mitgelieferten Tools `jdb`
- bei betriebssystemspezifischen Problemen oder der Verwendung von nativen Bibliotheken kann auch die Verwendung von `gdb` beziehungsweise einem anderen Debugger auf C/C++-Ebene notwendig sein
- mithilfe von Log-Ausgaben, die auch im finalen Programm erhalten bleiben und gegebenenfalls zur Laufzeit durch feingranulare Konfiguration aktiviert werden
- oder im einfachsten Fall durch das Hinzufügen von `println`-Ausgaben im Code

Alle genannten Methoden haben eine Gemeinsamkeit: Sie bedingen eine interaktive Bedienung des Debuggers. Eine weitere Methode ist der in diesem Artikel beschriebene Debugger *YouDebug*, der nicht-interaktives Debugging ermöglicht.

Dieser Artikel stellt *YouDebug* vor, einen vor vielen Jahren von Jenkins-Schöpfer Kohsuke Kawaguchi entwickelten Scriptable Java Debugger. *YouDebug* ist ein kleines, nicht-interaktives Werkzeug, das mit Groovy-basierten Scripts Breakpoints setzt, das Laden von Klassen und die Ausführung von Methoden und Funktionen nachverfolgt, Bedingungen auswertet und Daten ausgeben kann. Oder auch quasi im Vorübergehen ein Problem patchen beziehungsweise umgehen kann, alles nur mit der JDI-API auf der JVM.

Debugging mit YouDebug

Abbildung 1: YouDebug verwendet eine Groovy-DSL als Wrapper um das Java Debug Interface. (Quelle: Wolfgang Schell)

YouDebug verwendet eine *Groovy*-basierte domänenspezifische Sprache (DSL) als Wrapper für das Java Debug Interface (JDI) und führt ein Debug-Script aus, das die Logik für die durchzuführende Untersuchung und/oder Programm-Anpassung enthält *(siehe Abbildung 1)*.

YouDebug bietet alles, was man mit einem Debugger machen kann:

- Setzen von Breakpoints,
- Auswerten von Ausdrücken,
- Auflisten aller Threads in der Anwendung mit deren jeweiligem Stack-Trace,
- Auslesen und Verändern von lokalen Variablen,
- Ausführen von Methoden,
- Erstellen eines Thread-Dumps,
- Erstellen eines Heap-Dumps.

Breakpoints können dabei auf verschiedene Weise definiert werden:

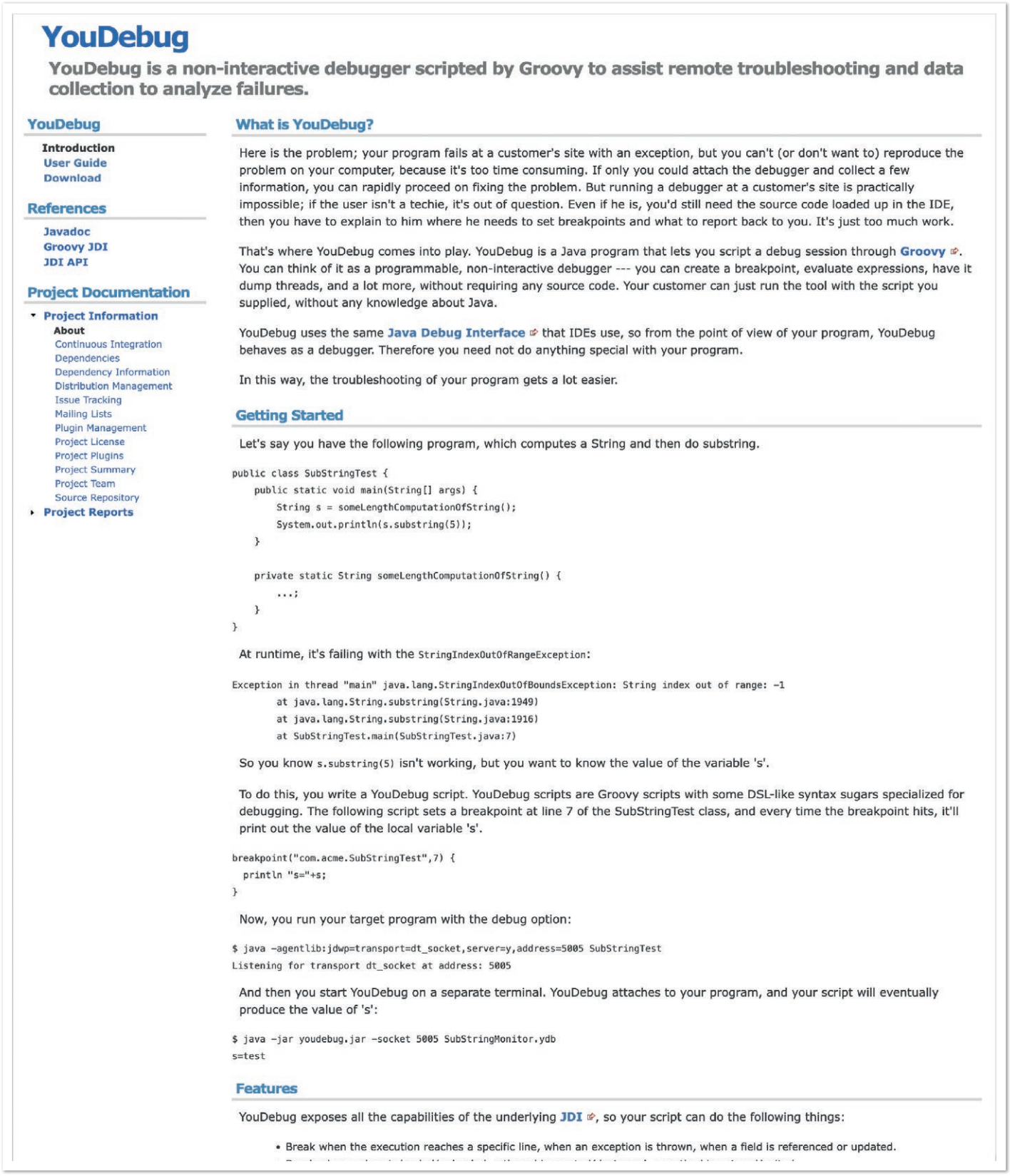
- beim Erreichen einer bestimmten Programmzeile,
- wenn eine Exception geworfen wird,
- wenn ein Feld referenziert oder verändert wird,
- wenn eine Klasse ge- oder entladen wird,
- wenn ein Thread gestartet oder beendet wird,
- wenn eine Methode aufgerufen wird oder zurückkehrt
- oder wenn ein Synchronisierungsblock betreten oder verlassen wird.

Details zur Funktionsweise werden später in diesem Artikel beschrieben. Die *YouDebug*-Website *[1]* hält eine Einführung sowie einen umfangreicher User Guide bereit *[2] (siehe Abbildung 2)*. Der Quellcode ist auf GitHub *[3]* und das Jar-File im Maven Repository *[4]* zu finden.

Anwendungsfälle

YouDebug kann vor allem dann glänzen, wenn manuelles Debuggen mit klassischen Tools mühsam oder unmöglich ist:

- bei iterativer Fehlersuche mit wiederholter Ausführung von mehreren Schritten



YouDebug

YouDebug is a non-interactive debugger scripted by Groovy to assist remote troubleshooting and data collection to analyze failures.

YouDebug
- Introduction
- User Guide
- Download

References
- Javadoc
- Groovy JDI
- JDI API

Project Documentation
- Project Information
  - About
  - Continuous Integration
  - Dependencies
  - Dependency Information
  - Distribution Management
  - Issue Tracking
  - Mailing Lists
  - Plugin Management
  - Project License
  - Project Plugins
  - Project Summary
  - Project Team
  - Source Repository
- Project Reports

What is YouDebug?

Here is the problem; your program fails at a customer's site with an exception, but you can't (or don't want to) reproduce the problem on your computer, because it's too time consuming. If only you could attach the debugger and collect a few information, you can rapidly proceed on fixing the problem. But running a debugger at a customer's site is practically impossible; if the user isn't a techie, it's out of question. Even if he is, you'd still need the source code loaded up in the IDE, then you have to explain to him where he needs to set breakpoints and what to report back to you. It's just too much work.

That's where YouDebug comes into play. YouDebug is a Java program that lets you script a debug session through Groovy. You can think of it as a programmable, non-interactive debugger --- you can create a breakpoint, evaluate expressions, have it dump threads, and a lot more, without requiring any source code. Your customer can just run the tool with the script you supplied, without any knowledge about Java.

YouDebug uses the same Java Debug Interface that IDEs use, so from the point of view of your program, YouDebug behaves as a debugger. Therefore you need not do anything special with your program.

In this way, the troubleshooting of your program gets a lot easier.

Getting Started

Let's say you have the following program, which computes a String and then do substring.

```
public class SubStringTest {
    public static void main(String[] args) {
        String s = someLengthComputationOfString();
        System.out.println(s.substring(5));
    }

    private static String someLengthComputationOfString() {
        ...;
    }
}
```

At runtime, it's failing with the `StringIndexOutOfRangeException`:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: -1
        at java.lang.String.substring(String.java:1949)
        at java.lang.String.substring(String.java:1916)
        at SubStringTest.main(SubStringTest.java:7)
```

So you know `s.substring(5)` isn't working, but you want to know the value of the variable 's'.

To do this, you write a YouDebug script. YouDebug scripts are Groovy scripts with some DSL-like syntax sugars specialized for debugging. The following script sets a breakpoint at line 7 of the SubStringTest class, and every time the breakpoint hits, it'll print out the value of the local variable 's'.

```
breakpoint("com.acme.SubStringTest",7) {
  println "s="+s;
}
```

Now, you run your target program with the debug option:

```
$ java -agentlib:jdwp=transport=dt_socket,server=y,address=5005 SubStringTest
Listening for transport dt_socket at address: 5005
```

And then you start YouDebug on a separate terminal. YouDebug attaches to your program, and your script will eventually produce the value of 's':

```
$ java -jar youdebug.jar -socket 5005 SubStringMonitor.ydb
s=test
```

Features

YouDebug exposes all the capabilities of the underlying JDI, so your script can do the following things:

- Break when the execution reaches a specific line, when an exception is thrown, when a field is referenced or updated.

Abbildung 2: YouDebug, Einführung und User Guide (Quelle: YouDebug webseite [1], [2])

- bei der Verwendung des Debuggers, um Informationen zum Status und für das Monitoring zu sammeln, die anderweitig nicht verfügbar sind
- um Deadlocks aufzubrechen, die den weiteren Programmfluss blockieren (kann durch *drop-to-previous-frame* erfolgen)
- sogenanntes *Monkey Patching*, das Verändern des Programmflusses zur Laufzeit, um Fehler zu umgehen
- wenn es nur beschränkten Zugriff auf das Zielsystem gibt
- wenn ein Deployment aus mehr als einem System besteht, etwa in einer verteilten Umgebung oder mit mehreren Replikas
- wenn der Bug nur sporadisch auftritt, man darauf warten muss und dabei die Anwendung nicht durch interaktives Debugging blockieren will

Auf einer Entwicklungsmaschine oder in einer Testumgebung wie in *Abbildung 3* ist das Debuggen in der Regel problemlos möglich. Die Anwendung ist via direktem Netzwerkzugriff erreichbar, man hat Zugang zu Daten in Form von Dateien oder Datenbanken, hat Quellcode und Dokumentation im Blick und kann sich direkt im System mit Testbenutzern anmelden sowie die Konfiguration beeinflussen.

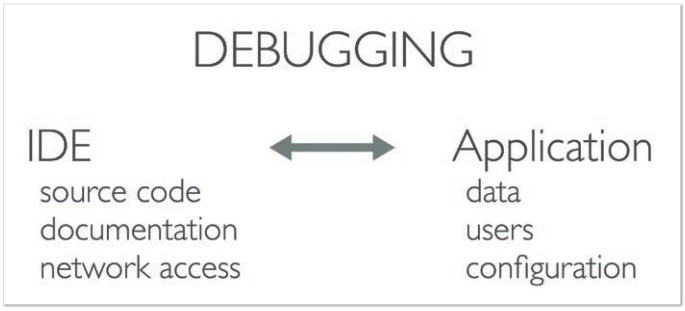


Abbildung 3: Debugging in einer Testumgebung (Quelle: Wolfgang Schell)

Doch was ist, wenn der Fehler nur in einer Produktionsumgebung auftritt?

Wie in *Abbildung 4* ersichtlich, hat man bei komplexen Installationen oder gar in einer produktiven Umgebung in der Regel viele Einschränkungen. Eine Entwicklungsumgebung ist nicht verfügbar und kann auch nicht installiert werden. Quellcode und Dokumentation sind vertraulich und können nicht zum Debugging bereitgestellt werden. Eine Firewall verhindert den direkten Netzwerkzugriff und Daten und Benutzerzugängen unterliegen Compliance- und Datenschutz-Richtlinien. Die Konfiguration ist aus Sicherheitsgründen nicht einsehbar und kann nicht verändert werden.

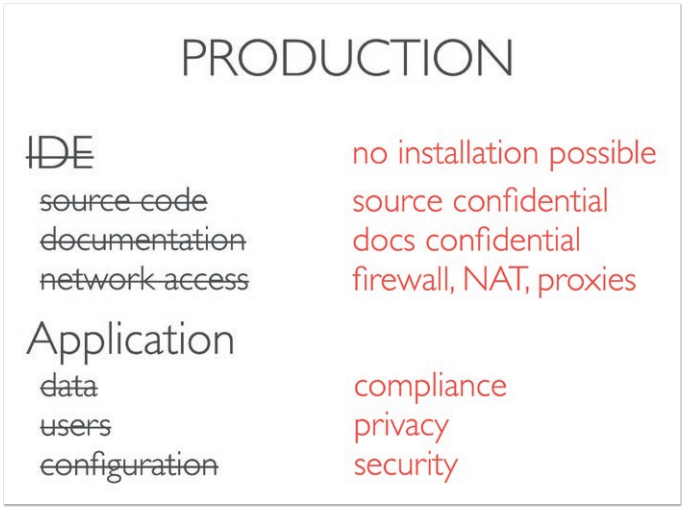


Abbildung 4: Debugging in einer Produktivumgebung (Quelle: Wolfgang Schell)

Die Lösung? Bringen wir das Debug-Script zur Anwendung! *Abbildung 5* zeigt, wie mithilfe eines kleinen Debug-Scripts die oben beschriebenen Einschränkungen umgangen und die Fehlersuche auch in produktiven Umgebungen ermöglicht wird.

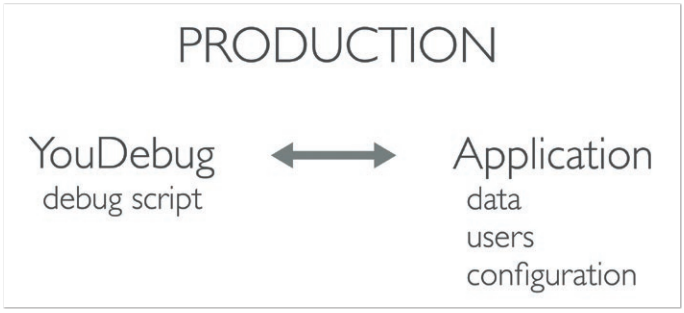


Abbildung 5: Debugging mit YouDebug in einer Produktivumgebung (Quelle: Wolfgang Schell)

Grundlegende Vorgehensweise

Die grundlegende Vorgehensweise zur Verwendung von *YouDebug* ist dabei immer gleich:

1. Starten der Anwendung im Debug-Modus
2. Erstellen eines Debug-Scripts
3. Ausführen des Debug-Scripts in der Zielumgebung

Das Starten der Anwendung im Debug-Modus erfolgt durch zusätzliche Parameter beim Programmstart, wie in *Listing 1* beschrieben.

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005 -jar myapp.jar
```

Listing 1: Starten einer Java-Anwendung mit aktiviertem Debugging

Dies funktioniert mit jeder Java-Anwendung, auch innerhalb eines Containers oder mit einem *Service Wrapper*. Weitere relevante Details werden später in diesem Artikel beschrieben.

Ein einfaches Debug-Script wird in *Listing 2* gezeigt. Es legt einen Breakpoint in der `main()`-Methode an, schreibt bei Erreichen des Breakpoints eine Meldung auf die Konsole und fährt dann mit der Abarbeitung des Programms fort.

```
vm.methodEntryBreakpoint("com.my.EntryPoint", "main") {
  method -> println "hey, we are in main()"
}
```

Listing 2: Einfaches Debug-Script mit einem Breakpoint in main()

Das Debug-Script kann dann wie in *Listing 3* beschrieben gestartet und auf die auf Port 5005 exponierte Anwendung angesetzt werden.

```
java -jar youdebug.jar -socket 5005 yourscript.ydb
```

Listing 3: Starten von YouDebug mit dem Debug-Script

Neben `youdebug.jar` brauchen wir auf dem Classpath noch `groovy-all.jar` sowie `args4j.jar`. Auf neueren Java-Versionen ab Java

9 muss zusätzlich das `tools`-Modul aktiv sein. Die Verwendung von YouDebug mit neueren Java-Versionen wird weiter unten beschrieben.

Beispiele für Debug-Scripts

Listing 4 zeigt, wie Breakpoints an einer bestimmten Programmzeile einer Klasse gesetzt werden können. Wichtig dabei ist, dass die Zeileninformation nicht exakt ist, da eine Zeile mehrere Ausdrücke oder Lambda-Funktionen enthalten kann und der Bytecode somit nur eine ungefähre Abbildung von Zeile zu Code bieten kann. Dies gilt allerdings genauso für jeden anderen Java-Debugger.

```
vm.breakpoint("net.jetztgrad.buggyweb.MyServlet",35) {
  println "at buggy position in MyServlet, line 35"
}
```

Listing 4: Anlegen eines Breakpoints an einer gegebenen Programmzeile

Listing 5 demonstriert den Zugriff auf und die Veränderung von Werten in der Anwendung. Neben dem lesenden und schreibenden Zugriff auf die Variable `age` sehen wir hier die Ausführung der Funktion `request.getParameter()`.

```
vm.breakpoint("net.jetztgrad.buggyweb.MyServlet",35) {
  println "age=" + age
  String ageParam = request.getParameter("age")
  println "setting age to $ageParam"
  age = ageParam
}
```

Listing 5: Anlegen eines Breakpoints an einer gegebenen Programmzeile

Manchmal ist es notwendig, zu verstehen, was die Anwendung gerade macht. Ein Thread-Dump ist hierfür das Mittel der Wahl, da man sieht, welche Funktion gerade ausgeführt wird. *Listing 6* zeigt, wie die Anwendung pausiert wird, Informationen zur JVM der Anwendung ausgegeben werden und wie für jeden Thread ein Thread-Dump getriggert wird, bevor die Ausführung fortgesetzt und die Debug-Verbindung geschlossen wird.

```
vm.suspend()
println vm.virtualMachine.name()
println vm.virtualMachine.description()
vm.threads*.dumpThread()
vm.resume()
vm.close()
```

Listing 6: Stoppen der Anwendung und Auslösen eines Thread-Dumps

Debug-Ereignisse

Das JDI kennt verschiedene Arten von Debug-Ereignissen, die vom Debugger abgefangen werden können:

- `breakpoint(location)`
- `exceptionBreakpoint(type)`
- `methodEntryBreakpoint(type, method)`
- `methodExitBreakpoint(type, method)`
- `accessWatchpoint(field)`
- `modificationWatchpoint(field)`
- `monitorWait(type)` (*)
- `monitorWaited(type)` (*)

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String name = request.getParameter(name:"name");
    String age = request.getParameter(name:"name");
    PrintWriter out = response.getWriter();
    try {
        out.println(x:"<html><head><title>Hello</title></head><body>");
        out.println("<h1>Hello, " + (name != null ? name : "stranger") + "</h1>");
        if (age != null && age.length() > 0) {
            out.println("<p>In " + (100 - Integer.parseInt(age)) + " years you'll be 100!</p>");
        }
        out.println(x:"<p><form>");
        out.println("Name: <input type='text' name='name' value='" + (name != null ? name : "") + "'></input><br>");
        out.println("Age: <input type='text' name='age' value='" + (age != null ? age : "") + "'></input><br>");
        out.println(x:"<input type='submit' name='Send'></input><br>");
        out.println(x:"</form></p>");
        out.println(x:"</body></html>");
    }
    catch (Exception e) {
        out.println(x:"<h3>An error occured:</h3>");
        out.println(x:"<p><pre>");
        out.println(e.getClass().getSimpleName() + " " + e.getMessage());
        e.printStackTrace(out);
        out.println(x:"</pre></p>");
    }
    finally {
        try {
            out.close();
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

Abbildung 6: doGet()-Methode in einem Servlet (Quelle: Wolfgang Schell)

- `monitorContendedEnter(type)` (*)
- `monitorContendedEntered(type)` (*)

Die mit einem * markierten Event-Typen sind mit der aktuellen *YouDebug*-Version noch nicht ansprechbar, können aber in einer künftigen Version ergänzt werden.

Referenzen und Aktionen

YouDebug ermöglicht das Referenzieren von Variablen, Feldern und Klassen in der debuggten Anwendung sowie das Auslösen von diversen Aktionen:

- `ref(type)`
- `_new(type)`
- `forEachClass(type)`
- `loadClass(type)`
- `suspend()`
- `resume()`
- `dumpHeap(String)`
- `dumpAllThreads()`
- `withFrozenWorld(Closure)`
- `exit(status)`
- `instanceCounts(types[])`

Anwendungsbeispiel

Die Mächtigkeit und Funktionen von *YouDebug* sollen nun anhand eines konkreten Beispiels demonstriert werden *[11]*. In einer einfachen Web-Anwendung steckt ein Fehler, der in mehreren Schritten mithilfe von Debug-Scripts identifiziert und bis zur Behebung und dem Redeployment notdürftig umgangen wird.

Die in *Abbildung 6* gezeigte `doGet()`-Methode eines Servlets der Beispielanwendung zeigt ein Formular an und wertet die übergebenen Parameter aus. *Abbildung 7* zeigt die Beispielanwendung in Aktion.

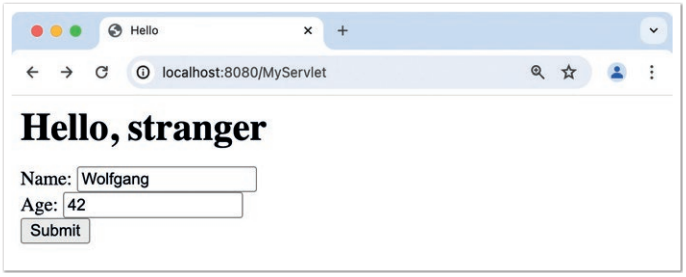


Abbildung 7: Beispielanwendung in Aktion (Quelle: Wolfgang Schell)

Ein Fehler verhindert leider die korrekte Anzeige des Ergebnisses, stattdessen erhalten wir nur eine Fehlermeldung *(siehe Abbildung 8)*. Zur Fehlersuche können wir uns nun durch mehrere Iterationen eines Debug-Scripts an das Problem annähern und es notdürftig beheben.

Abbildung 8: Ein Fehler in der Beispielanwendung. (Quelle: Wolfgang Schell)

Zunächst wollen wir die Fehlerstelle eingrenzen und loggen alle Exceptions *(siehe Listing 7)*.

```
vm.exceptionBreakpoint("java.lang.Exception") { e ->
  println "caught exception:"
  e.dumpStackTrace(System.out);
}
```

Listing 7: Loggen aller Exceptions

Das Script aus *Listing 7* kann mit dem Kommando aus *Listing 8* ausgeführt werden.

```
youdebug -socket $DEBUG_PORT exception.ydb
```

Listing 8: Starten von YouDebug mit einem Debug-Script

Anhand des Stacktrace der geworfenen Exception können wir die Zeilennummer ermitteln und dort einen Breakpoint setzen. Die `NumberFormatException` legt einen Fehler beim Parsen des Alters nahe. Aber warum? Schauen wir uns den Wert genauer an *(siehe Listing 9)*.

Wir stellen fest, dass dort der im Formular übergebene Name und nicht das Alter steht. Irgendwas läuft an dieser Stelle schief, aber wir beheben das Problem erst einmal provisorisch durch das Setzen eines konstanten Alters *(siehe Listing 10)*.

```
vm.breakpoint("net.jetztgrad.buggyweb.MyServlet",35) {
  println "age=" + age
  println "setting age to 50"
  System.out.flush()
  age = "50"
}
```

Listing 10: Die lokale Variable age wird im Debugger überschrieben.

Bei genauer Betrachtung des Quellcodes der Anwendung *(siehe Abbildung 5)* stellen wir fest, dass die lokale Variable `age` falsch gesetzt

```
vm.breakpoint("net.jetztgrad.buggyweb.MyServlet",35) {
  println "at buggy position in net.jetztgrad.buggyweb.MyServlet, line 35"
  println "age=" + age
  System.out.flush()
}
```

Listing 9: Breakpoint mit Ausgabe der lokalen Variable age

wird und wir stattdessen den richtigen Formularparameter verwenden müssen.

An dieser Stelle würde man nun einfach den Fehler im Quellcode beheben, die Anwendung neu bauen, ausliefern und installieren.

Doch was machen wir, wenn der Fehler kritisch ist und es bis zur Auslieferung eine Weile dauern kann, etwa um ausreichende Tests durchzuführen oder weitere Fehlerbehebungen oder gar neue Features in einem Release mit auszuliefern? Mit *YouDebug* kann man den Fehler einfach provisorisch durch sogenanntes *Monkey Patching* beheben. In *Listing 11* ergänzen wir die korrekte Logik im Debug-Script und solange der Debugger läuft, wird dieser Logikpfad durchlaufen. Das ist zwar keine dauerhafte Lösung, kann aber schon einmal eine Alternative zu ständigen Fehlern sein und die Dauer bis zur Behebung durch ein neues Update überbrücken.

```
vm.breakpoint("net.jetztgrad.buggyweb.MyServlet",35) {
  println "age=" + age
  System.out.flush()
  String ageParam = request.getParameter("age")
  println "setting age to $ageParam"
  System.out.flush()
  age = ageParam
}
```

Listing 11: Korrektes Auslesen des Parameters age im Debugger

Monitoring und Tracing

Neben dem Suchen und Überbrücken von Fehlern kann man *YouDebug* auch hervorragend einsetzen, um Tracing-Information zu sammeln. So kann man sich mit dem in *Listing 6* gezeigten Script nähere Informationen zur JVM einer Anwendung sowie einen Thread-Dump ausgeben lassen, der zeigt, was gerade passiert. Breakpoints

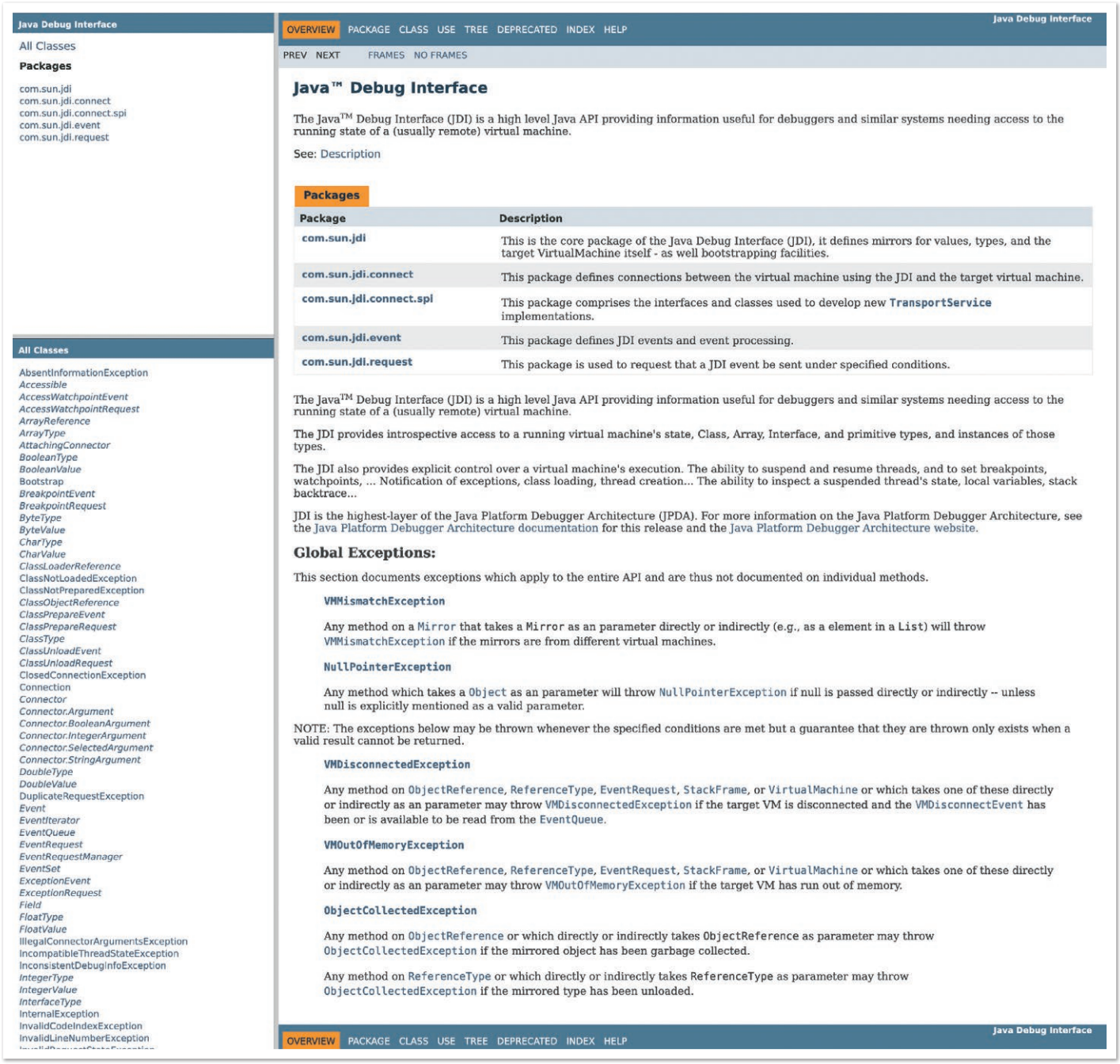


Abbildung 9: JavaDoc für das Java Debug Interface (Quelle: Oracle JavaDoc Webseite [5])

an kritischen oder interessanten Stellen können weitere Daten sammeln und periodisch Informationen ausgeben und somit einen tiefen Einblick in die Anwendung geben, die weit über Logging hinaus gehen.

Hinter den Kulissen

YouDebug verwendet eine *Groovy*-basierte domänenspezifische Sprache (DSL) als Wrapper für das Java Debug Interface (JDI).

Das *Java Debug Interface* *[5]* definiert eine Schnittstelle zwischen der Java Virtual Machine (JVM) und einem Debugger. JDI ermöglicht das Starten von oder eine Verbindung zu einer laufenden JVM-basierten Anwendung. Die Verbindung kann dabei lokal auf dem gleichen Rechner oder über das Netzwerk zu einem entfernten Server erfolgen und folgt dem *Java Debug Wire Protocol* (JDWP).

Die Schnittstelle bietet die Möglichkeit, Breakpoints zu setzen sowie bei deren Erreichen den Zustand der Anwendung zu inspizieren und gegebenenfalls auch mit ihr zu interagieren, etwa durch das Verändern von Daten oder den Aufruf von Funktionen. Alles, was die JVM erlaubt, ist möglich, ebenfalls der Aufruf von Betriebssystemfunktionen. *Abbildung 9* zeigt das umfangreiche *Java Debug Interface* mit vielen Klassen und Interfaces.

Was ist Groovy?

Groovy *[6]* ist eine dynamische Programmiersprache basierend auf der JVM. Ähnlich wie *Python* oder *Ruby* ist Groovy-Code sehr lesbar und verzichtet auf verbose Konstrukte. Gleichzeitig ist Groovy vollständig kompatibel zu Java. Funktionsaufrufe können entweder direkt in die jeweilige Implementierung springen oder aber abgefangen und beliebig interpretiert werden. Letztere Eigenschaft, die allen dynamischen Sprachen gemein ist, erlaubt die Übersetzung von Variablen-Zugriffen (lesend oder schreibend) sowie Funktionsaufrufen in entsprechende Zugriffe in der Anwendung, die in einem eigenen Prozess und gegebenenfalls sogar auf einem anderen Computer läuft und nur über Netzwerk erreichbar ist.

Risiken und Nebenwirkungen

Zur Verwendung von *YouDebug* ist es zuerst notwendig, dass der Debug-Modus aktiv ist. Hierzu gibt es zwei wichtige Punkte: Sicherheit und Performance.

Über das *Java Debug Interface* ist alles möglich, was der überwachte Prozess machen kann. Dazu gehört auch das Ausführen von beliebigem Java-Code sowie das Aufrufen des Betriebssystems und das Absetzen von Kommandos auf dem Zielsystem. Aus diesem Grund sollte der Debug-Port nur offen sein, wenn es unbedingt notwendig ist und auch entsprechend abgesichert werden. JDI beziehungsweise JDWP haben keinerlei eigene Sicherheitsmechanismen. Daher sollte der Port mittels einer Firewall abgesichert und der Zugriff nur lokal und/oder via SSH-Tunnel ermöglicht werden.

In neueren Java-Versionen gibt es ebenso die Möglichkeit, den Prozess im Debug-Modus zu starten, den Port aber erst auf Zuruf zu öffnen. Diese Funktionalität wurde auch bis Java 11 zurück portiert. Dazu wird der Prozess mit einem zusätzlichen Kommando gestartet *(siehe Listing 12)* und kann später mittels des Kommandos `jcmd` den Port zur Laufzeit öffnen *(siehe Listing 13)*.

Eine ausführliche Beschreibung dieses wenig bekannten Features findet sich im Blog des OpenJDK-Entwicklers Johannes Bechberger *[8]*.

```
java -agentlib:jdwp=transport=dt_socket,server=y,
suspend=n,address=5005,onjcmd=y -jar myapp.jar
```

Listing 12: Starten der Anwendung mit Debugger auf stand-by

```
jcmd <pid> VM.start_java_debugging
```

Listing 13: Starten des Debuggers und Öffnen des Ports

Lange sei der Debug-Modus als inperformant angesehen worden. Dies sei aber durch einige wichtige Verbesserungen behoben worden, wie Johannes Bechberger ebenfalls in seinem Blog beschreibt *[9]*. Demnach spricht aus Performance-Sicht wenig dagegen, den Debug-Modus auch auf Produktivsystemen dauerhaft zu aktivieren, falls Debugging potenziell notwendig ist.

Es gibt zurzeit eine Diskussion, ob das nachträgliche Öffnen des Debug-Ports wieder entfernt werden soll, aber aktuell unterstützen alle Releases seit Java 11 diese Funktionalität.

Umgang mit Fehlern im Debug-Script

Auch das Debug-Script besteht nur aus Quellcode, der wiederum Fehler haben kann. Wenn ein Breakpoint erreicht und der Debugger aktiv wird, dann ist die Zielanwendung im pausierten Zustand. Wenn im Debug-Script ein Fehler auftritt, kann sich der Debugger beenden und die Anwendung im pausierten Zustand verharren. Daher empfiehlt es sich, sicherzustellen, dass die Anwendung beim Trennen des Debuggers auch im Fehlerfall auf jeden Fall wieder weitergeführt wird.

Zugriff auf den Zustand der Anwendung

Das JDI spiegelt Elemente wie Typen (Klassen), Objekte, Methoden, Felder und Variablen. *YouDebug* versteckt die Komplexität des Zugriffs auf diese Elemente hinter einer bequemen Abstraktion. Dies kann allerdings ebenfalls zu subtilen Fehlern führen, wenn nicht ganz klar ist, ob eine Operation wie ein Variablen-Zugriff oder ein Funktionsaufruf nun im Debugger-Prozess oder der überwachten Anwendung ausgeführt wird.

Wichtig ist, dass Zugriffe auf den Zustand der Anwendung sowie der Aufruf von Methoden nur möglich sind, wenn die Anwendung pausiert, sprich in einem Breakpoint ist. Wenn man die Zielanwendung aktiv zu einem beliebigen Zeitpunkt per `suspend()`-Aufruf unterbricht, dann ist kein Zugriff auf Felder und Methoden möglich.

Beware of the *Heisenbug*!

Der Physiker Werner Heisenberg hat festgestellt, dass eine Beobachtung eines Systems immer auch zu einer Veränderung führt. Demnach kann das Debuggen eines Programms auch den Fehler maskieren, beispielsweise durch Laufzeitveränderungen oder andere Effekte, also zum sogenannten *Heisenbug* führen *[10]*.

Manchmal ist also doch ein einfaches `println` in der Anwendung

Heisenbug

Im Jargon der Computerprogrammierung ist ein Heisenbug ein Softwarefehler, der zu verschwinden oder sein Verhalten zu verändern scheint, wenn man versucht, ihn zu untersuchen. Der Begriff ist ein Wortspiel mit dem Namen Werner Heisenberg, dem Physiker, der als erster den Beobachtereffekt der Quantenmechanik geltend machte, der besagt, dass der Akt der Beobachtung eines Systems unweigerlich seinen Zustand verändert. In der Elektronik ist der traditionelle Begriff der Sondeneffekt, bei dem das Anbringen einer Testsonde an ein Gerät dessen Verhalten verändert.

(Quelle: Wikipedia [10])

die beste Wahl, um Beobachtungen und Zustände zu ermitteln und auszugeben.

Debugging in Container-basierten Anwendungen

Java-Anwendung werden heute oft als Container-Image ausgeliefert und betrieben. Auch ein Container ist nur ein Prozess und der Debug-Modus kann einfach durch Setzen einer Umgebungsvariable (zum Beispie `JAVA_OPTS` oder `JAVA_TOOL_OPTIONS`) aktiviert werden.

Bei der Verwendung von *Docker Compose* kann man dies einfach in einer `docker-compose.override.yml`-Datei eintragen und somit zusätzliche Parameter an den Container übergeben. *YouDebug* kann ebenfalls in Form eines Containers gestartet und an das gleiche virtuelle Netzwerk angehängt werden, um auf den Debug-Port zuzugreifen.

In einer *Kubernetes*-Umgebung kann *YouDebug* beispielsweise mithilfe von `kubectl debug` als Debug-Container im gleichen Netzwerk-Namespace eines Pods gestartet werden *[12]*.

In beiden Fällen ist darauf zu achten, dass der Debug-Port nur für berechtigte Zugriffe zugänglich gemacht wird!

Verwendung von *YouDebug* mit neueren Java-Versionen

YouDebug in der Originalversion von Kohsuke Kawaguchi läuft mit Java 6 und Groovy 1.6. Es läuft in einem eigenen Prozess und wegen der Rückwärtskompatibilität des *Java Debug Interfaces* ist es kein Problem, unterschiedliche Java-Versionen für Debugger und Anwendung zu verwenden.

Ich habe *YouDebug* auf Java 11 und Groovy 4 portiert *[7]* und Kohsuke Kawaguchi kontaktiert, um den Debugger auf einen aktuellen Stand zu bringen.

Neben der Unterstützung für einige neueren Debug-Events hat dies allerdings kaum Auswirkungen auf die Funktionsweise. Lediglich die Syntax für Lambda-Ausdrücke in Debug-Scripts wurden durch die neuere *Groovy*-Version an die von Java-Lambdas angeglichen. Während die JDI-Klassen bis Java 8 Teil des `tools.jar` waren, das auch den Java-Compiler und weitere Tools enthält, muss man auf neueren Java-Versionen ab Java 9 stattdessen das `tools`-Modul mithilfe zusätzlicher Kommandozeilenparameter aktivieren *(siehe Listing 14)*.

```
export JAVA_TOOL_OPTS="$JAVA_TOOL_OPTS \
 -Djava.awt.headless=true \
 --add-opens=jdk.jdi/com.sun.tools.jdi=ALL-UNNAMED"
```

Listing 14: Aktivieren des `tools`-Moduls, das die JDI-Klassen enthält

Fazit

YouDebug verspricht Abhilfe in Fällen, in denen interaktives Debuggen von Anwendungen nicht möglich ist. Auch obskure Fälle wie das Sammeln von Informationen zur Laufzeit, das in der Anwendung nicht vorgesehen ist, kann mithilfe der Debug-Schnittstelle ermöglicht werden. Es ist vermutlich kein Tool für jeden Tag, aber in den beschriebenen Situationen kann es unschätzbar nützlich sein.

Quellen

[1] Einführung zu YouDebug auf *https://youdebug.kohsuke.org/*

[2] YouDebug User Guide auf *https://youdebug.kohsuke.org/user-guide.html*

[3] YouDebug-Projekt auf GitHub: *https://github.com/kohsuke/youdebug*

[4] Download von YouDebug im Maven Repository: *https://mvnrepository.com/artifact/org.kohsuke/youdebug*

[5] JavaDoc des Java Debug Interface auf *https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/index.html*

[6] Website der Programmiersprache Groovy auf *https://groovy-lang.org/*

[7] YouDebug-Port basierend auf Java 11 und Groovy 4 bei *https://github.com/jetztgradnet/youdebug/tree/update-deps*

[8] On-Demand-Aktivierung des Debug-Modus im Blog von Johannes Bechberger auf *https://mostlynerdless.de/blog/2023/10/03/level-up-your-java-debugging-skills-with-on-demand-debugging/*

[9] Untersuchungen zur Performance im Debug-Modus im Blog von Johannes Bechberger auf *https://mostlynerdless.de/blog/2024/02/09/is-jdwps-onjcmd-feature-worth-using/*

[10] Der Heisenbug auf Wikipedia *https://de.wikipedia.org/wiki/Heisenbug*

[11] Beispielanwendung auf GitHub *https://github.com/jetztgradnet/youdebug-talk*

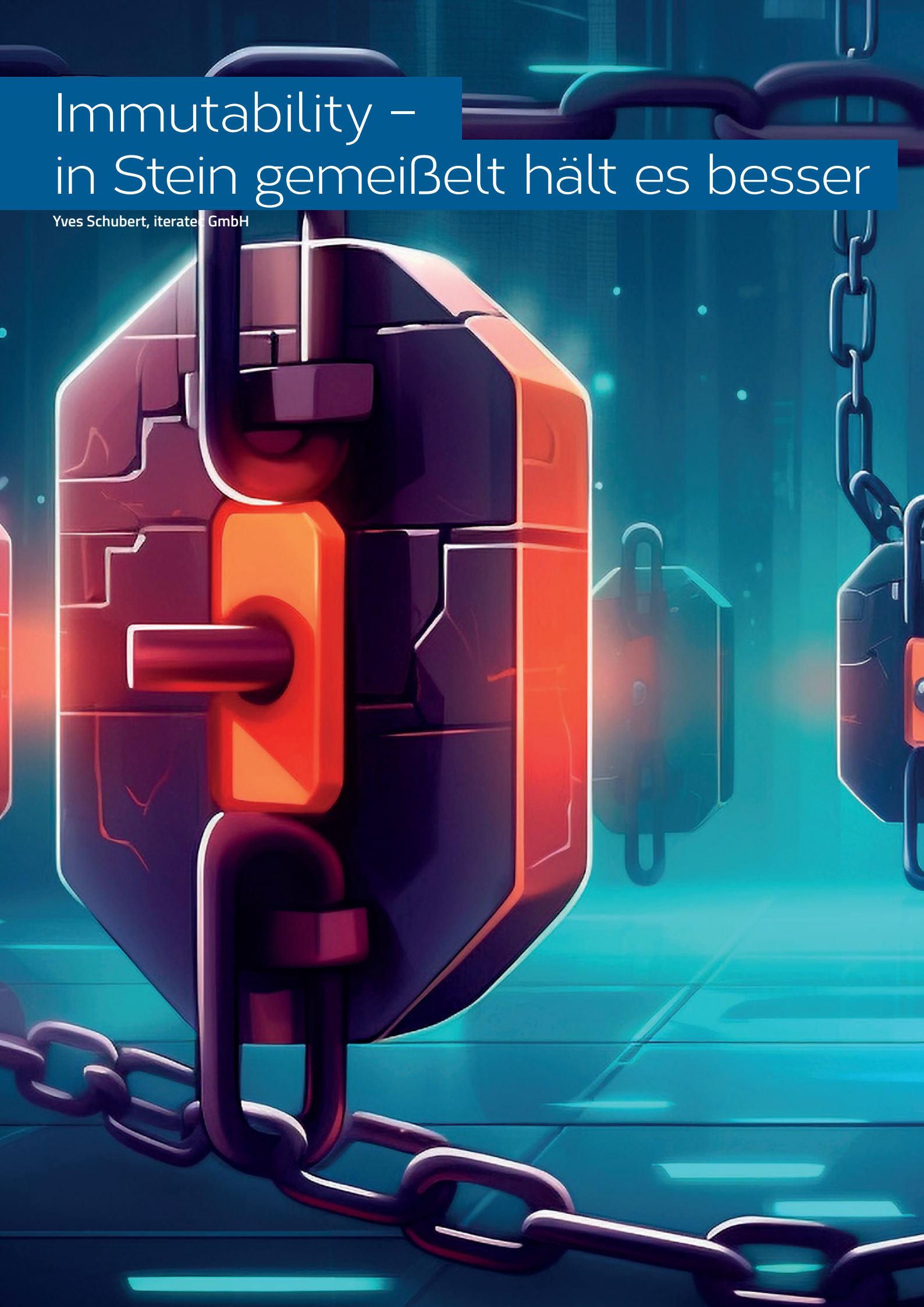
[12] kubectl-debug-Kommando *https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#debug*

Wolfgang Schell
metaphacts GmbH

Wolfgang Schell arbeitet als Principal Software Architect bei der metaphacts GmbH und entwickelt zusammen mit dem Software-Engineering-Team wartbare Softwareprodukte im Umfeld von Knowledge-Management basierend auf Knowledge-Graphen. Als enthusiastischer Softwareentwickler ist er auch Co-Organisator der Mannheim Java User Group (Majug2) und Mentor für Kids im Rahmen des JugendHackt Labs Mannheim.

Immutability – in Stein gemeißelt hält es besser

Yves Schubert, iterate GmbH



Variable Objekte sind in der Softwareentwicklung wie eine tickende Zeitbombe. Das Verändern der Zustände von Objekten kann zu fachlichen Inkonsistenzen und Fehlern führen. Die Folge sind unvorhergesehene Verhaltensweisen und Seiteneffekte.

Mutable Objekte zwingen Softwareentwickler:innen dazu, über diese Risiken nachzudenken und komplexe Lösungsmöglichkeiten zu schaffen. Die beste Umgangsweise damit ist es aber, diesen mentalen Ballast komplett loszuwerden. Ein Werkzeug hierfür ist die Verwendung von Immutability – die Idee, das Verändern von Zuständen gänzlich zu verbieten und Datenstrukturen zu schaffen, die auf vielen Ebenen für eine stabile Grundlage und zuverlässige und sichere Software sorgen. Nicht zuletzt ist Immutability auch ein Kernkonzept in der funktionalen Programmierung. Seit Java 14 sind die sogenannten Records in den Sprachkern mit aufgenommen, womit Immutability auch hier salonfähig geworden ist.

Dieser Artikel zeigt auf, was hinter dem Konzept Immutability steckt und wie die Verwendung von Immutability die Code-Komplexität reduzieren kann und die Entwicklung und Handhabung von Software und Daten erleichtert.

Um *Immutability* zu verstehen, müssen wir etwas ausholen und zunächst die Grundbegriffe *State*, *Value* und *Identity* klären. Denn es geht bei *Immutability* nicht nur um technischen Aspekt der Unveränderlichkeit, sondern auch vielmehr um ein Programmiermodell und eine bestimmte Herangehensweise an Komplexität.

State

In einem Computerprogramm kann man den State auf verschiedenen Ebenen erklären. Auf der kleinsten Ebene ist der State eines Programms (oder sogar des gesamten Rechners) nichts weiter als die momentanen Inhalte der einzelnen CPU-Register, des Hauptspeichers und gegebenenfalls irgendwelcher Caches. Auf hoher Flughöhe hat der Zustand eines Programms eine rein fachliche Bedeutung. Das folgende Beispiel soll das erläutern.

In *Abbildung 1* ist ein stark vereinfachtes State-Diagramm dargestellt, das unseren Beispielfall „Auto fahren" darstellt. Dabei gibt es

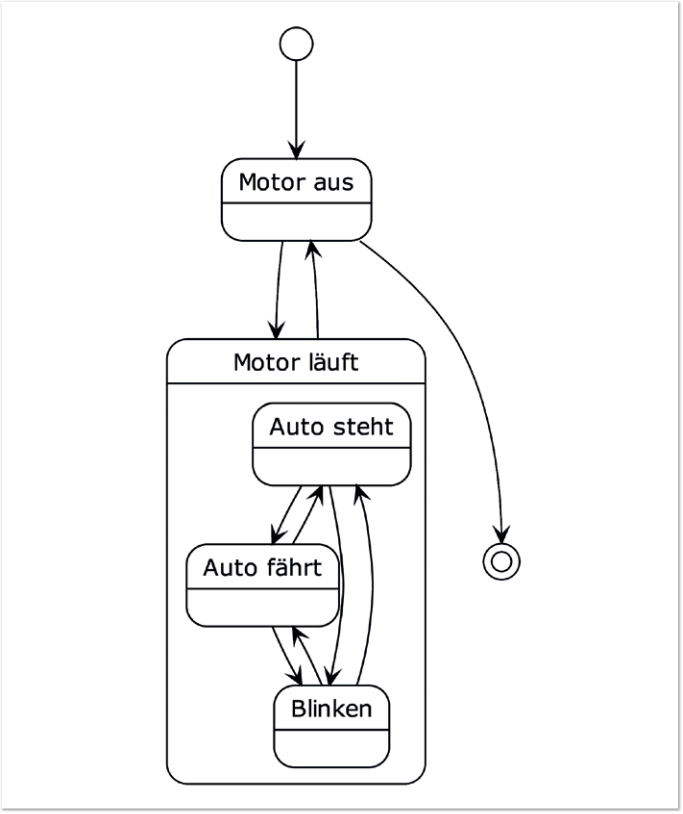


Abbildung 1: Ein State-Diagramm für unser Beispiel „Auto fahren"

die Zustände „Motor aus" und „Motor läuft" mit den verschachtelten Zuständen „Auto steht", „Auto fährt" und „Blinken". Wir gehen davon aus, dass dieses Diagramm aus einer Anforderungsanalyse entstanden ist und bisher auch keine Sonderfälle oder Fehlerfälle beinhaltet (zum Beispiel „Blinken, wenn der Motor aus ist" oder „Ausschalten des Motors während der Fahrt"). Im Code könnte dann die Implementierung des Autos wie in *Listing 1* dargestellt aussehen.

```
public class Auto {

      boolean motorAn;
      boolean rollt;
      boolean blinkt;

      void einschalten() {
            motorAn = true;
      }

      void ausschalten() {
            motorAn = false;
      }

      void gasGeben() {
            rollt = true;
      }

      void bremsen() {
            rollt = false;
      }

      void blinkerAn() {
            blinkt = true;
      }

      void blinkerAus() {
            blinkt = false;
      }
}
```

Listing 1: Beispiel-Implementierung eines Autos

Der *State* einer Instanz dieser Klasse sind die Werte der drei Felder „motorAn", „rollt" und „blinkt". Die Methoden, die den *State* des Objekts verändern, nennt man die „Mutator-Methods".

So weit ist alles noch sehr einfach zu überschauen und wir haben zugegebenermaßen eine recht naive Implementierung vorliegen. Es geht hier aber um die folgenden Gedankengänge: Wir wollen nun den Zustand des Objekts verändern und rufen dafür die beiden Methoden „ausschalten" und „gasGeben" in Folge auf. Und das ist der Kern einer ganzen Reihe von Problemen: Wir haben einen fachlich nicht korrekten Zustand im Objekt hergestellt! Im Zweifel kann es dadurch, dass dieses Objekt in andere Module transportiert wird, zu ernsthaften Laufzeitfehlern kommen. Es müsste in dieser Klasse, oder schlimmer noch woanders, eine mehr oder weniger komplexe Validierungslogik erstellt werden, die solche Inkonsistenzen verhindert.

Ein Zitat von Charles Scalfani bringt es auf den Punkt: „Did you ever wonder why most solutions to program glitches are fixed by rebooting your computer or restarting the offending application? That's because of state. The program has corrupted its state." *[1]*

Values

In der Informationstechnologie beschäftigen wir uns, wie der Name schon sagt, mit Informationen. Was aber sind Informationen? Bei genauerer Betrachtung gelangt man zu dem Schluss, dass Informationen nichts anderes sind als Tatsachen. Tatsachen selbst verändern sich nicht. Sie gelten für einen bestimmten Zeitpunkt und bleiben darauf bezogen stets unverändert. Tatsachen werden nicht verändert, sondern durch neue Tatsachen ersetzt. In der Programmierung spricht man dann nicht von Tatsachen, sondern von Werten, auf Englisch *Values*.

Values sind zum Beispiel Zahlen wie 42 oder 1/3. Die Zahlen können nicht verändert werden. Eine 42 bleibt eine 42. Wenn eine 1 dazu gezählt wird, dann entsteht eine komplett neue Zahl. Auch ein Datum kann nicht verändert werden. Der 18. März 2024 ist und bleibt das gleiche Datum. Das Addieren von 5 Tagen führt zu einem neuen Datum.

Es lassen sich folgende vorteilhafte Eigenschaften aufzählen: *Values* können verteilt werden. Da man sich sicher sein kann, dass ein *Value* niemals verändert wird, ist es klar, dass man später auch genau diesen *Value* wieder erhält, oder dass auch genau dieser *Value* an weitere Parteien verteilt wird und immer erhalten bleibt. Bei der Übertragung der *Values* wird der Wert selbst übertragen und keine Referenz. Man erhält dadurch reproduzierbare Ergebnisse. Das Schöne an *Values* ist außerdem, dass sie in der Regel einfach zu erstellen und sprachunabhängig sind. Eine weitere wichtige Eigenschaft ist, dass ein Aggregat, also eine Zusammenstellung, von *Values* wieder einen *Value* ergeben!

Würde man also die Zeit anhalten, so würde der gesamte Zustand des Universums dieses einen Zeitpunkts ein aus vielen *Values* aggregierter *Value* sein. Und genau das ist der *State*, von dem im vorherigen Abschnitt die Rede war. Der *State* ist die Menge aller zugehörigen *Values* zu einem bestimmten Zeitpunkt.

Identity

Die Natur eines *Values* kann man sich auch verdeutlichen, wenn man über Identität redet. Ein Objekt mit einer eindeutigen ID ist ein Container für ein Ding, das real oder virtuell existiert. Die ID kann als ein Zeiger verwendet werden (zum Beispiel in einer Datenbank als Fremdschlüssel). Die ID des Objekts selbst muss unveränderbar sein, aber der Inhalt könnte verändert werden.

Ein *Value* dagegen wird allein durch seine Inhalte identifiziert und benötigt keine ID. Zwei Values, die unterschiedliche Inhalte haben, sind nicht dieselben Objekte.

In Java unterscheidet man hier zwischen „Call-By-Value", bei dem ein *Value* direkt übertragen wird, und „Call-By-Reference", bei dem eine Referenz zu einem Objekt übertragen wird.

Immutability

Was genau ist *Immutability* in der Programmierung und wie hängt das nun mit den zuvor genannten Punkten zusammen? Zunächst kann man festhalten: Ein Objekt ist unveränderlich, wenn sein *State* nach der Erzeugung nicht mehr verändert werden kann.

In Java gibt es seit einiger Zeit die *Java Records*, die als Sprachkonstrukt sehr gut für unseren Zweck geeignet sind.

In unserem Beispiel haben wir eine Adresse als *Java Record* vorliegen *(siehe Listing 2)*. Dieses *Record* wird einmal instanziiert und die Felder können dann mit den von Java bereitgestellten Zugriffsmethoden ausgelesen werden. Verändert werden kann dieses Objekt aber nicht. Bei einem Umzug muss eine neue Adresse erstellt werden.

```
record Address(String street,
                      String city,
                      String state,
                      String zipCode,
                      String country) {}

...

// spätere Verwendung
var address = new Address("Zettachring 6", "Stuttgart",
"BW", "70567", "Germany");
```

Listing 2: Eine Adresse als Java Record

Aber Achtung: *Records* in Java sind nicht per se *immutable*. Wenn eines der Felder den Typ einer veränderbaren Klasse hat (zum Beispiel eine Liste oder eine Klasse mit *Mutator*-Methoden), ist die Instanz dieses *Records mutable*, also veränderbar.

Wenn die Sprache also nicht bereits syntaktisch eine *Immutability* einfordert, wie es in funktionalen Programmiersprachen der Fall ist, obliegt es den Softwareentwicklern und -entwicklerinnen diese herzustellen.

Mit Listen verhält es sich ähnlich. Im ersten Moment liegt der berechtigte Einwand vor, dass *immutable* Listen (wie sie zum Beispiel in Java durch `List.of(...)` erstellt werden) sehr ineffizient sind. Die Antwort hierauf lautet: *Persistent Data Structures*. Ein Vertreter dieser besonderen Datenstrukturen ist der sogenannte Präfixbaum oder Trie. Mit diesem werden Listen nicht flach, sondern in einer Baumstruktur gehalten. Die Baumstruktur erlaubt es, sehr schnell

darin zu suchen, aber insbesondere auch einzelne Listenelemente mit hoher Effizienz mit neuen Elementen zu ersetzen, ohne dabei die Ursprungsliste zu verändern (daher „persistent"). Stattdessen wird ein neuer Wurzelknoten erstellt, alle Knoten auf dem Weg zum entsprechenden Blatt kopiert und die Zeiger entsprechend „umgehängt". Diese Operation kann sogar schneller sein als das Einfügen eines Elements in eine Liste auf die „klassische" Art. Alle Elemente und sogar die vorherige Liste bleiben dabei erhalten, was beispielsweise für Undo-/Redo-Funktionen hilfreich sein kann.

Als ein weiterer Einwand kursiert der Mythos, dass die bei der konsequenten Anwendung von *Immutability* vermehrt vorkommende Erstellung und Entsorgung von Objekten zu Performance-Einbußen führen würden. In modernen Programmiersprachen, zu denen auch aktuelle Versionen von Java gehören, ist das kein nennenswertes Problem. Wie im vorherigen Beispiel erwähnt, kann die Tatsache, dass ein Objekt unveränderlich ist, sogar zu besserer Performance verhelfen. Allein, dass *immutable* Objekte inhärent *thread-safe* sind, verbessert die Performance von Programmen. Es sind keinerlei Lock-Mechanismen notwendig und das Teilen der Objekte auf mehrere Threads ist ohne Risiko, denn *immutable* Objekte sind frei von Seiteneffekten.

Weitere Vorteile spielen *immutable* Objekte auf der Wartungsseite aus. Durch die Unveränderlichkeit sind diese Objekte einfacher zu verstehen und beim Lesen des Codes wird die mentale Last verringert, Stichwort ist hier das „Prinzip der geringsten Überraschung". Das schlägt sich auch in einer geringeren Codekomplexität nieder. Des Weiteren sind Klassen dieser Objekte einfacher zu testen, da es weniger Sonderfälle gibt.

Kein Geringerer als einer der Java-Architekten Brian Goetz, sagte einmal: „Making objects immutable eliminates entire classes of programming errors." *[2]*

Einer der größten Vorteile ist aber, dass *immutable* Objekte, die einmal fachlich korrekt erzeugt wurden, dies auch bleiben, denn: der *State* kann nicht verändert werden.

Hier wird auch klar, dass *Immutability* nicht nur eine technische Besonderheit ist. Die Anwendung von *Immutability* verändert auch die Art und Weise, wie man programmiert. Es ist eine Veränderung im Mindset.

Die Transformation

Die klare Empfehlung dieses Artikels ist es, so gut es geht, insbesondere in der Domain-Logik, mit *immutable* Objekten zu arbeiten. An dem vorherigen Beispiel „Auto fahren" kann man nun zeigen, wie eine bestehende Klasse so umgestaltet werden kann, dass sie *immutable* Objekte erzeugt *(siehe Listing 3)*.

Egal, in welcher Reihenfolge die Methoden aufgerufen werden, der Zustand ist stets fachlich korrekt. Es liegt auf der Hand, dass die Validierung auch mit einer *mutable* Klasse erledigt werden könnte. Allerdings würde diese insbesondere bei einem komplexeren Zustandsmodell niemals die oben genannten Vorteile erfüllen.

In the Wild

Wenn man sich in der Welt der Softwareentwicklung umsieht, so stößt man an vielen Stellen ebenfalls auf das Konzept der Unveränderbarkeit. Die folgende Aufzählung ist dabei bei weitem nicht vollständig:

Event Sourcing

Im Event-Sourcing werden alle Zustandsänderungen als Ereignisse (Objekt erstellt, Feld XY geändert, ...) betrachtet. Diese Ereignisse werden persistiert und sind unveränderbar. Die Wiederherstellung des Zustands eines bestimmten Zeitpunkts geschieht über das Replay, also das Ausführen, aller Ereignisse bis zu diesem Zeitpunkt.

```
public record Auto(
            boolean motorAn,
            boolean rollt,
            boolean blinkt) {

      static Auto einschalten() {
            return new Auto(true, false, false);
      }

      public Auto gasGeben() {
            return new Auto(true, true, this.blinkt());
      }

      public Auto bremsen() {
            return new Auto(true, false, this.blinkt());
      }

      public Auto blinkerAn() {
            return new Auto(true, this.rollt(), true);
      }

      public Auto blinkerAus() {
            return new Auto(true, this.rollt(), false);
      }

      public Auto ausschalten() {
            return new Auto(false, false, false);
      }
}
```

Listing 3: Ein immutable Auto

Docker

Die Images, die für die Materialisierung von Containern verwendet werden, bestehen aus sogenannten Schichten (*Layer*). Die *Layer* sind (bis auf die „oberste") *immutable*. Bei jeder Änderung wird eine neue Schicht erzeugt („Copy-On-Write"-Prinzip).

Blockchain

Ähnlich wie beim Event-Sourcing werden in der Blockchain alle Transaktionen unveränderlich gespeichert.

Git

Jeder Commit in Git ist unveränderlich.

Infrastructure-as-Code

Bei Infrastructure-as-Code werden Server-Topologien und -Konfigurationen deklarativ in Code gegossen. Statt sich manuell per SSH auf einem der Server anzumelden und dort Änderungen vorzunehmen, wird die Infrastruktur bei jeder Änderung des Codes „weggeworfen" und neu aufgesetzt. Unter der Annahme, dass intern sicherlich effiziente Delta-Mechanismen ausgeführt werden, ist der Vorgang aus Anwendersicht *immutable*.

React

React macht sich den Umstand zunutze, dass die Property-Objekte, die in die Komponenten gegeben werden, *immutable* sind. Die Erkennung, dass sich die Properties verändert haben geschieht hier nicht durch einen teuren Vergleich der Inhalte der Properties, sondern nur über die Referenz des Objekts. Solange die Objekt-Referenz unverändert ist, geht man davon aus, dass sich auch die Properties nicht verändert haben.

Fazit

Es liegt auf der Hand – obwohl *Immutability* auf den ersten Blick unhandlich erscheint, ist es auf den zweiten Blick genau das Gegenteil. Das Arbeiten mit *Immutables* verändert die Art und Weise, wie man Software entwickelt. Durch die Anwendung fügen sich viele Dinge zusammen und Probleme wie *Thread-Safety* oder Testbarkeit lösen sich in Wohlgefallen auf, die Vorteile überwiegen. Auf lange Sicht wird durch *Immutability* die Codequalität verbessert, sodass hier die dringende Empfehlung ausgesprochen wird, dieses Konzept so gut es geht einzusetzen. *Immutability* ändert einfach alles!

Quellen

[1] Charles Scalfani (2016), Why Programmers Need Limits, *https://cscalfani.medium.com/why-programmers-need-limits-3d96e1a0a6db*

[2] Brian Goetz (2021), Garbage collection and performance, *https://web.archive.org/web/20210224000208/http://www.ibm.com/developerworks/library/j-jtp01274/*

Yves Schubert
Iteratec GmbH
yves.schubert@iteratec.com

Seit über 15 Jahren bin ich beruflich und auch privat an vielen Softwareprojekten unterschiedlichster Technologiebereiche maßgeblich beteiligt. Von neuen Technologien lasse ich mich immer begeistern und probiere daher, möglichst viel von dieser schnelllebigen Softwarewelt mitzunehmen. Neue Trends klopfe ich dabei aber auf ihre Nachhaltigkeit hin ab und wäge kritisch ab, ob und wie diese eingesetzt werden können. In der Praxis wende ich die Prinzipien des Software Craftsmanship an.

Strukturzementierende Tests und wie man sie vermeidet, Teil 2: Konzepte der TestDsl

Richard Gross, MaibornWolff GmbH

Tests sollten auf Verhaltensänderungen reagieren, aber unempfindlich gegenüber Strukturveränderungen sein. Tests, die die zweite Bedingung nicht erfüllen, nennen wir struktur-zementierend. In Teil 1 haben wir eine TestDsl gebaut, mit der wir die Zementierung durch das Test-Setup komplett vermeiden können. In diesem Teil gehen wir näher auf die Konzepte der DSL ein und zeigen, warum man mit ihr Tests schreiben kann, die nach Änderung von nur einer Zeile von Integration- zu Unit-Tests werden.

Die in Teil 1 implementierte DSL (*Domain-specific Language [1]*) ist eine Abstraktionsschicht zwischen Test und Test-Setup. Man nutzt sie, um den eigentlichen Test von den Vorbedingungen zu entkoppeln und um sich genauer anzuschauen, wie Vorbedingungen hergestellt werden. Beispielsweise könnte eine zu testende Methode `rentBook(bookId, userId)` in einem `RentingService` die Vorbedingung haben, dass Buch und Anwender in einem `Repository` hinterlegt sein müssen, bevor wir `rentBook` aufrufen. Ein Anwender hat vielleicht auch eine `Role`, die bereits hinterlegt sein muss, genauso wie die Permission der Rolle.

Würde man all die Vorbedingungen in jedem Test von Hand herstellen, dann hätte man nicht nur viel Redundanz geschaffen, sondern (genügend Tests vorausgesetzt) damit auch die Struktur aller Vorbedingungen zementiert. Implementierungsdetails, wie eine `Role` aussieht, wie diese in ihr Repository kommt, wie der `RentingService` an jenes `Repository` kommt, zementieren sich mit jedem redundanten Test-Setup. Der jeweilige Engineer hat dann schlicht und einfach keine Lust, x Dateien anzufassen, um eine eigentlich sinnvolle Strukturänderung im Nachhinein einzubauen. Unsere Struktur ist zementiert.

Wenn das gesamte Setup der Tests über die DSL gemacht wird *(siehe Abbildung 1)*, ist bei Strukturänderungen auch nur noch die DSL betroffen und der Test ist von der Struktur entkoppelt. Wir können ohne Probleme Strukturveränderungen im Produktionscode vornehmen, weil wir Änderungen nur noch an einer Stelle, in der DSL, nachziehen müssen.

Die TestDsl besteht aus den Teilen:

- TestState
- Floor
- Entity-(Combo-)Builder
- Test Doubles (statt strukturzementierender Mocks)
- Service-Configurator (optional)
- JUnit-Extension (optional)

Merkbar viel Code also, jedoch nicht viel Logik. Es geht eigentlich immer darum, zu delegieren oder Werte zu setzen. Die DSL soll möglichst **wenig denken**, damit wir uns kein Wartungsproblem schaffen.

Ganz klar ist, wir müssen Code investieren. Dieser zahlt allerdings schnell Dividende und erlaubt uns auch, sehr knappe Tests zu schreiben (*siehe Listing 1*, der finale Test aus Teil 1):

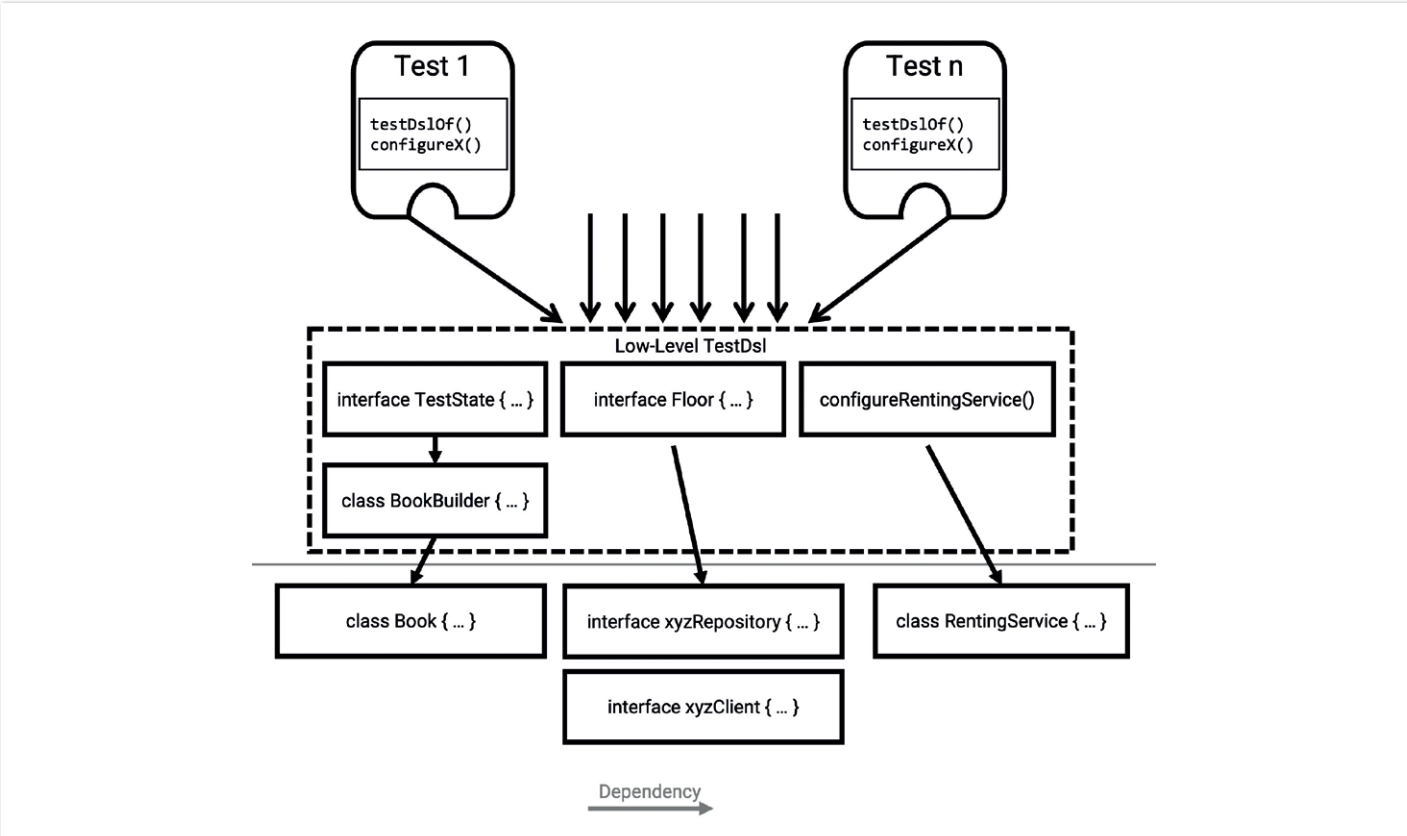


Abbildung 1: Die TestDsl schiebt sich zwischen Tests und die Struktur des Produktionscodes. (© Richard Gross)

```
@Integration @Test // <1>
void should_be_able_to_rent_book(TestState a, Floor floor){ // <2>
  // given
  var book = a.book();
  var userCombo = a.userCombo(it -> it.hasPermission("CAN_RENT_BOOK"));
  a.saveTo(floor);

  var testee = new RentingService(floor);
  // WHEN
  var result = testee.rentBook(book, userCombo.user());
  // THEN
  assertThat(result.isRented()).isTrue();
}
```

Listing 1: Vollständiger Test mit TestDsl Extension

Den Test aus *Listing 1* können wir jetzt nur noch durch Ändern der @Integration-Annotation zu einem schnellen @Unit-Test machen. Davon profitiert insbesondere Legacy-Code, weil wir vor dem Sanieren oft noch viel Logik in der Datenbank haben und auf Integrationsebene testen müssen. Mit der Zeit landet diese Logik in der Domäne und wir können bestehende Tests mit einem Einzeiler zu deutlich schnelleren Unit-Tests machen. Ohne eine TestDsl müsste man sie auf Unit-Ebene komplett neu schreiben, weswegen viele Teams das nicht machen, auf langsamen Integration-Tests sitzen bleiben und trotz Tests nicht schneller iterieren können.

Warum das geht und welche Konzepte hinter der DSL stehen, sehen wir im Folgenden.

Unit- und Integration-Tests

Unit- und Integration-Tests sind industrieweit sehr schwammige Begriffe *[3]*. Selbst in kleinen Teams gibt es keine eindeutige Definition, jeder hat sein eigenes Verständnis. Wir sollten also kurz pausieren und festlegen, wovon wir eigentlich reden, wenn wir sagen `@Unit @Test`.

Der in Teil 1 geschriebene Test ist ein *[4]* und folgt der Unit-Test-Definition von Michael Feathers *[5]*:

> *A test is not a unit test if:*
>
> - *It talks to the database*
> - *It communicates across the network*
> - *It touches the file system*
> - *It can't run at the same time as any of your other unit tests*
> - *You have to do special things to your environment (such as editing config files) to run it.*
>
> *Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.*

Um 2010 herum kannte Google diese Definition nicht oder ihre Teams konnten sich nicht darauf einigen. Sie wussten aber, dass es für den firmeninternen Austausch praktisch ist, wenn alle das gleiche Verständnis von Tests haben und führten daher neue *data-driven naming conventions* für Tests ein. Ihre Definition von einem Small-Test entspricht dabei ziemlich genau der Definition von Feathers *(siehe Tabelle 1)* und der Medium-Test liefert eine ziemlich gute Definition von einem Integration-Test.

Feature	Small (Unit)	Medium (Integration)	Large (Acceptance)
Database	No	Yes	Yes
Network access	No	localhost only	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

Tabelle 1: Google Test Sizes [3]

Die Tabelle zeigt gut, warum Unit-Tests so schnell sind: Es gibt kein *Out-of-Process* mit dem unser getesteter Code interagieren muss. Alles läuft *In-Process* und *In-Memory* und ohne *Network*. Eine perfekte Grundlage für den Großteil unserer Tests, denn die nächste Stufe Integration beziehungsweise Medium kann bereits deutlich langsamer sein. Je nach Test-Runner und Infrastruktur sind Unit-Tests in Kundenprojekten zwischen 4- bis 10-mal schneller als Integration-Tests. Den Faktor 4 konnten wir mit unseren Integration-Tests überhaupt erst erreichen, indem wir sie mit einem kleinen Trick parallelisiert hatten. Wenn jeder Test seinen eigenen Namespace in der Datenbank bekommt (in MongoDB wäre das ein Schema), dann kann jeder Integration-Test nur seine eigenen Daten sehen und nur seine eigenen Daten modifizieren. Die Test-Isolation ist damit wiederhergestellt.

Von Integration zu Unit Test

Unseren Test *(siehe Listing 1)* können wir mit einem Einzeiler von @Integration zu @Unit umwandeln. Die JUnit-Extension tauscht im Hintergrund jetzt alle Repositories aus. Aus `InMemoryBooksDouble` wird so das Produktions-Repository `JpaBooks`. Die API der TestDsl bleibt gleich, weswegen wir am Test nichts mehr ändern müssen. Am getesteten Code müssen wir auch nichts ändern, weil dieser nur das `interface Books { add(Book book); /* ... */ }` kennt und nicht weiß, welche Implementierung dahinter steht.

Damit dieser Wechsel so reibungslos funktioniert, müssen sich die *InMemory* und die *Jpa* Repositories allerdings auch gleich verhalten. Im folgenden Kapitel sehen wir, wie wir das mit den sogenannten *Port Contract* Tests kontinuierlich sicherstellen können.

Es ist allerdings nicht immer sinnvoll, alle Methoden von `Books` auch in `InMemoryBooksDouble` zu implementieren und mit *Port Contract Tests* synchron zu halten. Manchmal brauchen wir die mächtigen Query-Funktionalitäten von Datenbanken nicht für Business-Logik, sondern für Suchfunktionen im UI. Dann wäre es zum einen ein großer Overhead, diese InMemory für ein paar @Unit-Tests nachzubauen. Zum anderen würden diese Tests dann nur unsere InMemory-Repository-Implementierung testen. In solchen Fällen werfen wir im InMemory Double lieber eine `NotImplementedException` und bleiben (erstmal) bei @Integration Tests. Wir können uns immer noch umentscheiden, falls die Business-Logik tatsächlich die Query-Methode benötigt.

Doubles mit Port Contract Tests synchron zu Produktionscode halten

Wir sind bisher davon ausgegangen, dass man ein *Jpa*- immer durch ein *InMemory* Repository austauschen kann. Das geht, weil wir die Ports & Adapters Architektur *[6]* mit sogenannten *Port Contract Tests [7]* kombinieren.

JpaBooks implementiert das Interface *Books*. Das Interface ist ein so genannter **Port**. Alle Klassen, die das Interface implementieren, sind **Adapter** davon. Die Domänen-Logik kennt allerdings nur die Ports und weiß nicht, welche Implementierung dahintersteckt. Damit haben wir die Domänen-Logik davon entkoppelt, wie der Code, der mit der Außenwelt kommuniziert, tatsächlich aussieht. Theoretisch muss beim Schreiben der Domänen-Logik noch nicht mal eine Implementierung vom Port existieren.

Die *Ports & Adapters Architektur [6]* hilft uns zum einen besser zu modellieren. Wir können zuerst die Domänen-Logik modellieren, bevor wir uns den Implementierungsdetails zuwenden müssen. Zum anderen bietet uns die Architektur auch die Möglichkeit, für Tests echte Adapter durch Test Doubles *[8]* auszutauschen. In unseren Unit-Tests nutzen wir daher ein *InMemoryBooksDouble* statt eines langsameren und teureren *JpaBooks* Repositories.

InMemoryBooksDouble ist eine bestimmte Art von Double, ein sogenannter *Fake [9]*. Im Gegensatz zu den anderen Double-Arten (Dummies, Stubs, Spies und Mocks *[10]*) sind Fakes funktionierende Implementierungen von Ports, die aber Abkürzungen nehmen, die der Produktionscode nicht nehmen kann – in diesem Fall die InMemory-Lösung.

Im Gegensatz zu anderen Doubles muss der Fake allerdings die Erwartungen erfüllen, die der Domänen-Code an den Port hat. Bei Repositories erwartet der Domänen-Code beispielsweise, dass eine Entity, die mit `add()` hinzugefügt wurde, danach auch mit einem `find()` wiedergefunden werden kann. Die Erwartungen, die die Domäne an den Port hat, nennen wir **Contract** und wir können sie mit einem **Port Contract Test** überprüfen *(siehe Listing 2)*.

<1> Der Contract ist abstract. Erst durch die Implementierung wird er zum ausführbaren Test.

```
public abstract class BooksContract { // <1>
  abstract Books testee(); // <2>

  @Test
  void should_remember_book(TestState a){ // <3>
     // given
     var book = a.book();
     var testee = testee();
     // when
     testee.add(book);
     // then
     assertThat(testee.findById(book.id())).
isEqualTo(book);
  }
}
```

Listing 2: Port Contract Test unseres Ports

<2> Wir kennen im Test nur den Port, nicht die Implementierung.
<3> Jeder Test beschreibt ein Verhalten, das wir vom Port erwarten.

Der Test der Implementierung ist sowohl für den Fake *(siehe Listing 3)* als auch den Produktionsadapter denkbar kurz.

```
@Unit
public class InMemoryBooksTest extends BooksContract {

  @Override
  Books testee() { // <1>
     return new InMemoryBooks();
  }
}
```

Listing 3: Test des Port-Adapters

<1> Adapter-Tests implementieren in der Regel nur die Methode, die den `testee` erstellt.

Auch der Fake ist denkbar einfach zu schreiben *(siehe Listing 4)*, dank einer wiederverwendbaren Basis *(siehe Listing 5)*.

```
public class InMemoryBooksDouble
        extends BaseInMemoryDouble<BookId, Book>
        implements Books {
     // <1>, <2>
}
```

Listing 4: Ein InMemory Fake ist dank der Basisklasse schnell zu schreiben.

<1> In den meisten Repositories benötigen wir keine besonderen Methoden. Wir implementieren und nutzen nur, was die Basis auch hat.
<2> Besondere Methoden entstehen hier meist nur durch Queries. Einfache Queries können wir mit der `filter(predicate)`-Methode aus der Basisklasse *(siehe Listing 5)* lösen. Bei komplexeren Filtermethoden können wir allerdings immer sagen, dass wir diese nicht implementieren und lieber auf einen langsameren Integration-Test setzen.

<1> Für Tests brauchen wir nur eine `HashMap`. Falls wir vorhaben, auch parallelen Code zu testen, sollten wir allerdings gleich auf eine `ConcurrentHashMap` setzen.
<2> Einfache Queries sind per `predicate` lösbar. Für unsere Unit-

European NetSuite User Days

November
2024
18 - 19

at NCC Ost
Nuremberg

netsuite.doag.org
#NetSuiteUserDays

```
public abstract class BaseInMemoryDouble<TId, TEntity extends Entity<TId>> {
  private Map<TId, TEntity> entities = new HashMap<>(); // <1>

  public List<TEntity> filter(Predicate<TEntity> predicate){
     return this.entities.values().stream()
           .filter(predicate)
           .toList(); // <2>
  }

  // <3>
}
```

Listing 5: Die Basisklasse hat wenig Logik und delegiert immer an die JDK Map.

Tests brauchen wir hier keine komplizierten Indizes, da unsere `HashMap` nur ein paar wenige Entities beinhaltet.
<3> Weitere Methoden wie `findById()`, `add()`, `remove()`, `removeIf()` und `count()` reichen nur an die (`Concurrent`)`HashMap` durch. Wir implementieren an dieser Stelle nichts Besonderes, sondern nutzen, was uns das JDK gibt.

Mit diesen Tests können wir nun garantieren, dass sich alle Adapter des Ports gleich verhalten. Sie werden immer synchron zu dem sein, was wir in den Tests als Erwartung (aka Contract) definieren.

Contract-Tests sind eine Idee von J. B. Rainsberger *[11]*. Wir nennen sie hier nur ***Port*** *Contract Tests*, um expliziter zu machen, welchen Contract sie testen wollen. Dadurch sind sie auch abgegrenzt von den ***Integration*** *Contract Tests* *[12]* und dem *Consumer-Driven-Contracts*-Ansatz *[13]*. Ein alternativer Name für die *Port Contract Tests* ist Role-Tests *[14]*.

Strukturzementierende Mocks und flexible Doubles

In unserem Test haben wir bisher nur eine Form der Test Doubles *[8]* genutzt, die InMemory Fakes *[9]*. Neben den Fakes gibt es noch Stubs, Spies und Mocks. Sie sind folgendermaßen definiert:

- *Fakes [9]* sind funktionierende Implementierungen, die aber Abkürzungen nehmen können, die der Produktionscode nicht nehmen kann. Wir halten sie mit *Port Contract Tests* synchron. Fakes erkennt man daran, dass ihre Implementierung grob dasselbe macht, wie die Produktionsimplementierung.

- *Stubs [15]* erlauben uns, **indirekte Inputs** in unseren *Testee* einzugeben. Indirekt, weil man diese Inputs nicht als Parameter an den *Testee* übergibt, sondern dieser sich die Inputs selbst zieht. Stubs erkennt man daran, dass wir ihnen Testdaten mitgeben, die sie auf Anfrage des *Testee* möglichst stumpf zurückgeben. Es gibt hier keine großartige Logik.

- *Mocks [16]* erlauben uns, **indirekte Outputs** von unseren *Testees* zu überprüfen. Indirekt, weil man diese Outputs nicht als Rückgabewert vom *Testee* bekommt, sondern über Umwege abgreifen und verifizieren muss. Das Ganze ist auch als Verhaltensverifikation bekannt. Mocks erkennt man daran, dass man den Mock direkt auffordert, zu verifizieren, ob er (mit bestimmten Parametern) aufgerufen wurde. Der Test ruft eine Framework-Methode (`verify(mock).sth(param)`) oder eine selbstgeschriebene Methode (`mock.verifyAddWasCalled()`) auf.

Alle drei Formen von *Test Doubles* können mit einem Mocking-Framework umgesetzt werden, aber nur für die Mocks ist ein Mocking-Framework überhaupt sinnvoll. Da man Mocks nur sehr selten benötigt, braucht man ein Mocking-Framework auch nur sehr selten. Das ist gut, weil gerade der exzessive Einsatz des Frameworks ebenfalls zur Strukturzementierung führt *(siehe Abbildung 2)*.

Wenn wir in *n* Tests immer wieder die gleichen Methoden reimplementieren, dann …

1. zementieren wir das Design auf Type-Ebene.
2. kann unsere Reimplementierung vom echten Code abweichen. Die Abweichung kann sogar so stark sein, dass wir die Encapsulation des Ports brechen *[17]*.

Ersteres nimmt uns die Möglichkeit, unsere Struktur zu ändern. Letzteres ist aber vielleicht noch schlimmer, denn unser Test kann mit dem Mock grün sein, obwohl er mit dem Produktionscode rot wäre. In der Folge vertrauen wir dann unseren Tests nicht mehr.

In „The Art of Unit Testing" *[18]* lautet die Empfehlung, Mocks nur dann zu nutzen, wenn wir die Interaktion mit einem *external Service* testen wollen. Dann braucht man Mocks nur in 2 % bis 5 % der Unit-Tests.

Für die allermeisten Tests nutzen wir also entweder gar kein Double (Methode, die nur berechnet und wir können auf den Rückgabewert

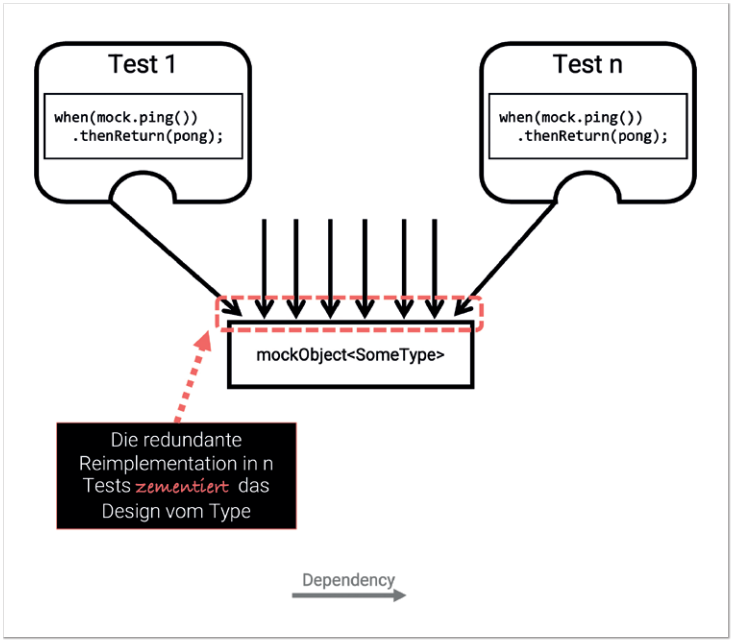


Abbildung 2: Die Reimplementation derselben Methode in n Tests führt zur Strukturzementierung. (© Richard Gross)

asserten), einen (InMemory-) Fake oder einen Stub und diese schreiben wir dann schnell von Hand (für Fake, siehe *Listing 5* und für Stub, siehe *Listing 6*).

```
public class IsbnApiEchoDouble { // <1>

  private final String bookTitleEcho;

  public SomeRemoteApiEchoDouble(String bookTitleEcho){
     this.bookTitleEcho = bookTitleEcho != null
                          ? bookTitleEcho
                          : "Refactoring";
  }

  public String findTitle(Isbn isbn) {
     return this.bookTitleEcho; // <2>
  }
}
```

Listing 6: Ein simpler Stub.

<1> Es gibt unterschiedliche Arten von Stubs. Dieser gibt immer ein Echo der Werte zurück, die er im Constructor bekommen hat.
<2> Keine besondere Logik hier. Es wird nur zurückgeben, was man im Constructor bekommen hat.

Das Selbstschreiben gibt uns dann auch einen einzigen Ort, an dem man Strukturänderungen des echten Ports nachpflegen kann, ohne dass der Test davon betroffen sein muss.

Builder Design

Die generische `with()`-Methode beschleunigt das Schreiben des initialen Builders *(siehe Listing 7)*, benötigt dafür aber *public* Fields.

Bisher sind wir den Trade-off eingegangen. Es gibt aber Situationen, in denen man umschwenken kann und spezifischere `withX()` oder `isX()` sinnvoll werden:

1. Neuen Methoden, die Tests angenehmer machen.
2. Falls zwei oder mehr Fields nur gemeinsam verändert werden können.

Ersteres passiert, wenn wir uns zum Beispiel entscheiden, den Autorennamen nicht mit *stringly* sondern **strongly** *[19]* als `AuthorName` zu typisieren, um ähnlich der Ids mehr *Compile-Time-Safety* zu haben. Dann ist die generische `with()`-Methode nicht mehr ganz so angenehm zu nutzen, weil wir immer `with(it -> { author = new AuthorName("Alistair"); })` schreiben müssen. Wir können dann einen `withAuthor(String name)` und einen `withAuthor(AuthorName name)` Overload einführen und damit Tests flexibler und lesbarer machen.

Wenn ein `Book` noch ein Field `rentedOn` bekommt, kann es zum Beispiel passieren, dass zwei oder mehr Felder voneinander abhängen. `rentedOn` muss zeitlich immer nach `createdOn` liegen. Mit unserem generischen `with()` können wir allerdings ein Objekt erzeugen, das invalide ist, weil wir nur `rentedOn` gesetzt haben. Das ist kein großes Problem, wenn wir im Konstruktor einer Klasse oder Record immer validieren, ob die Fields (beziehungsweise der State) korrekt sind. `BookBuilder` würde dann allerdings etwas zulassen, was `Book` zur Runtime mit einer `IllegalArgumentException` quittiert.

Um wieder mehr Compile-Time-Safety zu haben, können wir `rentedOn` im Builder wieder *private* machen und `isRentedOn(Duration rentedAfterCreate)` mitsamt dem Overload `isRentedOn(Instant createdOn, Duration rentedAfterCreate)` einführen. Das neue Präfix `is` zeigt uns, dass die Methode konzeptionell etwas anderes macht, als ein `with`. `is` deklariert, wodurch die Methode mehrere voneinander abhängige Werte setzt. Der Overload zeigt uns, von welchem Wert der Parameter `rentedAfterCreate` abhängig ist.

Das neue Präfix ist ebenso dafür da, damit wir merken, ob unser Builder anfängt, zu komplex zu werden. Wenn die Anzahl der `is`-Methoden die `with`-Methoden übersteigt, dann bewegt sich unser Builder in gefährlichen Wassern.

```
public class BookBuilder extends TestBuilder<Book> {

  public BookId id = ids.next(BookId.class);
  public String title = "Refactoring";
  public String author = "Martin Fowler";
  public Instant createdOn = clock.now();

  public BookBuilder(Clock clock, Ids ids){
     super(clock, ids);
  }

  public Book build(){
     return new Book(id, title);
  }

  public BookBuilder with(Consumer<? super BookBuilder> action) { // <4>
     action.accept(this);
     return this;
  }

}
```

Listing 7: Entity-TestBuilder

TestDsl in Kombination mit Spring

Die in Teil 1 geschriebene JUnit-Extension kann auch mit `@SpringBootTest` kompatibel gemacht werden. Die Extension muss nur prüfen, ob ein *ApplicationContext* existiert. Falls ja, zieht sie den Floor aus dem Spring *DI-Container* statt aus dem JUnit Store *(siehe Listing 8)*.

```
@Override
public Object resolveParameter(
        ParameterContext parameterContext,
        ExtensionContext extensionContext
  ) throws ParameterResolutionException {
  // ...
  var springFloor = SpringExtension
        .getApplicationContext(extensionContext)
        .getBeanProvider(Floor::class.java)
        .ifAvailable;
  // ...
}
```

Listing 8: TestDsl mit Floor aus Spring

Durch die Annotation können wir jetzt den Test aus *Listing 9* schreiben.

<1> Wir kombinieren die SpringBootTest-Annotation mit der TestDsl-Annotation.
<2> Wir bitten Spring, uns den *testee* zu injecten.
<3> Wir können hier die TestDsl wie in jedem anderen Test nutzen. Die Repositories, die Spring kennt, und die der TestDsl sind dieselben.

Wenn man auf `@SpringBootTest` setzt, muss man allerdings aufpassen, wie man seine Tests schreibt und wie umfangreich sie sind. Der *Spring Application Context* wird für Tests gecached und hebelt dann die Test-Isolation aus. Modifikationen, die ein Test macht, können einen später laufenden Test zum Fehlschlagen bringen. Dadurch kann man Tests nicht mehr parallelisieren.

Unit-Tests sollten daher (fachliche Domänen-) Logik ohne Spring testen. Das entspricht auch der Empfehlung, die das Spring-Framework seit Version 2 *[20]* gibt und bis zur aktuellen Version 6 *[21]* beibehalten hat. Ein `@Integration @SpringBootTest` kommt sporadisch für wichtige Testpfade durch die Anwendung hinzu.

Low und High Level Test DSLs

Die bisher gezeigte TestDsl für `@Unit` und `@Integration` ist eine **Low-Level** TestDsl. Sie wirkt der Strukturzementierung entgegen und macht Tests „unter der Haube" angenehmer zu schreiben. Durch den direkten Zugriff auf Domänen-Objekte sind wir sehr flexibel, welche Testzustände wir hervorrufen können. Wir können mit ihr den *Happy Path*, die *Sad Paths* und auch sehr viele *Strange Paths* überprüfen – also Pfade, die eigentlich nie auftreten sollten.

Sie ist allerdings nicht aus Nutzersicht geschrieben und man kann mit ihr nicht verifizieren, dass das System sich aus Nutzersicht richtig verhält. Für solche Tests brauchen wir ein laufendes System, auf das wir von außen per Browser, HTTP-API oder ähnlichem zugreifen. Google würde diese Tests „Large" nennen *[3] (siehe Tabelle 1)*. Andere geläufige Namen sind System-Tests oder User-Acceptance-Tests.

Wir brauchen für diese Tests eine neue DSL mit einem anderen Aufbau, aber auch einem sehr ähnlichen Konzept dahinter. Diese **High-Level** `@Acceptance` DSL hat allerdings nichts mehr mit Strukturzementierung zu tun, sondern mit UI- beziehungsweise API-zementierung. Je mehr Tests wir haben, die einen bestimmten Button oder ein bestimmtes Widget verlangen, desto zementierter ist diese UI-Komponente. Bei einem öffentlichen API ist diese Zementierung vielleicht noch gewollt, man will anderen ja ein stabiles API anbieten. Aber auch dann ist eine DSL empfehlenswert, denn sie macht die Tests deutlich lesbarer und wartbarer.

Die im Folgenden kurz skizzierte High-Level TestDsl ist die Umsetzung der 4-Layer-Acceptance-Test-Struktur von Dave Farley *[22]*. Die vier Layer sind:

1. Top: unser Test
2. DSL pro Domäne: renting, buying, etc.
3. Protocol drivers: UI, API, External System Stub
4. Das zu testende System

Unser Test greift also nicht mehr direkt auf das API unseres Systems zu. Es gibt kein `http.get("/api/users")`. Der Test klickt auch nicht direkt im Browser. Es gibt kein `page.navigate()` oder `page.click()`. Der Test kennt nur den nächsten Layer, die DSL.

Die DSL bietet nur Domänen-spezifische User-Ziele, gibt jedoch nicht an, wie die Ziele technisch umgesetzt werden (mit der `renting`-DSL könnten wir zum Beispiel `.findBook("Refactoring").rent()` umsetzen). Sie kennt nur die *Protocol Driver* und delegiert die Umsetzung an diese.

Erst die Driver kennen das zu testende System. Der UI-Driver weiß, wie man die Ziele mit zum Beispiel *Playwright* umsetzt, während der *API Protocol Driver* die Ziele beispielsweise per *RestAssured* umsetzen kann. Welchen Driver man nutzt, steuert man per Annotation *(siehe Listing 10)*.

<1> Diesen Test führen wir über den Browser, aber auch über das HTTP-API aus.

```
@Integration @SpringBootTest @Test // <1>
void should_be_able_to_rent_book_via_api(TestState a, Floor floor, @Autowired BookController testee){ // <2>
  // rest of test
  // <3>
}
```

Listing 9: TestDsl mit Floor aus Spring

```
@Acceptance @UiProtocol @ApiProtocol @Test // <1>
void should_be_able_to_rent_book(InventoryDsl inventory, RentingDsl renting){
  // given
  inventory.addBook("Refactoring"); // <2>, <3>
  var book = renting.findBook("Refactoring");
  // when
  book.rent();
  // then
  assertThat(book.isRented()).isTrue();
}
```

Listing 10: High-Level TestDsl

<2> Genauso wie bei der Low-Level-DSL muss jeder Test seinen kompletten State erstellen.
<3> Anders als bei der Low-Level-DSL, macht diese DSL allerdings deutlich größere Schritte. Ein Buch anzulegen, kann aus sehr vielen Browser-Aktionen beziehungsweise API-Calls bestehen. Wenn einer der Zwischenschritte fehlschlägt, bricht die DSL sofort ab und gibt das spezifische Feedback, welcher der Zwischenschritte nicht funktioniert hat.

Parallelisieren kann man diese Tests auch auf eine ähnliche Weise, wie wir das bei Integration-Tests gemacht haben: Wir können entweder direkt in unserem zu testenden System einen Namespace pro Test vorsehen oder das über unsere DSL lösen.

Ersteres ist möglich, wenn man von Anfang an Mandantenfähigkeit in sein System einbaut. Jede Entity benötigt dann eine `MandantenId` und man muss vorsehen, dass jeder nur die Daten des eigenen Mandanten sehen kann. Wenn man pro Test nun einen neuen Mandanten anlegt und der Test auch alle Vorbedingungen in Form von Entities anlegt, dann sind die Tests über die `MandantenId` voneinander isoliert und somit parallelisierbar.

Wenn die `MandantenId` nicht direkt in das System eingebaut werden kann, nutzt man das von Dave Farley genannte *Test Data Aliasing [22]*. Bei diesem Vorgehen sorgt die TestDsl selbst dafür, dass die Testdaten eindeutig sind. Sie ergänzt bei Fields dann einen Test-eindeutigen Key. `addBook("Refactoring")` legt nicht das Buch „Refactoring" an, sondern das Buch „Refactoring dbac1q23". `findBook("Refactoring")` sucht nicht nach „Refactoring", sondern nach „Refactoring dbac1q23". Beim Testschreiben muss man allerdings aufpassen, dass man nicht auf die Anzahl der Bücher oder ähnlichem assertet, denn diese könnte sich durch parallellaufende Tests fortlaufend ändern.

Insgesamt ergänzt die hier geschilderte High-Level DSL die Low-Level DSL um die Anwendersicht. Den Großteil der Tests schreiben wir mit der Low-Level DSL, aber gerade kritische Anwendungsfelder testen wir mit der High-Level DSL aus Anwendersicht.

Ausblick

Die TestDsl kombiniert bestehende Konzepte wie Builder, Ports *[6]*, Port Contract Tests *[23]*, Stubs *[10]* und Fakes *[9]* und bietet ein einheitliches API für alle unsere Unit- und Integration-Tests. Mit der TestDsl konnten wir die Strukturzementierung durch redundantes Test-Setup lösen. Wie wir die TestDsl einsetzen, um Strukturzementierung durch Tests auf falscher Ebene zu verhindern, werden wir in Teil 3 zeigen.

Wer sich mehr für das Thema interessiert, kann TestDsl-Beispielcode online einsehen *[24]* oder sich den Vortrag zur „Beehive Architecture" *[25]* anschauen, der sich auch um die TestDsl dreht.

Quellen

[1] M. Fowler, „Domain Specific Language". 2008. Verfügbar unter: *https://martinfowler.com/bliki/DomainSpecificLanguage.html*
[2] M. Fowler, „On the Diverse And Fantastical Shapes of Testing". 2021. Verfügbar unter: *https://martinfowler.com/articles/2021-test-shapes.html*
[3] S. Stewart, „Test Sizes". 2010. Verfügbar unter: *https://testing.googleblog.com/2010/12/test-sizes.html*
[4] M. Fowler, „Unit Test". 2014. Verfügbar unter: *https://martinfowler.com/bliki/UnitTest.html#SolitaryOrSociable*
[5] M. Feathers, „A Set of Unit Testing Rules". 2005. Verfügbar unter: *https://www.artima.com/weblogs/viewpost.jsp?thread=126923*
[6] A. Cockburn, „Hexagonal architecture". 2005. Verfügbar unter: *https://alistair.cockburn.us/hexagonal-architecture/*
[7] R. Gross, „Contract Tests in Kotlin". 2020. Verfügbar unter: *http://richargh.de/posts/Contract-Tests-in-Kotlin*
[8] G. Meszaros, „Test Double". 2011. Verfügbar unter: *http://xunitpatterns.com/Test%20Double.html*
[9] G. Meszaros, „Fake Object". 2011. Verfügbar unter: *http://xunitpatterns.com/Fake%20Object.html*
[10] M. Fowler, „Mocks Aren't Stubs". 2007. Verfügbar unter: *https://martinfowler.com/articles/mocksArentStubs.html*
[11] J. B. Rainsberger, „Getting Started with Contract Tests". 2017. Verfügbar unter: *https://blog.thecodewhisperer.com/permalink/getting-started-with-contract-tests*
[12] M. Fowler, „Integration Contract Test". 2011. Verfügbar unter: *https://martinfowler.com/bliki/ContractTest.html*
[13] I. Robinson, „Consumer-Driven Contracts: A Service Evolution Pattern". 2006. Verfügbar unter: *https://martinfowler.com/articles/consumerDrivenContracts.html*
[14] M. Rivero, „Role tests for implementation of interfaces discovered through TDD". 2022. Verfügbar unter: *https://codesai.com/posts/2022/04/role-tests*
[15] G. Meszaros, „Test Stub". 2011. Verfügbar unter: *http://xunitpatterns.com/Test%20Stub.html*
[16] G. Meszaros, „Mock Object". 2011. Verfügbar unter: *http://xunitpatterns.com/Mock%20Object.html*
[17] M. Seemann, „Stubs and mocks break encapsulation". 2022. Verfügbar unter: *https://blog.ploeh.dk/2022/10/17/stubs-and-mocks-break-encapsulation/*

[18] R. Osherove, „Art of Unit Testing (3. Edition)“. 2024. Verfügbar unter: *https://www.artofunittesting.com/*
[19] T. Spring, „Stringly Typed vs Strongly Typed“. 2022. Verfügbar unter: *https://www.hanselman.com/blog/stringly-typed-vs-strongly-typed*
[20] T. Spring, „Unit Testing“. 2006. Verfügbar unter: *https://docs.spring.io/spring-framework/docs/2.0.4/reference/testing.html#unit-testing*
[21] T. Spring, „Unit Testing“. 2022. Verfügbar unter: *https://docs.spring.io/spring-framework/docs/6.0.0/reference/html/testing.html#unit-testing*
[22] D. Farley, „Acceptance Testing for Continuous Delivery [#AgileIndia2019]“. 2019. Verfügbar unter: *https://www.youtube.com/watch?v=Rmz3xobXyV4*
[23] R. Gross, „Contract Tests in Kotlin“. 2020. Verfügbar unter: *http://richargh.de/posts/Contract-Tests-in-Kotlin*
[24] R. Gross, „TestDsl (Avoid structure-cementing Tests)“. 2024. Verfügbar unter: *https://github.com/Richargh/testdsl*
[25] R. Gross, „Beehive Architecture“. 2023. Verfügbar unter: *http://richargh.de/talks/#beehive-architecture*

Richard Gross
richard.gross@maibornwolff.de

Richard Gross ist IT-Archäologe und arbeitet seit 2013 für den Bereich IT-Sanierung. Seine Schwerpunkte sind hexagonale Architekturen, Hypermedia-driven APIs sowie die ausdrucksstarke und eindeutige Modellierung der Domäne als Code. Daher ist er von vielen eher jüngeren Programmiersprachen wie Kotlin oder F# begeistert, die ihn bei der Modellierung unterstützten. Außerdem hat er auch lange Zeit das F&E Projekt CodeCharta begleitet, mit dem auch Nicht-Entwickler ein Verständnis für die Qualität ihrer Software bekommen können. Privat saniert Richard auch; sein Eigenheim.

KI und generierte Unittests: Eine Analyse

Jan-Hendrik Telke, Neozo GmbH

Einfach die generative KI Unittests schreiben lassen? Ist das wirklich eine gute Idee? Wie gut sind die generierten Tests? In diesem Artikel wird eine detaillierte Analyse des entstandenen Codes vorgestellt und aufgezeigt, wie man selbst Tests generieren lassen kann. Der Artikel hebt viele Schwachstellen, aber auch einige positive Aspekte von KI-generiertem Code hervor und zeigt dabei auch die Möglichkeiten generativer KIs im Coding auf.

Testgenerierung ist ein Konzept, das schon vor einigen Jahren aufgekommen ist, sich allerdings in der Praxis nie flächendeckend durchgesetzt hat. Das komplexe Problem konnte mit herkömmlichen Algorithmen nicht zufriedenstellend gelöst werden *[1]*. Man kann annehmen, dass die jüngsten Fortschritte von LLM (*Large Language Models*) im Bereich der Code-Generierung auch auf die Generierung von Unittests anwendbar sein könnten. Der Frage, ob dem so ist, widmet sich dieser Artikel.

Prompt Engineering

Prompt Engineering beschreibt den Prozess, Anfragen an LLM möglichst so zu formulieren, dass die Ausgabe der gewünschten Form entspricht. Insbesondere beim Generieren von Code mit LLM ist es wichtig, im Prompt enge Grenzen und syntaktische Anforderungen zu definieren. Neben der Anfrage, Tests für eine gegebene Methode zu generieren, werden im Prompt auch Randbedingungen explizit eingefordert *(siehe Listing 1)*. So wird neben der Ausgabe-Programmiersprache auch das zu nutzende Test-Framework (*JUnit 4*) festgelegt.

Dieser Prompt kann als Rahmen genutzt werden, um wiederholte Anfragen mit möglichst reproduzierbaren Ergebnissen zu erhalten. Dabei kann das Tool *TestSpark* *[2]* helfen, das eine Integration in *IntelliJ* bietet und das Template direkt aus dem Kontext-Menü aufrufbar macht.

Einfaches Codebeispiel

Im nachfolgenden Abschnitt werden Fragmente des Open-Source-Projekts *AssertJ* *[3]* verwendet. Das vorliegende Codebeispiel *(siehe Listing 2)* soll einige Eigenschaften des generierten Codes darlegen. Für diese simple Methode sollen nun einige Unittests generiert werden. Das sollte keine allzu große Herausforderung darstellen. Für die Generierung wird *ChatGPT* von *OpenAI* benutzt (insbesondere GPT-3.5, Stand März 2023).

In diesem Beispiel *(siehe Listing 3)* liefert *ChatGPT* den korrekten Code und erklärt diesen. Mit den drei Testfällen werden alle Ausführungspfade abgedeckt. Mithilfe von *TestSpark* wird der Test auch direkt in die IDE eingebunden und eine neue Klasse erstellt.

Komplexere Anwendungsfälle

In diesem Teil soll es darum gehen, die Grenzen derzeitiger LLMs im Bereich der Testgenerierung durch ein komplexeres Beispiel *(siehe Listing 4)* auszuloten.

Es wird wieder der Prompt aus dem ersten Beispiel genutzt. Die Methode *(siehe Listing 5)* ist allerdings etwas komplexer und nutzt einige interne Klassen und Methoden des *AssertJ*-Projekts.

Die Klasse des Methoden-Parameters `AssertionInfo` ist *ChatGPT* nicht bekannt. Hier wird davon ausgegangen, dass `AssertionInfo` einen Konstruktor hat, dem ist jedoch nicht so. Auch ein Entwickler müsste hier in die verwendeten Klassen schauen, um einen validen Test zu schreiben. Gibt man nun allerdings die Klasse als zusätzlichen Kontext an *ChatGPT*, erreicht man hier leider keine Verbesserung.

Das Problem bleibt bestehen und *ChatGPT* trifft fehlerhafte Annahmen. Das ist zunächst kein größeres Problem und kann von Hand schnell behoben werden. Allerdings lässt sich so kein größeres Testset automatisiert generieren.

Bei größeren Projekten wird auch die Kontextgröße von LLMs zum Problem. Die Kontextgröße zum Beispiel von GPT-4 liegt bei 32.000 Token. Das sind weniger als 5.000 Zeilen Code und nicht genug, um komplexe Projekte komplett zu beachten.

Insgesamt scheinen sich LLMs derzeit mit stark typisierter Sprache noch schwer zu tun. Dieses Bild wird auch in der Analyse noch verstärkt. Einfache Methoden, die Java-Standard-Bibliotheken und -Schnittstellen verwenden, wurden gut erkannt und getestet. Me-

```
Don't use @Before and @After test methods.
All tests should be for $TESTING_PLATFORM.
In case of mocking, use $MOCKING_FRAMEWORK. But, do not use mocking for all tests.
Name all methods according to the template - [MethodUnderTest][Scenario]Test, and use only English letters.
The source code of method under test is as follows:
$CODE
```

Listing 1: Beispielprompt aus TestSpark

```
Generate unit tests in Java. Don't  use @Before and @After test methods. Make tests as atomic as possible. All tests
should be for JUnit5. In case of mocking, use Mockito. But, do not use mocking for all tests. Name all methods ac-
cording to the template - [MethodUnderTest][Scenario]Test, and use only English letters. Here is the line that needs
to be tested it can be found in the class Arrays
public static boolean isArray(Object o) {
return o != null && o.getClass().isArray();
}
```

Listing 2: Codeausschnitt, AssertJ

thoden, die verstärkt projekteigenen Code verwenden, wurden tendenziell schlechter verarbeitet. Im nächsten Abschnitt gibt es einen kleinen Exkurs zu Qualitätsmerkmalen und anschließend wird der hier gewonnene Eindruck mit einigen Zahlen hinterlegt.

Exkurs: Qualitätsmerkmale von Unittests

Wie lassen sich nun objektiv die Mehrwerte von generierten Unittests bewerten? Dazu muss man einen näheren Blick auf die Qualitätskriterien von Tests werfen. Das am meisten verwendete – und am meisten diskutierte – Merkmal ist sicherlich die Testabdeckung. Eine hohe Testabdeckung sollte aber mehr als Indiz für eine hohe Anzahl an Tests und nicht als echtes Qualitätsmerkmal der Tests angesehen werden.

Ein echtes Qualitätsmerkmal, das in realen Projekten verwendet werden kann, ist die Anzahl der *Escaped Bugs* [6]. Man zählt Bugs, die nicht durch Unittests gefangen wurden. Das ist sehr aufwändig und kann nur schwer in diesem Rahmen ermittelt werden.

Here are the unit tests in Java using JUnit 5 for the provided method:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class ArraysTest {

     @Test
     public void isArrayNotNullArrayTest() {
            // Given
            Object array = new int[5];

            // When
            boolean result = Arrays.isArray(array);

            // Then
            assertTrue(result);
     }

     @Test
     public void isArrayNotNullNonArrayTest() {
            // Given
            Object nonArray = "Not an array";

            // When
            boolean result = Arrays.isArray(nonArray);

            // Then
            assertFalse(result);
     }

     @Test
     public void isArrayNullTest() {
            // Given
            Object nullObject = null;

            // When
            boolean result = Arrays.isArray(nullObject);

            // Then
            assertFalse(result);
     }
}
```

These tests cover the basic scenarios of the `isArray()` method without using any mocking. Each test is self-contained and does not rely on the state of other tests, making them atomic.

Listing 3: ChatGPT-Ausgabe

```
Generate unit tests in Java. Don't use @Before and @After test methods. Make tests as atomic as possible. All tests
should be for JUnit5. In case of mocking, use Mockito. But, do not use mocking for all tests. Name all methods ac-
cording to the template - [MethodUnderTest][Scenario]Test, and use only English letters. Here is the line that needs
to be tested it can be found in the class Uris
public void assertHasPath(AssertionInfo info, URI actual, String path) {
  assertNotNull(info, actual);
  if (!Objects.equals(actual.getPath(), path)) throw failures.failure(info, shouldHavePath(actual, path));
}
```

Listing 4: Prompt

```
Here are the unit tests for the assertHasPath() method in the Uris class:
java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class UrisTest {

     @Test
     void assertHasPathMatchingPathTest() {
           // Arrange
           AssertionInfo info = new AssertionInfo("Test Info");
           URI actual = URI.create("http://example.com/test");
           String path = "/test";

           // Act and Assert
           assertDoesNotThrow(() -> {
                 Uris uris = new Uris();
                 uris.assertHasPath(info, actual, path);
           });
     }

     @Test
     void assertHasPathNotMatchingPathTest() {
           // Arrange
           AssertionInfo info = new AssertionInfo("Test Info");
           URI actual = URI.create("http://example.com/test");
           String path = "/wrongpath";

           // Act
           Uris uris = new Uris();

           // Assert
           AssertionError exception = assertThrows(AssertionError.class, () -> {
                 uris.assertHasPath(info, actual, path);
           });
           assertTrue(exception.getMessage().contains("Expected path: </wrongpath> but was: </test>"));
     }

     @Test
     void assertHasPathWithNullInfoTest() {
           // Arrange
           AssertionInfo info = null;
           URI actual = URI.create("http://example.com/test");
           String path = "/test";

           // Act and Assert
           assertThrows(NullPointerException.class, () -> {
                 Uris uris = new Uris();
                 uris.assertHasPath(info, actual, path);
           });
     }

     @Test
     void assertHasPathWithNullActualUriTest() {
           // Arrange
           AssertionInfo info = new AssertionInfo("Test Info");
           URI actual = null;
           String path = "/test";

           // Act and Assert
           assertThrows(NullPointerException.class, () -> {
                 Uris uris = new Uris();
                 uris.assertHasPath(info, actual, path);
           });
     }

     @Test
     void assertHasPathWithNullPathTest() {
           // Arrange
           AssertionInfo info = new AssertionInfo("Test Info");
           URI actual = URI.create("http://example.com/test");
           String path = null;

           // Act and Assert
           assertThrows(NullPointerException.class, () -> {
                 Uris uris = new Uris();
                 uris.assertHasPath(info, actual, path);
           });
     }
}
These tests cover different scenarios such as matching path, not matching path, null assertion info, null actual
URI, and null path. They adhere to the naming convention specified and are designed to be as atomic as possible.
```

Listing 5: Response

Zwei weitere Merkmale, die sich auch leichter künstlich nachbilden lassen, sind *Defect Resolution* und *False Defect Detection*. Um die *Defect Resolution* zu messen, werden zufällig Fehler im Code eingestreut, der Anteil der detektierten Fehler sollte dann möglichst groß sein. Bei der *False Defect Resolution* wird gemessen, wie viele Fehler nach gültigen Refactorings im Code erkannt werden.

Eine weitere Möglichkeit ist es, den generierten Code von Hand auszuwerten und auf bekannte *Code Smells* zu untersuchen.

Empirische Analyse

Um eine gute Basis für die Testgenerierung zu haben, wurden die bestehenden Tests des Open-Source Projekts *AssertJ* [4] gelöscht und neue Tests generiert.

In der Analyse wurde ein Teil des *AssertJ*-Projekts mit 783 Klassen betrachtet. Dabei konnte bei 483 Klassen keine syntaktisch korrekte Testklasse erstellt werden. Durch kleinere händische Korrekturen könnte hier die Zahl der kompilierenden Tests erhöht werden. Darauf wurde hier bewusst verzichtet.

Von insgesamt 203 erfolgreich generierten Testklassen schlugen ungefähr 54 % bei einem Testdurchlauf fehl. Es ist anzunehmen, dass der Code größtenteils fehlerfrei ist, deswegen muss hier von fehlerhaft generierten Tests ausgegangen werden.

Die erzeugte *Code Coverage* durch die generierten Tests ist akzeptabel und liegt bei knapp über 80 %.

Im Code wurden dann an insgesamt 50 Stellen boolesche Operatoren umgedreht, sodass der Code sich fehlerhaft verhält. Von diesen 50 Stellen wurden 28 erkannt. Hier hätten die generierten Tests in einem realen Test einen echten Mehrwert geboten und potenzielle Bugs verhindert.

Fazit

Insgesamt muss sich die Code-Generierung mit LLMs noch weiterentwickeln, um unumstritten einen Mehrwert zu bieten. Außerdem werden sich spezialisierte Ansätze gegenüber allgemeinen Ansätzen durchsetzen, sollte die Nachfrage nach generierten Tests steigen. So haben Entwickler bei *Meta* [5] mit einem spezialisierten Ansatz deutlich größere Erfolge vorweisen können.

Für diesen spezialisierten Ansatz wurde die bestehenden Test-Suite als Input an das LLM-Modell gegeben, wodurch die bestehende Test-Suite erfolgreich erweitert werden konnte. Solch spezialisierte Ansätze sind derzeit noch nicht als Produkt verfügbar, könnten aber vielversprechend sein. In der resultierenden Test-Suite konnten 57 % der Tests erfolgreich ausgeführt werden und 73 % der Tests wurde bei einem späteren Review durch Entwickler in den produktiven Code integriert.

Quellen

[1] Ted Kurmaku et al., 2022, Human-based Test Design versus Automated Test Generation: A Literature Review and Meta-Analysis, ISEC 2022, Gandhinagar India. *https://dl.acm.org/doi/pdf/10.1145/3511430.3511433*

[2] Arkadii Sapozhnikov and Mitchell Olsthoorn and Annibale Panichella and Vladimir Kovalenko and Pouria Derakhshanfar, 2024, TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion, arXiv., *https://arxiv.org/abs/2401.06580*

[3] *https://github.com/assertj/assertj*

[4] Mohammed Latif Siddiq, 2024, An Empirical Study of Using Large Language Models for Unit Test Generation, *https://arxiv.org/html/2305.00418v3*

[5] Nadia Alshahwan et al., Automated Unit Test Improvement using Large Language Models at Meta, 2024, *https://arxiv.org/pdf/2402.09171.pdf*

Jan-Hendrik Telke

telke@neozo.de

Jan-Hendrik ist Principal Consultant bei der Neozo GmbH. Seine Erfahrung mit der Skalierung von rechenintensiven Optimierungsanwendungen setzt er heute bei der Implementierung und Planung von komplexen, verteilten Web-Systemen in verschiedenen Industrien wie Robotik, Logistik oder E-Commerce ein. Außerdem schreibt er regelmäßig in Fachmagazinen zu den Themen KI, Java und Code Qualität.

KI-Sprachmodelle im öffentlichen Sektor: Die Vorteile von Open Source für besondere Anforderungen

Eldar Sultanow & Thomas Heimann, Capgemini, Matthias Seßler, Bundesagentur für Arbeit

Wer kennt sie nicht, die großen Sprachmodelle (englisch: Large Language Model, kurz LMM) für Generative Künstliche Intelligenz? Es sind die Modelle, die den Chatbots wie ChatGPT, Gemini oder Neuroflash zugrunde liegen. Sie erleichtern Menschen das Leben, geben ihnen Antworten auf ihre Fragen, formulieren Texte und nehmen viel Arbeit ab. Und dies privat sowie beruflich – in unterschiedlichen Bereichen des Lebens und Branchen. Das Potenzial solcher Sprachmodelle ist auch im öffentlichen Sektor enorm, ihre Implementierung allerdings nicht trivial.

Der öffentliche Sektor erkennt das enorme Potenzial von LLMs an

Laut dem dbb Beamtenbund und Tarifunion fehlen dem Staat aktuell mindestens 551.500 Beschäftigte und in den nächsten 10 Jahren gehen 1,36 Millionen Beschäftigte in den Ruhestand *[1]*. Künstliche Intelligenz (KI) kann ein Instrument sein, um dem Fachkräftemangel entgegenzuwirken. Aus diesem Grund wird KI im öffentlichen Sektor beispielsweise auf Bundesebene, in Bayern und Baden-Württemberg verstärkt eingesetzt, um das bestehende Potenzial zu heben.

Beispielsweise hat die Bayerische Digitalagentur Byte in Partnerschaft mit Aleph Alpha ein Projekt initiiert, um administrative Prozesse zu optimieren und Behördensprache ins Englische zu übersetzen, was die internationale Kommunikation erleichtern soll. Baden-Württemberg hat ebenfalls die Initiative ergriffen und mit Aleph Alpha einen digitalen Assistenten namens F13 entwickelt, der die Textarbeit von Beamten erleichtern soll. Auf Bundesebene zeigt das Beispiel der Registermodernisierung, dass Verknüpfung und Digitalisierung verschiedener Register große Möglichkeiten für den effektiven Einsatz von KI-Anwendungen eröffnet, um administrative Prozesse zu unterstützen und zu automatisieren. Und es gibt weitere Beispiele.

Welche Rolle spielt Open-Source-KI für den öffentlichen Sektor?

Open-Source-LLMs bieten dabei Vorteile, die speziell auf die hohen Anforderungen an Datenschutz, Transparenz und Verantwortlichkeit im öffentlichen Sektor ausgerichtet sind. Sie können sicherheitstechnisch abgeschottet und feingranular konfiguriert werden – sogar so, dass die Chatbots für bestimmte Fälle oder bei bestimmten Personen die gestellten Fragen und gelieferten Antworten aus Compliance-Gründen „vergessen".

Warum das Ganze? Im öffentlichen Sektor ist die Datenarchitektur komplex, sie reflektiert die geltenden Gesetze und Bestimmungen mitsamt Sicherheitsanforderungen. So unterliegen etwa Daten im SGB-Umfeld unterschiedlichen Rechtskreisen mit unterschiedlichen Schutzklassen – und die dürfen nicht einfach vermischt werden. Open-Source-LLMs bieten die dafür benötigte Granularität der Sprachmodell-Anpassung und Konfiguration. Das gilt auch für LLMs, die zwar nicht Open-Source-lizensiert sind, aber bei denen der Hersteller seinen Kunden auch für einen On-Premise-Betrieb die Lizenz übergibt und ihnen den Sourcecode unter bestimmten Voraussetzungen zugänglich macht. Diese LLMs bieten sogar die Möglichkeit zur Schaffung eines hochsicheren europaweiten KI-Sprachmodell-Hubs.

Vier Gründe für Open-Source-KI im öffentlichen Sektor

Erstens haben Entscheidungen im öffentlichen Sektor weitreichende Folgen für die Gesellschaft:

Gerade im öffentlichen Sektor müssen Entscheidungen transparent und Informationen verlässlich sein. Die Transparenz und Nachvollziehbarkeit von KI-basierten Entscheidungen sind immer noch eine Herausforderung und Forschungsgegenstand in der Entwicklung von Künstlicher Intelligenz. Aleph Alpha strebt mit seinem Sprachmodell *Luminous* an, eine größere Nachvollziehbarkeit in die Entscheidungsfindung der KI zu bringen. Dieses Modell generiert nicht nur Texte und beantwortet Fragen, sondern legt auch die Herkunft seiner Informationen offen, um Menschen die Möglichkeit zu geben, irreführende oder inkorrekte Ausgaben leichter zu erkennen.

Zweitens rücken hohe Datenschutzanforderungen Open Source-Überlegungen in den Fokus:

Durch den offenen Zugang zum Quellcode der Open-Source-LLMs können Behörden und die Öffentlichkeit die Funktionsweise und Entscheidungsprozesse besser nachvollziehen. Natürlich ist Zugang zu den Quellen nur ein Transparenzbaustein. Ein wichtiger Baustein sind auch Konzepte für sogenannte *Trustworthy AI*, die durch technische Modelltrainingsbeobachtung sicherstellen, dass Modelle keine Vorurteile und negative Stereotypen kodieren, sich also nicht „negativ" entwickeln, sodass die KI vertrauenswürdig ist und bleibt. Transparenz ist entscheidend für das Vertrauen in KI-Systeme, insbesondere wenn sie in sensiblen Bereichen wie der öffentlichen Verwaltung eingesetzt werden, wo Entscheidungen tiefgreifende Auswirkungen auf Bürger haben können. Öffentliche Institutionen können die Modelle an spezifische Anforderungen und Richtlinien, etwa zur Einhaltung von Datenschutzgesetzen wie der DSGVO, anpassen. Das macht überall dort Sinn, wo Datenschutz eine entscheidende Rolle spielt.

Drittens bietet Open Source eine kosteneffizientere Lösung:

Für Open-Source-LLMs fallen keine Lizenzgebühren an. Das fördert eine breitere Akzeptanz von KI-Technologien im öffentlichen Sektor, selbst in Kommunen oder Abteilungen mit strenger geregelten und mit begrenzten Budgets. Dies bezieht sich aber nur auf die Lizenzierung und nicht auf den Betrieb der Software. Die Anpassungsfähigkeit von Open-Source-Software erlaubt die Umsetzung spezifischer Anforderungen und lokaler Bedingungen ohne den Kauf maßgeschneiderter Zusatzlösungen. Das führt zu einer besseren Abstimmung der Technologie auf den Bedarf und die Herausforderungen des öffentlichen Sektors.

Viertens fördert der Open-Source-Gedanke die Innovation und Zusammenarbeit am Code:

Durch die gemeinsame Nutzung und Weiterentwicklung von Open-Source-Projekten profitieren öffentliche Einrichtungen von den Erfahrungen, Best Practices und Entwicklungen anderer und können eigene

Innovationen beisteuern. Die Zusammenarbeit von Behörden und der Open-Source-Gemeinschaft hätte nicht nur das Potenzial, die Qualität und Sicherheit der Lösung zu verbessern, sondern auch die Umsetzung maßgeschneiderter Module zu beschleunigen – vom Bug-Fix bis hin zur Umsetzung neuer Feature-Requests. Darüber hinaus schaffen Partnerschaften zwischen öffentlichen und akademischen Einrichtungen Zugang zu technischem Support und zu den neuesten Forschungserkenntnissen. Dies überbrückt die Lücke zwischen der sich schnell entwickelnden Technologie und der praktischen Anwendung von Open-Source-LLMs in der öffentlichen Verwaltung. So entstehen ein Expertennetzwerk und Nährboden für innovative öffentliche Dienste.

Die Umsetzung ist alles andere als einfach, die Zielarchitektur hochanspruchsvoll

Jede Medaille hat zwei Seiten. Der Einsatz von Open-Source-LLMs bringt hohe Infrastrukturvoraussetzungen und Anforderungen mit sich, insbesondere mit Blick auf die Wartung und Aktualisierung der Modelle. Öffentliche Einrichtungen müssen sicherstellen, dass sie über die erforderlichen Ressourcen und Kompetenzen verfügen, um diese Modelle effektiv zu nutzen und weiterzuentwickeln.

Die Industrie demonstriert, wie eine technische Umsetzung großer und skalierender Lösungen mit Open Source aussehen kann. Einige Unternehmen integrieren das quelloffene multimodale Modell OpenFlamingo, das Open-Source-LLM Mixtral oder Llama 3. Um eine nahtlose Integration und fortlaufende Pflege dieser Modelle zu gewährleisten, ist es von entscheidender Bedeutung, dass der öffentliche Sektor entsprechendes Fachpersonal und technische Infrastrukturen aufbaut. Neben den initialen Einrichtungskosten entstehen auch laufende Kosten für die Personalschulung und regelmäßige Systemaktualisierung.

Abbildung 1 zeigt das schematische Vorgehen *[2]* für den Einsatz domänenspezifischer LLMs im abgeschotteten Netzwerk der Organisation. Ein vortrainiertes Sprachmodell wird von „draußen" in das sichere Netzwerk der Organisation „reingeholt", dann einem Fine-Tuning mit domänenspezifischen (auch internen) Dokumenten unterzogen und zusätzlich mit Dokumenten augmentiert. Alternativ ließe sich auch ein komplett domänenspezifisches Modell aufbauen, indem es von der Pike auf mit internen und domänenspezifischen externen Daten (wie zum Beispiel die öffentlichen Weisungen der Bundesagentur für Arbeit) trainiert wird.

Drei Beispiele für Behörden-übergreifenden Datenbezug aus dem SGB-Bereich

Die Daten im SGB-Umfeld liegen in unterschiedlichen Rechtskreisen. Und genau das muss die Zielarchitektur abbilden. Es gibt Daten, welche besonders schutzbedürftig sind, – das sind zum Beispiel Daten im Bereich SGB II wie Sozialversicherungsdaten. Und es gibt Fragen, deren Antwort die Informationen und Koordination zwischen mindestens zwei verschiedenen Behörden erfordert. Ein Bürger könnte beispielsweise fragen:

> *„Wie kann ich als neu zugezogener Elternteil feststellen, ob ich für das Elterngeld berechtigt bin und gleichzeitig überprüfen, ob mein Kind bereits in der neuen Gemeinde für einen Kindergartenplatz angemeldet ist?"*

Das Jugendamt oder die für das Elterngeld zuständige Stelle müsste Informationen über die Berechtigung und das Antragsverfahren für das Elterngeld bereitstellen, was oft von Faktoren wie Einkommen, Erwerbstätigkeit und Wohnsitz abhängt. Das Einwohnermeldeamt oder die zuständige kommunale Stelle für Kindertagesstätten müsste überprüfen, ob das Kind im System für einen Kindergartenplatz in der neuen Gemeinde angemeldet ist, was wiederum eine gültige Wohnsitzanmeldung voraussetzt.

Etwas komplexer ist die folgende Frage:
Die Gewerbebehörde in Gemeinde B müsste bestätigen, dass das

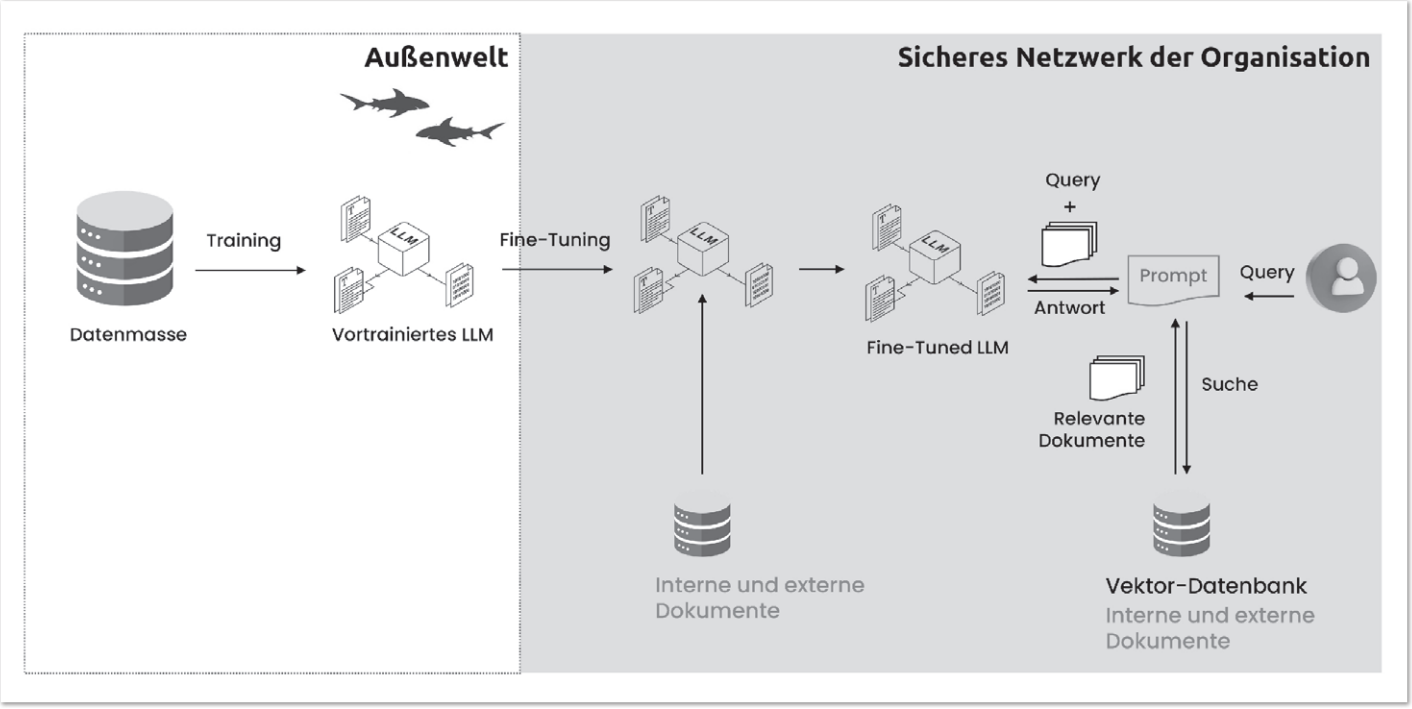


Abbildung 1: LLMs für schutzbedürftige Daten liegen im abgeschotteten Netzwerk der Organisation. (© Eldar Sultanow)

„Ich habe vor kurzem mein Gewerbe in Gemeinde A abgemeldet und in Gemeinde B neu angemeldet. Wie kann ich sicherstellen, dass mein Gewerbe nun von den lokalen Steuerbehörden in Gemeinde B erfasst wird und ich gleichzeitig für die lokale Wirtschaftsförderung in Frage komme?“

Gewerbe ordnungsgemäß angemeldet wurde und alle formalen Kriterien erfüllt sind. Das Finanzamt oder die lokale Steuerbehörde in Gemeinde B muss das Gewerbe für die lokale Gewerbesteuer erfassen und gegebenenfalls eine neue Steuernummer zuweisen. Die Wirtschaftsförderung in Gemeinde B müsste überprüfen, ob das neu angemeldete Gewerbe für lokale Förderprogramme oder Unterstützungsmaßnahmen in Frage kommt, was wiederum von der Art des Gewerbes, der Größe und anderen Faktoren abhängt.

Verzwickter wird es, wenn die Frage Informationen und Koordination zwischen zwei Behörden mit unterschiedlichen Datenschutzklassen erfordert:
Die Beantwortung dieser Frage benötigt Informationen der Bun-

„Wie wirkt sich mein Umzug von Gemeinde A nach Gemeinde B auf meine Leistungen nach dem SGB II (Grundsicherung für Arbeitsuchende) aus und welche Schritte muss ich unternehmen, um meine Gesundheitsversorgung in der neuen Gemeinde sicherzustellen?“

desagentur für Arbeit und der lokalen Gesundheitsbehörden. Für Informationen und Leistungen nach dem SGB II ist das Jobcenter zuständig. Hier geht es um sozialrechtliche Aspekte, die unter das Sozialgesetzbuch fallen und einen hohen Datenschutzstandard erfordern. Wie wirkt sich der Umzug des Bürgers auf seine beanspruchten Leistungen aus? Muss er sich bei einem neuen Jobcenter anmelden? Welche Unterlagen braucht er? Für die Gesundheitsversorgung in der neuen Gemeinde sind die lokalen Behörden, das kommunale Einwohnermeldeamt und Gesundheitsamt zuständig. Dabei könnte es um die Ummeldung des Wohnsitzes und die Information über die lokalen Gesundheitseinrichtungen sowie die Anmeldung bei einem neuen Hausarzt gehen. Diese Daten fallen unter eine andere Schutzklasse, da sie teils personenbezogene Daten umfassen, die nicht direkt sozialrechtlichen Regelungen unterliegen.

Multi-Domänen-Sicherheit für Sprachmodelle

Ein Ansatz ist die Einführung von Multi-Domänen-Sicherheit (englisch: Multi Domain Security), angewendet auf die unterschiedlichen Rechtskreise und Schutzklassen der Daten. Das heißt, für einem Rechtskreis zugeordnete Datenbestände gibt es jeweils ein Sprachmodell, das innerhalb einer entsprechenden Sicherheitsdomäne liegt. Die Domänen sind durch einzelne Netzwerke strikt segmentiert. Diese Netzwerke agieren als isolierte Umgebungen, in denen Sprachmodelle operieren (etwa nach dem Multi-Agenten-Ansatz untereinander kommunizieren) können, ohne dass Daten unautorisiert zwischen den Domänen übertragen werden. Dies verhindert, dass sensible Informationen aus hochsicheren Bereichen in weniger sichere Domänen gelangen oder gar in die Öffentlichkeit. Darüber hinaus benötigen wir Zugriffskontrollmechanismen, die sicherstellen, dass nur autorisiertes Personal auf bestimmte Daten zugreifen kann. Rollenbasierte, dynamische und je nach Domäne anpassbare Zugriffsrechte gewährleisten, dass Mitarbeiter nur die Informationen abfragen, die für ihre spezifische Aufgabe notwendig sind. Die Zugriffe innerhalb der segmentierten Sprachmodell-Domänen müssen überwacht werden, um sicherzustellen, dass die Sicherheitsprotokolle eingehalten werden und die Integrität der abgefragten und verarbeiteten Daten gewahrt bleibt.

Brauchen wir ein deutschlandweites oder sogar europaweites KI-Sprachmodell-Hub?

Zu gewährleisten, dass KI-Anwendungen die Schutzbedürftigkeit von Daten entsprechend geltenden Regeln einhalten, ist nicht trivial. Eine Idee könnte ein deutschlandweites oder europaweites KI-Sprachmodell-Hub sein. Die zentrale Koordination solcher Sprachmodelle würde die Standardisierung der KI-Anwendungen über

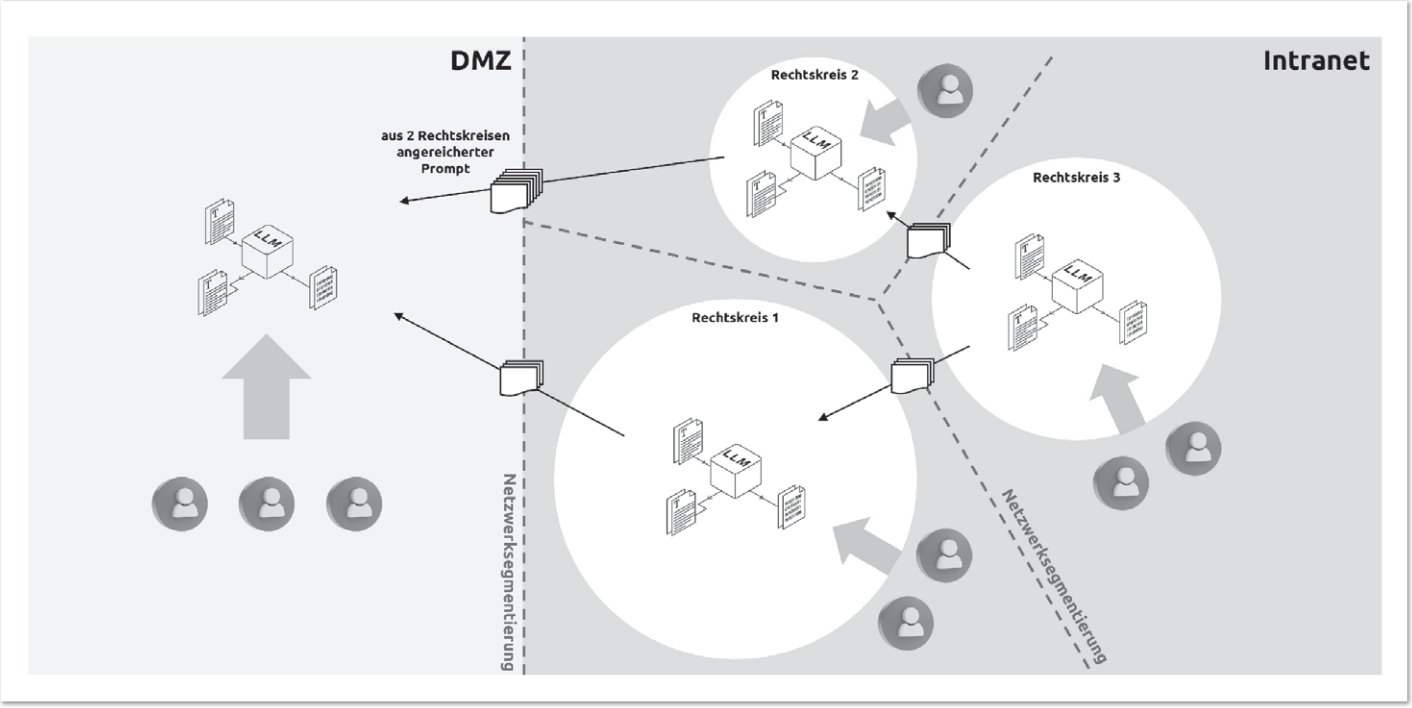


Abbildung 2: Verschieden mächtige Sprachmodelle (Multi-Agenten-Ansatz) in unterschiedlichen Rechtskreisen (© Eldar Sultanow)

verschiedene Behörden und Regionen hinweg fördern. Zu erwarten wäre davon nicht nur eine höhere Effizienz in der öffentlichen Verwaltung, sondern auch eine größere Qualität und Einheitlichkeit der Dienste innerhalb der Behörden und der Bürgerdienste.

Ein zentraler, von der öffentlichen Verwaltung bereitgestellter Hub könnte die Konformität von behördlichen KI-Anwendungen gegenüber den gesetzlichen Rahmenbedingungen des jeweiligen Landes oder der jeweiligen Region gewährleisten. Die zentrale Bereitstellung und Verwaltung der Sprachmodelle bündelt Ressourcen und senkt Kosten, da nicht jedes Verfahren – oder größer gedacht jede Behörde – eigene, individuelle Lösungen entwickeln und auf diese warten muss. Der Hub erlaubt eine effektive Kontrolle und das Management von Zugriffsrechten auf die Daten. Darüber hinaus könnte ein solcher Hub als Hyperscaler für Rechenleistung (über die nicht jede Behörde einfach so verfügt) und als Innovationszentrum dienen.

Ein deutschlandweiter oder europaweiter KI-Sprachmodell-Hub bietet eine strategische Plattform für die Modernisierung des öffentlichen Sektors, stärkt die digitale Souveränität Europas in einer global vernetzten Welt und liefert harmonisierte Lösungen für die sich ständig wandelnden und rasant wachsenden rechtlichen und gesellschaftlichen Anforderungen.

Fazit und Ausblick

Was die Entwicklung, Einführung und das Ausschöpfen der Potenziale von Sprachmodellen angeht, liegen wir wahrscheinlich erst bei einem Prozent. LLMs im öffentlichen Sektor können dazu beitragen, dem Fachkräftemangel zu begegnen, die Effizienz steigern und die Zugänglichkeit öffentlicher Dienste für Bürgerinnen und Bürger verbessern. Open Source kann eine valide Antwort auf die hohen Datenschutz- und Transparenz-Anforderungen des öffentlichen Sektors sein. Allerdings verlangt die strikte Trennung der Daten aus verschiedenen Rechtskreisen eine ausgeklügelte Zielarchitektur und Multi-Domänen-Sicherheit, die isolierte Umgebungen für die Verarbeitung unterschiedlich sensibler und schutzbedürftiger Sprachmodelle schafft.

Der gemeinsame Einsatz und die Weiterentwicklung von Open-Source-Sprachmodellen können ein nachhaltiges Netzwerk schaffen, das innovative Lösungen für spezifische Anforderungen des öffentlichen Sektors hervorbringt.

Quellen

[1] dbb beamtenbund und tarifunion (2023). Dem Staat fehlen über 500.000 Beschäftigte. *https://www.dbb.de/artikel/dem-staat-fehlen-ueber-500000-beschaeftigte.html*

[2] Najeeb Nabwani (2023). Full Fine-Tuning, PEFT, Prompt Engineering, and RAG: Which One Is Right for You? *https://deci.ai/blog/fine-tuning-peft-prompt-engineering-and-rag-which-one-is-right-for-you/*

Eldar Sultanow

eldar.sultanow@capgemini.com

Eldar Sultanow ist CTO des Technologiebereichs Insights & Data bei Capgemini in Deutschland, einer der weltweit führenden Beratungsgesellschaften für digitale Transformation.

Thomas Heimann

thomas.heimann@capgemini.com

Als ausgebildeter Informatiker arbeitet Thomas Heimann seit mehr als 20 Jahren für Capgemini in verschiedenen Rollen. Aktuell ist er Enterprise Architect mit Fokus auf den öffentlichen Sektor und berät zu digitalen Strategien.

Matthias Seßler

Matthias.Sessler@arbeitsagentur.de

Matthias Seßler ist im IT-Systemhaus der Bundesagentur für Arbeit der FT-V (fachlich-technischer Verantwortlicher) für Datenbanktechnologien und -services. Davor war er 16 Jahre bei Microsoft im Bereich Data Insights tätig.

10 JAHRE JAVALAND

JavaLand

www.javaland.eu

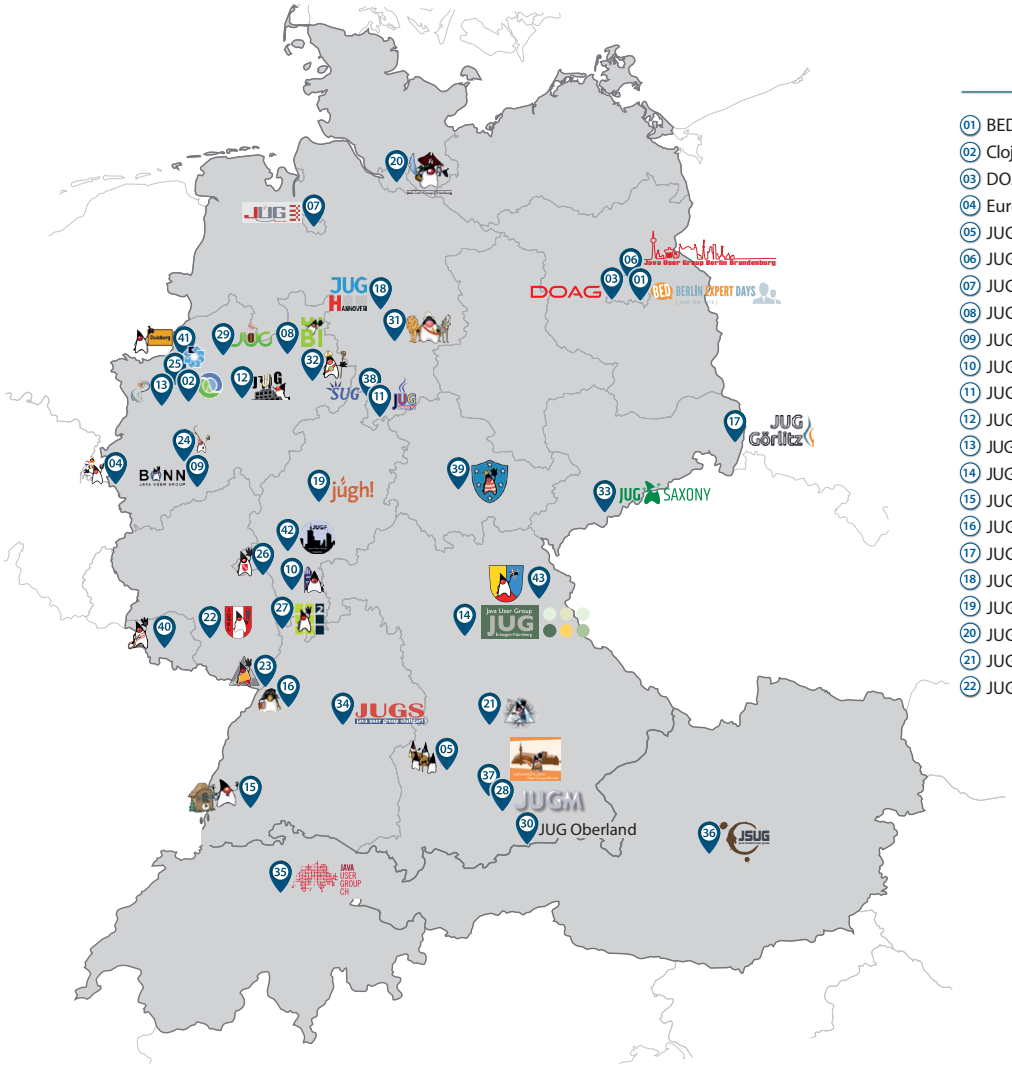
JAVALAND 2024 VERPASST?

ALLE ON-DEMAND-ANGEBOTE IM TICKETSHOP

JETZT ON-DEMAND-TICKET BUCHEN UND
VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

Veranstalter: JavaLand



Mitglieder des iJUG

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

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.

DOAG e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. DOAG e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. DOAG 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: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz

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

Bildnachweis:
Titel: Bild © Designed by freepik
https://freepik.com
S. 10 + 11: Sloth McSloth
https://stock.adobe.com
S. 13: Bild © DOAG
https://doag.org
S. 15: Bilder © DOAG
https://doag.org
S. 16 + 17: Bild © Designed by teravector
https://freepik.com
S. 24 + 25: Bild © Designed by freepik
https://freepik.com
S. 30 + 31: Bild © Designed by freepik
https://freepik.com
S. 38: Bild © Designed by gstudioimagen
https://freepik.com
S. 42 + 43: Bild © Designed by sentavio
https://freepik.com
S. 52 + 53: Bild © Designed by freepik
https://freepik.com
S. 58 + 59: Bild © Designed by freepik
https://freepik.com
S. 70 + 71: Bild © Ash
https://stock.adobe.com
S. 76 + 77: Bild © The Little Hut
https://stock.adobe.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org
Mediadaten und Preise:
www.doag.org/go/mediadaten

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

Inserentenverzeichnis

DIE DOAG ANWENDERKONFERENZ

ANWENDERKONFERENZ.DOAG.ORG

Save the Date

19. BIS 22. NOVEMBER

IN NÜRNBERG

2024

DOAG

Konferenz + Ausstellung