Ausgabe 02/2022

Java aktuell

iJUG Verbund
www.ijug.eu

Temurin 17
Interview mit Hendrik Ebbers

Architektur
Clean Architecture, Rule Engines, CQRS/ES und mehr

Softwareentwicklung
Besserer Code durch Zufriedenheit

Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 02

29. JUNI BIS 2. JULI 2022

CLOUD NATIVE FESTIVAL – 4 TAGE UND 4 THEMEN

im Phantasialand in Brühl

CloudLand
www.cloudland.org

Liebe Leser/innen der Java aktuell,

nach dem Java-Tagebuch und der Eclipse Corner beginnt diese Ausgabe mit einem Interview mit Hendrik Ebbers, der als Projektleiter von Eclipse Adoptium das Temurin-Projekt begleitet hat. Er berichtet zur Zusammenarbeit in der Eclipse Foundation und den Produkten AQAvit und Temurin.

Darauf folgt direkt das Schwerpunktthema dieser Ausgabe: Architektur. Dazu zeigt uns Matthias Eschhold in seiner Artikelreihe, wie wir durch die Verwendung von Clean Code für eine flexible Anwendungsarchitektur sorgen. Der erste Teil beschäftigt sich mit der Anwendung dieses Konzepts auf die Paket- und Klassenebene innerhalb einer Anwendung. Was Rule Engines sind und was es damit auf sich hat, verrät uns Merlin Bögershausen ab Seite 22. Dabei demonstriert er, wie diese auf System- und Softwarearchitekturebene genutzt werden können. Wie mithilfe von Event Modeling und der Verwendung des Axon-Frameworks eine CQRS/ES-basierte Architektur umgesetzt werden kann, zeigen uns Frank Neugebauer und Nikolai Steimle. Die beiden Autoren beleuchten außerdem die Vorteile dieser Umsetzung.

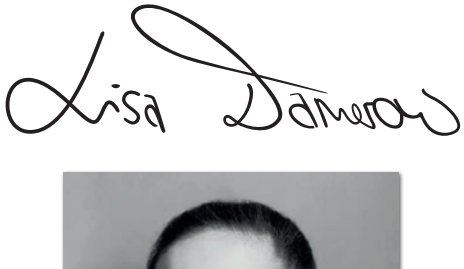
Wartbare und zukunftsfähige Software-Architekturen in Java – geht das? Und ob! Wie dies mit Annotation-Processing, Spring Boot und Cloud-Technologien umgesetzt werden kann, veranschaulicht Klaus Sausen ab Seite 33. Um eventbasierte Schnittstellen mit Amazon EventBridge und Spring geht es im darauffolgenden Beitrag von Jan Sauer und Simon Jakubowski. Das Criteria-API ermöglicht den Zugriff auf Datenbanken auch für komplexe Anfragen, erhöht gleichzeitig aber die Komplexität des eigenen Codes. Nicolai Mainiero vergleicht verschiedene Alternativen mit der Verwendung des Criteria-API.

In „A Tale of Fail" erläutert Bastian Glöckle anhand einer beispielhaften Geschichte die Herausforderungen, die bei der Entwicklung nachhaltig erfolgreicher Cloud-Produkte auftreten können und wie man diese überwinden kann. Der vielen Entwickler/innen bekannte Grundsatz „Write once, use everywhere" ist in der Praxis häufig nicht leicht umsetzbar. Elmar Dott nimmt uns mit auf die Suche nach Lösungen für die Wiederverwendung von Anwendungen im richtigen Leben. Tipps und Tricks für effizientere Softwareentwicklung unter anderem im Hinblick auf Kopplung beleuchtet Ingmar Kellner ab Seite 58.

Zum Abschluss dieser Ausgabe erläutert und begründet Thorsten Diekhof seine Hypothese, dass nicht das Schreiben von gutem Code Entwickler/innen zufrieden macht, sondern dass zufriedene Entwickler/innen besseren Code schreiben. Einige Ansätze zur Verbesserung der eigenen Zufriedenheit präsentiert er in seinem Artikel ab Seite 64.

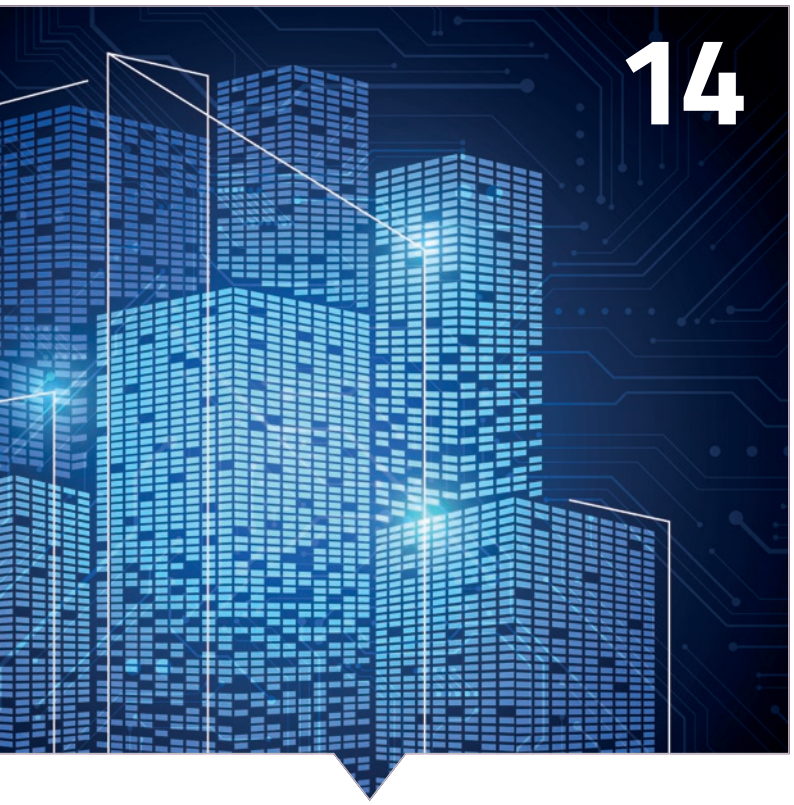
Wir wünschen euch viel Spaß beim Lesen!

Eure



Lisa Damerow
Redaktionsleitung Java aktuell

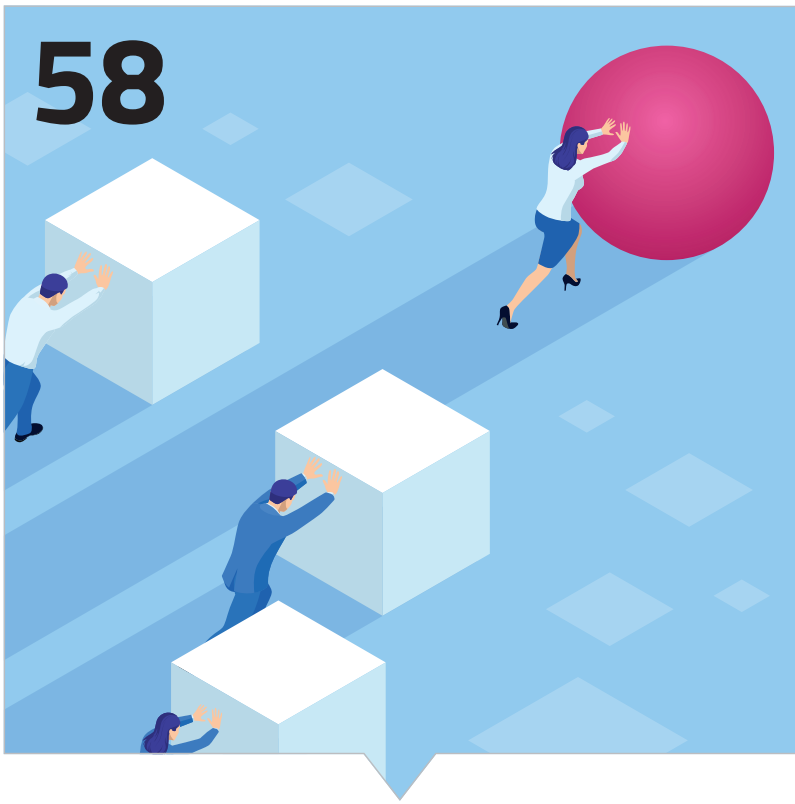
Inhalt



Teil 1: Die Grundlagen für Clean Architecture

Rule Engines: Einführung und Überblick

Nachhaltig erfolgreiche Cloud-Produkte entwickeln



Minimieren und Kontrollieren von Kopplungen für effizientere Software

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

19. Oktober 2021

Graal VM 21.3 mit Java 17

GraalVM unterstützt in seiner neuen Version 21.3 jetzt auch Java 17. Es enthält einige neue Optimierungsfeatures, zum Beispiel die „Infeasible Path Correlation Optimization", bei der anhand von, nun, korrelierenden Bedingungen, die im Code verteilt sein können, genau hingeschaut wird, ob bestimmte Pfade überhaupt erreichbar sind. Oder die „Strip Mining"-Optimierung für „non-counted loops", die wohl häufiger vorkommen, als ich erwartet hätte. Bei entsprechenden Workloads soll die Option bis zu 15 % Geschwindigkeit herausholen. Einen Eindruck, wie viel Theorie hinter so einem Feature steckt, vermittelt vielleicht dieses Paper von 2018 dazu *[1]*.

22. Oktober 2021

Vermisst: Java-8-Videos

Es ist Ende 2021, und noch immer sind da draußen viele Migrationsunwillige unterwegs. Auch der abnehmende Support kann die Skeptiker nicht davon überzeugen, Java 9 oder höher auf ihre Rechner zu lassen. Jetzt haben sich scheinbar finstere Mächte neue Tricks ausgedacht, um den Druck zu erhöhen: Von YouTube sind die Videos des „Java 8 Massive Online Open Course" verschwunden. Bevor die ersten Verschwörungsmythen entstehen, erreicht aber eine Anfrage das JUG-Leader-Forum unter jugs.groups.io – und der Sachverhalt wird schnell geklärt: YouTube hat viele alte Videos, die vor 2017 hochgeladen wurden, als „privat" markiert, sodass sie nicht mehr sichtbar sind – das betrifft nicht nur Java-Videos.

Problem: Ohne Zugriff auf die ursprünglichen Accounts lassen sie sich nicht wieder sichtbar machen. Simon Ritter von Azul hat die Java-8-Videos in seiner Zeit bei Oracle selbst aufgenommen und besitzt noch Kopien. Ohne Zustimmung von Oracle kann er sie aber nicht neu hochladen. Hoffentlich kann das dort jemand direkt entscheiden; wenn die Rechtsabteilung ins Spiel kommt, könnte es mal wieder länger dauern.

3. November 2021

JavaLand 2022: Programm online

Das Programm für die JavaLand 2022 steht fest, mit mehr als 130 Vorträgen plus Community-Aktivitäten. Am 15. und 16. März – und Schulungstag am 17. – soll die Konferenz wieder vor Ort im Phantasialand durchgeführt werden – so es die Umstände zulassen. Bis Ende Januar soll eine konkrete Einschätzung vorliegen, sodass die Veranstaltung gegebenenfalls in ein Online-Format umgewandelt werden kann. Hoffen wir, dass das nicht nötig ist. So sehr sich diese Formate in den letzten Jahren verbessert haben: Ein vollwertiger Ersatz wird es erst werden, wenn wir auch den Quatsch mit der VR-Brille und Ähnlichem hinter uns gelassen haben und nur noch Gehirne in einem Tank sind. Oder ist das schon passiert?

4. November 2021

Microsoft wird Mitglied im JCP

Microsoft umarmt Java noch ein wenig fester (Entschuldigung für mein schlechtes Englisch) und wird jetzt auch Mitglied im Java Community Process. Der hat zwar für Enterprise Java seine Bedeutung verloren (wo Microsoft aber bereits innerhalb der Eclipse Foundation mitarbeitet) – ist aber für die Sprache und die SE-Plattform weiterhin das zentrale Gremium, das über die Spezifikationen wacht *[2]*.

10. November 2021

Neues Projekt, alter Name: Jakarta Commons

Das Jakarta-EE-Plattform-Projekt will ein neues „Commons"-Projekt starten. API-Teile inklusive Annotations sowie Utility-Klassen, die grundsätzlich generisch und über verschiedene Einzel-Spezifikationen hinweg nutzbar sind, sollen in ein neues „Jakarta Commons" aufgenommen werden können. Den älteren Java-Entwicklerinnen und -Entwicklern wird dieser Name noch bekannt vorkommen. Das alte Apache-Tools-Projekt ist aber bereits 2011 archiviert worden und die Apache Software Foundation hat der Namensübernahme durch Eclipse zugestimmt.

24. November 2021

MicroProfile: Ziele für 2022

Die MicroProfile Working Group hat die Ziele für das nächste Jahr beschlossen. Zentraler Bestandteil sind drei MP-Releases und eine Fortsetzung der Zusammenarbeit mit Jakarta EE in der „CN4J Alliance". Aus Marketingsicht soll unter anderem ein „MicroProfile Champions"-Programm gestartet werden, das Individuen belohnt, die sich um MicroProfile verdient gemacht haben (ob durch Code oder Beiträge anderer Art). Aus technischer Sicht soll außerdem das OpenTracing-API überarbeitet werden, hin zu einer Unterstützung von OpenTelemetry. Auch der Bedarf an einer eigenen Spezifikation für gRPC soll zumindest geprüft werden, während für existierende „Stand-alone"-Spezifikationen ein Umzug anstehen könnte (beispielsweise GraphQL in die MicroProfile-Plattform, Context-Propagation ins Jakarta-Projekt).

30. November 2021

ML mit Eclipse Gran Sasso

Maschinelles Lernen hält Einzug in Eclipse EE4J – und damit zukünftig vielleicht auch als API-Spezifikation in Jakarta EE: Mit einem Pilotprojekt unter dem Namen Eclipse Gran Sasso (der ursprüngliche Arbeitstitel war wohl Eclipse Morpheus) sollen existierende ML-Werkzeuge und APIs im Kontext von Applikationsservern integriert werden, die verlässliche Vorhersagen über Ressourcen-Nutzung beziehungsweise -Allokation und die Kosten in Bezug auf bestimmte Lastszenarien ermöglichen. Als Grundlage dienen dafür unter anderem Ergebnisse aus dem 2020 abgeschlossenen JSR-381 („Visual Recognition Specification"), den die Initiatoren von Gran Sasso geleitet haben. Im nächsten Schritt könnte dann daraus ein Standard-API für Jakarta EE entstehen. Ob der Name (ein italienisches Gebirgsmassiv) etwas mit den erwarteten Herausforderungen zu tun hat, ist leider nicht überliefert.

7. Dezember 2021

MicroProfile 5.0

MicroProfile 5.0 sieht oberflächlich betrachtet mal nach tiefgreifenden Änderungen in allen Teilen des „Umbrella" aus. Jede Einzelspezifikation ist auf ein neues Major-Release gehoben worden, auch die fünf von Jakarta übernommenen (von CDI 3.0 bis Annotations 2.0). Nur bei den Stand-alone-Spezifikationen ist nichts passiert – mit Ausnahme von „Context Propagation" (jetzt 1.3). Geschuldet ist das einer nicht abwärtskompatiblen Anpassung: Alle APIs sind jetzt kompatibel mit Jakarta EE 9.1, dafür war es nun mal erforderlich, alle Abhängigkeiten von den alten java.x-Packages durch Jakarta-Packages zu ersetzen.

Das war es dann auch schon mit den wesentlichen Änderungen – 5.0 ist nur die Grundlage für zukünftige Releases, die es ab jetzt dreimal im Jahr geben soll (im Februar, Juni und Oktober). Die von der MP Working Group Charter für ein Release vorgeschriebene kompatible Implementierung ist Open Liberty (22.0.0.1-beta). Unter anderem das WildFly-Projekt steht auch kurz vor einem kompatiblen Release.

11. Dezember 2021

Log4j macht teure Anrufe

Aufregung zu Beginn des Adventswochenendes. Sicherheitswarnungen zu CVE-2021-44228 machen die Runde – ein „Exploit" in log4j – der mithin einen großen Teil aller Java-Applikationen weltweit betrifft. Es werden bereits (erfolgreiche) Versuche von Hackern beobachtet, die Lücke zu nutzen. Log4j ist ein ziemlich mächtiges Framework und kann mit seinen „Lookup Plug-ins" Log-Informationen dynamisch ergänzen – inklusive JNDI-Lookups beim Formatieren eines Log-Statements mit Parametern. Das Feature, denn das ist es tatsächlich, ist bereits seit 2013 in log4j-core enthalten, aber trotz Open Source erst jetzt aufgefallen (oder ist es naiv, das zu glauben?).

Im Jahr 2013 war vielleicht das Gespür für Sicherheitslücken noch nicht so ausgeprägt, aber der Nutzer-Wunsch nach einfacher Anreicherung des Logs mit Metadaten der Runtimes schnell erfüllt. Seine volle Wirkung entfaltet das Feature dank der teilweise absurden Remote-Code-Execution-Fähigkeiten von Java. Habe ich auch schon einmal genutzt – ungefähr 2001. Hand hoch – wer braucht das wirklich? Kann man das nicht per Default erst mal komplett abklemmen? Neuere Java-Versionen schließen die Lücken zwar bereits teilweise (etwa Java 8 Update 121 für RMI und 191 für LDAP), aber eben nur teilweise. Irgendjemand, der Tomcat nutzt? Dann steht hier Interessantes zu seiner BeanFactory *[3]*.

Die Updates für log4j kamen zwar sehr schnell (Plural, da Schnelligkeit und Gründlichkeit keine guten Freunde sind), aber wer kann schon sagen, wie viele ähnliche Lücken noch (hoffentlich) unentdeckt sind? Bis dahin: Den Apps verbieten, dass sie unbeschränkt nach draußen telefonieren dürfen. Neben Netzwerk-/Firewall- /Egress-Konfigurationen und Ähnlichem könnte das auch der SecurityManager in Java übernehmen. Der soll allerdings bald verschwinden. Angeblich ist er für den Betrieb als „Insellösung" zu umständlich und wird daher kaum genutzt. Wer den SecurityManager (weiter) nutzen möchte, sollte sich daher schleunigst an das OpenJDK-Projekt wenden – oder an den iJUG.

15. Dezember 2021

Jakarta EE 10, SE 11 und Records

Im Jakarta-EE-Platform-Project wird in den wöchentlichen Calls regelmäßig der Status des kommenden EE 10 Release unter die Lupe genommen. Bislang war es für Q1/2022 geplant, was erfordern würde, dass alle Einzelspezifikationen, die ins Release wollen, zügig fertig werden. Es sieht so aus, als ob sich das Release ins zweite Quartal verschieben wird.

EE 10 soll nach den Vorbereitungen mit 9 und 9.1 das erste Release unter dem Namen Jakarta mit „wirklichen" neuen Features werden. Das Plattform-Release enthält das neue „Core Profile", das für Entwicklung und Betrieb von Microservices gedacht ist (ein bisschen mehr als die Schnittmenge mit dem MicroProfile). Ebenso wie 9.1 setzt EE 10 auf Java SE 11 für die APIs (Source und Binaries) – Ja-

karta-kompatible Implementierungen dürfen allerdings auch höhere Runtime-Versionen fordern. Die Diskussion „11 oder 17" wird mit Jakarta EE 11 wiederkommen. Zuletzt lief die Diskussion ungefähr darauf hinaus: „Das aus EE-Sicht einzig wirklich wichtige Feature zwischen Java SE 11 und 17 sind Records – macht es das wirklich den Aufwand wert, alle zur Migration zu zwingen?" Zu Records gibt es eh noch eine eigene Diskussion, da ihre Nutzung mit den Anforderungen von JPA nach „No-Arg"-Konstruktoren bricht. Oder sagen wir: „Bisherigen Anforderungen…" Wenn es genügend Leute nervt, gibt es ja vielleicht in näherer Zukunft eine Lösung dafür *[4]*.

17. Dezember 2021

The Hottest JIT

„Cloud Native Compiler" ist Teil von Azuls kommerzieller OpenJDK-Variante „Platform Prime" und soll die Just-in-Time-Kompilierung in der JVM auf das nächste Level bringen. Dazu wird der Default-JIT-Compiler C2 des OpenJDK durch den Falcon Compiler ausgetauscht und als geteilter Optimierungsservice auf Kubernetes betrieben (im selben Cluster wie die zu optimierenden JVMs oder zumindest von diesen aus mit möglichst wenig Latenz erreichbar). Dadurch werden dann nicht nur die JVMs gerade während ihrer Startphase entlastet, sondern die aus potenziell vielen einzelnen JVMs zusammengetragenen Informationen sollen zu einer deutlich besseren Optimierung führen, als wenn jede JVM die Optimierung für sich macht. Zumindest für Evaluierungszwecke lässt sich das Tool kostenlos herunterladen und in beliebigen Clouds oder lokal auf Minikube betreiben.

Referenzen

[1] *https://www.davidleopoldseder.com/publications/manlang2018-fast_path_unrolling_authorpreprint.pdf*
[2] *https://devblogs.microsoft.com/java/microsoft-deepens-its-investments-in-java/*
[3] *https://devcentral.f5.com/s/articles/Mitigating-New-Gadget-Leveraging-JNDI-Injection-into-Remote-Code-Execution-Using-Advanced-WAF*
[4] *https://www.eclipse.org/lists/jakarta.ee-community/msg02852.html*

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

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

Markus' eclipse-Corner

Ich freue mich sehr, an dieser Stelle den ersten Stipendiaten im Rahmen des iJUG-Open-Source-Stipendiums vorzustellen! In den kommenden Monaten werde ich als Mentor Jeremias Weber dabei unterstützen, den Committer-Status bei Jakarta-REST zu erhalten. Hierzu stehe ich ihm regelmäßig mit Rat und Tat, aber auch mit gelegentlichen Herausforderungen, zur Seite. Seine ersten Schritte hat er erfolgreich absolviert: Jeremias hat das JAX-RS-Team dabei unterstützt, Konformitätstests aus dem Jakarta-EE-TCK herauszuoperieren, auf eine neue Technologie (Arquilian und JUnit 5) zu migrieren und in das neu geschaffene Jakarta-REST-TCK zu integrieren. Für diese aufwendige und komplexe Operation dankt das ganze Team, denn diese Tests sind für die anstehende Veröffentlichung von JAX-RS 3.1 zwingend notwendig und waren längst überfällig. Auch unliebsame Aufgaben wie die Entfernung von Warnings im Build-Lauf hat er übernommen und sich mit dem Studium des Eclipse-Regelwerks die nötigen organisatorischen Kenntnisse für die zukünftige Verantwortung als Committer erarbeitet. Ich bin schon gespannt, welches Thema Jeremias als Nächstes adressiert, und stelle ihn euch auf der JavaLand dann gerne auch nochmal persönlich vor!

Das Stipendium steht in der laufenden Kampagne übrigens nicht nur Contributors zu Jakarta EE offen, sondern auch zu MicroProfile und Adoptium. Wenn ihr euch dauerhaft diesen Projekten verschreiben wollt, dann bewerbt euch doch einfach. Alles Weitere findet ihr im Web.

Zukünftig möchte der iJUG weitere Open-Source-Projekte mit Committern unterstützen. In der Diskussion ist beispielsweise Maven. Würde euch das reizen? Dann schreibt doch einfach mal an stipendium@ijug.eu!

Ach ja, ich bin euch ja noch das in der letzten Ausgabe angekündigte Interview mit Hendrik Ebbers zu Adoptium schuldig. Ihr findet es in dieser Ausgabe auf Seite 10.

Referenzen

[1] *https://github.com/ijug-ev/Stipendium iJUG-Open-Source-Stipendium im Web*

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.

Temurin 17: Interview mit Hendrik Ebbers, Projektleiter Eclipse Adoptium

Als Oracle vor einigen Jahren sein Lizenzmodell für JDK und JRE kostenpflichtig verändert hatte, erwuchs der Bedarf nach einer kostenlosen und trotzdem professionell gepflegten Java-Distribution. Die Gründung des AdoptOpenJDK-Projekts war die konsequente Antwort der Community, einfach eine eigene Distribution zu lancieren. Seit Kurzem nun ist das Projekt unter dem Namen „Eclipse Adoptium" (siehe Abbildung 1) bei der Eclipse Foundation angesiedelt und hat seine finanzielle Stabilität und professionelle Rückendeckung damit langfristig gesichert.

Markus Karg sprach für die Java aktuell mit Java-Champion Hendrik Ebbers, der als Projektleiter bei Adoptium diesen Weg begleitet hat.

***Markus:** Hendrik, danke, dass du dir die Zeit nimmst, unseren Leserinnen und Lesern die Hintergründe von Eclipse Adoptium näherzubringen! Wie kamst du eigentlich zu AdoptOpenJDK und was ist deine Aufgabe bei Adoptium?*

Hendrik: Ich empfand die aktive „Mitarbeit an Java" schon immer als spannend. Aus diesem Grund habe ich in der Vergangenheit ja auch beispielsweise bereits in Expert-Groups für JavaEE-/JakartaEE-Spezifikationen mitgearbeitet. Da der Einstieg in das Thema OpenJDK immer eine riesige Hürde war, habe ich mir schon vor einigen Jahren AdoptOpenJDK angeschaut, als es größtenteils aus Contributions von IBM entstanden ist.

Allerdings habe ich schnell gemerkt, dass eine direkte Mitarbeit an den Builds des OpenJDK in der Freizeit nur schwer zu stemmen ist. Daher

ADOPTIUM

Abbildung 2 (© Eclipse Foundation)

habe ich mich um andere Aufgaben, wie etwa die Website von AdoptOpenJDK und Ähnliches, gekümmert. Auch habe ich dabei geholfen, IcedTeaWeb von Red Hat in die AdoptOpenJDK-Community zu migrieren. Darüber hinaus habe ich einen Teil der Community-Arbeit übernommen. Anfang 2019 bin ich dann gefragt worden, ob ich Mitglied im Technical Steering Committee (TSC) von AdoptOpenJDK werden möchte. Auch wenn der Einstieg dort sehr hart war, da ich an vielen Stellen einfach nicht genug technisches Verständnis vom Bau des OpenJDK hatte, habe ich da in den letzten drei Jahren unglaublich viel gelernt. Ich bin froh, dass ich mittlerweile auch bei den tiefen technischen Themen mitreden und so das Projekt weiter in die eine gute Richtung lenken kann. Zum besseren Verständnis, worum es hier genau geht: Aktuell diskutieren wir im TSC zum Beispiel, wie Alpine-Linux-/Build-Container basierend auf Musl am besten aufgebaut sein sollen.

Markus: Hast du Zahlen für uns, wie viele Leute bereits Adoptium verwenden? Oracle hat ja kürzlich wieder kostenlosen Support angekündigt, sicherlich eine Reaktion auf euren Erfolg?

Hendrik: Wie eigentlich alles sind natürlich auch unsere Downloadzahlen öffentlich zugänglich! Informationen zu den über 350 Millionen Downloads von AdoptOpenJDK-Builds kann man sich hier anschauen: *[1]*. Für Adoptium ist das Ganze unter *[2]* verfügbar.

Hier hat der Zähler natürlich wieder bei 0 begonnen, aber wir können trotzdem schon über 10 Millionen Downloads verzeichnen. Zu Relationen zwischen den Distributionen gibt es keine offiziellen Zahlen, da Oracle mitunter seine Downloadzahlen nicht öffentlich zugänglich macht. Wenn man mit dem einen oder anderen führenden Cloud-Dienstleister spricht, kann man jedoch schon raushören, dass AdoptOpenJDK (und in Zukunft dann die Distribution von Adoptium) die mittlerweile meistgenutzte Distribution ist.

Markus: Der Umzug zur Eclipse Foundation hat ja eine ganze Zeit gedauert, umso spannender ist es zu sehen, wer denn treibende Kraft dahinter ist. Stimmt es, dass neben dem Open-Source-Riesen Red Hat da nun sogar Microsoft mitmacht – eine Firma, die den Älteren unter uns eher für den Markenstreit mit Sun Microsystems bekannt war?

Hendrik: Ja, und Microsoft macht da wirklich einen super Job. Allerdings muss man auch sehen, dass Microsoft da sehr strategisch eingekauft hat. Mit JClarity gab es eine auf OpenJDK, AdoptOpenJDK und Java-Performance spezialisierte Firma in Großbritannien, die Ende 2019 von Microsoft aufgekauft wurde. Mit dem Know-how dieser Firma und ein paar weiteren Neueinstellungen wurde dann sehr erfolgreich ein Java-/OpenJDK-Team innerhalb von Microsoft aufgebaut. Da einige der Leute hinter JClarity auch maßgeblich für AdoptOpenJDK mitverantwortlich waren, ist Microsoft hier ziemlich schnell vorne mit dabei gewesen. Aber wie gesagt: Ich sehe diese Entwicklung als zu 100 % positiv an.

Markus: Wie ist die Mitwirkung von IBM und Azul in Hinblick auf deren eigene, seit Jahren etablierte Distributionen zu sehen? Werden diese sterben oder sind dies mittlerweile gebrandete Distributionen des Adoptium-JDK beziehungsweise „Temurin", wir ihr es ja inzwischen genannt habt?

Hendrik: An den Distributionen der Working-Group- Mitglieder wird sich erst einmal nichts ändern. Jedes Mitglied kann natürlich seine eigene Distribution weiterführen und ich sehe auch keinen Grund darin, warum das nicht passieren sollte. Für die eigene Distribution haben die Firmen ja in der Regel eigene private Repositories, in denen Bugfixes für kommerzielle Kunden unabhängig vom OpenJDK schnell implementiert werden können und Bugfix-Releases solcher Änderungen ohne Umwege zum Kunden gelangen können. Hier werden solche Änderungen dann oft als Nacharbeit ins OpenJDK verschoben. Aufgrund des kommerziellen Supports hat der zahlende Kunde jedoch Vorrang. Das ist aus meiner Sicht auch völlig legitim. Somit hat jeder Benutzer die Wahl zwischen mehreren kommerziellen Hersteller-Angeboten oder der Temurin-Distribution von Adoptium, die herstellerneutral ist.

Da Temurin als neuer Name der Java-Distribution von Adoptium hier in diesem Interview zum ersten Mal verwendet wird, will ich mir einmal die Zeit nehmen und kurz etwas zum Namen sagen. Ich denke, das hilft, die Entstehung des Namens besser zu verstehen.

Wir wurden in der letzten Zeit oft gefragt, warum unsere Distribution nicht auch einfach Adoptium heißt. Hier haben wir uns aus zwei Gründen ganz klar dagegen entschieden: Zum einen planen wir als Adoptium-Working-Group noch weitere Projekte und haben mit AQAvit auch schon ein weiteres Projekt mit Produktnamen sowie offiziellem Logo *(siehe Abbildung 2)*. Dazu kommt, dass wir uns hier

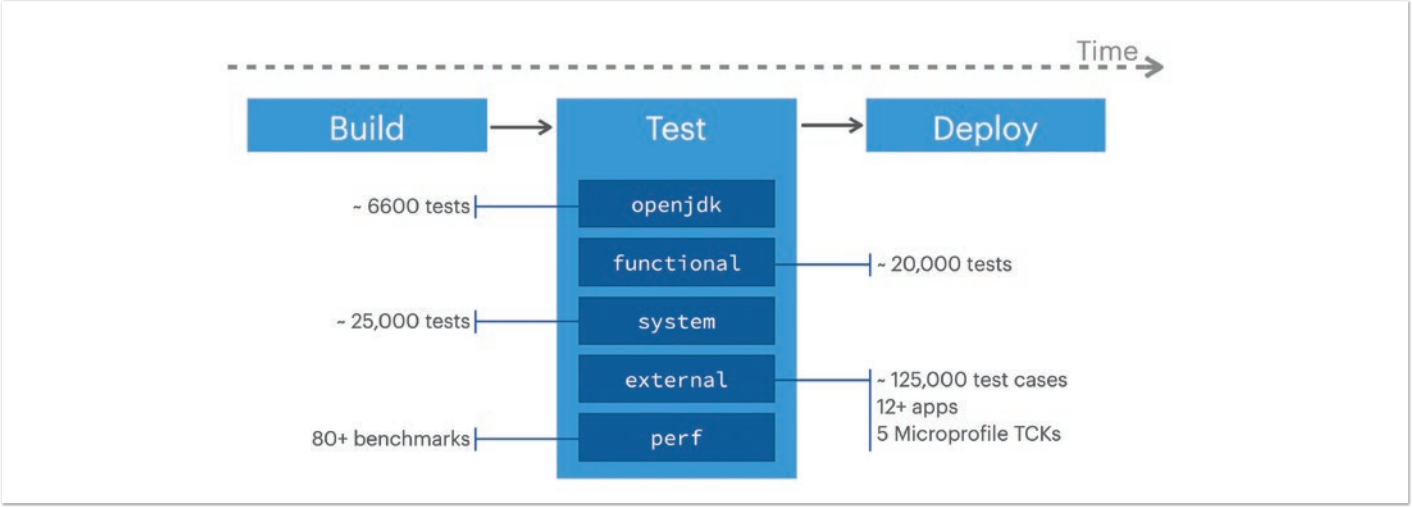


Abbildung 2 (© Hendrik Ebbers)

auch einfach an das Prozedere der anderen Hersteller gehalten haben. Die Distribution von Azul etwa heißt Zulu – und Bellsoft bietet mit Liberica eine Java-Distribution an. So kann man durch den Namen leicht Hersteller und Produkt unterscheiden. Der Name Temurin ist hierbei übrigens kein Kunstwort, sondern ein Anagramm von Runtime. Zusätzlich ist Temurin der Name der chemischen Verbindung, die Koffein am nächsten ist *[1] (siehe Abbildung 3)*. Aus diesen beiden Gründen ist Temurin aus meiner Sicht ganz klar der coolste Name für eine Java-Runtime!

Abbildung 3 (© Eclipse Foundation)

Markus: Gibt es überhaupt noch einen Markt für andere Distributionen – sei es Open Source wie etwa von Debian oder kommerziell wie von Azul – oder ist anzunehmen, dass es langfristig nur noch Oracle und Adoptium geben wird?

Hendrik: Wie erwähnt, gibt es einen solchen Markt aus meiner Sicht ganz klar. Bei Adoptium ist es sogar eines unserer Ziele, einen solchen Markt digital abzubilden. Hierbei geht es darum, dass alle durch das TCK und AQAvit zertifizierten Distributionen von verschiedenen Herstellern an einer zentralen Stelle angeboten werden. Dadurch, dass dieser Marketplace von Eclipse und unserer Working-Group verwaltet wird, gibt es eine wirklich neutrale Stelle zur Bereitstellung von Binaries, die die für uns wichtigen Kriterien zur Nutzbarkeit erfüllen.

Markus: Ist Temurin, also die JDK-Distribution von Eclipse Adoptium, wirklich produktiv einsetzbar oder nur für Entwicklungszwecke gedacht? Wie bekomme ich beispielsweise Support, wenn es mit Temurin Probleme bei meinen Kunden gibt?

Hendrik: Ich habe ja eben bereits von Zertifikaten gesprochen. Alle diese Überprüfungen gelten natürlich für Temurin genauso, wie etwa für Azul Zulu oder die OpenJDK-Distribution von Microsoft: Jeder Release-Build von Temurin wird gegen das TCK getestet, um sicherzustellen, dass es sich hierbei um eine Java-Runtime handelt, die den JavaSE-Standard implementiert. Durch AQAvit können wir mit zirka 200.000 zusätzlichen Unittests, Integrationstests und Benchmarks sicherstellen, dass die Binaries nicht nur dem Standard entsprechen, sondern auch alle Anforderungen für die Nutzung im Enterprise erfüllen *(siehe Abbildung 4)*.

Anders als das TCK, das Closed Source von Oracle entwickelt wird und über dessen konkreten Inhalt aus Lizenzgründen nicht geredet werden darf, ist die AQAvit Suite komplett Open Source und wird als Projekt der Adoptium-Working-Group weiterentwickelt. Eine nähere Beschreibung zu der Testsuite kann man hier finden *[4]*. Das Ganze zeigt auch sehr schön, dass wir bei Adoptium nicht einfach nur eine Distribution anbieten, sondern uns um das gesamte Ökosystem der Java-Laufzeitumgebung kümmern. Sobald der angesprochene Marketplace etabliert ist, werden auch sicherlich weitere Projekte folgen.

Markus: Ihr habt euch kürzlich entschieden, nur noch JDKs, aber keine JREs mehr zu veröffentlichen. Stattdessen gibt es eine Anleitung [5], wie man sich selbst ein JRE aus einem JDK erzeugt. Was steckt hinter dieser Entscheidung und ist die Mehrzahl der Nutzer wirklich begeistert von der Idee, erst mal selbst Hand anlegen zu müssen?

Hendrik: Definitiv ein Punkt, der sehr stark innerhalb der Working-Group und mit der Community diskutiert wurde. Aktuell gibt es auch für alle angeboten Java-Versionen JRE Builds für Temurin zum Download *[6]*. Auch für Java 17 bieten wir hier JRE-Binaries an.

Am Ende des Tages ist dies sicherlich ein perfektes Beispiel, um zu zeigen, wie wichtig uns die Community und die Zusammenarbeit mit der Community ist. Die Diskussion bezüglich JRE-Builds wurde öffentlich in einem GitHub-Issue geführt *[7]*. Neben den im Issue genannten Punkten gibt es verschiedene weitere Dinge, die klar gegen ein JRE sprechen. Im OpenJDK wurde bereits im Juni 2018 der Support für den Bau von JREs entfernt *[8]* und dadurch ist auch nicht hundertprozentig klar definiert, was genau in ein JRE für Java 17 gehört. Aus Sicht des OpenJDK ist die Nutzung von JLink zur Erstellung von eigenen und konkret angepassten Laufzeitumgebungen ganz klar die Zukunft.

Markus: Adoptium ist ein reinrassiges Open-Source-Projekt, in dem alle Teilnehmer gleichberechtigt sind. Gibt es trotzdem einen herausragenden Leader? Sind nur Firmen beteiligt oder macht es Sinn, sich auch als Einzelpersonen in das Projekt einzubringen?

Hendrik: Ich würde nicht sagen, dass es einen herausragenden Leader gibt. Klar ist, dass Red Hat und Microsoft mehrere Leute in Vollzeit auf das Projekt buchen und ich sehe das auch als einen extremen Mehrwert an. Ich habe aber weder zu Zeiten von AdoptOpenJDK noch zu Adoptium in irgendeinem Moment das Gefühl gehabt, dass eine Partei dieses Projekt allein leitet oder rein für die eigenen Zwecke nutzt. Vor allem in der Working-Group und im Project Management Committee begegnen sich alle auf Augenhöhe.

Auch, wenn ich aktuell aufgrund meiner leitenden Funktion bei Karakun nicht Vollzeit bei Adoptium arbeiten kann, werden meine Beiträge wertgeschätzt. Es wird darauf geachtet, dass ich bei Entscheidungen nicht umgangen werde und auch immer auf dem Laufenden bleibe.

Abbildung 4 (© Eclipse Foundation)

All diese Punkt sprechen klar dafür, dass es auch als Einzelperson Sinn machen kann, sich in das Projekt einzubringen. Will man hier tief in den Builds der Runtime mitarbeiten, muss man allerdings schon ein wenig Zeit mitbringen. Das Schöne ist jedoch, dass wir mit dem Marketplace, AQAvit, unserer Website und ein paar weiteren Projekten sehr gute Alternativen und „Good First Issues" anbieten können. Wir haben 2021 erneut erfolgreich am Hacktoberfest teilgenommen und unterstützen weitere Events und Organisationen, um neue Mitarbeiter in das Team zu bekommen und generell für das Thema Open Source zu begeistern. Das Thema liegt aktuell auf meinem Tisch und wir erarbeiten momentan noch bessere Workflows, um den Einstieg in Adoptium so einfach wie möglich zu gestalten *[9]*.

Ein weiterer Grund für deine Frage ist sicherlich das iJUG-Stipendium, das du letztes Jahr ins Leben gerufen hast *[10]*. Dieses „Stipendium" fordert die Mitarbeit von iJUG-Mitgliedern an Open-Source-Projekten durch Benefits wie Freikarten für die JavaLand oder das JavaForumNord. Ich habe mich deiner Idee angeschlossen und betreue hierbei mittlerweile einige Freiwillige, die durch das Stipendium zusammen an einer losgelösten Thematik arbeiten. Durch Bi-Weeklies und Pair-Programming ist hier eine kleine deutsche Community in Adoptium im Entstehen, in der bereits sehr viel Wissen aufgebaut und geteilt wird. So etwas zu sehen, ist als Projektleiter eines Open-Source-Projekts einfach wunderbar! Natürlich möchte ich an dieser Stelle auch noch einmal jeden bitten, sich bei Markus oder mir zu melden, falls Interesse an der Mitarbeit in einem Open-Source-Projekt besteht.

Markus: Bislang baut und testet ihr lediglich die Quellen des OpenJDK-Projekts. Ist zukünftig geplant, auch eigene Optimierungen an den Quellen vorzunehmen, etwa um bestimmte Segmente wie den Embedded-Bereich oder das Hochgeschwindigkeitsrechnen besser unterstützen zu können, die ja von OpenJDK selbst eher rudimentär abgedeckt werden?

Hendrik: Es ist nicht geplant, dass mit Temurin neue exklusive Funktionen angeboten werden. Alle Verbesserungen und Features, die durch Adoptium entstehen, werden Bestandteil des OpenJDK. Daher haben wir auch mehrere Leute im Team, die Commiter im OpenJDK sind. Dies bedeutet aber nicht, dass solche Ideen nicht innerhalb von Adoptium entstehen können und dann in das OpenJDK getragen werden. Allerdings sehe ich hier weiterhin das OpenJDK als die richtige Plattform für solche Entwicklungen und Diskussionen an. Das Ziel von Adoptium ist ja nicht, das OpenJDK oder die etablierten Workflows in irgendeiner Weise zu übernehmen. Vielmehr hoffe ich, dass wir einen positiven Einfluss auf verschiedene Bereiche haben können.

Markus: Was dürfen wir in Zukunft von Eclipse Adoptium erwarten? Wie sieht die mittelfristige Roadmap aus?

Hendrik: Das Projekt ist ja noch sehr jung und wir haben es trotzdem geschafft, mit Temurin bereits Downloads im zweistelligen Millionenbereich zu haben. Ich denke, dass wir hiermit schon einiges erreicht haben und das Projekt bereits eine gute Sichtbarkeit hat. Trotzdem liegt hier eine Hauptaufgabe in der nahen Zukunft: Mit über 350 Millionen Downloads ist AdoptOpenJDK noch immer die am meisten heruntergeladene Laufzeitumgebung und wir planen, diesen Erfolg mit Adoptium und Temurin noch zu vergrößern. Hierfür muss eine klare und einheitliche Message her, die wir dann unter Entwicklern, Administratoren und Entscheidenden verbreiten können. Genau zu diesem Zweck habe ich auch ein Marketing-Committee innerhalb von Adoptium ins Leben gerufen. Man kann also sicher sein, auf der einen oder anderen Konferenz in Zukunft von uns zu hören!

Ein weiterer wichtiger Meilenstein ist eine neue Website *[11]*. Der aktuelle Stand ist im Prinzip ein Fork von AdoptOpenJDK. Die neue Website soll Funktionen wie den Marketplace, Temurin und Dokumentationen unter einem Dach vereinheitlichen. Dies ist für die erste Jahreshälfte 2022 geplant und die ersten Ergebnisse sehen hier auch sehr vielversprechend aus.

Neben diesen Punkten und der klaren Weiterführung und Verbesserung der Temurin-Builds sowie der AQAvit-Testsuite wird es sicherlich in Zukunft auch weitere Projekte geben. Um genau an solchen Punkten zu arbeiten, haben wir ein Incubator-Projekt ins Leben gerufen, in dem auch bereits Ideen ausgetauscht werden.

Markus: Das waren interessante Antworten, die Lust machen, im nächsten Projekt Temurin zu evaluieren. Ich danke dir für das Gespräch und freue mich schon auf das nächste Release!

Hendrik: Ich danke, dass ich die Adoptium-Working-Group und Temurin vorstellen durfte!

Referenzen

[1] *https://dash.adoptopenjdk.net*
[2] *https://dash.adoptium.net*
[3] *https://en.wikipedia.org/w/index.php?title=Temurin&redirect=no*
[4] *https://guigarage.com/2020/02/21/adopt-tests.html*
[5] *https://blog.adoptium.net/2021/10/jlink-to-produce-own-runtime* - So erzeugt man mit jlink ein JRE aus einem JDK
[6] *https://adoptium.net/releases.html*
[7] *https://github.com/adoptium/temurin-build/issues/2683*
[8] *https://github.com/openjdk/jdk/commit/1791b54bf2f1d44b882756a41f204d49472c90d0#diff-bfbb5c87a00ff54b129b-7ffd68637b90a85c62c466eefeec1195d614ea4200ee*
[9] *https://github.com/adoptium/documentation/blob/main/first-timers-support/index.adoc*
[10] *https://github.com/ijug-ev/Stipendium*
[11] *https://adoptium.net*

Flexible Anwendungsarchitektur mit der Clean Architecture – Teil 1: Die Basis für mehr Flexibilität

Matthias Eschhold, Novatec Consulting GmbH

In der Softwareentwicklung versteckt sich der Wunsch nach Flexibilität auf vielen Architekturebenen. Soll Flexibilität in der Anwendungsarchitektur erreicht werden, ist das Clean-Architecture-Muster ein erfolgversprechender Ansatz. Dennoch ist dieses Muster nicht weit verbreitet und nicht einfach anzuwenden. Der erste Teil dieser Serie beschäftigt sich aus diesem Grund mit der Anwendung dieses Musters auf Paket- und Klassenebene.

Qualitätsmerkmale für Software sind in DIN ISO/IEC 25010 definiert. Die Wartbarkeit, als eines der acht Hauptmerkmale dieses Qualitätsmodells *(siehe Abbildung 1)*, besagt, dass ein Softwaresystem modular, änderbar, erweiterbar, analysierbar, stabil und testbar zu sein hat *[1]*. Änderbarkeit, Erweiterbarkeit und Stabilität sprechen implizit auch Flexibilität an. Flexibilität steht aber nicht im Fokus!

Moderne Architekturansätze, wie Microservices und Self-Contained-Systems (SCS), werden als flexible Architekturmodelle bezeichnet. Flexibilität wird hier adressiert, indem diese Architekturmodelle auf Austauschbarkeit, Technologiefreiheit, Skalierbarkeit sowie Änderbarkeit von Subsystemen abzielen *[2]*.

Abbildung 1: Qualitätsmodell DIN IS /IEC 25010 (© Matias Eschhold, in Anlehnung an [1])

Starke und *Hruschka* beschreiben flexible Eigenschaften von Softwarebausteinen durch **Flexibilität** zur **Laufzeit** und zur **Entwicklungszeit**. Flexibilität zur Laufzeit kann durch Konfiguration von Softwarebausteinen erfolgen. Flexibilität zur Entwicklungszeit entsteht, wenn durch Wiederverwendung Beschleunigung erzielt wird, indem eine Funktion in gleicher Weise von weiteren Konsumenten genutzt werden kann. Die hier beschriebene Flexibilität zielt primär auf Wiederverwendung ab, indem ein Baustein eine generische Schnittstelle für mehrere Konsumenten anbietet oder durch Konfiguration die Funktionalität eines Bausteins für einen konkreten Konsumenten ausgeprägt wird *[3]*. Flexibilität von Softwarebausteinen durch eine universelle Schnittstelle oder durch Konfiguration kann innerhalb eines Systems oder einer Produktlinie positiven Nutzen erzielen.

Beides trifft noch nicht das Verständnis über Flexibilität, dass in dieser Artikelserie als Qualitätseigenschaft für die Anwendungsarchitektur von Softwaresystemen verstanden wird. Einer der wichtigsten Erfolgsfaktoren für die Geschäftsmodelle vieler Unternehmen ist Time-to-Market *[4]*. **Time-to-Market** ist die Zeit, die benötigt wird, um einen definierten Umfang an Funktionalität zu beauftragen, zu entwickeln, zu testen und letztendlich stabil in Produktion zu bringen. Softwaresysteme unterliegen einem stetigen Wandel. Aufgrund fachlicher, technologischer und organisatorischer Veränderungen im Umfeld eines Softwaresystems muss dies kontinuierlich angepasst werden. Ändert sich die reale Welt, muss sich die digitale Welt ebenfalls verändern *[5]*.

Eine **flexible Anwendungsarchitektur** richtet sich vollkommen auf einen **schnellen** und **permanenten Anpassungsbedarf** aus. Erweiterbarkeit und Wartbarkeit reichen bei hohen Anforderungen an Time-to-Market in volatilen Umfeldern nicht aus. Die richtigen Lösungsstrategien helfen, Flexibilität in der Basis zu unterstützen und darauf aufbauend wahre Flexibilität in der Anwendungsarchitektur zu erreichen.

Architekturstil, Architekturmuster und Anwendungsarchitektur

Ein Architekturstil beschreibt eine strukturelle **Basisstrukturierung** für eine **Familie** von **Softwaresystemen** *[6]*. Die oben erwähnten Architekturmodelle für Flexibilität stellen Architekturstile dar. Alle Microservices folgen den Entwurfsprinzipien und -regeln, die der Architekturstil beschreibt. Die Kommunikation über standardisierte Schnittstellen oder die Konfiguration eines Microservice über seine Umgebung sind bekannte Prinzipien dieses Architekturstils *[2]*.

Ein Microservice benötigt eine innere Architektur. Diese wird als Anwendungsarchitektur bezeichnet. Für die Anwendungsarchitektur enthält der Architekturstil keine Prinzipien und Regeln. Jeder Microservice kann intern unterschiedlich strukturiert sein. Eine Gleichartigkeit wird durch Architekturmuster, wie Clean Architecture, erreicht. Architekturmuster sind kombinierte Entwurfsprinzipien und -regeln, mit dem Charakter der **Basisstrukturierung** eines **Softwaresystems** *[6]*. Architekturmuster nutzen als Strukturierungsmittel Verantwortungsbereiche und Beziehungsregeln, die für die Dekomposition eines

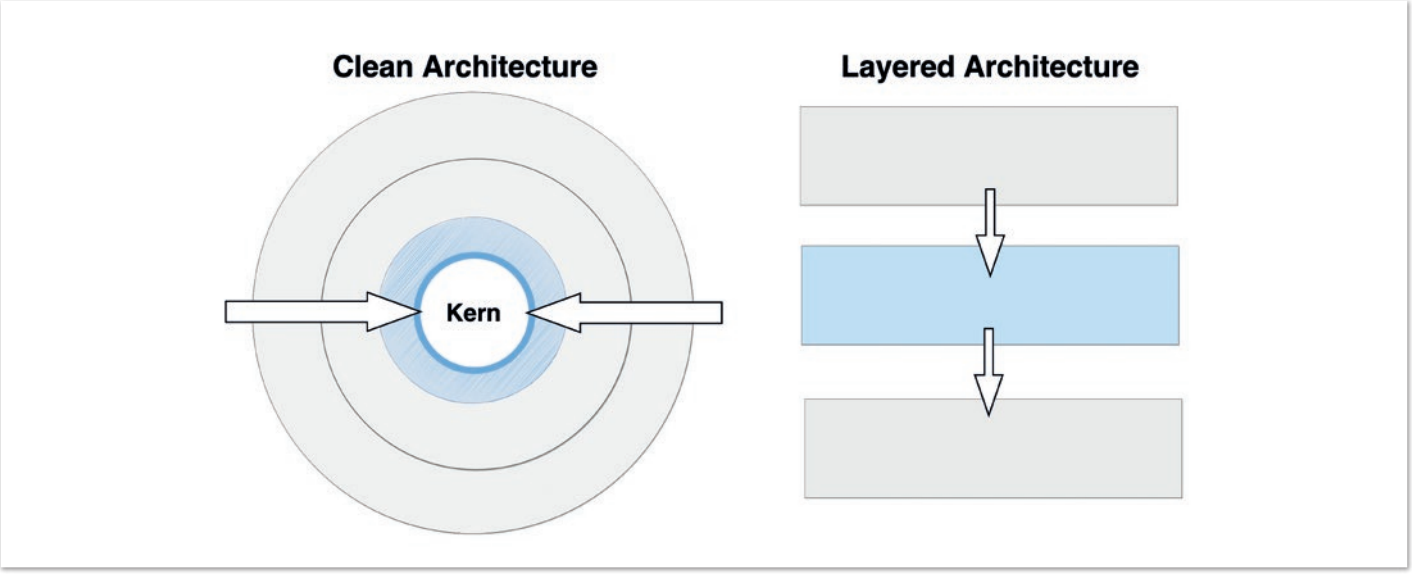


Abbildung 2: Vergleich Clean Architecture und Layered Architecture (© Matthias Eschhold)

Systems eingesetzt werden. Im Falle der Clean Architecture sind dies Ringe und die vorgegebene Beziehungsrichtung verläuft zum inneren Kern des Ringmodells *[7]*. Bei einer geschichteten Architektur (Layered Architecture) sind die Verantwortungsbereiche Schichten, die hierarchisch in Beziehung stehen *(siehe Abbildung 2) [6]*.

Das Basismuster Ports und Adapters

Das Ports-und-Adapters-Architekturmuster *(siehe Abbildung 3)* definiert den Grundsatz, dass ein System nur auf einem einheitlichen Weg genutzt werden darf. Und dies unabhängig davon, ob die Nutzung durch einen Menschen, ein Softwareprogramm, einen Test oder einen Batchlauf geschieht. Um dies zu erreichen, bedarf es einer klaren Trennung zwischen der fachlichen Domäne und der Infrastruktur. Die Nutzung der Domäne erfolgt durch eine Infrastrukturkomponente. Die Domäne selbst nutzt auch Infrastruktur, etwa für das Speichern von Daten. Zwischen der Domäne und der Infrastruktur befinden sich Adapter, die auf Ports der Domäne adaptieren. Dadurch wird die Domäne unabhängig *[8]*.

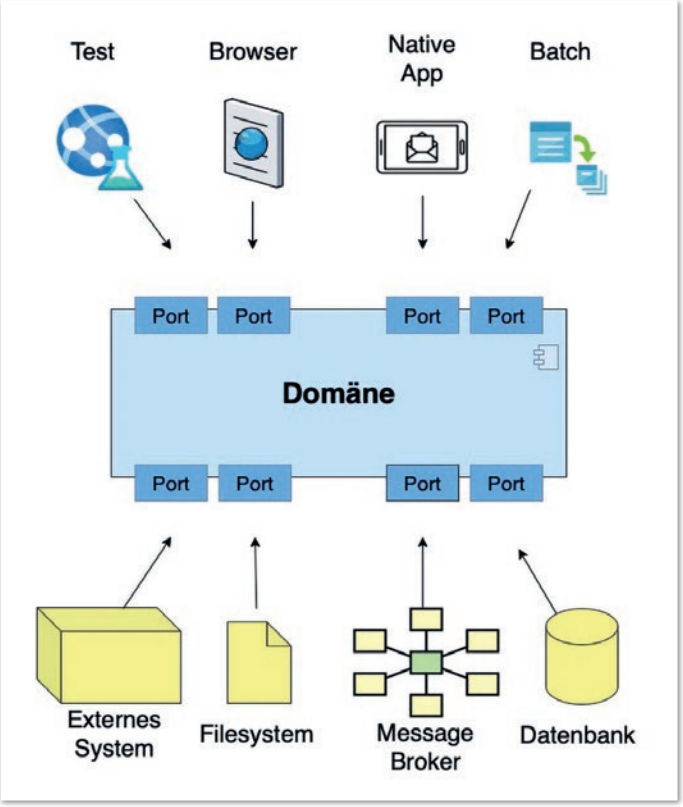


Abbildung 3: Ports-und-Adapters-Architekturmuster (© Matthias Eschhold, in Anlehnung an [9])

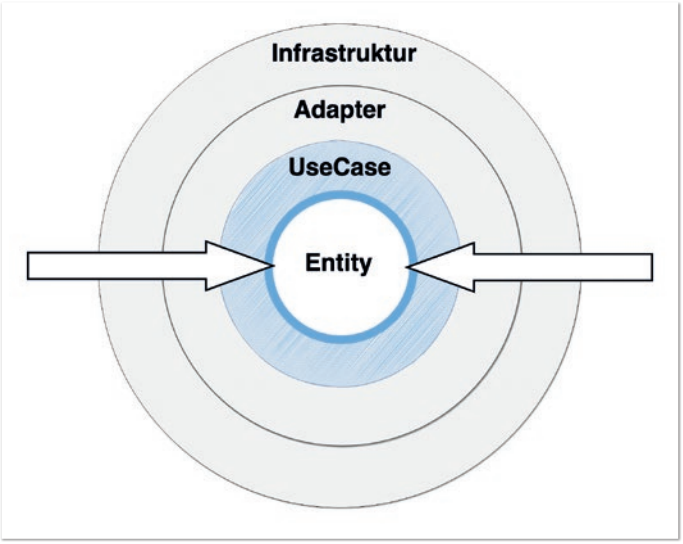


Abbildung 4: Clean Architecture nach Robert C. Martin (© Matthias Eschhold, in An-lehnung an [8])

Die Clean Architecture

Die Clean Architecture basiert auf dem Ports-und-Adapters-Muster. Wie bereits beschrieben, werden Ringe zu Abgrenzung von Verantwortungsbereichen eingesetzt. Die Domäne und die Infrastruktur werden im Ringmodell der Clean Architecture weiter verfeinert *(siehe Abbildung 4)*. Die Domäne unterteilt sich in Use Case und Entity. Die Infrastruktur wird um Adapter ergänzt.

In der Beschreibung von *Martin* umhüllt der Use Case Ring die Entitäten. Gemeinsam bilden sie den fachlichen Kern. Der Use Case ist eine **fachlich-motivierte Schnittstelle**, über die die Entität genutzt wird oder die Entität selbst Infrastruktur nutzt. Auf dem zweitäußersten Ring befinden sich Adapter. Die Infrastrukturkomponenten sind auf dem äußersten Ring verortet. Die Richtung der strukturellen Beziehung zwischen den Ringen verläuft stets nach innen in Richtung der Domäne *[8]*. Das bedeutet, dass die Klassen des Domänen-Kerns keine Kompilierungsabhängigkeiten zu Klassen der Infrastruktur aufweisen.

Ein *Repository* ist eine gängige Bezeichnung für einen Datenbank-Adapter. Das *Repository* hat die Verantwortung, den technischen Zugriff auf eine Datenbank zu kapseln. Der *Use Case* definiert aus Sicht der Anwendung die Schnittstelle auf die Datenbank. Der Adapter, also das *Repository*, implementiert diese Schnittstelle. Diese Beziehungsstruktur verwirklicht das Dependency Inversion Principle *(siehe Abbildung 5)*.

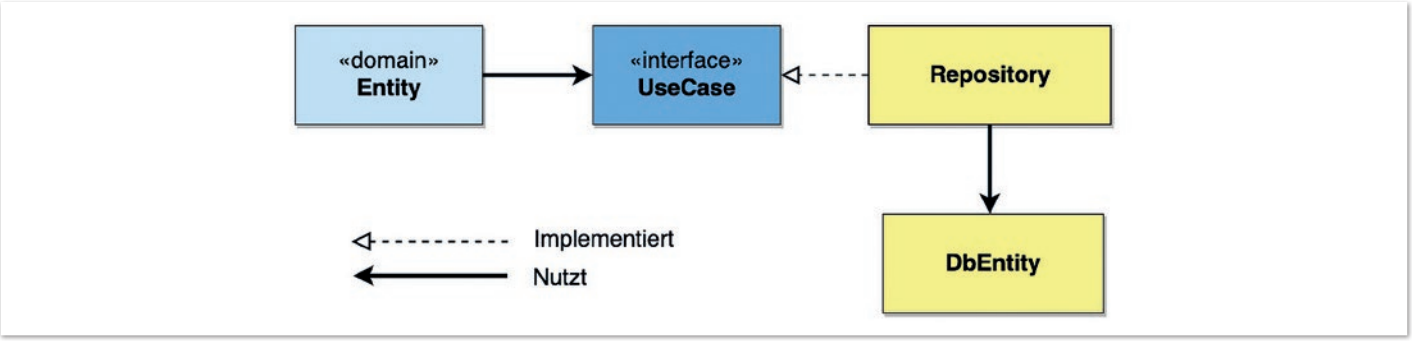


Abbildung 5: Datenbank-Adapter mit Anwendung des Dependency Inversion Princip-le (© Matthias Eschhold)

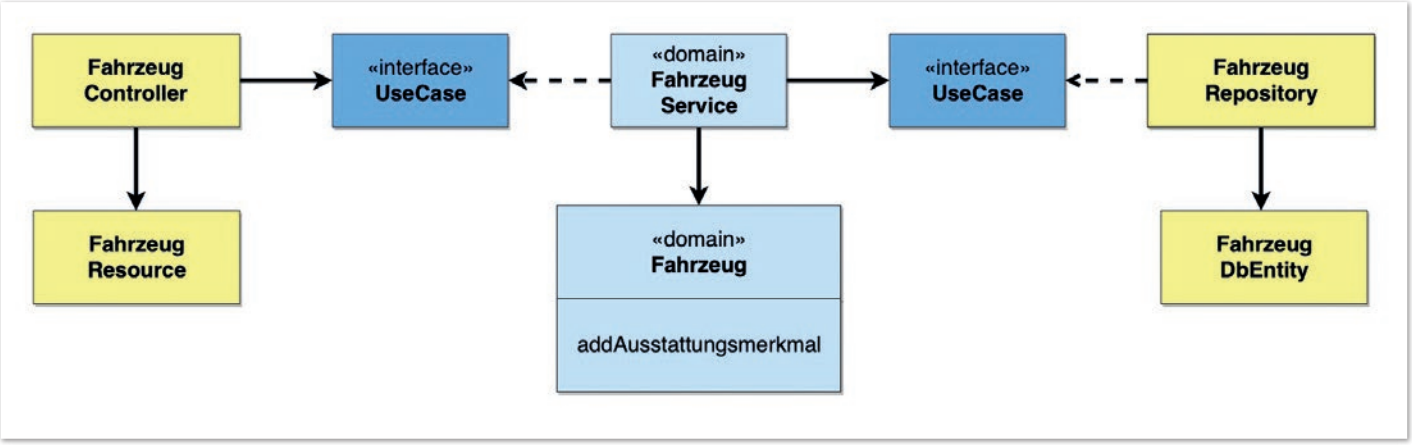


Abbildung 6: Mustersprache in der Clean Architecture am Beispiel der Entität Fahr-zeug (© Matthias Eschhold, in Anlehnung an [11])

Dependency Inversion Principle

Der Nutzer einer Dienstleistung sollte möglichst von Abstraktionen (das heißt abstrakten Klassen oder Interfaces), nicht aber von konkreten Implementierungen abhängig sein. Abstraktionen sollten nicht von konkreten Implementierungen abhängen [10].

Die Beziehungsstruktur zwischen den beteiligten Klassen wird durch das Dependency Inversion Principle umgekehrt. Drei Details bewirken diesen Effekt:

1. Der Use Case gehört der Domäne.
2. Der Adapter implementiert nach objektorientiertem Paradigma den Use Case.
3. Die Domäne kennt mithilfe von Dependency Injection nur den Use Case.

Durch Dependency Injection wird die Erzeugung der konkreten Adapter-Klasse auf einen Injector ausgelagert. Dies ist entscheidend für die Unabhängigkeit der Domäne *[11]*.

Flexibilität in der Technologie

Die strikte Trennung von Infrastruktur und Domäne erleichtert technologische Erneuerung. Ein neuer Adapter substituiert den bisherigen Adapter. Diese Flexibilität wird beispielsweise bei dem Umstieg auf eine neue Bibliothek für die Datenbank-Anbindung oder bei Ablösung eines „Legacy"-Service durch einen neuen benötigt.

Durch die Entkopplung der Domäne von der Infrastruktur sind technologische Erneuerungen mit weniger Risiko und Testaufwand verbunden. Außerdem können diese sukzessive durchgeführt werden. Dies erleichtert die Verzahnung von technischer Modernisierung und fachlichen Releases.

Fachlich ausdrucksstarke Anwendungsarchitektur

Ein Beispiel durch alle Ringe der Clean Architecture zeigt *Abbildung 6* anhand der Entität „Fahrzeug". Für das Klassendesign wird eine Mustersprache verwendet. Eine Mustersprache ist ein Verbund von Klassen, die durch ihr Zusammenwirken zusammengehörige fachliche Funktionen Ende-zu-Ende realisieren. Jede Klasse in diesem Verbund hat eine stereotypische Verantwortung, eine sogenannte „Single Responsibility". Hierbei werden fachliche und technische Verantwortungsbereiche voneinander abgegrenzt *[11]*. Um diesen Klassenverbund bildet sich ein fachliches Modul, wie etwa das Fahrzeug. Ein System besteht in den meisten Fällen aus mehreren fachlichen Modulen, die auf oberster Ebene der Paketstruktur sichtbar sind *(siehe Abbildung 7)*.

Die Funktionalität des Bausteins „Fahrzeug" wird über HTTP für das Web angeboten. Technischer Code für einen HTTP-Endpunkt und sein Datentransferobjekt finden sich in der Mustersprache durch die Stereotype Controller beziehungsweise *FahrzeugController* und *Resource* beziehungsweise *FahrzeugResource* wieder. Der *Controller* ist der Adapter der Infrastrukturkomponente „Web".

Der Klassenstereotyp *Service*

- implementiert und nutzt Use Cases,
- realisiert übergreifende Geschäftslogik und
- delegiert die Ausführung von Geschäftslogik an Entitäten.

Die *Entity* Fahrzeug ist verantwortlich für die Logik, etwa dafür, ein weiteres Ausstattungsmerkmal zum *Fahrzeug* hinzuzufügen. Auch

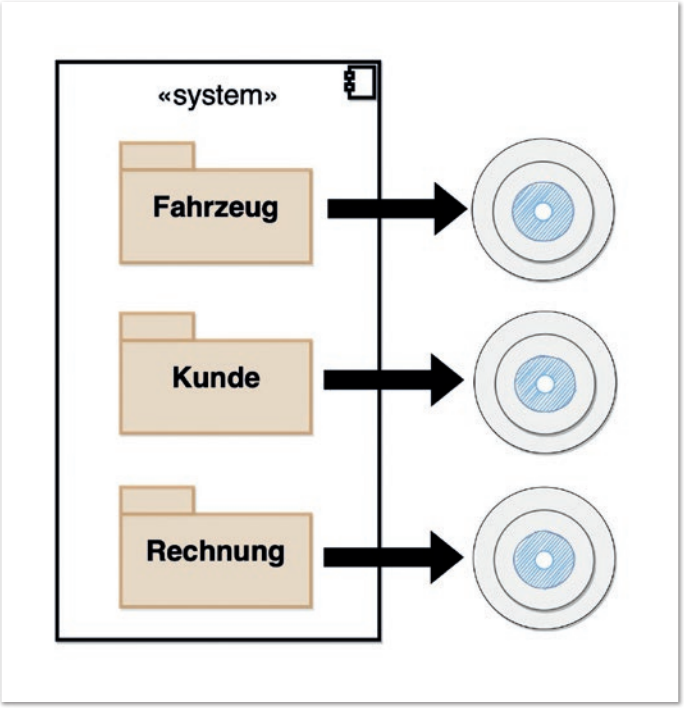


Abbildung 7: Fachliche Paketstrukturierung eines Systems am Beispiel der Entitäten Fahrzeug, Kunde und Rechnung (© Matthias Eschhold)

für die Validierung der eingehenden Daten ist die *Entity* selbst verantwortlich. Der *FahrzeugService* stößt beispielsweise die Abfrage eines Fahrzeugs aus der Datenbank an, lässt dann die entsprechende Zustandsveränderung (zum Beispiel neues Ausstattungsmerkmal) in der Entität ausführen, um abschließend den neuen Zustand wieder zu persistieren. Die Datenbank-Anbindung befindet sich in einem Adapter. Im vorliegenden Beispiel ist dies dargestellt durch das *FahrzeugRepository* und die *FahrzeugDbEntity*. Services und Use Cases referenzieren Entitäten als eingehende Parameter und Rückgabewerte. Die Adapter müssen auf die Entitäten adaptieren.

Neben Controller und Repository als Adapter sind *EventListener* oder der *ServiceClient* gängig. Der *EventListener* hat die Verantwortung, fachliche Ereignisse über die Infrastruktur zu empfangen. Ein *ServiceClient* kapselt die technische Integration externer Systeme. Ein größeres Spektrum von Infrastrukturkomponenten und Adapter wird in *Abbildung 8* dargestellt. Im Kern des Ringmodells befinden sich die Entitäten.

Use Cases als Flexibilisator

Die Schnittstellendefinition von *Use Cases* erfolgt Konsumenten-getrieben aus Sicht der Domäne. Dabei wird das Interface Segregation Principle eingesetzt.

Interface Segregation Principle

Schnittstellen gehören den Clients und nicht den Klassenhierarchien, die diese Schnittstellen implementieren. Das bedeutet, Schnittstellen sollten nach den Clients entworfen werden, die diese Schnittstellen benötigen. Auch sollten Clients nicht von Schnittstellen abhängen, die sie nicht benötigen [10].

Die Anwendung dieses Prinzips wird umso besser erfüllt, je kleiner und spezifischer die Schnittstelle gestaltet wird. Empfehlungen für den Entwurf von *Use Cases* sind die **Trennung** von **ein-** und **ausgehenden** *Use Cases* sowie die Trennung von Query und Command *Use Cases*. Die Unterscheidung in ein- und ausgehende Use Cases und auch Adapter erscheint am Beispiel *Controller* und *Repository* fast natürlich. Unter *Query Use Cases* werden alle lesenden Anfragen verstanden. Command *Use Cases* lösen Zustandsveränderungen aus. Die Anwendung dieser Praktiken motivieren sich wie folgt:

1. Command und Query stellen sinnvolle und granulare Modularisierungsgrößen dar.
2. Die Anwendungsarchitektur bleibt offen für eine tiefergehende Anwendung des Command Query Responsibility Segregation Principle.
3. Das Interface Segregation Principle wird eingehalten, da Klassen die nur *Query Use Cases* benötigen, nicht von *Command Use Cases* abhängig sind.

UML-Stereotype und die Vererbungsbeziehung (Ist-Ein-Beziehung) verdeutlichen die Unterschiede hinsichtlich *Use Cases* in der Modellierung *(siehe Abbildung 9)*. Auf Codeebene ist nicht zwangsläufig eine technische Vererbungsstruktur notwendig. Dies dient primär der konzeptionellen Beschreibung.

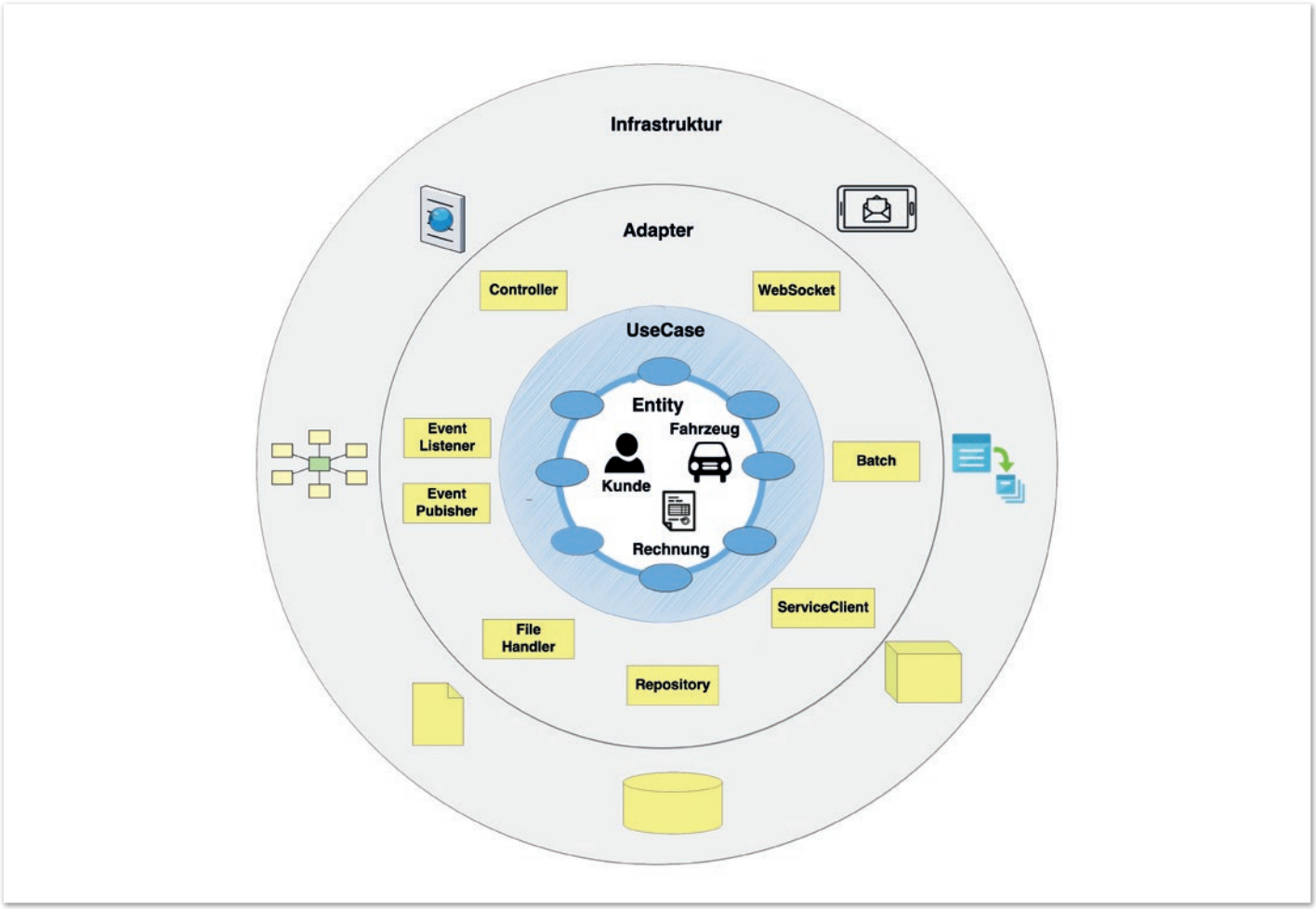


Abbildung 8: Infrastrukturkomponenten und Adapter in der Clean Architecture (© Matthias Eschhold)

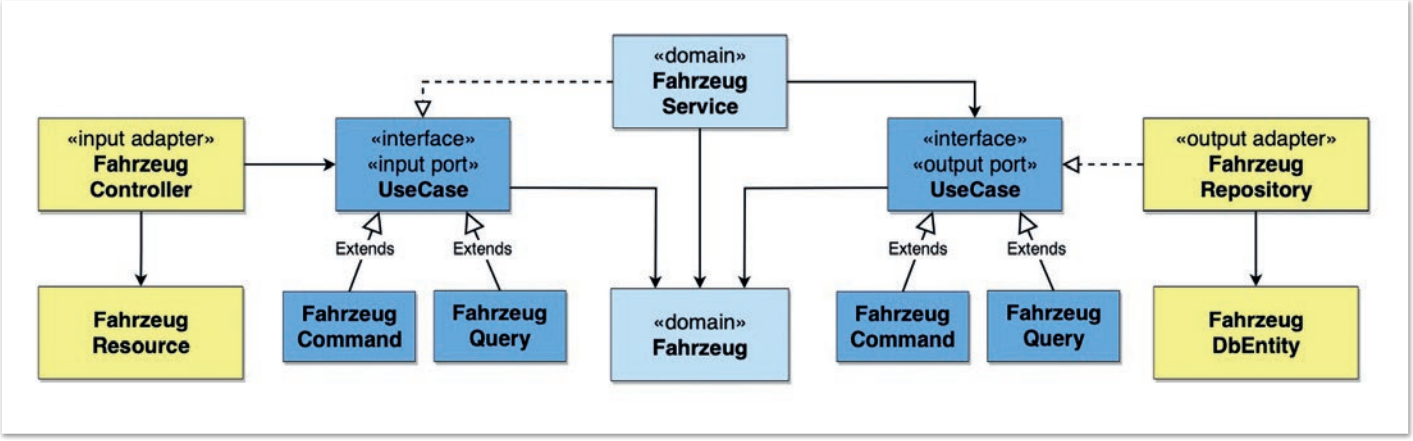


Abbildung 9: Erweiterte Use-Case-Definition in der Mustersprache (© Matthias Esch-hold)

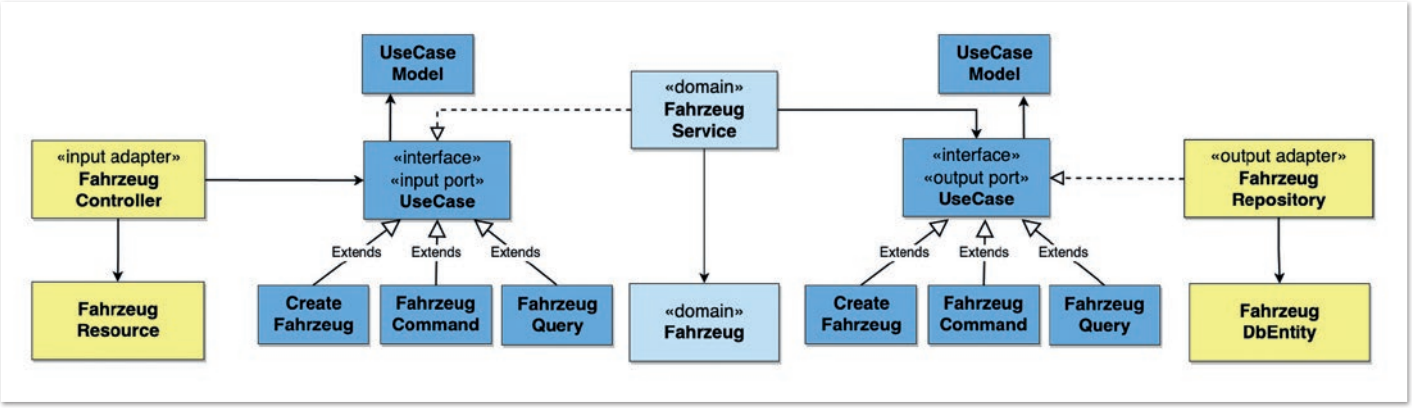


Abbildung 10: Feingranulare Use Cases mit eigenem Use-Case-Modell (© Matthias Eschhold, in Anlehnung an [11])

Hombergs beschreibt einen tiefergehenden Ansatz *(siehe Abbildung 10)*, indem *Use Cases* auf einem separaten Modell operieren und nicht nur klein, sondern **feingranular** geschnitten sind. Jeder *Use Case* definiert eine Aktion (beispielsweise *CreateFahrzeug*) mit einem eigenen Eingangsparameter und Rückgabewert (*Use Case Model*). Dies sollte auf jeden Fall Anwendung finden, wenn ansonsten das Interface Segregation Principle verletzt wird und dies zu Seiteneffekten führt. Auch empfiehlt sich dieser Ansatz bei fachlich sehr komplexen Anforderungen. Das Single Responsibility angewendet auf jeden Use Case reduziert nochmals stärker Komplexität und Kopplung *[11]*.

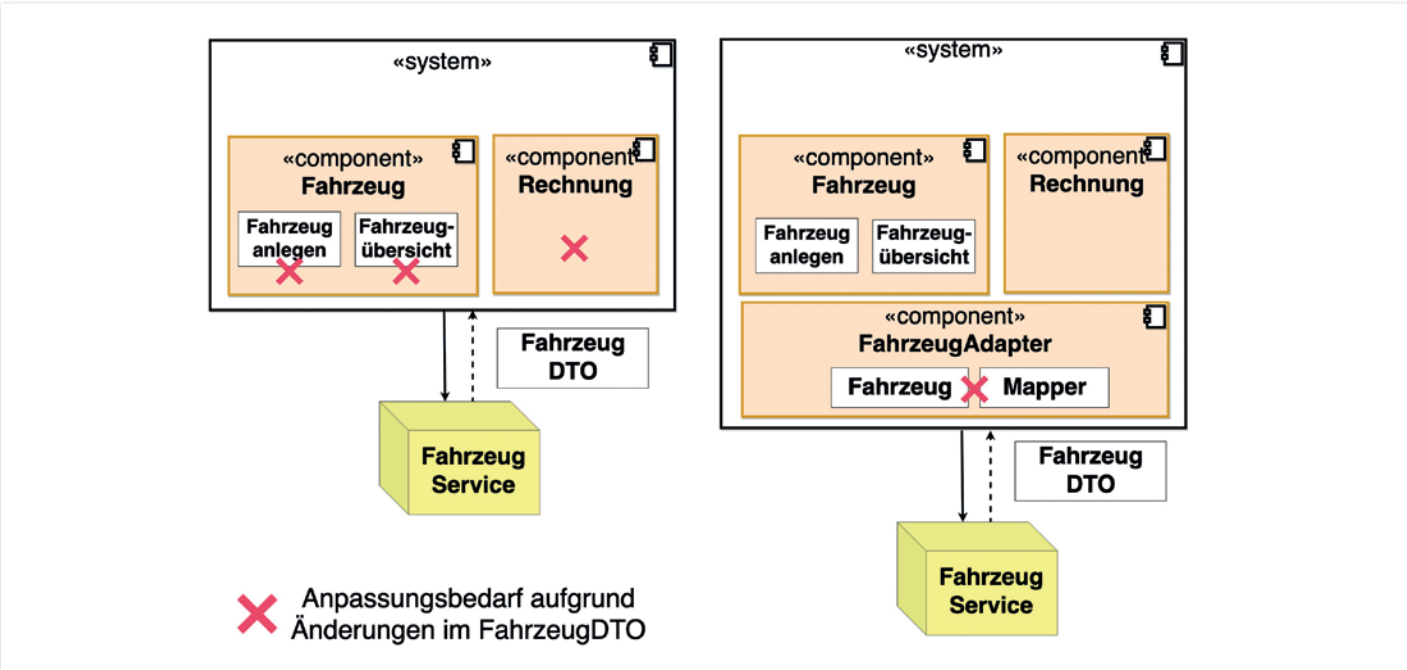


Abbildung 11: Vergleich des Anpassungsbedarfs ohne beziehungsweise mit Adapter (© Matthias Eschhold)

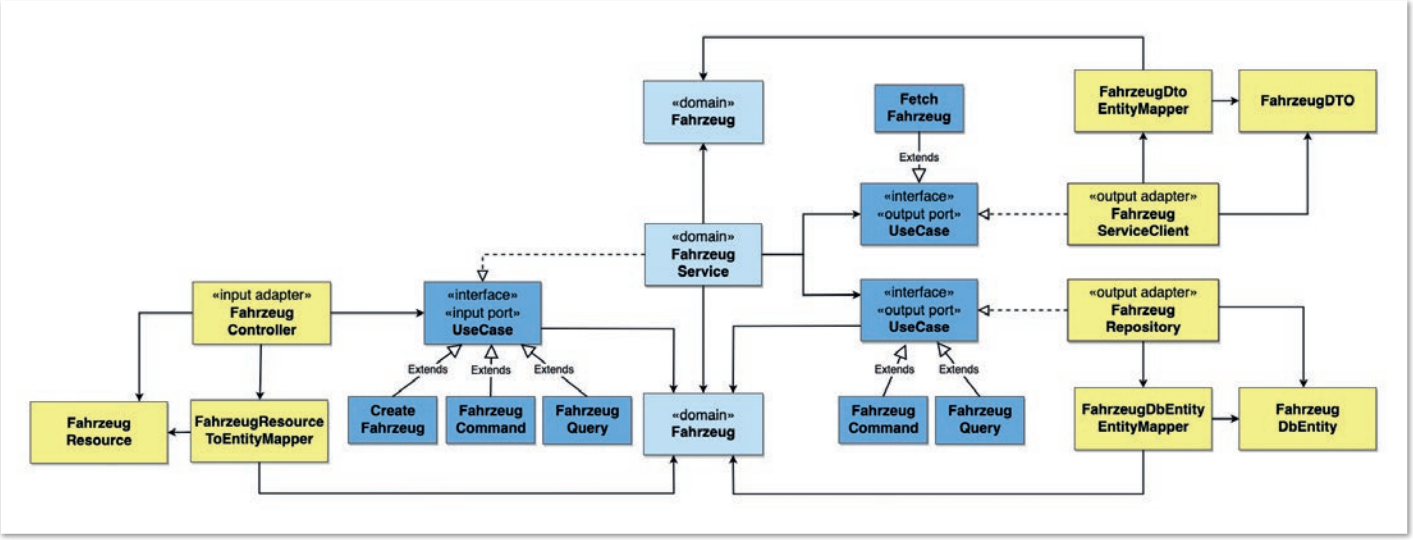


Abbildung 12: Mustersprache mit Two-Way-Mapping-Strategie (© Matthias Eschhold, in Anlehnung an [11])

Flexible Software ist anhand der mentalen Modelle der Fachexperten zu strukturieren. Vereinfacht ausgedrückt, entspricht dies einer Struktur wie in *Abbildung 7* dargestellt. Um dies zu erreichen, ist viel Kommunikation und permanente Anpassung notwendig. Bei der Analyse von fachlichen Anforderungen, bei der Implementierung einer User Story sowie beim Code Review werden verschiedene Perspektiven auf fachliche Anforderungen diskutiert und neues Wissen entsteht. Damit die Software schnell darauf reagieren kann, ist es notwendig, Anpassungen schnell, einfach und ohne Risiko durchführen zu können. Dies verhindert fachlich fehlerhaft abgebildete Architektur und technische Schulden. Je granularer die Verantwortung im System verteilt ist und je geringer der Kopplungsgrad zwischen Klassen im System ist, desto einfacher kann sich der Domänencode an fachliche Veränderungen anpassen *[12]*.

Flexibilität durch Stabilität

Für die vollständige Entkopplung zwischen Domäne und Infrastruktur durch einen Adapter ist eine Überführung zwischen externem Datentransferobjekt und dem Domänenmodell notwendig. Dies wird als Mapping bezeichnet und innerhalb eines Adapters ausgeführt.

Was würde passieren, wenn kein Mapping stattfindet und das externe Modell in der Domäne verwendet wird? *Abbildung 11* zeigt dies anhand der Verwendung von Fahrzeugdaten (*FahrzeugDTO*), die bei einem externen Service abgefragt werden. Werden die Daten in den Komponenten Fahrzeug und Rechnung ohne Adaption verwendet, führen Änderungen im externen Service zu vielen Änderungen in der Domäne. Wird ein Mapping implementiert, verursachen Änderungen im externen Service keinen Anpassungsbedarf in der Domäne.

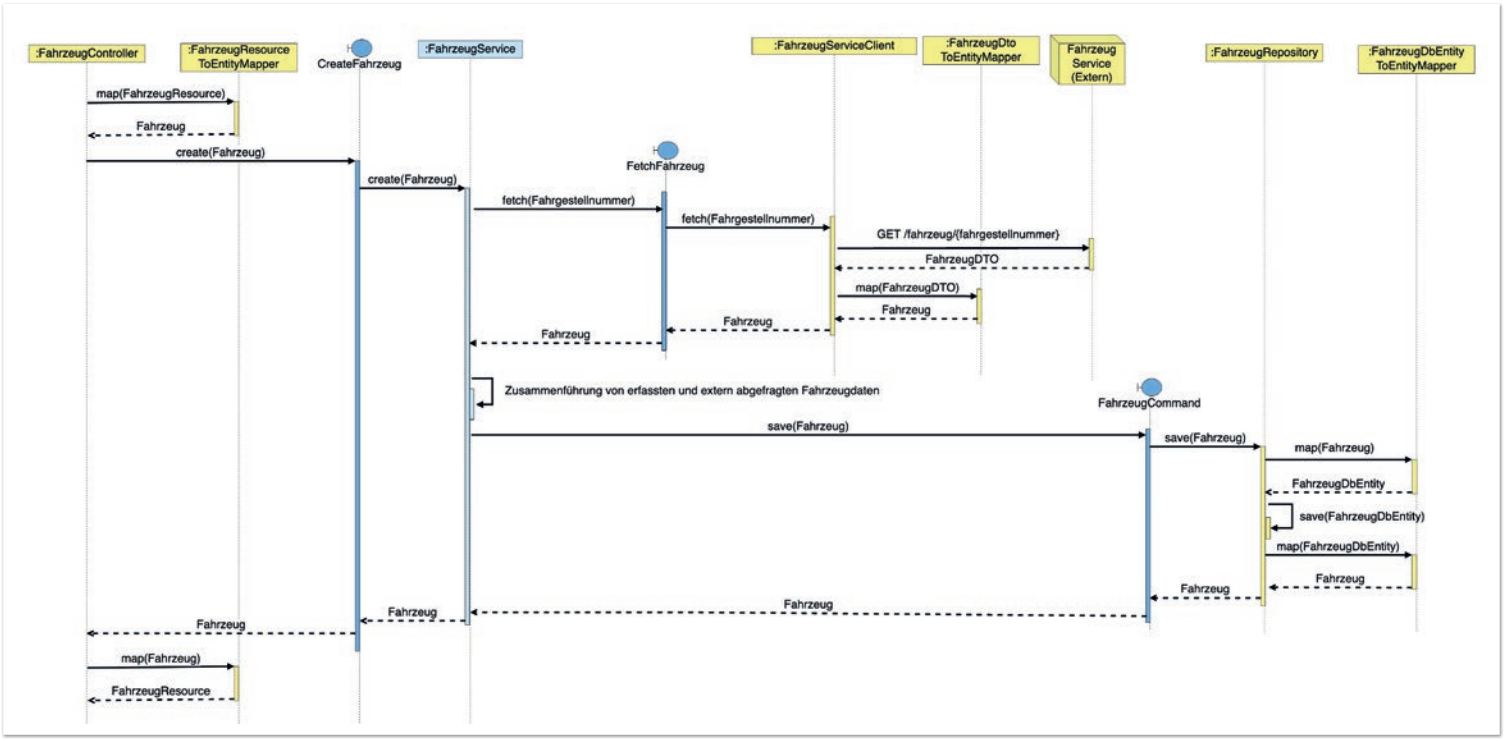


Abbildung 13: Laufzeitsicht Use Case „CreateFahrzeug" mit der Two-Way-Mapping-Strategie (© Matthias Eschhold)

Der Anpassungsbedarf ist ausschließlich im Adapter. Der Adapter stellt folglich einen „Single Point of Change" dar, der die Stabilität des Systems deutlich erhöht. Dies ermöglicht eine sehr schnelle Reaktion auf Veränderung und reduziert erneut Testaufwände.

Ein Mapping zwischen Objekten der Domäne und Datenobjekten der Infrastruktur-komponenten wird als Two-Way-Mapping bezeichnet. Durch das Mapping wird die Mustersprache um das Stereotyp *Mapper* ergänzt. Dies und den *ServiceClient* als weiteren Adapter zeigt *Abbildung 12*.

Neben der Two-Way-Mapping-Strategie beschreibt Hombergs auch die Full-Mapping-Strategie. Diese wird benötigt, wenn *Use Cases* ihr eigenes Modell (*Use Case Model*) verwenden. Hier muss ein Mapping zwischen Entität und *Use Case* sowie zwischen *Use Case* und Adapter realisiert werden *[11]*.

Abbildung 13 zeigt die Laufzeitsicht des Use Case *CreateFahrzeug*. Dieser wird ausgelöst vom *FahrzeugController*, der ebenfalls die *FahrzeugResource* auf die Entität *Fahrzeug* überführt. Im *FahrzeugService* werden Daten anhand der im *Fahrzeug* übergebenen *Fahrgestellnummer* von einem externen Service über den Use Case FetchFahrzeug abgerufen. Der *FahrzeugServiceClient* adaptiert die *FahrzeugDTO* auf die Domäne. Zuletzt wird das *Fahrzeug* über das *FahrzeugCommand* und das *FahrzeugRepository* persistiert.

Fazit und Ausblick

Die Basis für Flexibilität, also die Clean Architecture, ist nicht einfach in der Umsetzung. Sie denkt die Beziehungsrichtungen neu und kombiniert Ports und Adapters, Dependency Inversion, Dependency Injection und Interface Segregation. Auch denkt die Clean Architecture fachlich und stellt die Entität und die Domäne ins Zentrum der Architektur. Darauf aufbauend können konzeptionell gleichartige Fachmodule auf Basis einer Mustersprache entstehen. Dies führt zu einer verständlichen Architektur, in der Domäne und Infrastruktur klar voneinander getrennt sind. Stabilität und lose Kopplung verhilft der Architektur zu einer schnellen Reaktionszeit bei Veränderungen.

Es wird deutlich, dass die Anwendung der Clean Architecture einen höheren Aufwand erfordert. Ist der Wunsch nach Flexibilität vorhanden, sollte damit ein Nutzen verbunden sein. Dieser Nutzen muss durch Flexibilität erreicht werden. Was Nutzen und Flexibilität auf dem nächsten Level bedeuten, wird in Teil 2 thematisiert. Teil 3 sucht die Balance zwischen Aufwand und Nutzen. Der differenzierte Umgang mit Bausteinen, die flexibel sein müssen und halt den anderen , erfordert Flexibilität im Lösungsraum der Architektur und pragmatische Ansätze für die Zielerreichung bei Minimierung der Aufwände.

Quellen

[1] Mahbouda Gharb et al. (2019): Basiswissen für Softwarearchitekten. dpunkt.verlag, Heidelberg.

[2] Eberhard Wolff (2016): Microservices – Grundlagen flexibler Softwarearchitekturen. Dpunktverlag, Heidelberg.

[3] Gernot Starke und Peter Hruschka (2016): Kolumne: Knigge für Softwarearchitekten Softwarearchitektur. Online. *https://entwickler.de/software-architektur/kolumne-knigge-fur-softwarearchitekten-002*

[4] Stefan Tilkov (2021): Software-Architektur-Trends 2021: „Es wird immer wichtiger, dass Software sich schnell verändern kann". Online. *https://entwickler.de/software-architektur/software-architektur-trends-2021-es-wird-immer-wichtiger-dass-software-sich-schnell-verandern-kann*

[5] Bruce Johnson et al. (2005): Flexible Software Design. Auerbach Publications, London.

[6] Oliver Vogel et al. (2009): Software-Architektur: Grundlagen – Konzepte – Praxis. Spektrum, Heidelberg.

[7] Robert C. Martin (2012): The Clean Code Blog: The Clean Architecture. Online. *https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html*

[8] Alistair Cockburn (2021): Hexagonal architecture. Online. Abgerufen am: 10.12.2021. *https://alistair.cockburn.us/hexagonal-architecture/*

[9] Gernot Starke (2015): Effektive Softwarearchitekturen. Hanser Verlag. Heidelberg.

[10] Gernot Starke et al. (2010): Patterns Kompakt. Spektrum, Heidelberg.

[11] Tom Homsberg (2019): Get Your Hands Dirty on Clean Architecture. Packt Publishing, Birmingham.

[12] Muhammad Ali Babar et al. (2014): Agile Software Architecture. Morgan Kaufmann, Waltham.

Matthias Eschhold
Novatec Consulting GmbH
matthias.eschhold@novatec-gmbh.de

Warum ist es wichtig, dass Softwarearchitektur fachlich und sauber strukturiert ist? Seit 2014 beantwortet Matthias Eschhold in seinem beruflichen Alltag diese und weitere Fragen zum Architekturentwurf sowie zu Architekturmitteln, Entwurfsprinzipien und -mustern. Sein schönstes Erfolgserlebnis dabei: zu sehen, dass die getroffenen Maßnahmen und Entscheidungen wirken und Ziele wie Wartbarkeit, Verständlichkeit und Flexibilität erreicht werden. Darüber hinaus vermittelt Matthias sein Wissen leidenschaftlich als Trainer für Softwarearchitektur.

Rule Engines – Was? Wie? Wo?

Merlin Bögershausen, Adesso SE

Rule Engines sind ein oft unterschätztes Werkzeug, um kunden- oder kontextspezifische Anforderungen in Produkte zu integrieren. In diesem Artikel werde ich zeigen, wie Rule Engines auf der Ebene der System- und Softwarearchitektur verwendet werden können.

Seit Anbeginn der Zeit ist der Mensch bestrebt, seinen Arbeitsaufwand so klein wie möglich zu halten und dabei den größten Erfolg zu erzielen. Für die Softwareindustrie besteht das Optimum dieser Bestrebung darin, ein Produkt nur einmal zu entwickeln und an die ganze Weltbevölkerung zu verkaufen. Gerade in unserer digitalisierten Welt kommen wir mit Basisprodukten und Kundenanpassungen diesem Optimum sehr nahe. Viele Softwarefirmen versuchen deswegen, ihr Portfolio auszudünnen und auf spezifizierte Produkte zu setzen. Jedoch hat nicht jeder Kunde dieselben Anforderungen, was der Grund für die gängigen Individuallösungen ist. Es gibt mehrere Möglichkeiten, mit diesen kundenspezifischen Anforderungen umzugehen. Eine Lösung möchte ich in diesem Artikel vorstellen und zeige die Erweiterung von Softwareprodukten mit einer Rule Engine.

Was sind Rule Engines?

Rule Engines wurden in den 80ern als sogenannte Expertensysteme von Pegasystems, Fair Isaac Corp und ILOG entwickelt *[1]*. Damals war das Ziel der Entwicklung eine stetige Verfügbarkeit über das Wissen der knappen Ressource „Experte".

Ein Experte ist in diesem Kontext eine Person, die in der fachlichen Domäne tief verwurzelt ist und auf möglichst viele Fragestellungen eine Antwort geben kann. Eine Fachkraft oder eine Gruppe von Fachkundigen stellt Verhaltensregeln auf Basis des eigenen Wissens auf. Regeln sind hierbei eine Einheit, bestehend aus einer Vorbedingung und einer Aktion. Wenn die Vorbedingungen zutreffend sind, so werden die Aktionen ausgeführt. Gibt es im Prozess nun einen Punkt, der eine Entscheidung benötigt, die weitreichenderes Wissen und Erfahrung voraussetzt, so wird das Expertensystem gefragt. Dieses wendet die ihm bekannten Regeln auf die Eingabedaten an und berechnet eine Reaktion, die ausgeführt wird. Das ermöglicht, Arbeitsvorgänge durchzuführen, auch wenn die seltene Ressource „Experte" nicht vorhanden ist.

Aus diesem Ansatz haben sich in den letzten Jahrzehnten die Rule Engines, oder auf Deutsch regelbasierten Systeme, entwickelt. Moderne Rule Engines können hervorragend verwendet werden, um Anforderungen in einem System umzusetzen, ohne das Produkt verändern zu müssen. Der Einsatz ermöglicht ebenfalls eine Weiterentwicklung des Produkts ohne Einflussnahme auf die Kundenanpassungen.

Funktionsweise von Rule Engines

In *Abbildung 1* ist der Aufbau beziehungsweise die Funktionsweise von Rule Engines zu sehen. Von links wird die Datenbank der Rule Engine – die sogenannte Knowledge Base – von einem Experten mit Regeln befüllt. Diese Regeln folgen der Form: „*Wenn X, dann tue Y.*" Diese Regeln können auf verschiedenen Wegen formuliert werden. Von der Implementierung eines Interface, über eine Domain-specific Language bis hin zu einer Excel-basierten Entscheidungstabelle erhielt alles Einzug in die Welt der Rule Engines. Welche Art und Weise gewählt wird, hängt von dem Projektkontext und dem Einsatzzweck ab.

Auf der rechten Seite ist eine Arbeitskraft oder ein computergestützter Prozess zu sehen. Diese Arbeitskraft hat eine Problemstellung, für die eine Aktion notwendig ist. Mit dieser Situation wendet sie

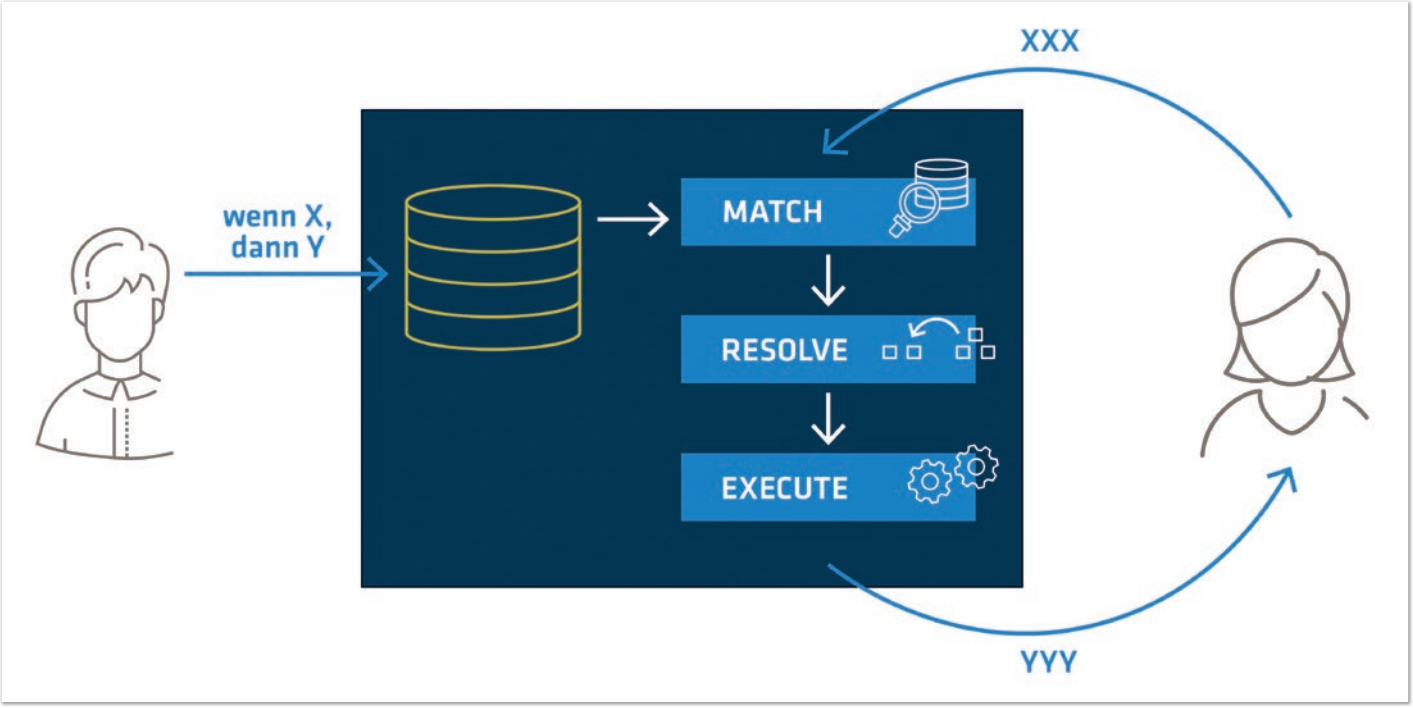


Abbildung 1: Die Rule Engine (dunkelblau) wählt zu jeder Frage des Benutzenden (rechts) Regeln aus, die der Experte (links) formuliert hat. Er bewertet die Reihenfolge und führt die Regeln aus, um eine Antwort zu generieren. (© Merlin Bögershausen, Adesso SE)

sich an die Rule Engine. Die Situation wird in Facts überführt. Vereinfacht gesagt sind Facts eine Sammlung von Key-Value-Paaren, die die aktuelle Situation beschreiben. Auf Basis dieser Facts wird das System eine Auswahl treffen und die entsprechenden Aktionen ausführen. Dazu durchläuft jede Rule Engine den gleichen Prozess. In den performanten Implementierungen dieser Prozessschritte unterscheiden sich gute von schlechten Rule Engines – dies soll in diesem Artikel aber nicht Thema sein.

Der Prozess gliedert sich in folgende Schritte:

1. **Match**: In diesem Schritt werden die Facts gegen die bekannten Vorbedingungen gematcht und die auszuführenden Aktionen gewählt.
2. **Resolve**: Hier wird zu den ausgewählten Aktionen eine Ausführungsreihenfolge erarbeitet.
3. **Execute**: Im letzten Schritt werden die ausgewählten Aktionen entsprechend der Ausführungsreihenfolge ausgeführt.

Die im Schritt Execute berechnete Antwort wird an die Arbeitskraft beziehungsweise die umgebenden Prozesse zurückgegeben oder als neue Eingabe für den Match-Schritt genutzt. Eine Rule Engine kann als konvergent konfiguriert sein, sodass der Prozess wiederholt angestoßen wird, bis im Match-Schritt keine Aktion mehr gefunden wird. Alternativ ist die Rule Engine derart konfigurierbar, dass sie sowohl eine feste Anzahl Iterationen wie auch eine einzige Iteration durchführen kann (iterative beziehungsweise single shot). Welche Einstellung die richtige ist, hängt von der Verwendung ab.

Mit der Zeit haben sich unterschiedliche Ausprägungen von Rule Engines gebildet. Die Bandbreite reicht von sehr Source-Code-nahen Rule Engines bis hin zu eigenständigen Deployments. In diesem Artikel habe ich sie anhand ihrer Zielgruppe aufgeteilt und behandelt.

Developer Rule Engines

Als Developer Rule Engine bezeichne ich solche, die in das Softwareprodukt integriert werden und fester Bestandteil dessen Source-Codes sind. Prominente Vertreter aus der Java-Welt sind JEasy EasyRules *[1]* oder RuleBook *[2]*. Beide Rule Engines sind komplett in Java geschrieben und als JAR einzubinden. Beide Engines unterstützen Definitionen von Regeln durch Annotationen an POJOs oder über ein Builder-Pattern und sind simpel in Enterprise-Anwendungen zu integrieren. Darüber hinaus bietet EasyRules auch die Möglichkeit, Regeln in YAML zu formulieren.

Developer Rule Engines werden vor allem auf Ebene der Softwarearchitektur verwendet. Hier werden sie eingesetzt, um die kundenspezifischen Anforderungen von denen des Basisprodukts zu trennen oder abhängig vom Kontext ein anderes Verhalten zu erzeugen. Der Service des Produkts wird um eine Instanz einer Rule Engine erweitert. Diese wird zur Ausführungszeit mit Regeln versorgt.

In *Listing 1* ist minimalistisch dargestellt, wie die Nutzung einer Developer Rule Engine auf Ebene der Softwarearchitektur möglich ist. In folgendem Beispiel wird ein Spring-Service im Kontext des Produkts um kundenspezifische Regeln erweitert. Der ProductService bekommt hierzu über den CDI-Kontext eine Liste von Regeln. Bei einem Aufruf von ProductService#addEntry wird zuerst die Basislogik ausgeführt und danach die Rule Engine mit den kundenspezifischen Regeln aufgerufen. Die kundenspezifischen Regeln werden über eine Spring-Konfiguration für jeden Kontext separat konfiguriert. Im vorliegenden Beispiel wird in der CustomerConfig definiert, dass für jeden neuen Customer eine Log-Zeile geschrieben werden soll. Durch das Austauschen der Konfiguration können so verschiedene Rule Sets an derselben Stelle ausgeführt werden.

```
// product artifact
@Service
public class ProductService {
  @Autowired
  private List<Rule<Entry>> addRuleSet;
  public void addEntry(Entry newEntry) {
     // do product add logic
     RuleEngine.execute(addRuleSet, newEntry);
  }
}
// customer solution
@Configuration
public class CustomerConfig {
  @Bean
  public List<Rule<Entry>> addRuleSet() {
     var loggingRule = new LogNewCustomerRule();
     return List.of(loggingRule);
  }
}
```

Listing 1: Simples Beispiel, wie eine Rule Engine auf Ebene der Softwarearchitektur mithilfe von Spring integriert werden kann. Dem ProductService können verschiedene Rule Sets über die Konfiguration bereitgestellt werden.

Durch den Einsatz einer Developer Rule Engine können nun alle kundenspezifisch-fachlichen Anforderungen aus dem ProductService gelöst werden. Hierdurch lässt sich der ProductService einfacher verstehen und testen. Auch die kundenspezifischen Regeln sind einfach testbar und leicht verständlich. Nachteilig ist die schwere Verständlichkeit des Programmflusses bei der Einführung einer Developer Rule Engine.

Business Rule Engines

Eine Business Rule Engine ist dem klassischen Expertensystem am ähnlichsten und lässt sich am besten auf der Ebene der Systemarchitektur ansiedeln. Die Rule Engines werden meistens eigenständig deployt und über einen Message-Bus oder über REST angesprochen. Weiterhin können sie komplett entkoppelt arbeiten und Datenbanktabellen überwachen. Red Hat Drools *[3]* und OpenL Tablets *[4]* sind zwei beispielhafte Vertreter von Business Rule Engines. Beide bieten die Möglichkeit, über eine komfortable Weboberfläche oder Excel-Import Regeln einzuspielen. Die Oberflächen sind für Businessanwender konzipiert und erinnern an das Schema der Entscheidungstabellen aus der Entscheidungslehre.

Abbildung 2 zeigt eine einfache Möglichkeit, eine Business Rule Engine in ein System mit Message Bus zu integrieren. Blaue Pfeile deuten hierbei die primäre Laufbahn von Daten und Requests an. Das Aktualisieren einer Entity ist ein Beispiel und wird von einem der Services ausgeführt. Nach der erfolgreichen Aktualisierung des Zustands in der Datenbank wird der Rest des Systems über die Än-

derung informiert. Diese Nachricht (grüne Pfeile) wird von der Business Rule Engine zum Anlass genommen, die aktuelle Situation zu bewerten und gegebenenfalls Aktionen auszuführen.

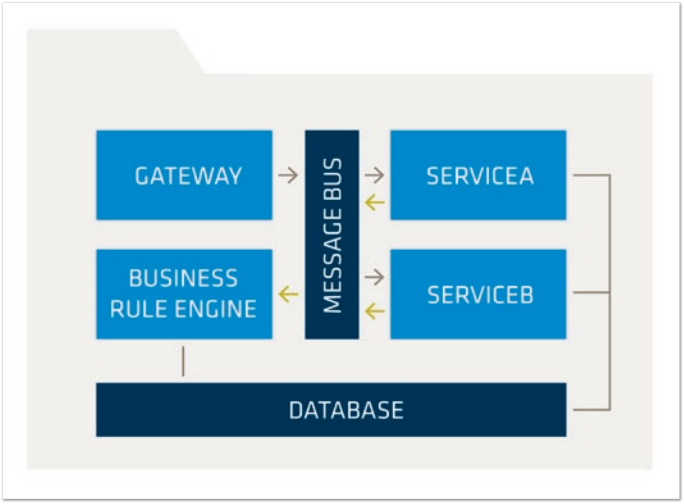


Abbildung 2: Möglicher Aufbau eines Systems mit einer Business Rule Engine. Blaue Pfeile deuten die Laufbahn von Requests an, grüne Pfeile den Verlauf von Events (© Merlin Bögershausen, Adesso SE)

Um eine Business Rule Engine in eine bestehende Architektur zu integrieren, bedarf es dementsprechend nur einer Anbindung an die bestehende interne Kommunikation. Die bereits implementierten und getesteten Services müssen nicht oder nur in geringem Maße angepasst werden. Es werden keine neuen Abhängigkeiten in das System eingefügt. Wegen der einfachen Definition von Regeln ist es sogar Nicht-Entwickelnden möglich, das Verhalten des Systems zu beeinflussen. Die Last auf das Entwicklungsteam verringert sich hierdurch. Nachteilig ist die schwierige Testbarkeit der erstellten Regeln in der Business Rule Engine. Die zusätzliche Komponente steigert die Komplexität des Systems. Es wird schwer zu verstehen, was wann wo passiert.

Fazit

Aus den gegebenen Beispielen wird klar, wie Rule Engines die Antwort auf die Frage „Wie bekomme ich kundenspezifische Anforderungen und Produktgeschäft unter einen Hut?" sein können. Rule Engines wirken sich also positiv auf die Weiterentwickelbarkeit und Wartbarkeit eines Systems aus. Es wird aber auch klar, dass der Einsatz von Rule Engines nicht ohne Nachteile ist. Sie bringen neue Komplexitäten und Abhängigkeiten in das System. Die Testbarkeit ist, abhängig von der gewählten Rule Engine, mal positiv und mal negativ beeinflusst.

Die Fragen nach der Performance von Rule Engine und danach, wann es Sinn ergibt, selbst eine Rule Engine für das eigene Projekt zu entwickeln, konnten aufgrund des Umfangs nicht behandelt werden. Ebenfalls nicht behandelt wurden die Fallstricke und theoretischen Grundlagen, was für sich genommen jeweils Material für weitere Artikel bietet.

Ich hoffe, euch einen guten Überblick über den möglichen Einsatz von Rule Engines gegeben zu haben. Kontaktiert mich gerne für Fragen und Anregungen.

Quellen

[1] *https://www.infoworld.com/article/2641011/rules-engines-and-soa.html*
[2] *https://github.com/j-easy/easy-rules*
[3] *https://github.com/deliveredtechnologies/rulebook*
[4] *https://www.drools.org*
[5] *http://openl-tablets.org*

Merlin Bögershausen
Adesso SE
Merlin.Boegershausen@adesso.de

Merlin ist Software Engineer und Speaker bei Adesso SE in Aachen mit dem Schwerpunkt Java. Er war in mehreren Stationen an der Entwicklung und Anpassung von Software-Produkten beteiligt. Mit seinen gewonnenen Erfahrungen möchte er anderen Entwicklern helfen, ihren Werkzeugkasten zu erweitern, um immer das richtige Werkzeug zur Hand zu haben. Abseits der Arbeit ist er leidenschaftlicher Segelflieger, Familienvater und Urban-Gardener.

Von Null auf CQRS/ES mit Event Modeling und Axon

Frank Steimle, Nikolai Neugebauer, Digital Frontiers GmbH & Co. KG

Immer noch werden klassische Schichtenarchitekturen eingesetzt, obwohl sie aktuellen Anforderungen nicht gerecht werden können. In diesem Artikel beleuchten wir die Vorteile einer modernen CQRS/ES-basierten Architektur. Wir zeigen, wie eine CQRS/ES-basierte Anwendung mithilfe von Event Modeling modelliert und entworfen werden kann und wie einfach diese mit dem Axon-Framework auf der JVM realisiert werden kann.

Schichtenarchitekturen sind eine seit vielen Jahren bewährte Abstraktion, um Softwareprojekte umzusetzen. Entwickler greifen gerne auf sie zurück, da sie sich vertraut anfühlen und man auf viele Erfahrungen zurückgreifen kann. Jedoch sind die Rahmenbedingungen heute typischerweise anders als zu der Zeit, als Schichtenarchitekturen erdacht wurden: Die Entwicklung findet meist agil statt und Produkte werden inkrementell entwickelt.

Eine große Herausforderung für Schichtenarchitekturen ist die Weiterentwicklung. Im Falle einer komplexen Fachlogik und eines dementsprechend komplexen Modells kann auch die kleinste Erweiterung des Fachmodells zu weitreichenden Änderungen in der Code-Basis führen.

Neue Anforderungen können auch neue Schnittstellen beinhalten, die initial nicht bedacht wurden. Im Extremfall führt die neue Schnittstelle bei jedem Aufruf teure Aggregation durch und macht so den Code schwerer verständlich und wartbar. Schlechtestenfalls können neue Anforderungen sogar eine komplette Überarbeitung des Datenmodells erfordern, wenn zum Beispiel die Änderungshistorie mitgespeichert werden soll.

Diesen Herausforderungen können wir mit der Kombination aus CQRS und ES begegnen.

Was ist eigentlich CQRS/ES?

Command-Query-Responsibility-Segregation (CQRS) *[1]* ist ein Architekturmuster, dessen Kernidee die strikte Trennung von schreibenden und lesenden Anfragen ist. Insbesondere bedeutet dies, dass schreibende Anfragen keine Daten zurückliefern. Schreibende Anfragen werden „Commands" genannt und lesende „Queries". Commands und Queries werden dabei von unterschiedlichen Datenmodellen verarbeitet. Für die Bearbeitung der Commands existiert ein Schreibmodell und zum Beantworten der Queries existieren ein oder mehrere Lesemodelle.

Wenn ein Client mit der Schnittstelle eines CQRS-basiertes System interagiert, erzeugt die Schnittstelle, je nach Operation, Commands und Queries. Empfängt das System ein Command , prüft ein sogenannter Command Handler, in dem die jeweilige Fachlogik implementiert ist, ob das Command alle benötigten Daten enthält und im aktuellen Zustand ausgeführt werden kann. Wenn es ausgeführt werden kann, werden zuerst Änderungen in den Daten persistiert und anschließend ein oder mehrere Events erzeugt (emittiert), die die Änderungen am Datenbestand abbilden. Zuletzt bestätigt der Command Handler gegenüber dem Client die erfolgreiche Ausführung des Command. Event Handler aktualisieren dann asynchron die Lesemodelle. Dieser Ablauf ist in *Abbildung 1* durch grüne Pfeile dargestellt. Falls das Command nicht ausgeführt werden kann, gibt der Command Handler eine Fehlermeldung zurück. In diesem Fall werden keine Änderungen geschrieben und damit auch keine Events erzeugt.

Wenn ein anderer Client lesend auf das System zugreifen möchte, sendet er eine Anfrage an die Schnittstelle, die sie wiederum in ein Query-Objekt transformiert und an einen Query Handler weiterleitet. Er greift nun auf die Lesemodelle zu, die zum Beantworten der Query nötig sind, fragt dort Daten ab und gibt sie zurück *(siehe Abbildung 1, lila Pfeile)*. Da die Verarbeitung von Events asynchron erfolgt, haben Query Handler nicht sofort nach einer vollzogenen Änderung am Schreibmodell Zugriff auf die geänderten Daten, sondern erst nach einer gewissen Zeit (Eventual Consistency).

Ein wesentlicher Vorteil von CQRS ist, dass es spezialisierte Lesemodelle unterstützt. Dadurch kann beispielsweise pro Anwendungsfall ein Modell erstellt werden, um diesen durch eine geeignete und spezialisierte Datenstruktur zu optimieren. Beispielsweise

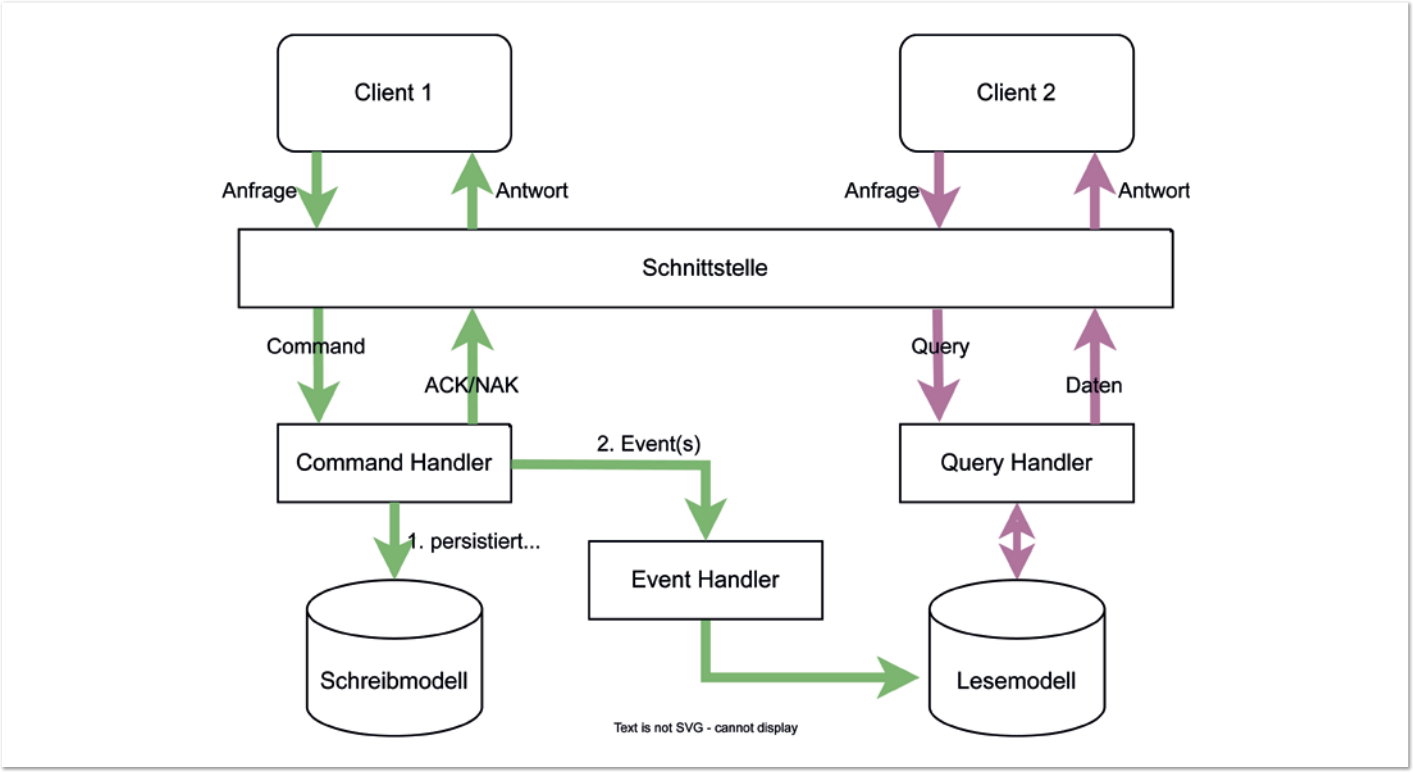


Abbildung 1: Komponenten und Datenfluss einer CQRS-basierten Anwendung (© Nikolai Neugebauer, Frank Steimle)

lässt sich für ein Online-Auktionshaus ein Lesemodell erstellen, das pro Auktion nur das aktuelle Höchstgebot enthält, und ein weiteres Lesemodell, das alle Gebote für eine Auktion beinhaltet. Ersteres wird benötigt, um Interessenten in einer Übersicht zu zeigen, welche Gegenstände zur Auktion stehen und wie hoch das aktuelle Höchstgebot ist. Letzteres kann auf der Detailseite verwendet werden, um eine Übersicht über alle abgegebenen Gebote anzuzeigen. Durch die Existenz dedizierter Lesemodelle ist es, im Gegensatz zur klassischen Architektur, auch möglich, gezielt einzelne Teile der Anwendung zu skalieren.

Zudem ist die Fachlogik in einer CQRS-basierten Architektur meist leichter erweiterbar, da sie zentralisiert an einer Stelle, dem Command Handler, vorliegt. Um die Fachlogik zu erweitern, genügt es, den jeweiligen Command Handler anzupassen und gegebenenfalls neue Events einzuführen. Diese Zentralisierung sorgt zusätzlich für eine gute Testbarkeit der Fachlogik.

Natürlich sind die Vorteile von CQRS auch mit Kosten verbunden. Die Aufteilung in Commands, Queries und Events erfordert ein Umdenken. Dieses zahlt sich jedoch aus, da so eine von technologischen Abhängigkeiten befreite Fachlogik entsteht, die sehr gut testbar ist. Des Weiteren muss ein Entwickler bedenken, dass die Lesemodelle unter Verwendung von Eventual Consistency aktualisiert werden. Da viele Systeme heutzutage jedoch als verteilte, Cloud-native Anwendung entwickelt werden und damit von Natur aus „eventually consistent" sind, fällt dieser Punkt nicht so sehr ins Gewicht. Die größere Anzahl von Lesemodellen bedeutet auch eine größere Code-Basis, die der Entwickler überblicken muss. Dieser Nachteil wird allerdings in Teilen durch die Partitionierung der Lesemodelle und damit ihre Vereinfachung wieder aufgehoben.

Während CQRS theoretisch mit vielen Formen der Persistenz umgesetzt werden kann, wird es in der Praxis meist mit Event Sourcing (ES) *[2]* kombiniert. Event Sourcing ist eine Form der Persistenz, bei der nicht der aggregierte Ist-Zustand der Daten gespeichert wird, sondern eine chronologische Liste der Änderungen. Dies hat den Vorteil, dass das Schreiben eines Events schneller geht als das Aktualisieren einer kompletten Entität. Außerdem ist jede Änderung am Datenbestand nachvollziehbar. Das Lesen aus dem Event-Store ist allerdings teurer, da jedes zu einer bestimmten Entität gehörende Event wieder eingelesen („Sourcing") werden muss, um den aktuellen Zustand zu erhalten. Durch die Kombination mit CQRS fällt dieser Nachteil allerdings nicht so sehr ins Gewicht, da das aufwendige Sourcing nur im selteneren Fall der Verarbeitung eines Command durchgeführt werden muss.

Die Kombination von CQRS mit ES ermöglicht es, beliebige neue Lesemodelle einzuführen und sie mithilfe der gesamten Event-Historie zu befüllen. Zusätzlich lässt sich durch ES jede Anforderung an Auditierbarkeit erfüllen. Bestimmte Funktionalitäten, beispielsweise Dashboards, profitieren ebenfalls davon, wenn sie nicht nur auf den aktuellen, aggregierten Datenbestand, sondern auch auf historische Daten zugreifen können, etwa um eine Replay-Funktionalität anbieten zu können.

CQRS/ES-basierte Architekturen sind damit den gängigen Herausforderungen im agilen, Cloud-basierten Umfeld gewachsen. Sie sind keineswegs nur für exotische Anwendungsfälle geeignet, sondern stellen eine ernstzunehmende und sinnvolle Alternative zu den etablierten Schichtenarchitekturen dar.

Event Modeling

Klassische Modellierung basiert auf aggregierten Zuständen, wobei implizit auch Abläufe modelliert werden. Eine CQRS/ES-basierte Architektur sieht die Verwendung eines zustandsbasierten Schreibmodells jedoch nicht vor. Durch den Fokus auf Events macht sie es erforderlich, die bisher implizit modellierten Abläufe in den Vordergrund zu stellen. Für die Modellierung einer CQRS/ES-basierten Anwendung ist es daher wichtig, eine passende, ablaufbasierte Form zu nutzen.

Eine Methode, um ablaufbasierte Anwendungen zu modellieren, ist Event Modeling *[3]*. Dabei erarbeiten Entwickler, Product Owner, Fachabteilung und alle weiteren Projektbeteiligten gemeinsam die Abläufe und Funktionalität der Software. So entsteht bereits früh ein gemeinsames Verständnis für die geplante Anwendung. Außerdem entwickeln alle Beteiligten dabei eine gemeinsame Sprache, mit der sie Fragestellungen der Domäne diskutieren können. Insbesondere für CQRS/ES-basierte Anwendungen ist Event Modeling interessant, da in beiden Fällen Events und Commands elementare Konzepte sind.

In einem Event-Modeling-Workshop erkunden die Teilnehmer kollaborativ die zu modellierende Domäne. Er besteht aus mehreren Phasen, in denen das Modell iterativ um Details erweitert wird. Dadurch wird das erarbeitete Modell mehrfach auf den Prüfstand gestellt und verfeinert. Das interaktive Format fördert dabei in starkem Maße Diskussionen und Wissenstransfer. Im Folgenden werden die verschiedenen Phasen eines Event-Modeling-Workshops am Beispiel des Online-Auktionshauses durchgeführt.

Die erste Phase im Event Modeling beschäftigt sich mit der Identifikation der fachlich relevanten Events. Dazu notieren die Teilnehmer Events, die ihnen einfallen, auf orange Post-its und heften sie an ein Whiteboard. Da wir mit diesen Events bereits geschehene Ereignisse modellieren, verwenden wir Namen in der Vergangenheitsform. Während dieser Phase werden oft Events mit ähnlichen Namen oder fachlich nicht-relevante Events erfasst. Die Bereinigung der Events ist Teil der nächsten Phase. Ein Ausschnitt aus Phase 1 zu unserer Beispieldomäne könnte diese Events umfassen:

- User wurde eingeloggt
- Gebot abgegeben
- Auktion gestartet
- Angebot erstellt

Abbildung 2: Whiteboard nach Abschluss der zweiten Phase, zeitlich geordnete Events (© Nikolai Neugebauer, Frank Steimle)

In Phase 2 werden die identifizierten Events dann in einen zeitlichen Ablauf gebracht. Dazu werden die Events auf dem Whiteboard entlang eines Zeitstrahls angeordnet. Dabei diskutieren die Teilnehmer auch die erfassten Events, wodurch sich eine Terminologie herauskristallisiert, die die Grundlage für die gemeinsame Sprache ist. Hierbei werden auch Duplikate zwischen Events sowie Zusammenhänge und Vorbedingungen sichtbar. So ist in unserem Beispiel „Angebot erstellt" ein Duplikat von „Auktion gestartet". Ebenfalls werden Events aussortiert, die nicht Teil der fachlichen Domäne sind. Ein Beispiel hierfür ist das Event „User wurde eingeloggt", da dieses nicht notwendig ist, um die Fachlogik einer Auktion abzubilden. Nach Phase 2 könnte ein Ausschnitt des Whiteboards so aussehen, wie in *Abbildung 2* dargestellt.

Häufig existieren in einer Anwendung verschiedene Rollen, die verschiedene Benutzeroberflächen benötigen. In Phase 3 werden diese Nutzerrollen und ihre Oberflächen identifiziert, erfasst und eingeordnet. Hierfür stellt man sich für jedes Event die Frage: „Wer verursacht dieses Event durch eine Interaktion mit der Anwendung?" Im Beispiel des Online-Auktionshauses sind beispielsweise die Rollen des Verkäufers und des Bieters identifiziert worden. Ein Verkäufer benötigt unter anderem eine Oberfläche „Auktion erstellen", um das Event „Auktion gestartet" zu verursachen. Grafisch werden die Rollen und zugehörige Oberflächen in eigenen Reihen über den Events angeordnet. Dazu werden einfache Platzhalter verwendet. Sie dienen zur Veranschaulichung, welche Information an welcher Stelle der Anwendung benötigt wird. In *Abbildung 3* ist das Beispiel um Rollen und Oberflächen ergänzt.

In der folgenden Phase 4 wird die Beziehung zwischen Oberfläche und Events explizit gemacht. Ein Nutzer interagiert mit der Anwendung, um den Zustand der Daten zu ändern, das heißt, bei der Interaktion mit der UI hat der Nutzer Änderungsabsichten. Diese werden in Form von Commands ausgedrückt. Ein erfolgreich verar-

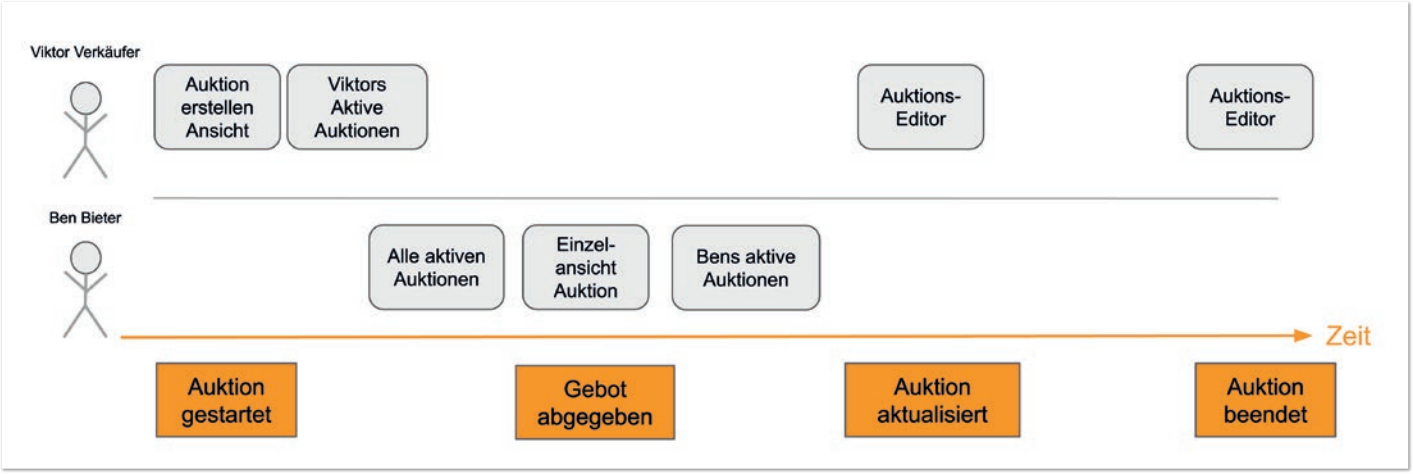


Abbildung 3: Whiteboard nach Abschluss der dritten Phase des Event-Modeling-Workshops (© Nikolai Neugebauer, Frank Steimle)

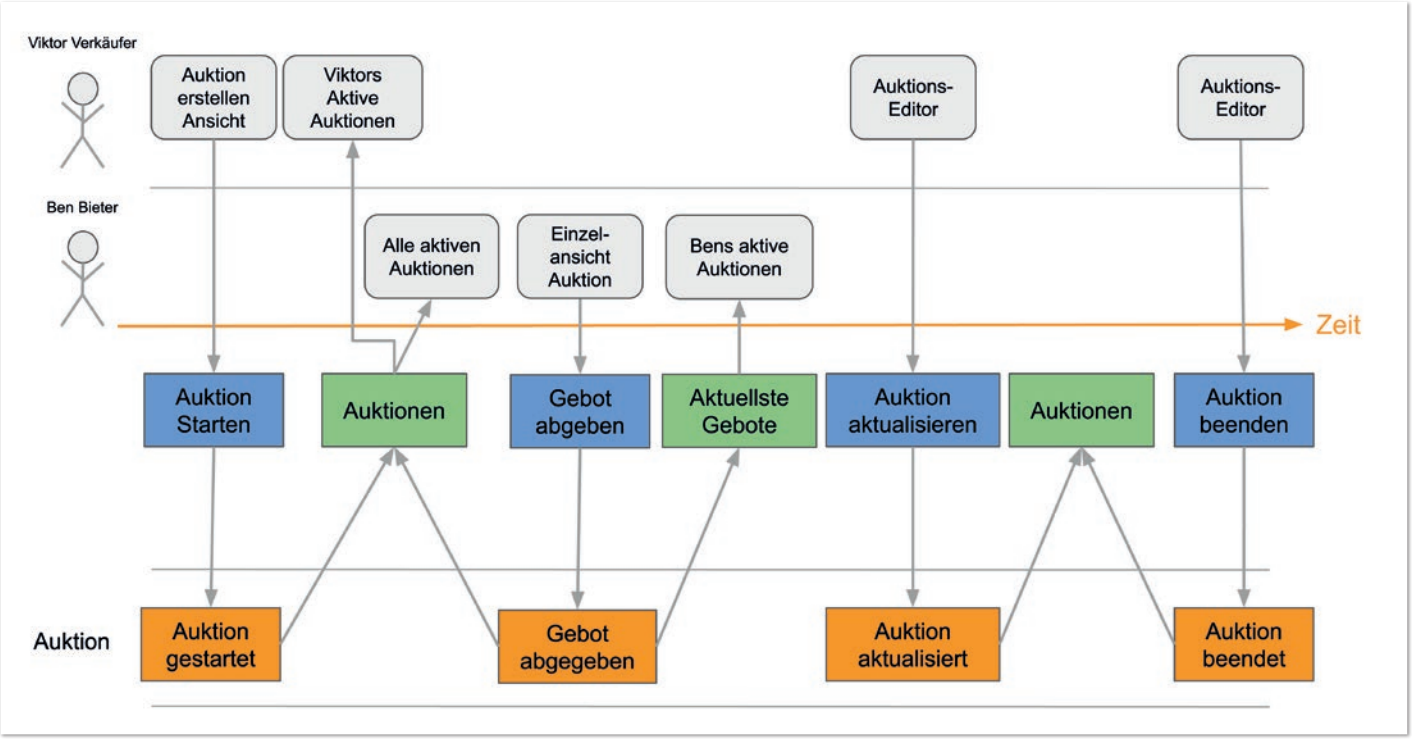


Abbildung 4: Whiteboard nach Abschluss des Event-Modeling-Workshops (© Nikolai Neugebauer, Frank Steimle)

beitetes Command erzeugt ein oder mehrere Events, die die erfolgte Änderung protokollieren. Am Whiteboard werden die Commands auf blauen Post-its erfasst und zwischen Oberflächen und Events platziert. Jede Oberfläche wird durch Pfeile mit den Commands verbunden, die sie auslöst. Alle Commands werden mit Pfeilen mit den Events verbunden, die durch die erfolgreiche Verarbeitung des Command ausgelöst werden. In der Beispiel-Domäne kann ein „Auktion starten"-Command zwischen der „Auktion erstellen"-Ansicht und dem Event „Auktion gestartet" platziert werden.

In Phase 5 wird herausgearbeitet, welche Oberfläche die Informationen welcher Events benötigt. Eine sogenannte View bündelt all diese Informationen für eine bestimmte Oberfläche. Sie konsumiert alle Events, die notwendig sind, um die Informationen zur Verfügung zu stellen. Views werden als grüne Post-its auf das Whiteboard geheftet und alle nötigen Events sowie die entsprechenden Oberflächen damit verbunden. Im Beispiel wird das Event „Auktion gestartet" von einer View „Auktionen" konsumiert, die alle aktiven Auktionen enthält und diese Daten einer Suchfunktion zur Verfügung stellt.

Das Ziel von Phase 6 ist es, die Events so zu gruppieren, dass eine Gruppe alle Events enthält, die zu einer fachlichen Entität gehören. Dies ermöglicht eine Modularisierung der Anwendung und eine Parallelisierung der Entwicklung. Hierfür werden pro Entität eine Zeile

```
data class StartAuctionCommand(
  @TargetAggregateIdentifier val id: UUID,
  val owner: UUID,
  val minPrice: Long,
  val title: String,
  val description: String
)

@Revision("1")
data class AuctionStartedEvent(
  val id: UUID,
  val owner: UUID,
  val minPrice: Long,
  val title: String,
  val description: String
)
```

Listing 1: StartAuctionCommand und AuctionStartedEvent

unter dem bestehenden Schaubild hinzugefügt und die Events, die zur Entität gehören, dort einsortiert. In unserem Beispiel beziehen sich alle Events auf dieselbe Entität, die Auktion. Sie verändert ihren Status durch das Verarbeiten von Commands und emittiert danach Events, die diese Änderungen protokollieren.

Nach Abschluss dieser Phase ist der erste Durchgang eines Event-Modeling-Workshops beendet. In *Abbildung 4* ist ein Ausschnitt

```
@Aggregate
class Auction {

  @AggregateIdentifier
  var id: UUID? = null
  var owner: UUID? = null
  var minPrice: Long? = null
  var title: String? = null
  var description: String? = null

  @CommandHandler
  @CreationPolicy(AggregateCreationPolicy.ALWAYS)
  fun handleStart(cmd: StartAuctionCommand, @Autowired users: UserRepository) {
     if (users.findUser(cmd.owner) != null) {
        AggregateLifecycle.apply(
           AuctionStarted(
              id = UUID.randomUUID(),
              minPrice = cmd.minPrice,
              description = cmd.description,
              title = cmd.title,
              owner = cmd.owner
           )
        )
     } else {
        throw IllegalStateException("User is not known")
     }
  }

  @EventSourcingHandler
  fun on(event: AuctionStartedEvent) {
     id = event.id
     owner = event.owner
     minPrice = event.minPrice
     title = event.title
     description = event.description
  }

}
```

Listing 2: Auction Aggregate mit CommandHandler und EventSourcingHandler

```
@Entity
data class ActiveAuction(
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "id")
  val id: Long? = null,
  val auctionId: UUID? = null,
  val title: String? = null,
  var bestBid: Long = 0L,
)

interface ActiveAuctionsRepository : CrudRepository<ActiveAuction, Long> {
  fun findOneByAuctionId(id: UUID): ActiveAuction
}

@Component
class ActiveAuctionsProjector(val repository: ActiveAuctionsRepository) {

  @EventHandler
  fun on(event: AuctionStartedEvent) {
     repository.save(
        ActiveAuction(auctionId = event.id, bestBid = event.minPrice, title = event.title)
     )
  }

  @EventHandler
  fun on(event: BidAddedEvent) {
     repository
        .findOneByAuctionId(id = event.id)
        .let { it.copy(bestBid = event.bid) }
        .run(repository::save)
  }

}
```

Listing 3: Lesemodell mit Spring Data-JPA

des Ergebnisses für das Online-Auktionshaus zu sehen. Das entstandene Modell ist eine strukturierte Darstellung aller fachlichen Events, der Commands, die sie erzeugen, und der Views, die sie konsumieren.

Implementierung mit dem Axon-Framework

Nach der Modellierung mit Event Modeling kann die Umsetzung beginnen. Im JVM-Umfeld lohnt sich für eine CQRS/ES-basierte Architektur ein Blick auf das etablierte Axon-Framework. Das Axon-Framework ermöglicht die Implementierung Event-getriebener Anwendungen auf Basis von CQRS, ES und Domain-driven Design. Es bietet eine Spring-Boot-Integration und ein intuitives Programmiermodell, dessen Konzepte große Ähnlichkeit mit denen des Event Modeling haben.

Um nun ein Event-Modeling-Modell in eine Axon-basierte Anwendung zu übersetzen, beginnt man mit Commands und Events. Sie können leicht vom Modell in den Code übernommen werden, indem DTO-Klassen erstellt werden. Dabei ist darauf zu achten, dass die Annotation @TargetAggregateIdentifier gesetzt wird. Sie sorgt dafür, dass Axon das Command der richtigen Aggregate-Instanz zuordnen kann. In *Listing 1* sind Kotlin-Beispiele für Commands und Events aus dem Online-Auktionshaus-Beispiel zu finden.

Im nächsten Schritt sollte man die Aggregates anlegen, da sie Methoden besitzen, die die erstellten Commands und Events als Parameter und Rückgabewerte verwenden. Ein Aggregate ist im Kontext des Axon-Frameworks eine Klasse, die mit der @AggregateIdentifier-Annotation versehen ist und ein Attribut besitzt, das mit @AggregateIdentifier annotiert wird *(Vergleich Listing 2)*. Für jedes Command, das von einem Aggregate verarbeitet werden soll, muss ein Command Handler implementiert werden. Das ist eine mit entsprechender Annotation gekennzeichnete Methode, die eine Instanz des Command als Parameter entgegennimmt. Wenn sie aufgerufen wird, erfüllt sie zwei Aufgaben: (1) Prüfen, ob das auszuführende Command im aktuellen Zustand ausgeführt werden kann, und (2) Veröffentlichen der Events, die die durchgeführten Änderungen protokollieren. Dies geschieht durch einen Aufruf von AggregateLifecycle.apply(). Der Command Handler kann auch entscheiden, dass die Ausführung des Command im aktuellen Zustand nicht zulässig ist, und beispielsweise eine Exception werfen.

Die Zustellung des Command an den passenden Command Handler wird von Axon übernommen. Im Fall der Spring-Integration stellt das Framework dazu eine CommandGateway-Bean bereit, das aus beliebigen Teilen der Anwendung aufgerufen werden kann, zum Beispiel aus einem REST-Controller.

Damit der Command Handler überprüfen kann, ob das Command im aktuellen Zustand des Aggregate angewendet werden kann, muss der Zustand in den Arbeitsspeicher geladen werden. Hierbei kommen Event Sourcing Handler zum Einsatz. Für jedes mögliche Event, das von einem Command Handler emittiert wird, muss ein solcher Handler implementiert sein. Er hat eine ähnliche Struktur wie ein Command Handler, eine Methode, die mit der entspre-

chenden Annotation gekennzeichnet ist und eine Instanz des Events als Parameter entgegennimmt. Während der Command Handler nur prüft, ob das übergebene Command angewendet werden kann, nimmt der Event Sourcing Handler die eigentliche Änderung am Aggregate vor.

Events dienen in Axon-basierten Anwendungen nicht nur dazu, per Event Sourcing den letzten Zustand eines Aggregate wiederherzustellen. Sie können auch dazu verwendet werden, beliebige Lesemodelle (beziehungsweise Views aus dem Event Modeling) aufzubauen. Spring-Beans können mithilfe von Event-Handlern auf den Event-Strom zugreifen und ihn verarbeiten. *Listing 3* zeigt ein Read-Modell aus dem Auktionshausbeispiel, das mit einer relationalen Datenbank und Spring Data JPA umgesetzt wurde. Dieses Read-Modell kann beispielsweise auch REST-Controllern zur Verfügung gestellt werden.

Axon bietet auch die Möglichkeit, Read-Modelle über einen Query-Mechanismus abzufragen. Dazu werden, analog zu Commands, Query-DTOs definiert, die von Query-Handlern verarbeitet werden. Für den Einstieg reicht es jedoch meist aus, die Read-Modelle direkt den Konsumenten zur Verfügung zu stellen.

Fazit

CQRS und Event Sourcing erleichtern es, erweiterbare Software zu entwickeln. Mithilfe von Event Modeling kann eine CQRS-basierte Anwendung intuitiv modelliert werden, wobei auch gleich eine gemeinsame Fachsprache entwickelt wird. Die anschließende Umsetzung mit dem Axon-Framework fällt, nach der Einarbeitung in das Programmiermodell, ebenfalls leicht. Durch die gute Integration mit dem verbreiteten Spring-Boot-Framework bietet Axon außerdem für viele Entwickler eine geringe Einstiegshürde.

Quellen

[1] *https://de.wikipedia.org/wiki/Command-Query-Responsibility-Segregation*

[2] *https://de.wikipedia.org/wiki/Event_Sourcing*

[3] *https://eventmodeling.org/*

Frank Steimle

Digital Frontiers GmbH & Co. KG

frank.steimle@digitalfrontiers.de

Frank Steimle ist Senior Consultant bei Digital Frontiers. Er beschäftigt sich mit agiler Softwareentwicklung im Umfeld von Domain-driven Design und CQRS/ES. Ein Schwerpunkt liegt auf der Durchführung von Event-Modeling-Workshops, mit deren Hilfe er Entwickler und Fachexperten in die Lage versetzen will, nützliche und wartbare Software zu erschaffen.

Nikolai Neugebauer

Digital Frontiers GmbH & Co. KG

nikolai.neugebauer@digitalfrontiers.de

Nikolai Neugebauer ist als Consultant für Digital Frontiers tätig. Sein Schwerpunkt liegt auf agiler Anforderungsanalyse sowie agiler Softwareentwicklung, vorwiegend im Java-/Kotlin- und Spring-Umfeld. Sein Wissen und seine Erfahrung gibt er regelmäßig in Kundenprojekten und auf Konferenzen weiter.

Wartbare, zukunftsfähige Software-Architekturen in Java mit Annotation-Processing, Spring Boot und Cloud-Technologien

Klaus Sausen

In großen Projekten kann es durch eine stetige Umsetzung von neuen Anforderungen schnell zu unübersichtlichen und monolithischen Strukturen kommen, die die Aufwände bei Erweiterungen leicht in die Höhe treiben. Ein Software-Architekt kann durch statische Analyse-Tools und den gezielten Einsatz von Unit-Tests beispielsweise die Vererbungstiefe und die Anzahl von Modul-Imports beschränken und so das Projekt auf Kurs halten. Mitglieder des Entwicklungsteams haben dann weniger Spielraum und die Produktivität sinkt oft, obwohl das Ziel nur die Verbesserung der Softwarequalität ist. Ein weiteres Mittel ist der bewusste Einsatz von Meta-Informationen an zentraler Stelle im Programm – dem Datenmodell. Dieser Artikel zeigt, wie mithilfe eines professionellen Annotation-Prozessors strukturierte Projekt-Entwicklung aussehen kann und wiederkehrende Patterns durch Erstellung von Vorlagen die Programm-Qualität steigert und zugleich das Team entlastet.

Viele Java-Projekte setzen das Spring-Framework ein, das sich als De-facto-Standard etabliert hat. Entwickler schätzen die Technologie aufgrund der hohen Produktivität, des umfangreichen Fundus an Erweiterungsbibliotheken und der Cloud-Unterstützung. Anhand einer Beispielanwendung zur Verwaltung von Fahrtkosten betrachtet der Artikel die verschiedenen Aspekte der Entwicklung aus Sicht einer wartbaren Architektur. Eine oft genutzte Bibliothek für Java-Datenbank-Anwendungen ist Hibernate. Dieser objektorientierte Mapper ist in Spring *[1]* integriert und macht genau das oben erwähnte Anreichern des Datenmodells mit Meta-Informationen zur Anbindung an das relationale Modell *(siehe Listing 1)*.

```
@Entity
public class Trip {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;

  @NotBlank
  private String source;

  @NotEmpty
  private String destination;

  @Size(min = 0, max = 5)
  private float distanceKm;

  private LocalDate tripStart = LocalDate.now(Clock.systemUTC());

  @Embedded
  private Address address;
//  [..]
```

Listing 1: Eine Entity-Definition

Man erkennt selbst als Nicht-Entwickler, dass es sich hier um eine Datenbankentität `@Entity` handelt und welche Felder in der relationalen Tabelle abgebildet sind. Hinsichtlich der Wartbarkeit ist die leichte Verständlichkeit von Programmcode ein wichtiger Aspekt. Gleiches gilt für die Definition von REST-Schnittstellen, auch hier bietet Spring mit dem `@RestController`-Stereotyp den gleichen Ansatz *(siehe Listing 2)*.

```
@SeifeClass
@RestController
public class TripController {

  private final EntityService entityService;

  @Autowired
  public TripController(EntityService entityService) {
     this.entityService = entityService;
  }

  @SeifeMethod
  @RequestMapping("/v1/trips/count")
  public long count() {
     return entityService.countTrips();
  }

  @SeifeMethod
  @RequestMapping("/v1/trips/create")
  public long create(@RequestBody Trip trip) {
     Trip created = entityService.createTrip(trip);
     return created.getId();
  }

  @SeifeMethod
  @RequestMapping("/v1/trips/{id}")
  public Trip get(@PathVariable("id") Long id) {
     return entityService.getTripById(id);
  }
//  [..]
```

Listing 2: Eine technische Datenschnittstelle

Software-Architekturen in der Cloud

Für hochverfügbare Microservice-Architekturen *[2]* in der Cloud sind diese Patterns für die Entwickler an der Tagesordnung. Natürlich ist das Projekt damit auf das Spring-Framework festgelegt und eine Erweiterung durch spezialisierte Tools wie Micronaut *[3]* für die

Microservices geht oft mit einer gewissen Code-Verdopplung und damit erhöhtem Wartungsaufwand einher. Sollte die Spritkosten-Verwaltung eines Tages auch durch eine mobile App genutzt werden, ist es sinnvoll, schon gleich zu Beginn die Datenbank mithilfe eines REST-API von der grafischen Benutzeroberfläche zu trennen. So lassen sich die Kernkomponenten später auch für die mobilen Clients verwenden, zudem sind mehrfach in der Cloud laufende REST-Endpunkte auf Redundanz ausgelegt *(siehe Abbildung 1)*. Sie verbessern die Verfügbarkeit und erlauben oft Softwareaktualisierungen mit Zero-Downtime, indem erst der eine Endpunkt und daraufhin der zweite aktualisiert wird.

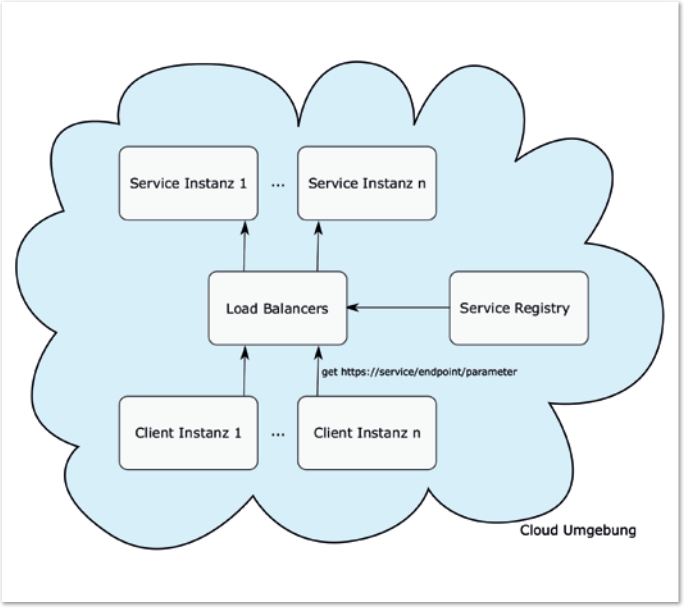


Abbildung 1: Redundante REST-APIs in der Cloud (© Klaus Sausen)

Die Modularisierung und die richtige Aufteilung von Funktionalität in einzelne Bibliotheken sind bekanntlich Best-Practices für gute Software. Daraus entstehen in der Cloud, in der einzelne Services unabhängig voneinander in Betrieb sind, aufgrund der notwendigen Kommunikation direkte technische Anforderungen. Eine solche Architektur ist in *Abbildung 2* dargestellt.

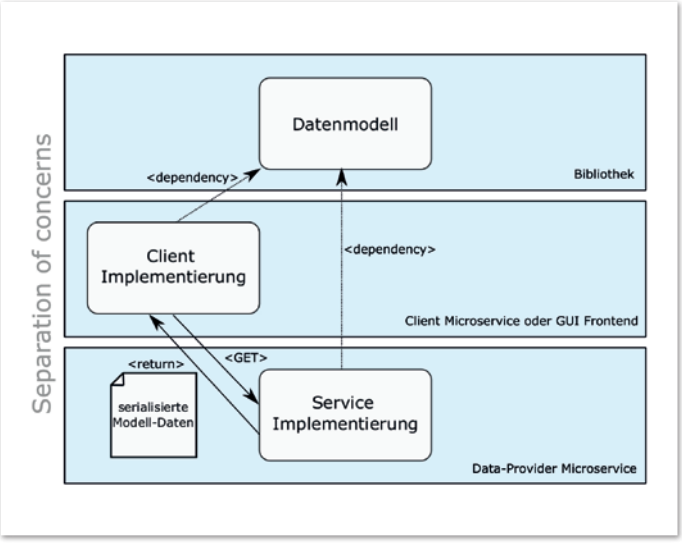


Abbildung 2: Anforderungen an die Architektur (© Klaus Sausen)

Die notwendigen Modulschnittstellen sind im besten Fall auf Sender- und Empfängerseite mit Unit-Tests, zum Beispiel mit Wiremock *[4]*, abgesichert und doch kann schon ein einfacher Versionskonflikt im Datenmodell zu Problemen führen. Die heute selbstverständliche Typsicherheit kann oft nicht mit vertretbaren Aufwänden gewährleistet werden, falls im Projekt einfache, interoperable Datenformate wie JSON eingesetzt sind.

Fehlerwahrscheinlichkeit verringern und Zeit sparen mit Annotations

Durch all diese Aspekte entstehen im Projekt viele Freiheitsgrade, die einen gewissen Wildwuchs begünstigen. Java-Annotation-Prozessoren können hier Abhilfe schaffen und die negativen Implikationen mindern. Sie funktionieren wie Plug-ins für den Java-Compiler und laufen zur Übersetzungszeit. Daher haben sie Zugang zum Klassenmodell, zu dessen Attributen, Methoden sowie Datentypen und unterstützen den Entwickler auf vielfältige Weise. Eines der bekanntesten ist Lombok. Im weitesten Sinne spart es Tipparbeit, indem repetitiver Code wie `getter`, `setter`, `equals()` und `hashcode()` automatisch erzeugt werden, was nicht nur die Lesbarkeit fördert, sondern natürlich auch Fehler wie ein vergessenes Feld im Objekt-Hashwert vermeidet.

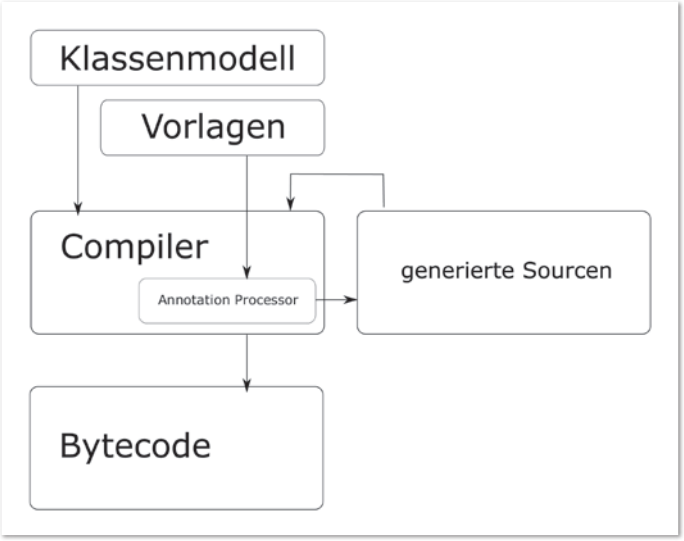


Abbildung 3: Funktionsweise eines Annotation-Prozessors (© Klaus Sausen)

Das Beispielprojekt nutzt einen Annotation-Prozessor *[5]*, der das Klassenmodell analysiert und im Projekt integrierte Programmvorlagen nutzt, um zur Compile-Zeit passenden Code aus dem Meta-Modell zu generieren. Er ist für Open-Source-Projekte kostenfrei und nutzt die Apache Velocity Template Engine *[6] (siehe Listing 3)* – diese unterstützt sogar Schleifen. Die Vorlagen müssen keinen Java-Code enthalten, sondern könnten genauso C#-Klassen für das .Net-Framework erzeugen und so Teile des Datenmodells für .Net oder Xamarin zugänglich machen. Um im Rahmen zu bleiben, wird in diesem Beispiel nur ein passender OpenFeign-Rest-Client generiert.

Erfahreneren Entwicklern ist klar, dass die Pflege solcher Vorlagen mehr Aufwand bedeutet – es lohnt sich sofort, wenn auch nur eine Änderung an allen Stellen im Bestandscode erforderlich wird oder

sobald man das zweite oder dritte API anbindet. Jede weitere neue Schnittstelle ist dann oft ohne Änderung der Vorlage auf Knopfdruck generiert.

Im Team ließen sich damit sogar projektspezifische Vorlagen für Tests erstellen, so weiß der Entwickler gleich, welche Programmteile noch Testabdeckung erfordern, bevor er eincheckt.

Annotations über Modulgrenzen hinweg nutzen

Eine Einschränkung des heute üblichen Annotation-Processing ist, dass Informationen über die Programmstruktur nur temporär während der Übersetzung vorliegen. Sobald ein Modul in ein Jar übersetzt wird, sind Lombok und ähnliche Tools nicht in der Lage, auf das Meta-Modell anderer Bibliotheken zuzugreifen, da diese Information nicht notwendigerweise in der .class-Datei der verwendeten Bibliotheken-enthalten ist. Das Compiler-Plug-in „sieht" nur den gerade verarbeiteten Programmcode. Wie soll also in unserem Fall der Webservice-Client Kenntnis über das Datenmodell des Microservice bekommen? Das eingesetzte Tool exportiert dazu auf Wunsch das aktuelle Meta-Modell eines Moduls in eine Datei. Andere Module oder eigenständige Services, die die Metadaten benötigen, sind so konfiguriert, dass sie diese Dependency während der Kompilierung entsprechend hinzuladen.

Databinding

Viele Softwaresysteme kommen nicht ohne eine grafische Benutzeroberfläche aus. Die komplette Anbindung der Eingabefelder an das Datenmodell ist für einen Entwickler je nach eingesetztem Framework nicht immer automatisch möglich. Das liegt meist einfach an recht spezifischen Anforderungen, die die meisten Systeme nicht vorab implementieren können. Ferner müssen zwei Ebenen der Validierung unterschieden werden, die Validierung in der Benutzeroberfläche muss zu den Datenbank-Constraints passen und darf höchstens strenger validieren. Die besten Ergebnisse erzielt man erfahrungsgemäß mit bereits sinnvoll auf der Datenbank konfigurierten Einschränkungen, denn diese sind dann auch überall gültig. Im Beispielprojekt ist eine Vaadin-Flow-Benutzeroberfläche *[7]* implementiert, die passende Felder zur Benzinkostenabrechnung bereitstellt. Bei der Komponente zur Dateneingabe wird dazu wiederum das Metamodell genutzt, um die Eingabefelder an die passenden Attribute anzubinden. Das Binding geschieht per „Convention over Configuration", jedes Eingabeelement wird so genannt wie das Attribut in der Klasse des Datenmodells.

Sollte die Zuordnung einmal nicht passen, kann sie explizit in der Annotation eingetragen werden. Natürlich spart das auch bei ersten GUI-Prototypen Zeit. Es genügt, alle Inputfelder zu deklarieren

```
@SeifeClass(classOptions = { "sourcemodel:class=com.examples.rest.TripController," +
     "prefix=/v1/trips/,replacement=/${rest.client.apilevel}/trips/" },
     generatorOptions = {
"merge.feignclient/client.vm=com.examples.restclient.TripClient",
"merge.feignclient/fallback.vm=com.examples.restclient.TripClientFallback",
})
@FeignClient(name = "tripclient", fallback = TripClientFallback.class)
public interface TripClient {

  // @seife automatically generated:

  @RequestMapping("/${rest.client.apilevel}/trips/count")
  long count();

  @RequestMapping("/${rest.client.apilevel}/trips/create")
  long create(Trip trip);

  @RequestMapping("/${rest.client.apilevel}/trips/{id}")
  Trip get(@PathVariable("id") Long id);

  // @seife auto-code end
}
```

Listing 3: Ein aus „TripController" generierter Client mit anderem Technologie-Stack

```
@SpringComponent
@Scope(SCOPE_PROTOTYPE)
@SeifeForm(forClass = Trip.class, generatorOptions = {"grid.gridClass"})
public class TripGrid extends Grid<Trip> {

  @SeifeBinding
  private Column<Trip> source;

  @SeifeBinding("destination")
  private Column<Trip> dest;

  @SeifeBinding("address.street")
  private Column<Trip> street;
//  [..]
```

Listing 4: Annotierte Eingabefelder

und ein oder zwei kleine Standardmethoden bereitzustellen *(siehe Listing 4)*.

Aus Sicht des Annotation-Processing gilt es zunächst, das Datenmodell zu analysieren und die Eigenschaften der einzelnen Datenfelder zu berücksichtigen. Das System verarbeitet nur solche Felder, die mit `@SeifeField` annotiert sind und pflegt diese in die internen Datenstrukturen ein, die es später für das Template-Processing benötigt. Es hat dabei auch Zugriff auf die Datenbank-Constraints und erzeugt automatisch entsprechende Validierungen, sofern die Templates darauf vorbereitet sind. Sollten sich Constraints an Attributen des Modells einmal ändern, zieht das System bei der nächsten Übersetzung automatisch alle betroffenen Stellen in der GUI nach. Die meisten vergleichbaren Systeme bieten zwar ein automatisches Binding über das Reflection-API an, jedoch müssen gerade die Validierer meist manuell gepflegt werden.

Im System eine physikalische Einheit oder Währung für einen Datentyp zu verwenden, bedeutet, eine passende Template-Vorlage für den Datentyp auszuwählen. Wann immer das entsprechende Attribut in einer GUI eingebunden wird, kann eine verfeinerte Vorlage genutzt werden und das System generiert darauf basierend Code mit passenden Datenkonversionen.

Der Charme eines ORM ist die Unterstützung objektorientierter Datenstrukturen. Dabei ist es wichtig, leicht auf verschachtelte Objekte zugreifen zu können. Über die Punkt-Notation, ähnlich wie auch in Hibernate Queries, lassen sich auch „tiefer gelegene" Felder ansprechen. Das System löst auch diese Datentypen auf und stellt einen Mechanismus bereit, den Pfad zum Datum beim Lesen und Schreiben zu nutzen.

Fazit

Annotation-Prozessoren sind nicht nur für kleine Programmschnipsel geeignet, richtig eingesetzt können sie das gesamte Projekt positiv beeinflussen. Sie sind in der Lage, die Typsicherheit über Modulgrenzen hinweg während der Interprozesskommunikation zu gewährleisten. Zugleich erlauben sie es, Einschränkungen auf Datenfeldern effektiv und projektbezogen im System zu definieren und an den nötigen Stellen durch die Vorlagen umzusetzen. Im Vergleich zu den über Reflection zur Laufzeit analysierten Annotationen geschieht die Verarbeitung einmal während der Kompilierung, das trägt zur Effizienz der Software bei und verbessert den Energieverbrauch bei Mobilgeräten.

Quellen

[1] Spring Boot: *https://spring.io/*
[2] Thomas Hunter II (2017): Advanced Microservices, APress, San Francisco
[3] Micronaut: *https://micronaut.io/*
[4] Wiremock: *http://wiremock.org/*
[5] Seife Annotation Processor: *https://uc-mobileapps.com/seife-annotation-processor/*
[6] Apache Velocity: *https://velocity.apache.org/*
[7] Vaadin-Flow: *https://vaadin.com/*
[8] Beispiel-Projekt auf GitHub: *https://github.com/uc-mobileapps/seife-samples-microservice*

Klaus Sausen
UC MobileApps
office@uc-mobileapps.com

Klaus Sausen ist freiberuflicher Softwareentwickler und hat in der Handels- und Lebensmittelbranche an Back-Ends auch im POS-Umfeld mitgearbeitet. Er entwickelt Cloud-Projekte mit Kafka, Spring und Micronaut für Kubernetes. Sollten Sie an einer Zusammenarbeit interessiert sein, sprechen Sie mich gerne an.

Eventbasierte Schnittstellen mit Amazon EventBridge und Spring

Simon Jakubowski und Jan Sauer, newcubator GmbH

In verteilten Systemen, wie beispielsweise in einer Microservice-Architektur, stellt die Kommunikation zwischen den Services eine große Herausforderung dar. Dabei gilt es einerseits, das Problem der Service Discovery zu lösen, sowie andererseits, geeignete Fehler- und Recovery-Strategien zu entwickeln. Darüber hinaus stehen wir vor der Aufgabe, bei der Entwicklung solcher Services ein geeignetes Entwicklungsverfahren in Zusammenarbeit mit anderen Teams oder Dienstleistern zu etablieren. An diesem Punkt bietet eine eventbasierte Architektur einen interessanten und frischen Ansatz.

Begonnen hat es mit einem Aufruf unseres REST-API. Immer, wenn sich ein neuer Nutzer auf der Webseite registrierte, sollte die Benutzerverwaltung unser API aufrufen, damit alle nötigen Vorbereitungen getroffen werden können. Allerdings verwaltet unsere Anwendung keine Nutzer, warum sollten wir also Nutzer in dem dazugehörigen REST-API anbieten? Die Anwendung möchte lediglich informiert werden, wenn sich ein neuer Nutzer registriert hat. In diesem Fall bietet sich der Einsatz von eventbasierter Kommunikation an, denn nicht jedes API muss RESTful sein.

Was eventbasierte Schnittstellen bieten

Statt einen Zustand in einem anderen System zu verändern, übertragen eventbasierte Schnittstellen nur die Information, dass und wann sich ein Zustand verändert hat. Beispiele hierfür sind etwa „Nutzer hat sich registriert", „eine Bestellung wurde erstellt" oder „das Thermostat hat 12° C gemessen". Dadurch muss der Absender eines Events keinerlei Informationen über das Datenmodell des Empfängers haben. Er muss nicht wissen, wie etwa ein Nutzer angelegt oder der gemessene Wert eines Thermostats aktualisiert wird. Wichtig ist nur, dass das Event vom Sender zum Empfänger kommt. Danach liegt es am Empfänger zu entscheiden, wie er aufgrund der Informationen eines Events weiter verfährt.

An dieser Stelle ist es von Vorteil, einen zentralen Event Bus einzuführen, statt Events direkt an andere Systeme zu senden. Dadurch werden diese voneinander entkoppelt und interagieren nicht mehr direkt miteinander. Infolgedessen werden Events direkt an den Event Bus gesendet, um sie von dort an alle Empfänger zu verteilen *(siehe Abbildung 1)*. Kommt ein neuer Empfänger hinzu, muss le-

diglich der Event Bus neu konfiguriert werden. Änderungen in der Anwendung, die das Event erstellt hat, sind dabei nicht mehr nötig.

In der Praxis hilft dies auch bei der Abstimmung mit anderen Teams oder Dienstleistern. Es muss sich nicht mehr über das „Wie" – das gibt der Event Bus vor –, sondern lediglich noch über das „Was" und „Wann" unterhalten werden. Somit werden interne Abstimmungs- und Kommunikationsprozesse verschlankt und effizienter gestaltet.

Die Einführung eines Event Bus kann zusätzlich helfen, Komplexität aus den einzelnen Systemen zu übernehmen. Ruft ein Service einen anderen auf, so muss sich im herkömmlichen System der aufrufende Service darum kümmern, dass diese Nachricht auch korrekt an den konsumierenden Service übertragen wird. Was ist jedoch, wenn der konsumierende Service gerade nicht erreichbar oder überlastet ist? Hier kann der Event Bus helfen, indem er diese Probleme zentral löst und dies nicht jedes System selbst übernehmen muss. So kann er versuchen, Events erneut auszuliefern, wenn die Zustellung beim ersten Mal nicht erfolgreich war. Ebenso ist er in der Lage, die Last auf ein System abzufedern, indem er limitiert, wie schnell Events zugestellt werden.

Was Amazon EventBridge liefert

Da viele unserer Kunden vollständig auf Amazon Web Services setzen, haben wir uns auf der Suche nach einem Event Bus auch den EventBridge-Service *[1]* von Amazon angesehen. Dieser übernimmt die vorgestellte Funktionalität eines Event Bus und erleichtert den Aufbau einer eventbasierten Architektur erheblich. Dazu lassen sich in EventBridge neben anderen Funktionen, auch einzelne Event Busses erstellen und Regeln verwalten, die definieren, welche Events wohin weitergesendet werden.

Die Nutzung eines Service nimmt uns hier den Aufbau und Betrieb einer vergleichbaren Infrastruktur ab. Mit dem Event Bus als Schnittstelle zwischen allen Systemen sind gerade hier Ausfallsicherheit und Skalierung wichtige Themen, die je nach Kunde und Projekt auch schnell zu unüberwindbaren Hindernissen werden können, wenn man sie selbst übernehmen muss. Allerdings setzt man bei Amazon EventBridge auf ein proprietäres Produkt. Im Gegensatz zu Projekten wie Apache Kafka und Co. bindet man sich an Amazon und das Selbstbetreiben ist keine Option.

EventBridge gehört zu Amazons Serverless-Angeboten und die Abrechnung erfolgt in erster Linie nach zugestellten Events und API-Aufrufen *[2]*.

Neben der Zustellung von Events gibt es auch die Möglichkeit, Events in einem Archiv mitprotokollieren zu lassen oder den Inhalt eines Archivs erneut zuzustellen. EventBridge ist jedoch keine Datenbank. Deshalb lassen sich Events nicht beliebig abfragen und es ist ebenfalls nicht möglich, nach alten Events zu suchen.

Aufbau eines Events

Events werden bei EventBridge in JSON encodiert und bestehen stets aus einem fest definierten Umschlag mit ihren Metadaten und einer benutzerdefinierten Nutzlast *(siehe Listing 1)*. Welche Informationen sich hinter `detail` befinden, ist dabei dem Nutzer überlassen. Einzige Voraussetzung ist, dass es sich in JSON abbilden lässt. Dabei sollte der `detail-type` angeben, um welche Art von Event es sich handelt und welche Informationen in der Nutzlast zu erwarten sind. Der `detail-type` wird auch oft in Regeln verwendet, um zu entscheiden, ob ein Event an ein System weitergeleitet werden soll.

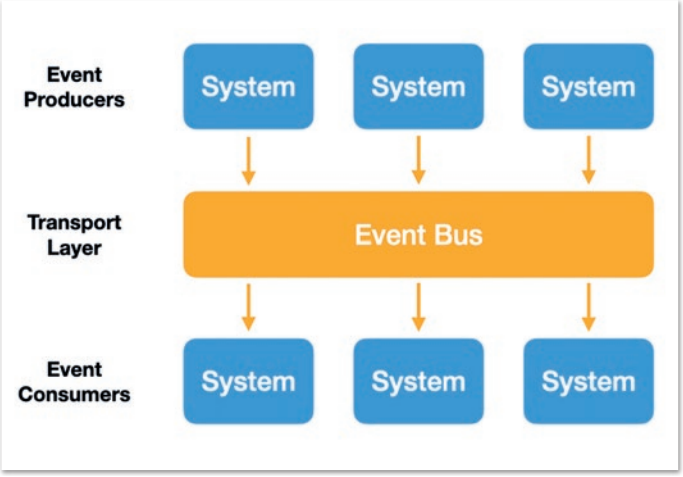


Abbildung 1: Schematische Darstellung einer Event-Bus-Architektur (© Jan Sauer, Simon Jakubowski)

```
{
  "version": "0",
  "id": "9193f4ce-890a-9397-08f8-767e1b60ed56",
  "detail-type": "UserCreatedV1",
  "source": "com.newcubator.event-bridge-demo",
  "account": "565575105323",
  "time": "2021-11-26T09:25:48Z",
  "region": "eu-central-1",
  "resources": [],
  "detail": {
     "name": "Tom",
     "source": "WEBAPP"
  }
}
```

Listing 1: JSON-Repräsentation eines Events

Das erste Event senden

Um in einer Java-Anwendung Events zu senden, nutzt man am einfachsten das SDK von Amazon *(siehe Listing 2)*. Dieses liefert Builder zum Erstellen der Event-Umschläge (`PutEventsRequestEntry`) und einen Client zum Senden. Dabei muss man die Nutzlast selbstständig als String serialisieren.

In Kundenprojekten sowie im Beispielprojekt zu diesem Artikel nutzen wir Value-Objekte für die Nutzdaten der Events. Wir betrachten Events als unveränderlich und wollen daher auch nicht erlauben, dass man sie nach ihrer Erstellung noch anpasst. Darüber hinaus hat es sich bei uns etabliert, Businesslogik und die Nutzung von SDKs voneinander zu trennen. Dies erleichtert, den Client zu mocken, um das Senden von Events als Teil der Businesslogik zu vertesten.

Events empfangen

Nur Events zu senden, hilft jedoch keinem Kunden. Stattdessen sollen andere Dienstleister und Anwendungen auf Events reagieren können. Hierfür bietet EventBridge eine Reihe von möglichen Zielen, denen der Service Events zustellen kann *[3]*. Neben anderen AWS Services wie SNS, SQS und Batch Job Queue setzen wir mit unseren

```
EventBridgeClient client = EventBridgeClient.builder()
  .region(Region.EU_CENTRAL_1)
  .build();

PutEventsRequestEntry eventEntry = PutEventsRequestEntry.builder()
  .eventBusName("EventBridgeDemo")
  .source("com.newcubator.event-bridge-demo")
  .detailType("UserCreatedV1")
  .detail("{ \"name\": \"Tom\", \"source\": \"WEBAPP\" }")
  .build();

PutEventsResponse result = client.putEvents(PutEventsRequest.builder()
  .entries(eventEntry)
  .build());
```

Listing 2: Senden eines Events

Kunden vor allem auf Lambda und API Destinations. In allen Fällen ist es nötig, eine Regel zu konfigurieren. Hiermit wird gefiltert, welche Events des Event Bus wohin zugestellt werden sollen.

Um Events mit einer Java-Anwendung zu empfangen, ist etwas mehr Aufwand nötig, als lediglich eine Regel zu konfigurieren. API Destinations erlauben es, Events an einen beliebigen HTTP-Endpunkt zu senden. Dafür muss dieser technisch gesehen nicht einmal innerhalb von AWS liegen. Die Konfiguration ist jedoch komplexer, da hierfür gleich drei verschiedene Ressourcen in EventBridge aufgesetzt werden müssen:

1. **Connection**: Diese regelt das Autorisierungsverfahren zwischen unserem System und Amazon. So lässt sich beispielsweise ein Token im HTTP-Request mitsenden, um sicherzustellen, dass ein eingehendes Event auch wirklich von Amazon stammt.
2. **API Destination**: Hier werden die BaseURL unseres Systems sowie die zu verwendende HTTP-Methode und Connection-Konfiguration angegeben.
3. **Regel**: Wie erwähnt gibt diese an, welche Events an eine API Destination übertragen werden sollen, und enthält weitere Details für die Zustellung wie HTTP-Header, Pfad und Query-Parameter.

Für unsere Spring-Anwendung ist ein Event nichts anderes als eine HTTP-Anfrage mit dem Event als JSON im Body *(siehe Listing 3)*. Um eine saubere Code-Struktur zu erreichen, empfiehlt es sich, für jeden Event-Typ einen eigenen Endpoint zu erstellen.

Wichtig ist, dass die Verarbeitung von Events in unter fünf Sekunden abgeschlossen sein muss. Sollte eine Verarbeitung länger andauern, betrachtet EventBridge die Zustellung als gescheitert. Sehr praktisch ist jedoch, dass sich mit einer Regel konfigurieren lässt, wie lange und wie oft eine erneute Zustellung versucht werden soll. Dies hilft auch, wenn die API Destination aus anderen Gründen nicht erreichbar ist oder die Verarbeitung eines Events aufgrund eines Fehlers nicht abgeschlossen werden konnte.

Event-Schema

In einer Architektur, die sich über den Austausch von Events definiert, sind in der Zusammenarbeit mit anderen Teams und Dienstleistern zwei Fragen besonders relevant: Welche Events gibt es und wie sind diese aufgebaut? EventBridge als Service hilft hierbei nicht direkt. So lässt sich nicht vorgeben, welche Typen von Events versendet werden können, und auch der Aufbau der Informationen eines Events wird nicht validiert.

Was EventBridge anbietet, ist eine Schema-Registry, in der sich der Aufbau von Events mittels OpenAPI beschreiben lässt. Diese Schemata werden automatisch aus Events generiert und lassen sich versionieren oder zur automatischen Generierung von Code nutzen. Für uns hat sich hier jedoch ein gemeinsames Git-Projekt etabliert. Dienstleister, die Events erzeugen, dokumentieren diese dort per JSON-Schema *(siehe Listing 4)*. Dadurch wird auch eine Zusammenarbeit mithilfe von Merge-Requests, Code Reviews und Issues ermöglicht, die insgesamt mehr unserem vorhandenen Workflow entspricht.

```
@RestController
@RequestMapping(value = "/api/events")
@Slf4j
public class EventsController {

  @PostMapping("/UserCreatedV1")
  @ResponseBody
  public void userCreatedV1(@RequestBody Event<UserCreatedV1> event) {
     log.info("Received {}", event);
     log.info("The new user is named '{}'", event.getDetail().getName());

     // do whatever is needed
  }

  // ... more endpoints for other events
}
```

Listing 3: Empfangen von Events

```
{
  "type": "object",
  "properties": {
     "name": {
        "type": "string",
        "description": "users first name"
     },
     "source": {
        "type": "string",
        "enum": [
           "WEBAPP",
           "IOSAPP"
        ],
        "description": "application used for the registration"
     }
  }
}
```

Listing 4: JSON-Schema-Dokumentation eines Events

Events entwickeln sich stetig weiter und ihr Aufbau ergänzt oder verändert sich. Änderungen am Aufbau eines Events vorzunehmen, ist eine sehr komplexe Aufgabe, da nicht mit Sicherheit gesagt werden kann, welche Auswirkung die Änderung auf die Verarbeitung in anderen Systemen hat. Daher sollten Events von Beginn an versioniert werden.

Recap

Die Nutzung von eventbasierten Schnittstellen als Architekturpattern hat unsere Entwicklungsperformance deutlich gesteigert. So haben wir in einem System zunächst die Events und deren Schemata definiert und diese schließlich mit den Entwicklungsteams abgestimmt und finalisiert. Dadurch konnten die Teams unabhängig voneinander ihre Anforderungen umsetzen, da eine Vielzahl der Schnittstellen über die Events gelöst wurde. Im Gegensatz zur herkömmlichen API-Dokumentation haben Events den Vorteil, dass sie die Businesslogik in den Vordergrund stellen.

Durch den Einsatz von Amazon EventBridge lassen sich die Kosten und Aufwände für die Inter-Service-Kommunikation deutlich reduzieren. Dies ermöglichte, mit einer kurzen Vorlaufzeit eine hoch performante und skalierbare Infrastruktur für unsere Events zu realisieren. Die Nutzung eines Event Bus erhöht darüber hinaus zusätzlich die Unabhängigkeit der Services voneinander.

Wir können nur jedem empfehlen, dieses Architektur-Pattern einmal auszuprobieren. Die hier verwendeten Beispiele gibt es als komplettes Demo-Projekt auch auf unserem GitHub-Account *[4]*. Für uns war der Einsatz ein voller Erfolg und wir werden es wieder nutzen.

Quellen

[1] *https://aws.amazon.com/de/eventbridge/*
[2] *https://aws.amazon.com/de/eventbridge/pricing/*
[3] *https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-targets.html*
[4] *https://github.com/newcubator/event-bridge-demo*

Jan Sauer
newcubator GmbH
jan.sauer@newcubator.com

Simon Jakubowski
newcubator GmbH
simon.jakubowski@newcubator.com

Simon Jakubowski und Jan Sauer sind als Softwareentwickler in der Java-Welt zuhause. Während Simon sich seit zwei Jahren bei newcubator den Themenschwerpunkten Enterprise Java und Datenbanken widmet, beschäftigt sich Jan bei newcubator mit Continuous Delivery und dem Einsatz von Java in Cloud-Umgebungen. Diese und andere Themen bringen Simon und Jan mit Kunden gemeinsam und erfolgreich in Projekte ein.

Alternativen zum Criteria-API

Nicolai Mainiero, sidion GmbH

Auch wenn es mit Spring Data oder Panache von Quarkus gut funktionierende Abstraktionen für den Zugriff auf die Datenbank gibt, kommt es manchmal zu Situationen, bei denen diese in dem, was über ihre APIs möglich ist, an ihre Grenzen stoßen. Sobald die where-Clause dynamisch zusammengebaut werden soll, muss in Spring Data auf Specifications und in Panache auf ungetypte Stringkonkatination ausgewichen werden. Falls noch komplexere Abfragen benötigt werden, kann auf das Criteria-API ausgewichen werden. Beides keine befriedigende Option, vor allem da das Criteria-API die Komplexität sowohl beim Schreiben als auch beim Lesen des Codes unnötig erhöht.

Beim lesenden Zugriff auf eine Datenbank ist es einfach, die Abfrage in SQL zu schreiben beziehungsweise sich durch SQL iterativ an die richtige Abfrage anzunähern. Die Umwandlung von SQL in Java-Code stellt dann aber eine große Herausforderung dar. Während einfache Abfragen mit den APIs von Spring Data oder Panache auch schnell umgesetzt werden können, kommt man bei komplexeren Abfragen mit mehreren Joins oder sogar Unions schnell an die Grenze dessen, was diese APIs unterstützen. An dieser Stelle muss dann auf die Grundlagen des Java Persistence API (JPA) *[1]* zurückgegriffen werden. Dieses ist nach wie vor das Standard-API in Java-SE- und Java-EE-Anwendungen für die Verwaltung der Persistenz und der Objekt-/Relationalzuordnung (ORM). Sie steht damit auch in allen Applikationen zur Verfügung, die zum Beispiel Spring Data oder Panache verwenden oder einen JPA Provider wie Hibernate *[2]* direkt integrieren.

Das bedeutet, innerhalb dieser Applikationen gibt es zwei zusätzliche Möglichkeiten, um Datenbankabfragen zu erstellen, zum einen JPQL (Java Persistence Query Language) und zum zweiten das Criteria-API. Bei JPQL kommt es ähnlich wie bei Panache schnell zu dem Problem, dass dynamische Abfragen durch Stringkonkatination zusammengebaut werden müssen und man Typsicherheit verliert. Außerdem eröffnet dies die Möglichkeit für Injection-Angriffe *[3]*, die es zu verhindern gilt. Als zweite Möglichkeit gibt es das Criteria-API, um sicher dynamische und komplexe Datenbankabfragen zu erzeugen. Im Folgenden sollen verschiedene Alternativen mit dem Criteria-API verglichen werden. Dazu wird folgendes Datenbankschema *(siehe Abbildung 1)* verwendet. Es handelt sich um ein stark vereinfachtes Modell eines Händlers mit Kunden, Produkten und Bestellungen.

Die folgenden beiden in *Listing 1* beschriebenen SQL-Abfragen sollen über das jeweilige API ausgeführt werden.
Listing 2 zeigt die Implementierung der Abfrage in Beispiel 1 mit dem Criteria-API. Zunächst wird ein `CriteriaBuilder` vom `EntityManager` geholt, dann eine `CriteriaQuery` für die Entität Customer erzeugt, um dann die where-Clause mithilfe des `CriteriaBuilder` (1) zu erzeugen. Am Ende wird die Abfrage erstellt und dann ausgeführt.

Zu erkennen ist, dass viel Setup-Code notwendig ist, um diese einfache Abfrage auszuführen. Außerdem ist es beim Lesen des Codes schwierig herauszufinden, welches SQL-Statement gegen die Datenbank schlussendlich ausgeführt wird. Die Implementierung von Bei-

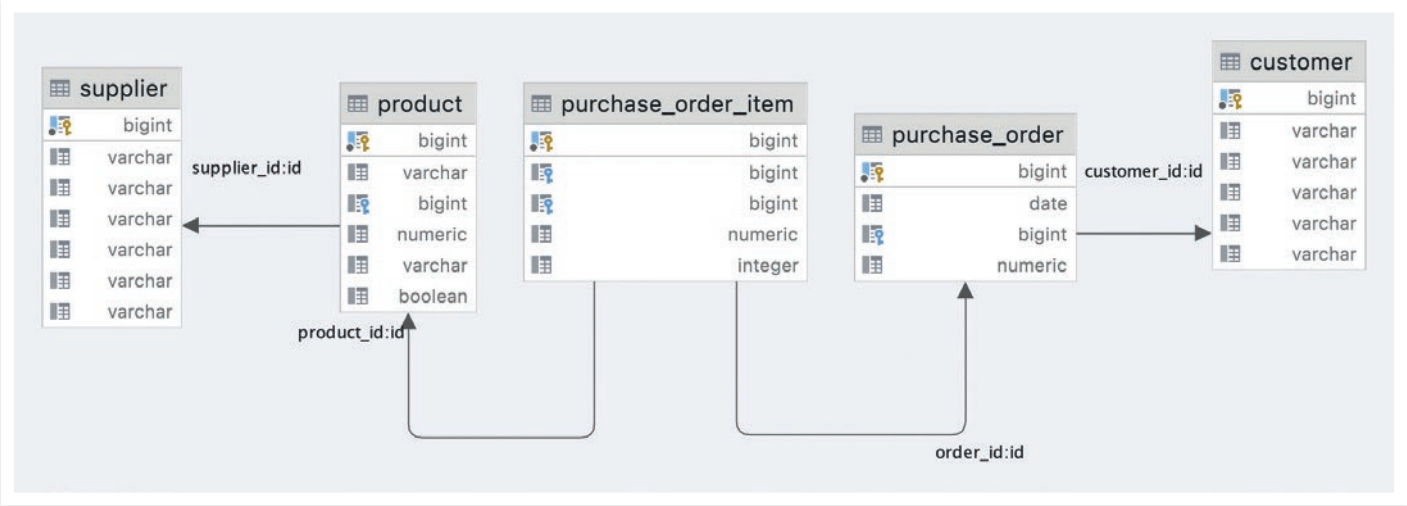


Abbildung 1: Datenbankschema (© Nicolai Mainiero)

```
SELECT * from customer WHERE first_name = :first_name and last_name = :last_name; Beispiel 1

SELECT * from purchase_order p JOIN customer c on p.customer_id = c.id WHERE last_name = :last_name; Beispiel 2
```

Listing 1: Beispiel 1 und 2 der SQL-Abfragen

```
public Customer findByName(String firstName, String lastName) {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Customer> cq = cb.createQuery(Customer.class);
  Root<Customer> customerRoot = cq.from(Customer.class);
  Predicate where = cb.and(cb.equal(customerRoot.get(Customer_.FIRST_NAME), firstName), cb.equal(customerRoot.
get(Customer_.LAST_NAME), lastName)); // 1
  CriteriaQuery<Customer> customerCriteriaQuery = cq
      .select(customerRoot)
      .where(where);
  return em.createQuery(customerCriteriaQuery).getSingleResult();
}
```

Listing 2: Beispiel 1 mit Criteria-API (zweiteilige where-Clause)

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<PurchaseOrder> query = cb.createQuery(PurchaseOrder.class);
  Root<PurchaseOrder> order = query.from(PurchaseOrder.class);
  Join<PurchaseOrder, Customer> customer = order.join(PurchaseOrder_.CUSTOMER);
  query.select(order).where(cb.equal(customer.get(Customer_.LAST_NAME), lastName));
  return em.createQuery(query).getSingleResult();
}
```

Listing 3: Beispiel 2 mit Criteria-API (join)

```
public Customer findByName(String firstName, String lastName) {
  SQLDialect configuration = SQLDialect.POSTGRES;
  SelectConditionStep<Record> query = DSL.using(configuration)
      .select()
      .from(CUSTOMER)
      .where(CUSTOMER.FIRST_NAME.eq(firstName).and(CUSTOMER.LAST_NAME.eq(lastName))); // 1
  return nativeQuery(em, query, Customer.class);
}
```

Listing 4: Beispiel 1 mit jOOQ

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  SQLDialect configuration = SQLDialect.POSTGRES;
  SelectConditionStep<Record> query = DSL.using(configuration)
      .select()
      .from(PURCHASE_ORDER)
      .join(CUSTOMER).on(PURCHASE_ORDER.CUSTOMER_ID.eq(CUSTOMER.ID))
      .where(CUSTOMER.LAST_NAME.eq(lastName)); // 1
  return nativeQuery(em, query, PurchaseOrder.class);
}
```

Listing 5: Beispiel 2 mit jOOQ

spiel 2 ist in *Listing 3* zu sehen. Zusätzlich zu der where-Clause ist hier auch noch ein Join notwendig, um die entsprechende Abfrage zu realisieren. Auch hier ist zu erkennen, dass viel Setup-Code benötigt wird, bis die eigentliche Abfrage formuliert und ausgeführt werden kann.

Anhand dieser beiden einfachen Beispiele werden drei Alternativen zum Criteria-API genauer betrachtet.

jOOQ

Die jOOQ-Bibliothek *[4]* verfolgt im Gegensatz zu JPA den Ansatz, dass zunächst die Datenbank modelliert und dann auf dieser Basis Java-Code erzeugt wird, mit dem Datenbankabfragen mithilfe eines Fluent-API beschrieben werden können. Dieses Vorgehen spielt vor allem bei bestehenden Datenbanken seine Vorteile aus, um dann in der weiteren Entwicklung nur mit dem jOOQ-API zu arbeiten. jOOQ bietet aber auch die Möglichkeit, aus bestehenden JPA-Entitäten das Metamodell von jOOQ zu erstellen, um dann sowohl JPA als auch das API von jOOQ zu kombinieren.

Dadurch können Abfragen typsicher und dynamisch erzeugt werden. Eine beliebte Kombination ist, die Daten mithilfe von JPA in die Datenbank zu schreiben und mit jOOQ oder einer der Alternativen zu lesen. Indem man das Schreiben der Daten an den Persistence-Provider delegiert, kann man sich den Aufwand, korrekte Insert- beziehungsweise Update-Statements zu schreiben, sparen. JPA kümmert sich darum, dass sowohl alle Felder der Entitäten berücksichtigt werden als auch die Reihenfolge der Inserts/Updates bei komplexen Objektgraphen korrekt ist. *Listing 4 und 5* zeigen die Implementierung der bekannten Beispielabfragen aus *Listing 1* mithilfe von jOOQ.

Wie bereits erwähnt, bietet jOOQ eine DSL, die SQL typsicher in Java nachbildet. Dadurch wird es sehr einfach, die Abfrage (1) in Java zu formulieren. Auch beim Lesen des Codes ist, wenn man SQL beherrscht, schnell klar, welche Abfrage ausgeführt werden wird.

Auch im zweiten Beispiel in *Listing 4* ist die Abfrage sofort erkennbar und erinnert stark an das äquivalente SQL-Statement. Sowohl die Tabelle, mit der das Join stattfindet, als auch die on-Bedingung werden explizit aufgeführt.

JPAStreamer

JPAStreamer ist noch eine recht junge Bibliothek, Version 1.0 erschien erst im Januar 2021, die einen interessanten Ansatz verfolgt. Datenbankabfragen werden mithilfe des Java-Stream-API beschrieben und können dann direkt mit den bekannten Methoden wie `filter()`, `sort()` oder `limit()` manipuliert werden.

Natürlich wird dabei nicht die gesamte Datenbank in den Speicher geladen. Der Ausdruck wird in ein entsprechendes SQL-Statement um-

gewandelt und dann zur Datenbank übermittelt. So wird zum Beispiel aus dem Prädikat in filter() eine where-Clause im SQL-Statement. *Tabelle 1* zeigt, welche Methode in welches SQL äquivalent umgewandelt wird. Für die Beispiele 1 und 2 ergeben sich dann mit dem JPAStreamer-API die Implementierungen, die in den *Listings 6 und 7* abgebildet sind. Die Abfrage nach Vor- und Nachnamen lässt sich direkt auf einen `filter()`-Aufruf abbilden. Hierbei ist zu beachten, dass das Prädikat über das von JPAStreamer erzeugte Metamodell erzeugt und keine Lambdaexpression verwendet wird, da sonst die Filter-Operation (1) nicht auf eine where-Clause abgebildet wird.

In *Listing 7* ist zu sehen, dass dennoch eine Lambdaexpression (1) verwendet worden ist. Das bedeutet, dass bei dieser Abfrage jede `PurchaseOrder` geladen wird, bis ein passendes Ergebnis gefunden worden ist, da die Filterung auf gejointe Tabellen wie von JPAStreamer noch nicht vollständig unterstützt wird. Es existiert aber bereits ein Issue *[5]*, das sich diesem Problem widmet.

Wer Gefallen an JPAStreamer gefunden hat, aber vorrangig mit Scala programmiert, hat mit Slick *[6]* und Quill *[7]* gleich zwei ähnliche Alternativen, die in Scala implementiert worden sind und sich besser in das Ökosystem einpassen.

QueryDSL

Die dritte Alternative ist QueryDSL, eine schon etwas ältere Bibliothek, die mehr als zwei Jahre kaum gepflegt wurde. Im vergangenen Jahr fand sich allerdings eine Gruppe Freiwilliger, die die Pflege übernommen hat. So wurde im Juli 2021 dann Version 5 veröffentlicht *[8]*, die nicht nur Abhängigkeiten aufgeräumt hat, sondern auch eine Menge Bugfixes und sogar neue Features mitgebracht hat. Wie die beiden zuvor beschriebenen Alternativen setzt auch

SQL	Java Stream
FROM	stream()
WHERE	filter() (before collecting)
ORDER BY	sorted()
OFFSET	skip()
LIMIT	limit()
COUNT	count()
GROUP BY	collect(groupingBy())
HAVING	filter() (after collecting)
DISTINCT	distinct()
SELECT	map()
UNION	concat(s0, s1).distinct()
JOIN	flatmap()

Tabelle 1: JPAStreamer Stream-API und SQL-Entsprechungen

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  StreamConfiguration<PurchaseOrder> joining = of(PurchaseOrder.class).joining(PurchaseOrder$.customer);
  Optional<PurchaseOrder> order = jpaStreamer
      .stream(of(PurchaseOrder.class).joining(PurchaseOrder$.customer)) // 1
      .filter(po -> po.getCustomer().getLastName().equals(lastName)) // 1
      .findFirst();
  return order.orElse(null);
}
```

Listing 7: Beispiel 2 mit JPAStreamer

```
public Customer findByName(String firstName, String lastName) {
  QCustomer customer = QCustomer.customer;
  JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(em);
  return jpaQueryFactory.selectFrom(customer)
      .where(customer.firstName.eq(firstName).and(customer.lastName.eq(lastName)))
      .fetchOne(); // 1
}
```

Listing 8: Beispiel 1 mit QueryDSL

```
public Customer findByName(String firstName, String lastName) {
  Optional<Customer> customer = jpaStreamer
      .stream(Customer.class)
      .filter(Customer$.firstName.equal(firstName).and(Customer$.lastName.equal(lastName))) // 1
      .findFirst();
  return customer.orElse(null);
}
```

Listing 6: Beispiel 1 mit JPAStreamer

```
public PurchaseOrder findOrderOfCustomer(String lastName) {
  QCustomer customer = QCustomer.customer;
  QPurchaseOrder purchaseOrder = QPurchaseOrder.purchaseOrder;
  JPAQueryFactory jpaQueryFactory = new JPAQueryFactory(em);
  return jpaQueryFactory.selectFrom(purchaseOrder)
      .innerJoin(purchaseOrder.customer, customer)
      .where(customer.lastName.eq(lastName))
      .fetchOne(); // 1
}
```

Listing 9: Beispiel 2 mit QueryDSL

QueryDSL auf die Generierung eines Metamodells, dass wie bei JPAStreamer durch einen Annotation-Prozessor erzeugt wird. Die Abfrage wird dann ähnlich zu jOOQ mithilfe einer DSL beschrieben und ausgeführt.

In *Listing 8* ist wieder die Suche nach einem Kunden anhand des Vor- und Nachnamens implementiert. Dank der DSL lässt sich die Abfrage (1) fast wie SQL schreiben. Zunächst wird die Tabelle ausgewählt (selectFrom), anschließend die where-Clause beschrieben (where) und dann genau ein Datensatz geladen (fetchOne).

Für das Beispiel 2 wird noch ein Join benötigt, die Implementierung ist in *Listing 9* zu sehen. Auch hier ist wieder zu erkennen, wie ähnlich die DSL zu SQL ist. Und wie direkt sich SQL-Statements nach Java übertragen lassen und typsichere Datenbankabfragen ermöglichen. Damit ist auch beim Lesen des Quellcodes direkt klar, welche Abfrage ausgeführt werden wird.

Fazit

Aus meiner Sicht ist JPAStreamer eine kreative Art, das Streams-API zu verwenden, um daraus SQL-Statements zu erzeugen. Die kognitive Last bei der Umwandlung zwischen Stream-API und SQL ist nicht zu vernachlässigen, vor allem bei komplexeren Abfragen. Dennoch denke ich, dass der Großteil der typischen Datenbankabfragen in einer Applikation sehr einfach mit filter() und sorted() abgebildet werden kann. Bei QueryDSL gefällt vor allem die sehr direkte Umwandlung zwischen Java und SQL. Bestehende SQL-Abfragen können praktisch 1:1 nach Java umgeschrieben werden. Das Gleiche gilt auch für jOOQ. Auch hier bietet die DSL die vollständige Kontrolle über das erzeugte SQL-Statement.

Sowohl QueryDSL als auch jOOQ punkten zusätzlich durch die Möglichkeit, die verwendeten Metamodelle aus bestehenden Datenbanken, ohne den Umweg über JPA, zu erzeugen. Damit bilden sie auch ein großartiges Bindeglied in Anwendungen, die sowohl Legacy- als auch JPA-basierte Datenbanken bedienen müssen. Für mich haben sich alle drei Bibliotheken als Möglichkeit erwiesen, komplexe Datenbankabfragen verständlich zu implementieren, ohne den Vorteil von einfachen Updates durch JPA aufzugeben. Der vollständige Quellcode der Beispiele, inklusive der Generierung der Metamodelle, findet sich auf GitHub *[9]*.

Quellen

[1] The Eclipse EE4J Project, „Jakarta Persistence project," [Online]. Available: *https://github.com/eclipse-ee4j/jpa-api*.

[2] Red Hat, „Hibernate ORM," [Online]. Available: *https://hibernate.org/orm/*.

[3] OWASP, „A03:2021 – Injection," [Online]. Available: *https://owasp.org/Top10/A03_2021-Injection/*.

[4] Data Geekery GmbH, „Great Reasons for Using jOOQ," [Online]. Available: *https://www.jooq.org/*.

[5] JPAStreamer, „Optimize Streams that yield Join-operations," [Online]. Available: *https://github.com/speedment/jpa-streamer/issues/43*.

[6] Slick, „ Functional Relational Mapping for Scala," [Online]. Available: *https://scala-slick.org/*.

[7] Quill, „Free/Libre Compile-time Language Integrated Queries for Scala," [Online]. Available: *https://getquill.io/*.

[8] QueryDSL, „QueryDSL Version 5.0.0," [Online]. Available: *https://github.com/querydsl/querydsl/releases/tag/QUERYDSL_5_0_0*.

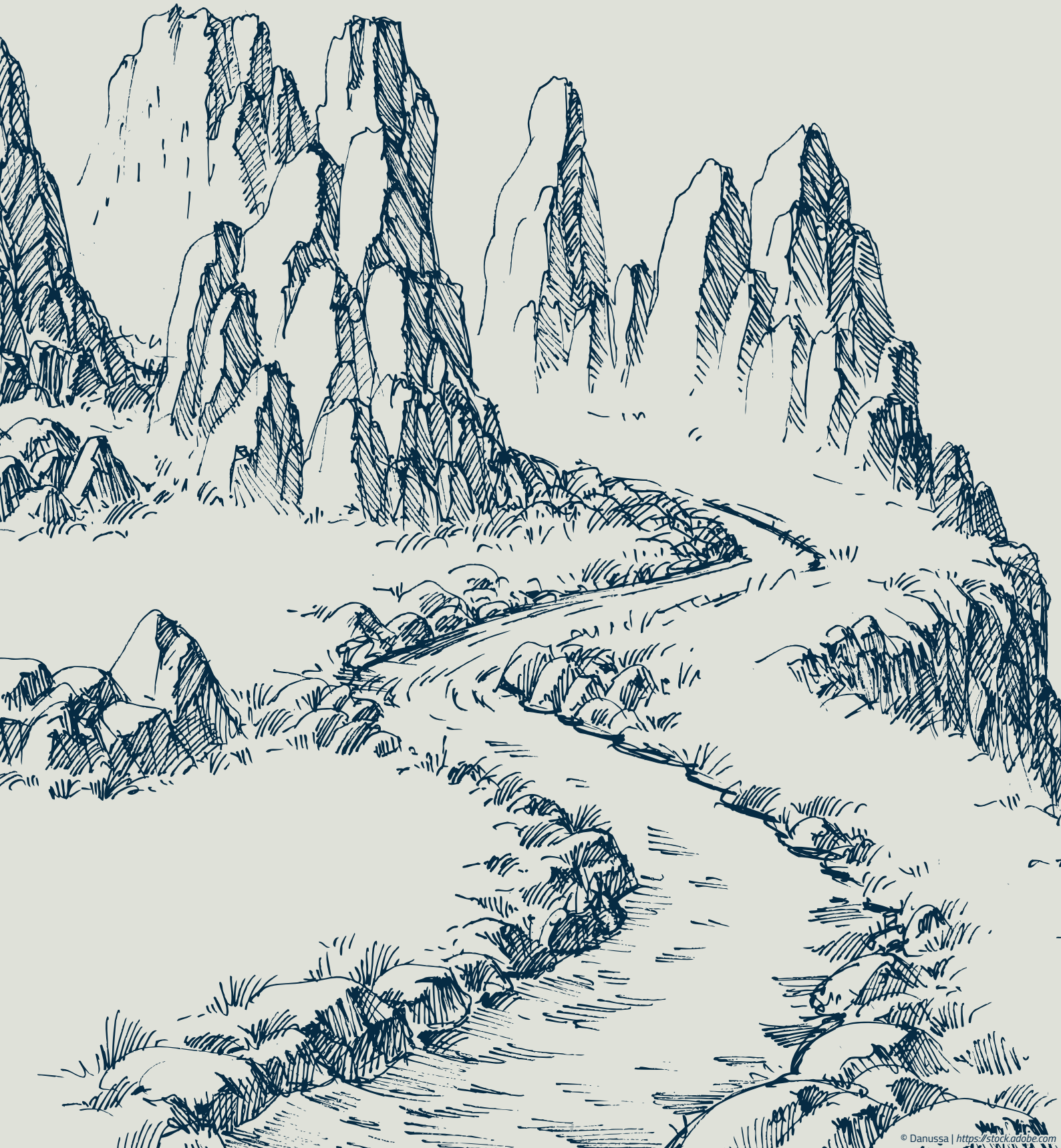
[9] N. Mainiero, „Comparison of 3 alternatives to the Criteria API," [Online]. Available: *https://github.com/nicolaimainiero/criteria-api-alternatives*.

Nicolai Mainiero
sidion GmbH
nicolai.mainiero@sidion.de

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

A Tale of Fail: Der steinige Weg zum nachhaltig erfolgreichen Cloud-Produkt

Bastian Glöckle, scailio GmbH



Nachhaltig erfolgreiche Cloud-Produkte zu entwickeln ist schwierig. Im Vergleich zur klassischen Enterprise-Softwareentwicklung unterscheiden sich insbesondere die technische Ausführungsumgebung sowie Anforderungen erheblich. Im Folgenden werden wir eine kleine Auswahl möglicher Probleme an einer beispielhaften Geschichte erleben und lernen, wie man das Risiko eines Scheiterns verringern kann: mit Technik, Architektur und Beachtung aller Teile des Gesamtsystems.

Unsere beiden Protagonistinnen sind Tanja Technik und Berta Business, die jeweils stellvertretend für die technischen Mitarbeiter/innen wie Softwareentwickler/innen und -architekten/innen sowie die eher wirtschaftlichen Mitarbeiter/innen wie beispielsweise Projektleiter/innen und Product Owner stehen.

Berta Business hat eine Geschäftsidee: „TheShop", eine neue App, mit der Kunden Artikel kaufen können. Zu jedem Zeitpunkt gibt es dabei immer nur genau einen Artikel zu kaufen, wobei die Gesamtstückzahl beschränkt ist. Nach Ablauf einer definierten Zeit steht der nächste Artikel zum Verkauf. Der Kniff von TheShop besteht darin, dass der Preis eines Artikels sinkt, je länger er zum Verkauf steht, die Verfügbarkeit aber gleichzeitig durch Verkäufe abnimmt. Dadurch erwartet Berta, dass die Kundinnen und Kunden geradezu an die App gefesselt werden und hohe Umsätze erzielt werden können.

Berta beauftragt die Umsetzung des Cloud-Backend bei Tanja. *Abbildung 1* zeigt die Requests der App für den Use-Case, der uns in dieser Geschichte begleiten wird: das Ansehen, Reservieren und Bestellen des aktuellen Artikels. Die Reservierung zu einem garantierten Preis wird sowohl von der App als auch vom Server nach fünf Minuten wieder aufgelöst, sollte keine Bestellung erfolgt sein. Die

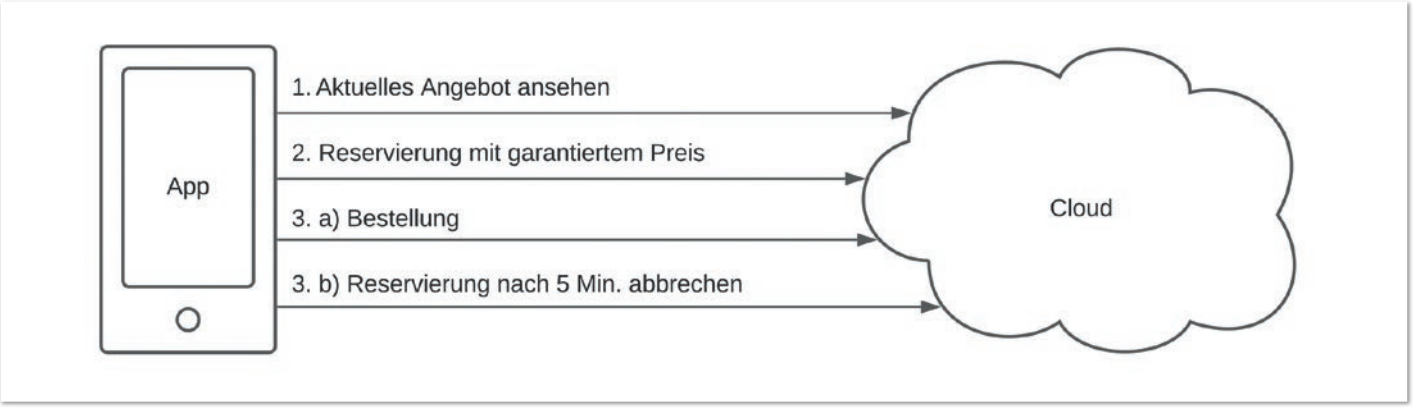


Abbildung 1: Requests der App (© Bastian Glöckle)

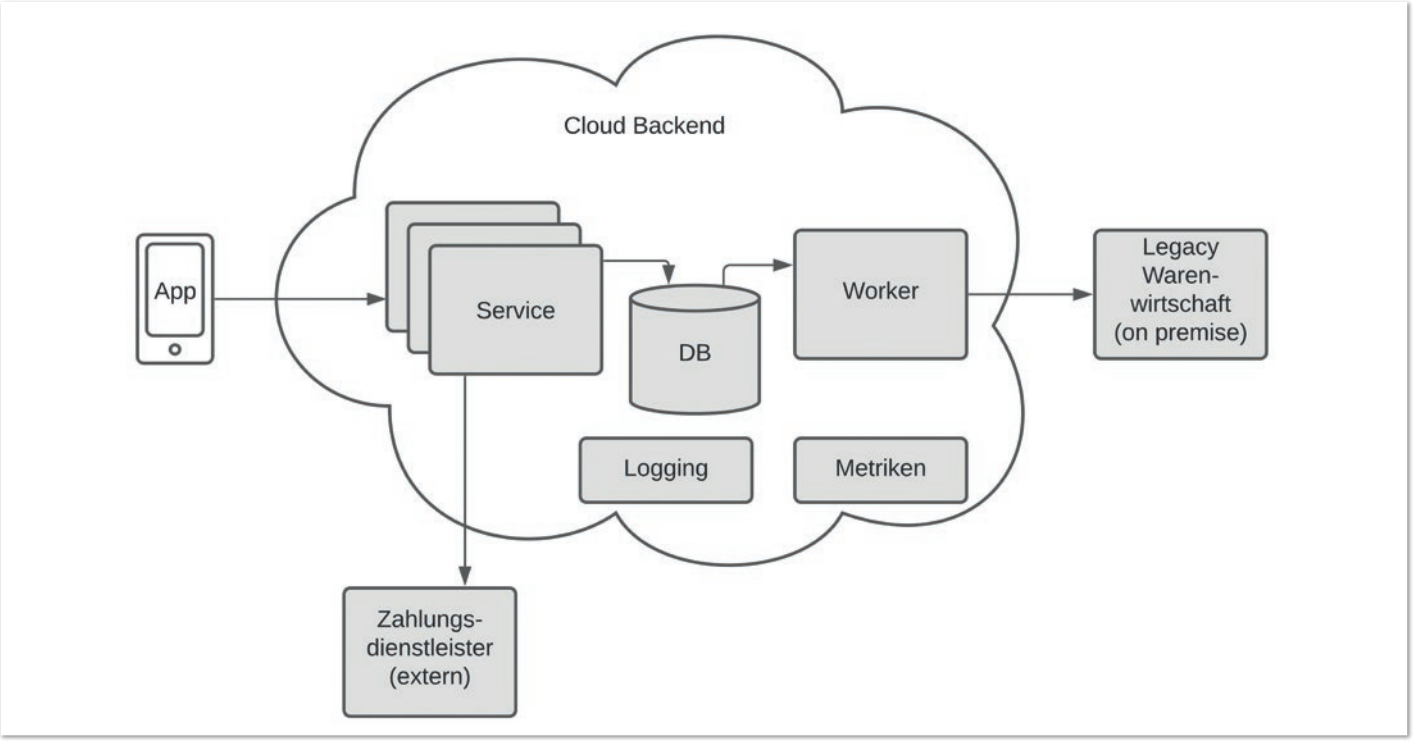


Abbildung 2: Systemübersicht und Datenfluss (© Bastian Glöckle)

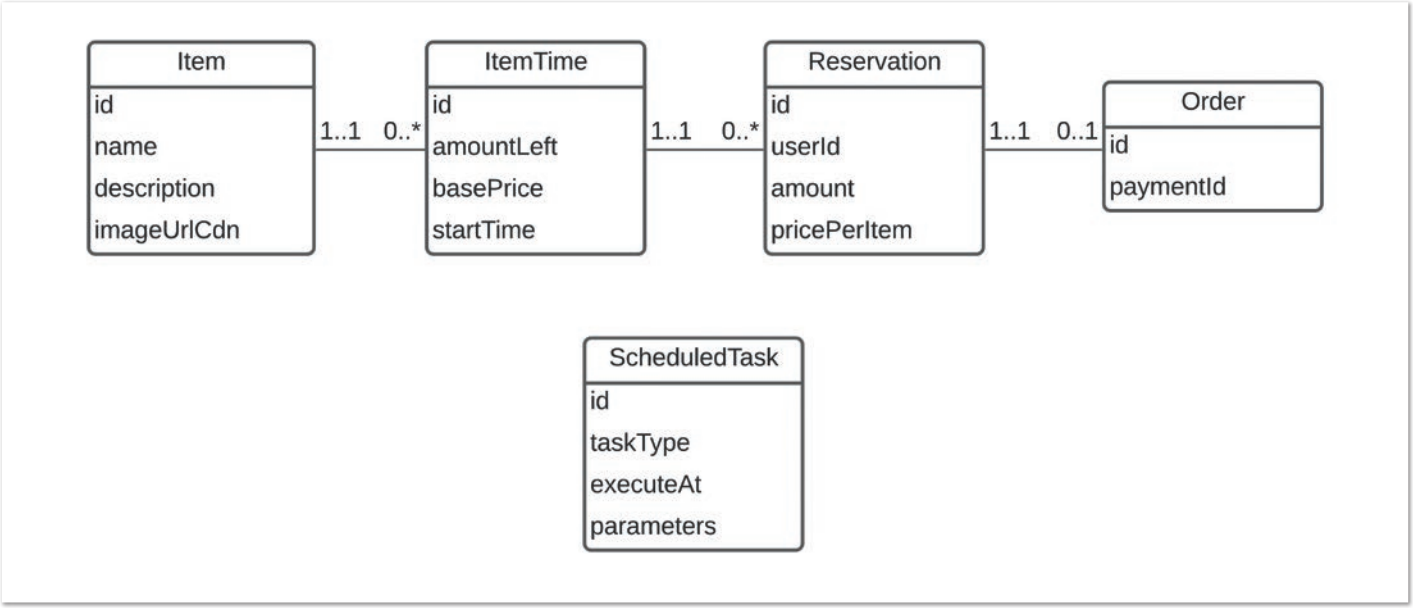


Abbildung 3: Datenmodell (© Bastian Glöckle)

App führt automatisch Retries aus, falls ein Request fehlschlägt, um etwa temporäre Verbindungsprobleme zu kaschieren.

In *Abbildung 2* sehen wir die Systemübersicht inklusive des Datenflusses: Die Aufrufe der App werden von einer der Instanzen eines zustandslosen Service bearbeitet. Dieser nutzt einen externen Zahlungsdienstleister via REST für Zahlungen und eine Datenbank zur Persistenz. Aus dieser werden die erfolgreichen Bestellungen von einem Worker asynchron in ein existierendes Warenwirtschaftssystem übernommen, das technisch entkoppelt sein soll und zum Versand der Bestellungen benötigt wird.

Außerdem ist, wie in der Cloud üblich, ein System für zentrales Logging (Beispiel: ELK Stack *[1]*) und eines für Metriken (Beispiel: Requests pro Sekunde in Prometheus mit Grafana *[2]*) integriert.

Im Rahmen dieses Artikels liegt der Fokus auf dem Service und den Interaktionen mit der Datenbank, wir lassen andere Aspekte wie Login, Security oder auch Loadbalancer außer Betracht.

Der Start: Agil ans Ziel

TheShop wird agil umgesetzt: Eine frühe Integration von App und Backend sowie schnelle Entwicklungszyklen sollen Kunden und Stakeholder überzeugen.

So starten Tanja und ihr Team mit der Entwicklung der Kernfunktionalität: dem Bestellprozess. Wie heute üblich, wird der Service zusammen mit der Datenbank in einem Kubernetes-Cluster deployt, das zum Beispiel sicherstellt, dass Services bei Fehlern korrekt neu gestartet werden. Das Cluster selbst wird von einem Public-Cloud-Anbieter betrieben. Tanja entscheidet sich für eine relationale SQL-Datenbank, mit der das Team Erfahrung hat, um schnell vorzeigbare Ergebnisse zu erzielen. *Abbildung 3* zeigt die Struktur der Tabellen in der Datenbank.

Die Tabelle ItemTime speichert die Stückzahl und die Zeit, ab wann ein Angebot verfügbar ist. Sowohl Item als auch ItemTime werden im Service zwischengespeichert, um Datenbankzugriffe zu minimieren. Der aktuelle Preis eines Artikels wird innerhalb des Service mithilfe einer Formel basierend auf Basispreis, Startzeitpunkt des Angebots und der aktuellen Zeit berechnet. Der externe Zahlungsdienstleister antwortet schließlich nach erfolgreicher Zahlung mit einer PaymentId, die mit der Bestellung gespeichert wird.

Listing 1 zeigt einen Ausschnitt des Codes für den Bestellprozess.

Die Entwicklung der zugehörigen App geht schnell voran und nach manuellen Tests sind auch anfängliche Fehler, wie fehlende Timeouts bei Remote-Aufrufen und deren fehlende explizite Behandlung, schnell beseitigt. In diesem Zustand entscheidet Berta Business mit allen Beteiligten, die App live zu schalten. Die Marketing- und Vertriebsmaschinerie startet gut und schafft es innerhalb kurzer Zeit, viele Tausend Kunden von TheShop zu überzeugen; die Käufe schnellen in die Höhe. Alles sieht gut aus.

Typische Probleme in der Cloud: Abrupte Neustarts und Inkonsistenzen

Nach einiger Zeit kommen aber immer mehr Rückmeldungen, dass bei Kunden Geld abgebucht wurde, ohne dass eine Bestellung erfasst wurde, und der Kunde somit auch keinen Artikel erhalten hat. Es entsteht mehrfach ein hoher buchhalterischer Aufwand, um die Zahlungen manuell zurückzubuchen. Gleichzeitig sind die Kunden verständlicherweise unzufrieden. Das führt zu schlechten Bewertungen der App in den App Stores. Berta ist besorgt.

Tanja und ihr Team stehen vor einem Rätsel: Die entsprechenden Kunden hatten eine Reservierung, die allerdings abgebrochen wurde, obwohl der Zahlungsdienstleister Erfolg gemeldet hatte – das Gesamtsystem ist inkonsistent. Schließlich fällt Tanja auf, dass Kubernetes die Serviceinstanzen wegen Ressourcenknappheit mehrfach neu gestartet hat. Da kommt ihr eine Idee: Was, wenn der Service gerade in dem Moment beendet wurde, nachdem die Bezahlung erledigt war, aber bevor die Bestellung in der Datenbank gespeichert wurde (vor `storeNewOrder` in *Listing 1*)? Durch Analyse der Logging-Daten bestätigt sich dieser Verdacht und der Grund für die fehlenden Bestellungen ist gefunden.

Lesson Learned

In der Cloud sind abrupte Neustarts von Services, fehlerhafte Disks und Netzwerke mit Schluckauf „normal", sowohl bei eigenen wie auch bei Drittanbieter-Services. Frei nach Murphy: „Anything that can go wrong will go wrong." Beim Schreiben jeder Zeile Code sollte bedacht werden, was passiert, wenn hier etwas fehlschlägt. Das Gesamtsystem darf niemals in einen (dauerhaft) inkonsistenten Zustand geraten.

Als Lösung entscheidet sich Tanja für die Implementierung einer „Transactional Outbox" *[3]*: Anstatt der direkten Ausführung der Zahlung und der Erstellung der Order wird nur eine Aufgabe in einer

```
@PostMapping("/reservation")
@Transactional
public Reservation newReservation(@RequestParam("a") int amount) {
  long userId = getUserFromSession();
  Reservation r = storeNewReservation(userId, amount);
  reduceAmountOrThrow(r.itemTime, amount);
  scheduleCancelAfterMin(r, 5);
  return r;
}

@PostMapping("/order")
public long order(@RequestParam("r") long reservationId) {
  Reservation r = getValidReservation(reservationId);
  String paymentId;
  try {
     paymentId = executePaymentViaRest(r);
  } catch (TimeoutException e) {
     throw new IntermediateException("Please retry", e);
  } catch (PaymentDeclinedException e) {
     cancelReservation(reservationId);
     throw new IllegalStateException("Payment failed", e);
  }

  try {
     return storeNewOrder(r, paymentId);
  } catch (OrderExistsException e) {
     rollbackPayment(paymentId);
     return loadOrderByReservation(r).id;
  }
}
```

Listing 1: Implementierung Bestellprozess

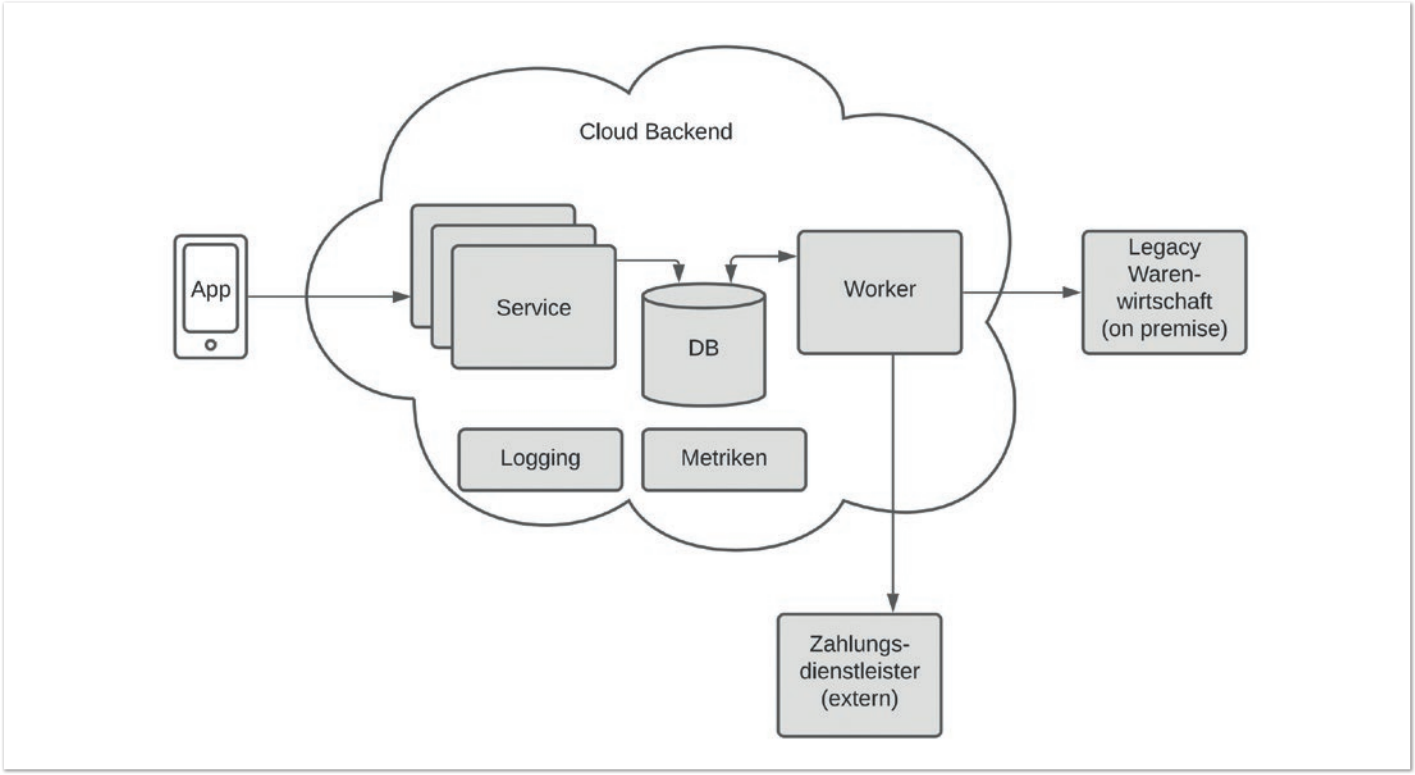


Abbildung 4: Angepasster Datenfluss (© Bastian Glöckle)

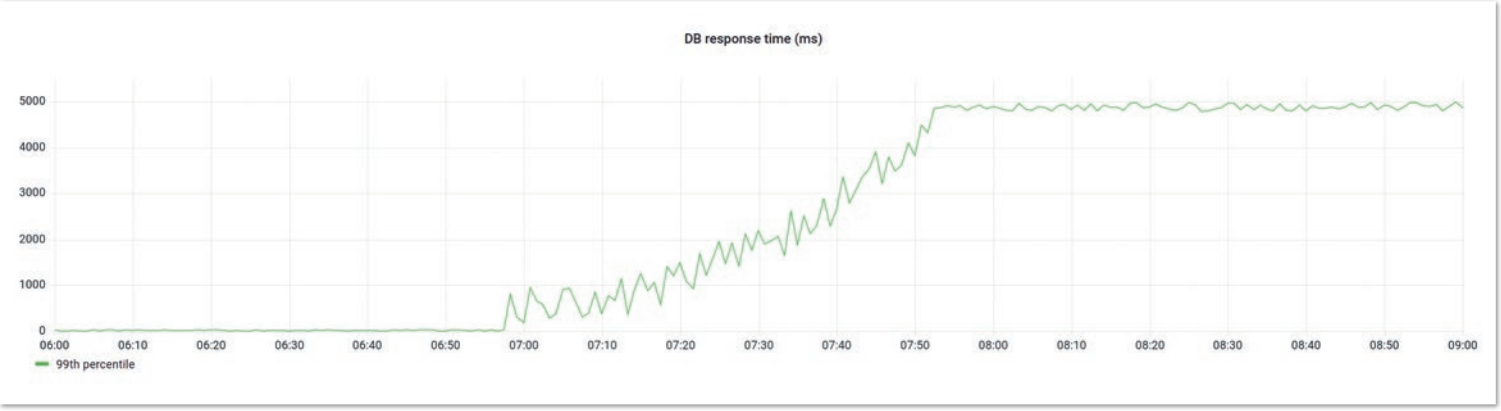


Abbildung 5: Antwortzeiten der Datenbank aus Sicht des Service

Outbox-Tabelle in der Datenbank angelegt und asynchron vom Worker abgearbeitet. Sollte der Worker während der Bearbeitung neustarten müssen, wird er an der Aufgabe weiterarbeiten, bei der er aufgehört hat. So wird eine schlussendlich vollständige Abarbeitung der notwendigen Schritte garantiert, ohne zusätzliche Komplexität durch nebenläufige Ausführung und ohne Gefahr für Doppelbuchungen durch Retries der App. Der neue Datenfluss ist in *Abbildung 4* dargestellt.

Die Anpassungen verändern allerdings die Garantie, die der Server bei der Antwort von `/order` gibt: Statt „abgeschlossen" ist die Bestellung nun lediglich gestartet. Dadurch sind auch Anpassungen am Code der App notwendig.

Lesson Learned

Konsistenzgarantien haben transitive Abhängigkeiten: Diejenigen der REST-Schicht hängen von denjenigen der Persistenzschicht ab. Wenn letztere wie bei der Einführung der Transactional-Outbox geändert werden, kann dies auch Änderungen an anderen Teilen des Gesamtsystems notwendig machen. Dies ist häufig sehr aufwendig und damit teuer – und in komplexen Projekten wesentlich schwieriger zu identifizieren als im vorliegenden Beispielprojekt.

Black Friday

Um die Gunst der Kunden zurückzugewinnen, plant Berta Business, beim nahenden Black Friday eine große Stückzahl von Artikeln zu sehr guten Preisen anzubieten. Die Marketingaktivitäten nehmen entsprechend zu und alle bereiten sich auf diesen Tag vor: Die Datenbank wird mit mehr CPU und Hauptspeicher ausgestattet und die Serviceinstanzen werden verdoppelt.

Leider kommt es dann aber so, wie es kommen musste: Kaum beginnt der Tag und die Kunden fangen an zu kaufen, steigt der Anteil der Fehler in den Services, bis das System schließlich vollständig zusammenbricht. Eine schnelle Analyse ergibt: Die Datenbank ist überlastet und antwortet, wenn überhaupt, nur noch sehr langsam – was Timeouts sowohl im Service als auch in der App auslöst, die dann wiederum Retries startet und die Situation weiter verschlimmert. *Abbildung 5* zeigt die Metrik der Antwortzeiten der Datenbank im gegebenen Zeitraum. Tanja bleibt kurzfristig nur eines übrig: Erhöhen der CPUs und des Speichers der Datenbank bis an die Grenzen dessen, was der Public-Cloud-Anbieter leisten kann, sowie eine massive Erhöhung der Serviceinstanzen – und hoffen, dass nun alles funktioniert.

Bis zum Abend ist dies erfolgreich. Dann kommt es aber erneut zu kurzen Ausfällen, die Tanja – am Ende ihrer Mittel – nicht mehr beheben kann.

Lesson Learned

Ohne ausreichend Metriken ist es sehr schwierig, Probleme in einem verteilten System schnell zu identifizieren.

Was war geschehen? Die tieferen Analysen in den folgenden Wochen ergeben, dass die Kunden ihr Verhalten geändert haben: Hatten bisher rund 3 % der Kunden einen Artikel reserviert, waren es wegen der guten Angebote plötzlich 30 %. Dadurch musste die Datenbank deutlich mehr Reservierungsanfragen verarbeiten *(siehe Listing 2)*. Diese Transaktionen sind ein Synchronisierungspunkt und benötigen eine Serialisierung aller Anfragen, da alle Vorgänge dasselbe Feld ändern: `amountAvailable`. Die meisten dieser Transaktionen müssen vergleichsweise aufwendig zurückgerollt werden, da der Artikel nicht ausreichend vorhanden ist. Durch die hohe Gesamtzahl von Transaktionen steigen außerdem die Zugriffe auf die Disk stark, da jede Transaktion zumindest kurzfristig persistiert werden muss – und so war die Disk der Datenbank selbst nach Erhöhung der Ressourcen nicht schnell genug.

Lesson Learned

Nutzer ändern ihr Verhalten plötzlich, was, unter anderem, Lasttests schwierig macht. Dies kann auch außerhalb der eigenen Kontrolle passieren, etwa bei unerwarteter Erwähnung des Produkts in reichweitenstarken Medien, durch die andere Kundensegmente aktiv werden.

Anstatt sich auf das bisherige Verhalten zu verlassen, sollten alle Synchronisationspunkte im Vorhinein identifiziert und architekto-

```
BEGIN;
  INSERT INTO Reservation (...) VALUES (...);
  UPDATE ItemTime
     SET amountAvailable = amountAvailable - :amount
     WHERE id = :itemTimeId
     RETURNING amountAvailable;
  -- in service: if amountAvailable < 0, rollback!
  INSERT INTO ScheduledTask (...) VALUES (...);
COMMIT;
```

Listing 2: Datenbanktransaktion „Reservierung"

nisch zumindest abgemildert werden (Stichwort „Graceful Degradation") – egal, in welchem Teil des Gesamtsystems sich diese befinden: interne Services, Datenbank oder externe Services.

Im Anschluss implementiert Tanja eine solche „Graceful Degradation", die den Synchronisationspunkt Datenbank im Fall der Fälle entlasten kann.

Listing 3 zeigt, wie nach einer manuellen Konfiguration ein bestimmter Prozentsatz der Reservierungsanfragen schon im Service grundlegend abgelehnt werden kann und damit nicht bis zur Datenbank durchgereicht wird, was diese entlastet. Die App führt in diesem Fall Retries aus – der Kunde muss dann gegebenenfalls länger warten, bis eine Reservierung gespeichert ist. Als Erweiterung kann der Prozentsatz der abgelehnten Anfragen auch automatisch angepasst werden, analog zum Circuit Breaker Pattern *[4]*. Es ist allerdings klar, dass dies nicht beliebig skaliert, da die Wartezeiten bei der Reservierung zu lang werden können und die Last auf den Service dabei sehr stark steigt.

Lesson Learned

Häufig verändern schon wenige Zeilen Code das Verhalten des Gesamtsystems maßgeblich.

Wir sind ja in der Cloud: 24/7

Berta Business hat auf Druck der Geschäftsleitung TheShop inzwischen auf verschiedene Länder in verschiedenen Zeitzonen ausgedehnt. Zusätzlich wird im Marketing das Rund-um-die-Uhr-Shopping hervorgehoben. Damit soll das Produkt wirtschaftlich gerettet werden. Wichtig ist aus Bertas Sicht, dass alle Märkte auf dasselbe System zugreifen. Das bedeutet, dass global nur ein Artikel gleichzeitig im Angebot sein darf und dieser nur eine zentrale verfügbare Stückzahl hat. Sorgen macht sich Berta hierbei nicht, denn „wir sind ja in der Cloud".

Lesson Learned

Unter „Cloud Software" verstehen viele, was sie von großen Internet-Plattformen wie etwa für Videostreaming geboten bekommen: Die Software ist immer verfügbar und skaliert beliebig.

Tanja hat neben den wachsenden Anforderungen an Skalierung auch mit Downtime zu kämpfen, die durch das Upgrade von Systemen entsteht: Die Datenbank und das Betriebssystem benötigen ein dringendes Sicherheitsupdate, durch das ein Neustart der Maschinen notwendig wird. Jede Downtime – egal zu welcher Zeit – bedeutet bei 24/7 automatisch den Verlust von Umsatz und steigende Unzufriedenheit bei Kunden.

Zur Lösung dieser Probleme könnte die SQL-Datenbank weiter optimiert werden. Die Verfügbarkeit könnte mittels eines Leader/Follower-Setups erhöht werden. Die Datenbank zu skalieren ist schwieriger, da selbst bei einem Sharding *[5]*, bei dem etwa Daten von Nutzern anhand ihrer `userId` auf separaten Datenbankinstanzen gespeichert werden, der Synchronisationspunkt bei der Reservierung bestehen bleibt: Nutzer aus verschiedenen Shards reservieren alle denselben Artikel. Die entsprechende Transaktion würde sogar zu einer verteilten Transaktion werden, was zusätzliche Komplexität im Gesamtsystem erzeugt.

Eine nachhaltige Alternative dazu ist der Wechsel zu verteilten Datenstrukturen für `amountAvailable` und einer passenden Da-

```
@PostMapping("/reservation")
public Reservation newReservation(@RequestParam("amount") int amount) {
  if (ThreadLocalRandom.current().nextInt(100) + 1 <= configuredPercent) {
     throw new IllegalStateException("Could not execute");
  }
  entityManager.getTransaction().begin();
  <...>
}
```

Listing 3: Ablehnung Reservierungsanfragen im Service

tenbank, die mehr Verfügbarkeit und Skalierung bietet. Allerdings würde dies Abstriche bei der Konsistenz bedeuten und damit nach Einführung der Transactional-Outbox wieder eine Änderung, die große Anpassungen auch in weiteren Teilen des Systems notwendig machen könnte. Die fehlende Erfahrung des Teams und die Möglichkeit, dass die Datenbank ihre selbst gegebenen Garantien nicht einhalten kann *[6]*, machen diese Variante nicht nur aufwendig, sondern auch risikoreich.

TheShop am Scheideweg

An dieser Stelle verlassen wir Berta, Tanja und TheShop. Das Produkt steht am Scheideweg. Entweder man investiert in große Architekturänderungen wie die Einführung verteilter Datenstrukturen oder man reduziert die Anforderungen an das System: Vollständig getrennte Deployments für einzelne Märkte und damit getrennte Stückzahlen pro Markt könnten ein möglicher Weg ohne Konsistenzänderungen sein, allerdings nur, wenn die einzelnen Märkte nicht zu stark wachsen.

Fazit

Wir haben an diesem zugegebenermaßen einfachen und konstruierten Beispiel gesehen, wie viele Cloud-Projekte im Kern ablaufen: Nichtfunktionale Anforderungen werden initial nur schlecht kommuniziert oder ändern sich stark und der Fokus liegt auf schnellen Entwicklungszyklen. Die Techniker setzen dementsprechend auf Techniken, die einfach zu benutzen sind, aber rückblickend große Architekturänderungen notwendig machen.

Durch klare Definition auch der nichtfunktionalen Anforderungen – insbesondere hinsichtlich Verfügbarkeit und Skalierung –, durch konsequentes architektonisches Behandeln der Synchronisationspunkte und schließlich durch einen konstanten Blick auf Fehlersituationen und das System als Ganzes schon vor dem Go-live können diese Probleme minimiert werden.

Dabei geht es nicht darum, schon von Anfang an die technische Nonplusultra-Variante zu implementieren, sondern Kompromisse bewusst einzugehen: Man kann die Serviceschicht beispielsweise von Anfang an so entwerfen, dass sie bei einem eventuell notwendigen Wechsel zu einer Datenbank mit weniger Konsistenzgarantien nicht angepasst werden muss.

Abschließend zu unterstreichen ist, dass ein Cloud-System in der Public Cloud immer auf Hard- und Software des Anbieters basiert. Diese liegt außerhalb der eigenen Kontrolle und entwickelt sich selbstständig weiter. Das bedeutet, dass ein System, das gestern noch gut funktioniert hat, morgen plötzlich ausfallen oder sein Verhalten ändern kann oder sogar abgekündigt wird und somit teure Architekturänderungen im eigenen System kurzfristig notwendig werden.

Ein Cloud-System ist nie ein stabiles System, das einmalig implementiert und dann nur noch betrieben werden muss. Ein Cloud-System lebt. Und der Kunde ist daran interessiert, dass das Gesamtsystem funktioniert, nicht daran, welcher Teil ausgefallen ist und wer daran Schuld hat.

Quellen

[1]: Was ist der ELK Stack?, *https://www.elastic.co/de/what-is/elk-stack*

[2]: Grafana Support for Prometheus, *https://prometheus.io/docs/visualization/grafana/*

[3]: Pattern: Transactional outbox, *https://microservices.io/patterns/data/transactional-outbox.html*

[4]: Circuit breaker design pattern, *https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern*

[5]: Shard (database architecture), *https://en.wikipedia.org/wiki/Shard_(database_architecture)*

[6]: Jepsen Analyses, *https://jepsen.io/analyses*

Bastian Glöckle
scailio GmbH
bastian@scail.io

Bastian unterstützt mit der scailio GmbH Firmen dabei, ihre Cloud-Produkte nachhaltig erfolgreich zu entwickeln und betreiben, mit einem Fokus auf Skalierung und Verfügbarkeit. Als Hands-on-Architekt überbrückt er die Lücke zwischen dem Businessplan und der technischen Realität auf allen Ebenen des Systems. Er kann dabei auf umfangreiche Erfahrung zurückgreifen, unter anderem auf seine Entwicklertätigkeit bei einem großen Suchmaschinenunternehmen, aber auch auf den Entwurf, die Entwicklung und den Betrieb von Systemen mit mehreren Hunderttausend aktiven Nutzern.

Der grüne Punkt – Mythos Wiederverwendung

Elmar Dott

Als mir im Studium die Vorzüge der objektorientierten Programmierung mit Java schmackhaft gemacht wurden, war ein sehr beliebtes Argument die Wiederverwendung. Dass der Grundsatz „write once use everywhere" in der Praxis dann doch nicht so leicht umzusetzen ist, wie es die Theorie suggeriert, haben die meisten Entwickler bereits am eigenen Leib erfahren. Woran liegt es also, dass die Idee der Wiederverwendung in realen Projekten so schwer umzusetzen ist? Machen wir also einen gemeinsamen Streifzug durch die Welt der Informatik und betrachten verschiedene Vorhaben aus sicherer Distanz.

Wenn ich daran denke, wie viel Zeit ich während meines Studiums investiert habe, um eine Präsentationsvorlage für Referate zu erstellen. Voller Motivation habe ich alle erdenklichen Ansichten in weiser Voraussicht erstellt. Selbst rückblickend war das damalige Layout für einen Nichtgrafiker ganz gut gelungen. Trotzdem kam die tolle Vorlage nur wenige Male zum Einsatz und wenn ich im Nachhinein einmal Resümee ziehe, komme ich zu dem Schluss, dass die investierte Arbeitszeit in Bezug auf die tatsächliche Verwendung in keinem Verhältnis gestanden hat. Von den vielen verschiedenen Ansichten habe ich zum Schluss exakt zwei verwendet, das Deckblatt und eine allgemeine Inhaltsseite, mit der alle restlichen Darstellungen umgesetzt wurden. Die restlichen 15 waren halt da, falls man das künftig noch brauchen würde. Nach dieser Erfahrung plane ich keine eventuell zukünftig eintreffenden Anforderungen mehr im Voraus. Denn den wichtigsten Grundsatz in Sachen Wiederverwendung habe ich mit dieser Lektion für mich gelernt: Nichts ist so beständig wie die Änderung.

Diese kleine Anekdote trifft das Thema bereits im Kern. Denn viele Zeilen Code werden genau aus der gleichen Motivation heraus geschrieben. Der Kunde hat es noch nicht beauftragt, doch die Funktion wird er ganz sicher noch brauchen. Wenn wir in diesem Zusam-

menhang einmal den wirtschaftlichen Kontext ausblenden, gibt es immer noch ausreichend handfeste Gründe, durch die Fachabteilung noch nicht spezifizierte Funktionalität nicht eigenmächtig im Voraus zu implementieren. Für mich ist nicht verwendeter, auf Halde produzierter Code – sogenannter toter Code – in erster Linie ein Sicherheitsrisiko. Zusätzlich verursachen diese Fragmente auch Wartungskosten, da bei Änderungen auch diese Bereiche möglicherweise mit angepasst werden müssen. Schließlich muss die gesamte Codebasis kompilierfähig bleiben. Zu guter Letzt kommt noch hinzu, dass die Kollegen oft nicht wissen, dass bereits eine ähnliche Funktion entwickelt wurde, und diese somit ebenfalls nicht verwenden. Die Arbeit wird also auch noch doppelt ausgeführt. Nicht zu vergessen ist auch das von mir in großen und langjährig entwickelten Applikationen oft beobachtete Phänomen, dass ungenutzte Fragmente aus Angst, etwas Wichtiges zu löschen, über Jahre hinweg mitgeschleppt werden. Damit kommen wir auch schon zum zweiten Axiom der Wiederverwendung: Erstens kommt es anders und zweitens als man denkt.

Über die vielen Jahre, genauer gesagt Jahrzehnte, in denen ich nun verschiedenste IT- beziehungsweise Softwareprojekte begleitet habe, habe ich ein Füllhorn an Geschichten aus der Kategorie „Das hätte ich mir sparen können!" angesammelt. Virtualisierung ist nicht erst seit Docker *[1]* auf der Bildfläche erschienen – es ist schon weitaus länger ein beliebtes Thema. Die Menge der von mir erstellten virtuellen Maschinen (VMs) kann ich kaum noch benennen – zumindest waren es sehr viele. Für alle erdenklichen Einsatzszenarien hatte ich etwas zusammengebaut. Auch bei diesen tollen Lösungen erging es mir letztlich nicht viel anders als bei meiner Office-Vorlage. Grundsätzlich gab es zwei Faktoren, die sich negativ ausgewirkt haben. Je mehr VMs erstellt wurden, desto mehr mussten dann auch gewertet werden. Ein Worst-Case-Szenario heutzutage wäre eine VM, die auf Windows 10 basiert, die dann jeweils als eine .NET- und eine Java-Entwicklungsumgebung oder Ähnliches spezialisiert wurde. Allein die Stunden, die man für Updates zubringt, wenn man die Systeme immer mal wieder hochfährt, summieren sich auf beachtliche Größen. Ein Grund für mich zudem, soweit es geht, einen großen Bogen um Windows 10 zu machen. Aus dieser Perspektive können selbsterstellte Docker-Container schnell vom Segen zum Fluch werden.

Dennoch darf man diese Dinge nicht gleich überbewerten, denn diese Aktivitäten können auch als Übung verbucht werden. Wichtig ist, dass solche „Spielereien" nicht ausarten und man neben den technischen Erfahrungen auch den Blick für tatsächliche Bedürfnisse auf lange Sicht schärft.

Gerade bei Docker bin ich aus persönlicher Erfahrung dazu übergegangen, mir die für mich notwendigen Anpassungen zu notieren und zu archivieren. Komplizierte Skripte mit Docker-Compose spare ich mir in der Regel. Der Grund ist recht einfach: Die einzelnen Komponenten müssen zu oft aktualisiert werden und der Einsatz für jedes Skript findet in meinem Fall genau einmal statt. Bis man nun ein lauffähiges Skript zusammengestellt hat, benötigt man, je nach Erfahrung, mehrere oder weniger Anläufe. Also modifiziere ich das RUN-Kommando für einen Container, bis dieser das tut, was ich von ihm erwarte. Das vollständige Kommando hinterlege ich in einer Textdatei, um es bei Bedarf wiederverwenden zu können. Dieses Vorgehen nutze ich für jeden Dienst, den ich mit Docker virtualisiere. Dadurch habe ich die Möglichkeit, verschiedenste Konstellationen mit minimalen Änderungen nach dem „Klemmbaustein"-Prinzip zu orchestrieren. Wenn sich abzeichnet, dass ein Container sehr oft unter gleichen Bedienungen instanziiert wird, ist es sehr hilfreich, diese Konfiguration zu automatisieren. Nicht ohne Grund gilt für Docker-Container die Regel, möglichst nur einen Dienst pro Container zu virtualisieren.

Aus diesen beiden kleinen Geschichten lässt sich bereits einiges für Implementierungsarbeiten am Code ableiten. Ein klassischer Stolperstein, der mir bei der täglichen Projektarbeit regelmäßig unterkommt, ist, dass man mit der entwickelten Applikation eine eierlegende Wollmilchsau – oder, wie es in Österreich heißt: ein Wunderwutzi – kreieren möchte. Die Teams nehmen sich oft zu viel vor und das Projektmanagement versucht, den Product Owner auch nicht zu bekehren, lieber auf Qualität statt auf Quantität zu setzen. Was ich mit dieser Aussage deutlich machen möchte, lässt sich an einem kleinen Beispiel verständlich machen.

Gehen wir einmal davon aus, dass für eigene Projekte eine kleine Basisbibliothek benötigt wird, in der immer wiederkehrende Problemstellungen zusammengefasst werden – konkret: das Verarbeiten von JSON-Objekten *[2]*. Nun könnte man versuchen, alle erdenklichen Variationen im Umgang mit JSON abzudecken. Abgesehen davon, dass viel Code produziert wird, erzielt ein solches Vorgehen wenig Nutzen. Denn für so etwas gibt es bereits fertige Lösungen – etwa die freie Bibliothek Jackson *[3]*. Anstelle sämtlicher denkbarer JSON-Manipulationen ist in Projekten vornehmlich das Serialisieren und das Deserialisieren gefragt. Also eine Möglichkeit, wie man aus einem Java-Objekt einen JSON-String erzeugt, und umgekehrt. Diese beiden Methoden lassen sich leicht über eine Wrapper-Klasse zentralisieren. Erfüllt nun künftig die verwendete JSON-Bibliothek die benötigten Anforderungen nicht mehr, kann sie leichter durch eine besser geeignete Bibliothek ersetzt werden. Ganz nebenbei erhöhen wir mit diesem Vorgehen auch die Kompatibilität *[4]* unserer Bibliothek für künftige Erweiterungen. Wenn JSON im Projekt eine neu eingeführte Technologie ist, kann durch die Minimal-Implementierung stückweise Wissen aufgebaut werden. Je stärker der JSON-Wrapper nun in eigenen Projekten zum Einsatz kommt, desto wahrscheinlicher ist es, dass neue Anforderungen hinzukommen, die dann erst umgesetzt werden, wenn sie durch ein Projekt angefragt werden. Denn wer kann schon abschätzen, wie der tatsächliche Bedarf einer Funktionalität ist, wenn so gut wie keine Erfahrungen zu der eingesetzten Technologie vorhanden sind?

Das soeben beschriebene Szenario läuft auf einen einfachen Merksatz hinaus: Eine neue Implementierung möglichst so allgemein wie möglich halten, um sie nach Bedarf immer weiter zu spezialisieren.

Bei komplexen Fachanwendungen hilft uns das Domain-driven Design (DDD) Paradigma, Abgrenzungen zu Domänen ausfindig zu machen. Auch hierfür lässt sich ein leicht verständliches, allgemein gefasstes Beispiel finden. Betrachten wir dazu einmal die Domäne einer Access Control List (ACL). In der ACL wird ein Nutzerkonto benötigt, mit dem Berechtigungen zu verschiedenen Ressourcen verknüpft werden. Nun könnte man auf die Idee kommen, im Account in der ACL sämtliche Benutzerinformationen wie Homepage, Postadresse und Ähnliches abzulegen. Genau dieser Fall würde die Domäne der ACL verletzen, denn das Benutzerkonto benötigt lediglich

Informationen, die zur Authentifizierung benötigt werden, um eine entsprechende Autorisierung zu ermöglichen.

Jede Anwendung hat für das Erfassen der benötigten Nutzerinformationen andere Anforderungen, weshalb diese Dinge nicht in eine ACL gehören sollten. Das würde die ACL zu sehr spezialisieren und stetige Änderungen verursachen. Daraus resultiert dann auch, dass die ACL nicht mehr universell einsatzfähig ist.

Man könnte nun auf die Idee kommen, eine sehr generische Lösung für den Speicher zusätzlicher Nutzerinformationen zu entwerfen und ihn in der ACL zu verwenden. Von diesem Ansatz möchte ich abraten. Ein wichtiger Grund ist, dass diese Lösung die Komplexität der ACL unnötig erhöht. Ich gehe obendrein so weit und möchte behaupten, dass unter ungünstigen Umständen sogar Code-Dubletten entstehen. Die Begründung dafür ist wie folgt: Ich sehe eine generische Lösung zum Speichern von Zusatzinformationen im klassischen Content Management (CMS) verortet. Die Verknüpfung zwischen ACL und CMS erfolgt über die Benutzer-ID aus der ACL. Somit haben wir gleichzeitig auch zwischen den einzelnen Domänen eine lose Kopplung etabliert, die uns bei der Umsetzung einer modularisierten Architektur sehr behilflich sein wird.

Zum Thema Modularisierung möchte ich auch kurz einwerfen, dass Monolithen *[5]* durchaus auch aus mehreren Modulen bestehen können und sogar sollten. Es ist nicht zwangsläufig eine Microservice-Architektur notwendig. Module können aus unterschiedlichen Blickwinkeln betrachtet werden. Einerseits erlauben sie es einem Team, in einem fest abgegrenzten Bereich ungestört zu arbeiten, zum anderen kann ein Modul mit einer klar abgegrenzten Domäne ohne viele Adaptionen tatsächlich in späteren Projekten wiederverwendet werden.

Nun ergibt sich klarerweise die Fragestellung, was mit dem Übergang von der Generalisierung zur Spezialisierung gemeint ist. Auch hier hilft uns das Beispiel der ACL weiter. Ein erster Entwurf könnte die Anforderung haben, dass, um unerwünschte Berechtigungen falsch konfigurierter Rollen zu vermeiden, die Vererbung von Rechten bestehender Rollen nicht erwünscht ist. Daraus ergibt sich dann der Umstand, dass jedem Nutzer genau eine Rolle zugewiesen werden kann. Nun könnte es sein, dass durch neue Anforderungen der Fachabteilung eine Mandantenfähigkeit eingeführt werden soll. Entsprechend muss nun in der ACL eine Möglichkeit geschaffen werden, um bestehende Rollen und auch Nutzeraccounts einem Mandanten zuzuordnen. Eine Domänen-Erweiterung dieser hinzugekommenen Anforderung ist nun basierend auf der bereits bestehenden Domäne durch das Hinzufügen neuer Tabellenspalten leicht umzusetzen.

Die bisher aufgeführten Beispiele beziehen sich ausschließlich auf die Implementierung der Fachlogik. Viel komplizierter verhält sich das Thema Wiederverwendung beim Punkt der grafischen Benutzerschnittelle (GUI). Das Problem, das sich hier ergibt, ist die Kurzlebigkeit vieler Technologien. Java Swing existiert zwar noch, aber vermutlich würde sich niemand, der heute eine neue Anwendung entwickelt, noch für Java Swing entscheiden. Der Grund liegt in veraltetem Look-and-Feel der Grafikkomponenten. Um eine Applikation auch verkaufen zu können, darf man den Aspekt der Optik nicht außen vor lassen. Denn auch das Auge isst bekanntlich mit. Gerade bei sogenannten Green-Field-Projekten ist der Wunsch, eine moderne, ansprechende Oberfläche anbieten zu können, implizit. Deswegen vertrete ich die Ansicht, dass das Thema Wiederverwendung für GUI – mit wenigen Ausnahmen – keine wirkliche Rolle spielt.

Lessons Learned

Sehr oft habe ich in der Vergangenheit erlebt, wie enthusiastisch bei Kick-off-Meetings die Möglichkeit der Wiederverwendung von Komponenten in Aussicht gestellt wurde. Dass dies bei den verantwortlichen Managern zu einem Glitzern in den Augen geführt hat, ist auch nicht verwunderlich. Als es dann allerdings zu ersten konkreten Anfragen gekommen ist, eine Komponente in einem anderen Projekt einzusetzen, mussten sich alle Beteiligten eingestehen, dass dieses Vorhaben gescheitert war. In den nachfolgenden Retrospektiven sind die Punkte, die ich in diesem Artikel vorgestellt habe, regelmäßig als Ursachen identifiziert worden. Im Übrigen genügt oft schon ein Blick in das Datenbankmodell oder auf die Architektur einer Anwendung, um eine Aussage treffen zu können, wie realistisch eine Wiederverwendung tatsächlich ist. Bei steigendem Komplexitätsgrad sinkt die Wahrscheinlichkeit, auch nur kleinste Segmente erfolgreich für eine Wiederverwendung herauslösen zu können.

Referenzen

[1] *https://www.docker.com*
[2] *https://www.json.org*
[3] *https://github.com/FasterXML/jackson/*
[4] *https://entwickler.de/api/futurama-nichts-ist-so-bestandig-wie-die-veranderung/*
[5] *https://entwickler.de/microservices/slice-down-the-monolith-001/*

Elmar Dott
elmar.dott@gmail.com

Elmar Dott (Marco Schulz) studierte an der HS Merseburg Diplominformatik und twittert regelmäßig als @ElmarDott über alle möglichen technischen Themen. Seine Schwerpunkte sind hauptsächlich Build- und Konfigurationsmanagement, Software-Architekturen und Release-Management. Seit über fünfzehn Jahren realisiert er in internationalen Projekten für namhafte Unternehmen umfangreiche Webapplikationen. Er ist freier Consultant/Trainer. Sein Wissen teilt er mit anderen Technikbegeisterten auf Konferenzen, wenn er nicht gerade wieder einmal an einem neuen Fachbeitrag schreibt. Seine Homepage finden Sie unter: *https://elmar-dott.com*

Design for Change: Effiziente Softwareentwicklung

Ingmar Kellner, hello2morrow GmbH

In einer global vernetzten Welt ist die Konkurrenz immens. Dies gilt besonders für den Bereich Softwareentwicklung, wo es meistens keine Rolle spielt, ob der Programmierer in Deutschland, Estland, Indien oder den USA sitzt. Um auf Dauer mit einem Projekt erfolgreich zu sein, wird es daher immer wichtiger, Funktionalität effizient umzusetzen. In diesem Artikel werde ich den Einfluss von Kopplung auf die effiziente Programmierung herausstellen und darüber hinaus, wie man diese minimiert und automatisiert kontrollieren kann.

Um herauszufinden, wie sich die Softwareentwicklung effizienter gestalten lässt, muss man sich darüber klar werden, wo und wann Probleme und Kosten entstehen: Üblicherweise besteht ein kleiner Teil der Kosten in Software-Projekten aus Hardware, beispielsweise für Laptops und Büroausstattung. Der Großteil liegt im personellen Bereich, das heißt, dominant sind hier die zu zahlenden Arbeitsstunden der Entwickler. Will man also eine effizientere Entwicklung, muss die Arbeitszeit der Entwickler besser genutzt werden.

Mit „agilen" Entwicklungsprozessen werden Probleme gelöst, die „Wasserfall"-artig durchgeführte Projekte mit sich bringen: Lange Analyse- und Design-Phasen mit später Validierung durch Anwender, wodurch oftmals nicht die richtigen Funktionalitäten entwickelt werden, werden ersetzt durch die schnelle Rückkopplung und den sinnvollen Fokus auf das Minimum Viable Product (MVP, zu Deutsch „minimal brauchbares Produkt"). Es werden permanent neue Features entwickelt oder existierende den geänderten Kundenwünschen angepasst. Dabei entwickelt sich auch das Wissen des Entwicklungsteams über die Domäne weiter und diese Erkenntnisse fließen über Refactorings wiederum als Änderungen in die Codebasis ein.

Es haben sich also Methoden etabliert, wie sich die „richtige" Funktionalität entwickeln lässt. Legen wir nun den Fokus darauf, wie besser „richtig" entwickelt werden kann und was notwendig ist, um Änderungen effizienter zu implementieren.

Dabei abstrahieren wir von Technologie und betrachten das Thema unabhängig von existierenden Frameworks wie etwa Spring oder Java EE.

Kosten von Änderungen

Die folgenden Sätze klingen sicher trivial, bilden jedoch die notwendige Begründung für die Wichtigkeit des Themas „Kopplung": Bei der Anpassung von Funktionalität muss man bestehenden Code ändern. In der Regel ist das auch der Fall, wenn neue Features eingebaut werden, denn diese werden in das bestehende System integriert. Je geringer die Wechselwirkungen im Code, desto weniger Code muss angepasst werden.

Die Wechselwirkung zwischen Teilen eines Systems bezeichnet man auch als „Kopplung" oder „Abhängigkeit". Am einfachsten ist diese zu erkennen, wenn die Abhängigkeiten durch Referenzen im Code erfolgen, etwa indem die Klasse die Klasse Customer verwendet, wie in *Abbildung 1* in Form eines UML-Diagramms dargestellt:

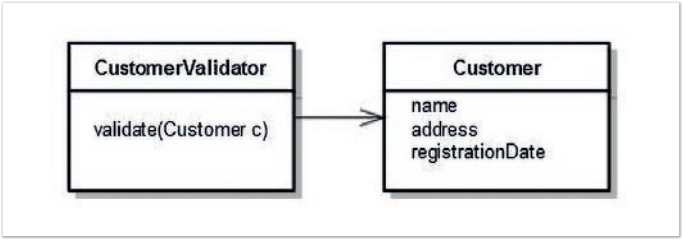


Abbildung 1: Kopplung zwischen Klassen (© Ingmar Kellner)

Ändert sich die Klasse Customer, etwa wenn ein neues Attribut hinzugefügt wird, muss die Klasse `CustomerValidator` ebenfalls angepasst werden, da alle Attribute validiert werden. Je weniger Referenzen auf Customer im Code existieren, desto weniger kann das System durch Änderungen an Customer beeinflusst werden, was zu geringeren Aufwänden und damit geringeren Kosten führt. Der Ansatz, die Abhängigkeiten zu minimieren (etwa, indem Interfaces eingeführt werden für die Umsetzung des „Dependency Inversion Principle"), wird auch als „lose Kopplung" bezeichnet.

Es gilt allerdings zu beachten, dass die oben genannte statische Kopplung nicht die einzige Form der Kopplung ist. Kopplung kann auch über gemeinsam verwendete Datenstrukturen entstehen, beispielsweise über Keys in Maps, zeitlich über ein festgelegtes Protokoll von Methodenaufrufen etc. Explizite Kopplung, die sich über Referenzen im Code nachverfolgen lässt, ist für den Entwickler am einfachsten zu verstehen und lässt sich über Werkzeuge automatisiert überwachen. Aber dazu später mehr.

Erstrebenswert ist also ein Design, in dem Änderungen möglichst lokale Auswirkungen haben. Hat man während der Umsetzung das Gefühl, dass sich die notwendigen Anpassungen wie ein Lauffeu-

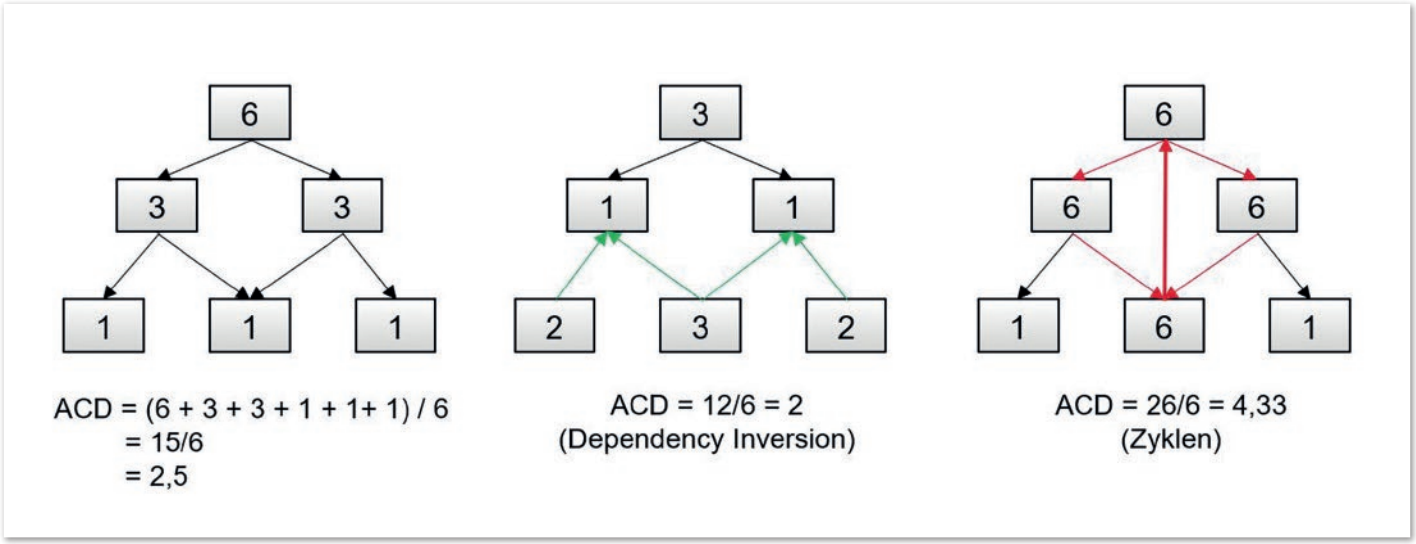


Abbildung 2: „Average Component Dependency"-Berechnung (© Ingmar Kellner)

er auf viele eigentlich unbeteiligte Klassen des Systems auswirken, dann ist das ein Warnzeichen für eine starke Kopplung. Schauen wir uns nun an, wie Kopplung messbar ist, bevor wir uns damit beschäftigen, wie sie sich minimieren lässt.

Messen von Kopplung

Es gibt einige Metriken, um die Kopplung eines gesamten Systems wie auch einzelner Klassen zu bestimmen. Am einfachsten zu verstehen finde ich die von John Lakos beschriebenen Metriken „Depends Upon" und „Average Component Dependency" (ACD) *[1]*. Der Begriff „Komponente" ist hierbei abstrakt zu verstehen und kann ein Modul, ein Package, eine Datei oder eine Klasse sein. „Depends Upon" berechnet sich aus der Anzahl direkter und indirekter Abhängigkeiten (+1 für die Komponente selbst). „Average Component Dependency" ist nun der Durchschnitt der „Depends Upon"-Werte. In *Abbildung 2* wird die Berechnung am Beispiel von wenigen Komponenten mit unterschiedlichen Abhängigkeiten deutlich. Der positive Effekt des „Dependency Inversion"-Prinzips (Mitte) auf die Kopplungsmetriken ist ebenso ersichtlich wie der negative Effekt von Zyklen (rechts).

Zyklische Abhängigkeiten erhöhen nicht nur abstrakt die Werte von Kopplungsmetriken, sondern auch ganz konkret die Auswirkung von Änderungen auf das System. In den seltensten Fällen sind zyklische Abhängigkeiten gewollt, da das Verständnis des Systems dadurch massiv erschwert wird. Diese Komponenten können nur gemeinsam verstanden, getestet und wiederverwendet werden. Es ist nicht möglich, eine Hierarchie aufzubauen und sich entweder Top-down oder Bottom-up in das System einzuarbeiten. Kleine Zyklen sehen noch nicht problematisch aus, in der Praxis sind Zyklen mit mehr als 100 Elementen keine Seltenheit. Visualisiert man diese Abhängigkeiten, entstehen schön anzusehende Bilder wie in *Abbildung 3*, die aber leider nur den existierenden „Big Ball of Mud" *[2]* repräsentieren: Eine klare Struktur ist nicht erkennbar, die Auswirkungen einer Änderung sind kaum abzuschätzen.

Konsistente Struktur auf allen Abstraktionsebenen

Aus dem vorangegangenen Abschnitt ist die Bedeutung der „Kopplung" auf die Änderbarkeit von Software deutlich geworden. Um eine Änderung vorzunehmen, muss der Entwickler aber auch wissen, welcher Code zu ändern ist beziehungsweise wo neuer Code eingebaut werden muss. Das heißt, die Software muss „navigierbar" und sinnvoll zerlegt sein. Die spannende Frage ist, wie das System in Einzelteile zerlegt werden sollte. Hier kommt die im vorigen Abschnitt erwähnte Kopplung ins Spiel: Es wird einfacher, die einzelnen Teile zu verstehen und zu implementieren, wenn man sie isoliert voneinander betrachten kann.

Eine weitere Hilfsgröße ist das Limit von Dingen, die das menschliche Hirn gleichzeitig vorhalten kann. Dieses Limit findet sich auch in anderen Bereichen wieder, beispielsweise in der Organisation von Unternehmen. Sobald ein Teil aus mehr als 15 bis 20 Einheiten besteht, ist eine Gruppierung in einen Container als weitere Abstraktion sinnvoll. Empfehlungen für die Dekomposition eines Systems mittels „Divide and Conquer" wurden bereits vor über 40 Jahren in „Structured Design" *[3]* dargelegt!
Hierbei ist neben der Kopplung auch ein weiteres Prinzip zu beach-

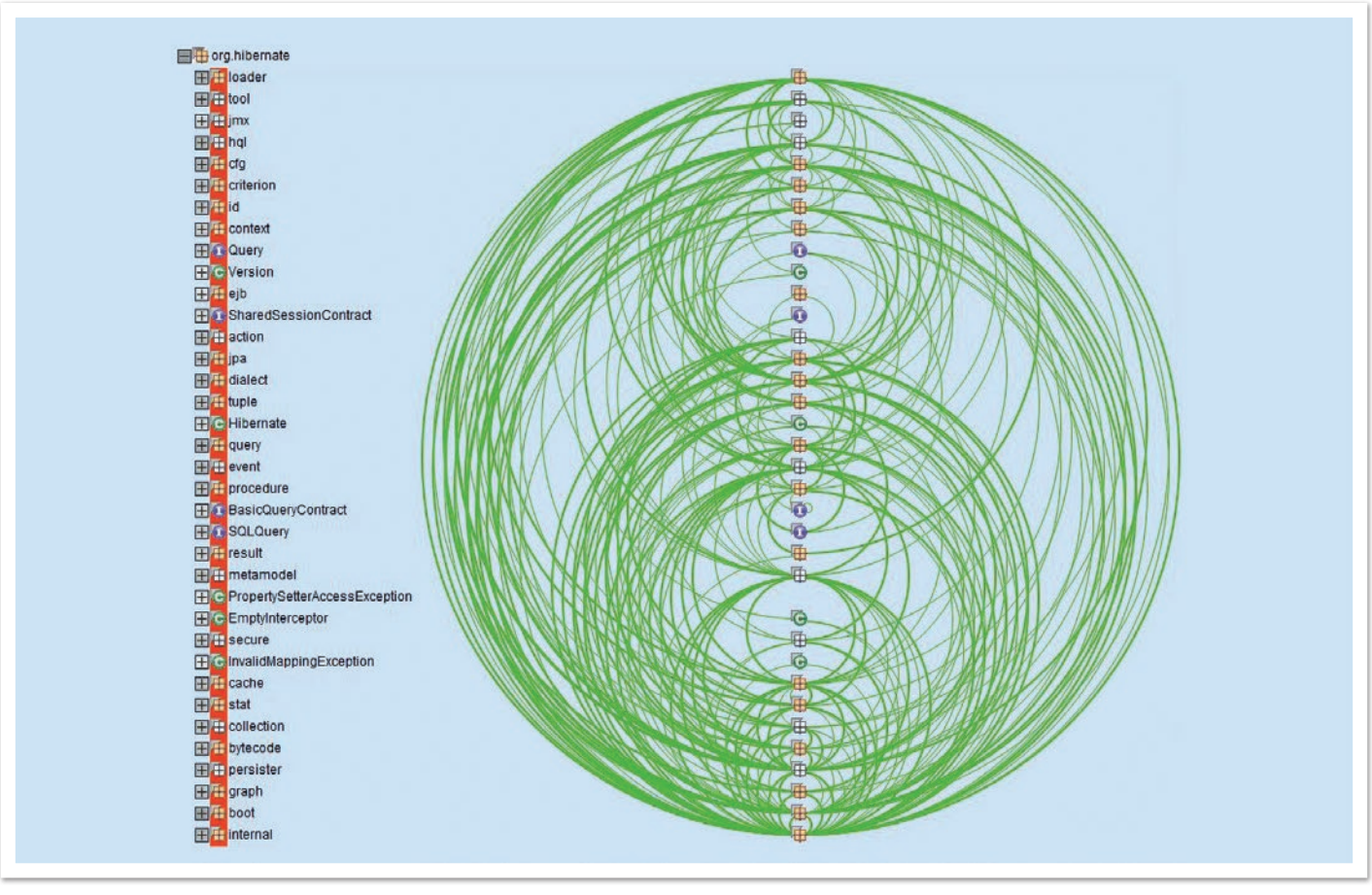


Abbildung 3: Visualisierung von zyklischen Abhängigkeiten (© Ingmar Kellner)

ten, die „Kohäsion“: Als Kohäsion wird dabei die Kopplung von einzelnen Elementen innerhalb eines Containers (zum Beispiel Methoden einer Klasse, Klassen eines Packages) verstanden. Das heißt, in einem Container sollten sich möglichst nur Elemente befinden, die auch miteinander zu tun haben. Bekannt ist dieses Prinzip auch als „Low Coupling, High Cohesion“. Wichtig ist es mir an dieser Stelle zu betonen, dass die Struktur der Software, auch Architektur bezeichnet, einen großen Einfluss darauf hat, wie schnell der Entwickler den Code verstehen und darin navigieren kann.

Das setzt voraus, dass die Elemente der Architektur sich im Code über Namenskonventionen wiedererkennen lassen und das Design der Software auf den verschiedenen Abstraktionsebenen konsistent ist. Es darf auf Code-Ebene keine Abhängigkeit bestehen, die in der Architektur nicht existiert. Ansonsten sind die Abstraktionen ungültig und helfen dem Entwickler nicht beim Verständnis.

Die Strukturierung des Systems ist keine einmalige Aktivität, sonst wäre man wieder beim Wasserfall-artigen „big design upfront“, das ja bereits eingangs als Anti-Pattern erwähnt wurde. Stattdessen muss die Architektur mit dem System „leben“ und laufend angepasst werden, sodass sie bestmöglich den Anforderungen an das System entspricht.

Strukturelle Erosion

Die bisher vorgestellten Konzepte „lose Kopplung“ und „konsistente Struktur“ mögen nicht besonders aufregend klingen und der ein oder andere mag bezweifeln, ob das wirklich relevante Dinge sind, auf die es sich zu achten lohnt. Meine Kollegen und ich haben in den vergangenen Jahren viele Software-Systeme analysiert. Bei den meisten Systemen existiert eine sehr starke Kopplung.

Wie schon erwähnt, werden stark gekoppelte Systeme auch als „Big Ball of Mud“ bezeichnet, da hier wenig von der ursprünglich angedachten Struktur erkennbar ist. Anhand von zwei Systemen möchte ich die Problematik verdeutlichen, dem Modul Hibernate-Core von Hibernate ORM *[4]* und dem Server-Teil der Corona-Warn-App *[5]*.

Analysiert man die Werte für Average Component Dependency (ACD) über einen längeren Zeitraum, lässt sich ein deutlicher Anstieg erken-

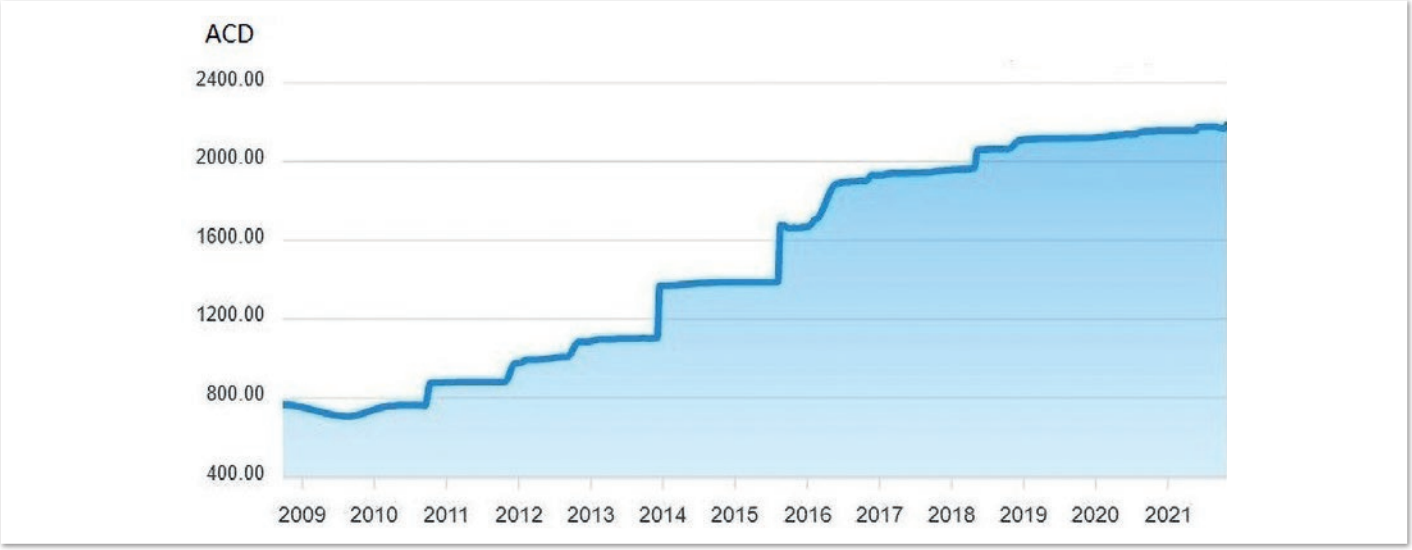


Abbildung 4: Entwicklung des ACD von Hibernate-Core (© Ingmar Kellner)

Abbildung 5: Dateien in der größten Zyklengruppe in Hibernate-Core (© Ingmar Kellner)

nen für Hibernate-Core, wie in *Abbildung 4* dargestellt. In der Version 5.6.1 liegt der Wert bei über 2.100! Das heißt, jedes Source-File im Projekt hängt durchschnittlich von 2.100 anderen Source-Files direkt oder indirekt ab. Bei einer Gesamtzahl von rund 3.500 Source-Files bedeutet dies, dass jede Datei an rund 60 % aller anderen Dateien gekoppelt ist. 90 % des Codes ist in Dateien, die an einem Zyklus beteiligt sind.

Im Release 5.6.1 sind rund 75 % des Codes in einem einzigen Zyklus bestehend aus 2.211 Dateien beteiligt. Auch dieser Zyklus ist kontinuierlich gewachsen, wie in *Abbildung 5* dargestellt, und so dominant, dass die Anzahl der beteiligten Dateien praktisch identisch mit dem ACD-Wert ist. Einen Ausschnitt aus dem Abhängigkeitsgeflecht konnte man bereits in *Abbildung 3* betrachten.

Im Server-Teil der noch jungen Corona-Warn-App lässt sich dieser Trend in kleinerem Ausmaß erkennen. Analysiert habe ich bis zur Version 2.15. Das System ist mit rund 17.000 Zeilen Code überschaubar (Tests und generierter Code ausgeklammert). Auch dort nimmt der an Zyklen beteiligte Code zu und beträgt momentan bereits fast 39 %. Der Code wird kontinuierlich auf Probleme geprüft, doch das verwendete Tool SonarQube bietet seit einigen Jahren keine Zyklenerkennung mehr. In Bezug auf die strukturelle Erosion ist das Projekt folglich blind.

Die Entwicklung dieser beiden Projekte ist ein gutes Beispiel für Software-Entropie, die unter anderem von M.M. Lehman beschrieben wird in „Laws of Program Evolution" *[6]*: „As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

Schauen wir uns an, wie sich die strukturelle Erosion vermeiden lässt.

Best Practice 1: Tooling

Es ist in großen Projekten mit vielen Entwicklern unmöglich, jede Änderung am System manuell zu prüfen. Aus unserer Erfahrung kann die oben genannte Erosion nur durch eine kontinuierliche und automatische Kontrolle erkannt werden. Werkzeuge wie Sonargraph, jQAssistant und andere bieten eine Fülle von Möglichkeiten, um sowohl die Struktur zu überwachen als auch Metriken zu berechnen und Anti-Patterns zu erkennen.

In einer Evaluierungsphase sollte man sich mit den eigenen Bedürfnissen vertraut machen und das Produkt wählen, das diese am besten erfüllt. Werden die Tools für bereits existierende Projekte eingeführt, besteht die Gefahr, dass man von der Menge der gemeldeten Probleme erschlagen wird. Hier bietet es sich an, nach der „Pfadfinder"-Regel („leave the camp ground cleaner than you found it") auf jeden Fall zu verhindern, dass neue Probleme hinzukommen, und die bestehenden dort aufzuarbeiten, wo Code geändert wird. Wichtig ist die kontinuierliche Prüfung und die Unterstützung durch das Management für notwendige Refactorings.

Best Practice 2: Keine Architekturprüfung durch Deployment-Struktur

Die meisten Projekte sind in mehrere Module unterteilt. Wenn für jedes Teil-Element in der Architektur ein eigenes Modul existiert, hat das zwar den Vorteil, dass sich über die erlaubten Modul-Abhängigkeiten keine schwerwiegenden Architekturverletzungen einbauen lassen, die Nachteile sind allerdings nicht zu vernachlässigen:

1. Die große Anzahl der Module erschwert das Verständnis des Systems
2. Die Pflege der Abhängigkeiten in den Modulen ist aufwendig

Noch unübersichtlicher wird es, wenn ein System in eine hohe Anzahl von Microservices aufgeteilt ist. Zwar werden auf diese Weise die Abhängigkeiten im Code auf den jeweiligen Service begrenzt und es können keine Zyklen im Code entstehen, die das gesamte System beinhalten; allerdings bestehen Abhängigkeiten zwischen den Services, die nun jedoch erst zur Laufzeit aufgelöst werden und dadurch noch schwerer nachzuvollziehen sind. Systeme, bei denen das Abhängigkeitsgeflecht nur noch schwer zu durchschauen ist, werden gerne als „Distributed Big-Ball of Mud" bezeichnet. Wir beobachten einen Trend wieder in Richtung größerer Services, die unter dem Label „Self-Contained Systems" *[7]* propagiert werden.

Meine Empfehlung ist, die Architekturprüfung nicht auf die Infrastruktur abzuwälzen! Verwenden Sie passende Werkzeuge, die genau dafür gebaut worden sind, und strukturieren Sie den Code in möglichst wenige Module, so wie es das Deployment später erfordert.

Best Practice 3: Programmier-Prinzipien

Aber nicht nur das passende Tooling ist wichtig. Auch bei der Programmierung selbst lässt sich die Kopplung reduzieren, indem man sich an ein paar Prinzipien orientiert.

Zum einen ist es die Verwendung von unveränderlichen Datenklassen, die seit Java 14 als „Records" direkt unterstützt werden. Zusammen mit „Pure Functions", also Funktionen ohne Seiteneffekte, die lediglich einen Rückgabewert liefern, bieten sie eine saubere Trennung in Datenstrukturen und Funktionen. Das passt meiner Meinung nach gut zu der Aufteilung in Value Objects und Services, die Eric Evans in „Domain-driven Design" *[8]* beschreibt.

Schwer verständliche Abhängigkeiten entstehen auch, wenn Code-Duplikate eliminiert werden, indem eine gemeinsame Oberklasse erstellt wird und der Code durch ein „Extract Method"-Refactoring dorthin verschoben wird. Schon oft habe ich mich selbst für einen durch eine tiefe Hierarchie mäandrierenden Kontrollfluss verflucht! Hier sollte möglichst das Prinzip „favor composition over inheritance" befolgt und der Code in eine separate, unabhängige Klasse ausgelagert werden.

God Classes mit mehreren 1.000 Zeilen Code, die oftmals eine Verletzung des „Single Responsibility"-Prinzips darstellen, lassen sich ebenfalls automatisiert erkennen. Und nicht zuletzt sollte man darauf achten, dass man nur die absolut notwendigen Dinge über Architekturgrenzen hinweg sichtbar macht („Encapsulation/Information Hiding"). Interfaces an Architekturgrenzen zur Implementierung des „Dependency Inversion"-Prinzips wirken wie Brandmauern gegen kaskadierende Änderungen.

Zusammenfassung

Dieser Artikel hat aufgezeigt, dass hohe Kopplung ein Kostentreiber in der Software-Entwicklung ist. Anhand von Beispielen wurde dar-

gestellt, wie sich Kopplung berechnen lässt. Die strukturelle Erosion, das heißt die Ausbreitung von ungewollter Kopplung, ist in vielen Software-Projekten erkennbar.

Eine Kontrolle der Kopplung ist immer notwendig und sinnvoll, egal ob es sich bei der Software um einen Monolithen oder eine verteilte Anwendung bestehend aus Dutzenden Microservices handelt. Ein gut strukturiertes System entsteht nur durch automatisierte Prüfung und regelmäßiges Refactoring, und nicht von selbst durch den Einsatz einer bestimmten Technologie. Grundvoraussetzung ist der Wille, im Team auf diesen Aspekt zu achten. Die Notwendigkeit einer sauberen Struktur wird seit Langem von Experten beschrieben und seit einigen Jahren gibt es dafür die Unterstützung durch passende Tools. Mit der Umsetzung weniger Best Practices lässt sich ein „Big Ball of Mud" zuverlässig vermeiden.

Investitionen an dieser Stelle zahlen sich aus: Einer unserer Kunden berichtet von einer Produktivitätssteigerung der Entwickler von 50 % *[9]*. Und nicht nur das: Es macht einfach mehr Spaß, an einer gut strukturierten Codebasis zu arbeiten, in der Veränderungen schnell realisierbar sind.

Quellen

[1] Lakos, J. (1996): Large-Scale C++ Software Design, Addison Wesley

[2] Foote, B.; Yoder, J. (1999): Big Ball of Mud, *http://www.laputan.org/mud/*

[3] Yourdon, E; Constantine, L (1979): Structured Design, Prentice-Hall

[4] Hibernate-ORM, *https://github.com/hibernate/hibernate-orm*

[5] Corona Warn App (Server), *https://github.com/corona-warn-app/cwa-server*

[6] Lehman, M. M.; Belady, L.A. (1985): Program evolution: processes of software change, Academic Press.

[7] Self-Contained Systems, *https://scs-architecture.org/*

[8] Evans, E. (2004): Domain-Driven Design, Addision Wesley

[9] Baldauf, T.; von Zitzewitz, A. (2015): Case Study Environment Agency Austria, *https://www.hello2morrow.com/whitepapers/50/download*

Ingmar Kellner
hello2morrow GmbH
i.kellner@hello2morrow.com

Ingmar Kellner programmiert seit 20 Jahren, mit einem starken Fokus auf Java. Seit 10 Jahren beschäftigt er sich bei der hello2morrow GmbH mit Code-Qualität und speziell mit der automatisierten strukturellen Analyse. Dort ist er einer der Entwickler des Sonargraph, eines Tools zur statischen Code-Analyse mit Schwerpunkt auf Architekturprüfung, und berät Kunden bei der Umsetzung von Qualitätsverbesserungen.

Wie du zufrieden wirst – und nebenbei besseren Code schreibst

Thorsten Diekhof, Abenteuer Softwareentwicklung

Ich bin überzeugt: Nicht guter Code macht Entwickler/innen zufrieden – zufriedene Entwickler/innen schreiben besseren Code. Zum Glück gibt es wissenschaftliche Erkenntnisse darüber, wie man ein zufriedenes Leben führen kann. In diesem Artikel begründe ich meine Hypothese, lassen wir unser inneres Feuer lodern, finden wir unsere Stärken und erleben Tagträume. Alles natürlich nur für bessere Software!

Der Tag, an dem ich aufhörte, normale Seminare zu geben, war ein dunkler Novembertag. Vielleicht sah der LKW-Fahrer mich an diesem verregneten Morgen nicht. Er fuhr, trotz massig Platz, direkt neben dem schmalen Gehweg durch eine riesige Pfütze, die sich in einem Schwall über mich ergoss. Da stand ich also. In zwei Stunden würde der zweite Seminartag über Softwarequalität mit JUnit beginnen. Ich war nass, in einer fremden Stadt und echt mies drauf.

In einem kleinen türkischen Café trocknete ich mich bei starkem Kaffee und enorm süßem Gebäck. Dabei machte ich mir ernsthafte Gedanken über den heutigen Tag. Schon der erste Seminartag war frustrierend gewesen. Nach kurzer Zeit war mir klar, dass das Problem im Team nicht das fehlende Wissen über die neusten Features von JUnit war. In kurzer Zeit hätte sich das Team JUnit auch selbst beibringen können – ohne ihre Probleme dadurch ernsthaft zu lösen.

Es ist oft der Fall, dass sich Teams gar nicht bewusst sind, wieso es an bestimmten Stellen hakt. Viel zu schnell werden dann einfach Seminare über irgendwelche Frameworks und Bibliotheken gebucht.

Typische Probleme – typische Lösungen

Meine Erfahrung sagt mir, dass so nur Level-1-Probleme gelöst werden können. So bezeichne ich Probleme, die auftauchen, weil hauptsächlich unerfahrene Entwickler/innen durch die scheinbare Komplexität neuer Frameworks oder Aufgaben überfordert sind. Viel seltener als gedacht sind es jedoch falschen Frameworks, die Schuld an den Problemen eines Teams sind.

Häufiger sind es Level-2-Probleme. Software wird fehlerhaft oder viel zu spät ausgeliefert. Bugs stören Kunden und Nutzer. Anpassungen und Änderungen im Code sind nur sehr aufwendig zu erledigen. Immer wieder tauchen alte Bekannte auf – Fehler, die schon mehrfach gefixt wurden. Level-2-Probleme sind sehr frustrierend und werden zu häufig mit Level-1-Lösungen angegangen. Neue Frameworks werden angeschafft – Seminare gebucht, um noch mehr Features der genutzten Technik zu lernen. Dabei ist das Problem häufig schlecht lesbarer Code, fehlendes Softwaredesign und überkomplexe Architektur. Statt also die neusten Features eines Tools zu lernen, bringt eine Vereinfachung der eigenen Codebasis häufig einen viel größeren Effekt.

Einen regelrechten Hype erleben aktuell die Lösungsmethoden der Level-3-Probleme. Software, die den Anforderungen der Kunden und des immer dynamischer werdenden Marktes nicht gerecht wird. Kunden, die sich über mangelnde Einflussnahme beschweren, und Entwickler/innen, die nicht wissen, wieso sie bestimmte Features überhaupt implementieren. Die naheliegendste Lösung ist hier, agile Methoden und agiles Arbeiten zu etablieren. Ich bin ein großer Freund agilen Denkens. Jedoch sollte einem bewusst sein, welche Art von Problemen damit gelöst werden können – und welche nicht.

Zurück zu jenem Novembertag. Ich bestellte mir noch einen Kaffee und versuchte, meine klebrigen Finger an einer Serviette zu reinigen. Welche Probleme lagen denn jetzt bei meinen Seminarteilnehmern und -teilnehmerinnen vor?

Die Teilnehmer/innen waren technisch fit. Ihr Trainings-Code im Seminar war lesbar und gut strukturiert. Auch arbeitete ihre Firma seit Längerem mehr oder weniger agil. Trotzdem gab es Anzeichen aller Arten von Problemen. Frameworks schienen zu komplex und wurden als „Dreckstools" bezeichnet. Fragen der Teilnehmer/innen deuteten auf enorm komplexen Code hin und immer wieder gab es Probleme mit dem Kunden und seinen scheinbar sinnlosen Anforderungen. Mir schien, als wären fast alle Teilnehmer/innen unzufrieden mit ihrer Arbeitssituation.

Zufriedene Entwickler/innen schreiben besseren Code

Beim dritten Kaffee kam mir dann die Idee. Meine Arbeitshypothese: Nicht guter Code macht Entwickler/innen zufrieden – zufriedene Entwickler/innen schreiben besseren Code.

Und mit zufrieden meine ich, zufrieden mit sich selbst, dem eingeschlagenen Weg im Leben und den eigenen Ideen der Zukunft. Entwickler/innen mit festen Werten und einer eigenen Haltung. Motivierte Entwickler/innen – weil sie wissen, welchen Baustein ihres Lebens sie gerade bearbeiten. Kleiner Spoiler: Genau dies ist es, was ich seitdem immer wieder erlebe.

Ich schmiss das komplette Seminar um.

Zum Glück gibt es hier Unterstützung von der Wissenschaft. Es gibt etliche Forschungsarbeiten zum Thema, was zu einem zufriedenen Leben gehört. In den letzten 20 Jahren etablierte sich ein Bereich der Wissenschaft, der sich fast ausschließlich damit beschäftigt, wie man ein gelungenes, zufriedenes Leben führt: die Positive Psychologie. Und zum Glück hatte ich mich genau damit vor einiger Zeit für mein eigenes Leben und meine eigene Zufriedenheit intensiv beschäftigt.

Ich schrieb stichpunktartig ein paar Faktoren auf, die laut Forschung zu mehr Zufriedenheit führen. Faktoren, die wir selbst in der Hand haben. Es wird angenommen, dass 50 % der eigenen Zufriedenheit zwar angeboren, aber immerhin 40 % Einstellungssache sind und 10 % an den äußeren Umständen liegen.

Viele Wege führen zur Zufriedenheit

Für den Anfang brauchte ich einen roten Faden. Ein schlüssiges Konzept, das sich gut umsetzen und schnell ins Leben integrieren lässt.

Ich umkreiste die Faktoren, an denen sich konkret, kurzfristig arbeiten lässt. Stimmige Ziele und angewandte Stärken. Darauf sollte sich das Seminar konzentrieren.

Die Motivationsforscherin Michaela Brohm-Badry erklärt in ihrem Buch „Das gute Glück" *[1]*, dass der synaptische Spalt, der unsere Neuronen voneinander trennt, je nach Situation durch Botenstoffe – die Neurotransmitter – überbrückt wird.

So erhöht Adrenalin, wenn wir unter Druck stehen, unsere Leistungsbereitschaft. Und Serotonin gibt uns ein Glücksgefühl, wenn wir Ziele erreicht haben. Dopamin entsteht, wenn wir Glück erwarten – also auf ein Ziel hinarbeiten. Es macht zufrieden und motiviert, auch anstrengende Tätigkeiten anzugehen und Hindernisse zu überwinden. Genau unser Botenstoff: Ziele zu haben und anzugehen, macht zufrieden.

Das ist ein schöner Gedanke. Der Weg zum Ziel ist also das, was zufrieden macht – nicht das Erreichen eines Zieles selbst. Das wusste wahrscheinlich schon Konfuzius, dem das Zitat „der Weg ist das Ziel" zugeschrieben wird. Fehlt ein Ziel, ist der Dopamingehalt niedrig, was unzufrieden macht.

Stimmige Ziele sind ein Muss

Leider ist es nicht damit getan, sich irgendein Ziel zu setzen. Der nächste Release oder das Fertigstellen des vom Kunden gewünschten Features reichen nicht aus. Es muss ein stimmiges Ziel sein. Ein Ziel, das mit den eigenen Stärken übereinstimmt – das zur eigenen Person passt. Nur dann macht das Dopamin seine Arbeit.

Das war der Weg, den ich mit den Teilnehmer/innen gehen wollte. Wie lassen sich stimmige Ziele im Leben entwickeln und erreichen?

Ich lud ein paar Listen und Fragebögen aus dem Netz herunter, trank den letzten Schluck lauwarmen Kaffee und ging durch einen kalten Nieselregen zum Seminarzentrum.

Zuerst musste ich den Teilnehmer/innen erklären, dass ich das Seminar umgestalten wollte. Ich erläuterte meine These. Alle waren interessiert und motiviert. Ich begab mich ins Risiko und wurde direkt mit einem satten Dopaminspiegel belohnt. Was konnte schiefgehen?

Reise zum Feuer

Im ersten Teil ging es um das innere Feuer. Eigene Stärken erkennen wir am besten, wenn wir uns selbst dabei beobachten, wie wir Dinge tun, die uns Freude bereiten und bei denen wir Flow *[2]* erleben. Erleichtert wird dies dadurch, dass unser Gehirn fast nicht unterscheiden kann, ob eine Situation gegenwärtig ist oder wir uns diese nur vorstellen. Es reicht also, uns in der Vorstellung zu beobachten.

Daher bestand die erste Übung darin, die Augen zu schließen und sich einige Zeit auf den eigenen Atem zu konzentrieren. Eine klassische Übung zur Entspannung, die für Entwickler/innen jedoch häufig recht ungewohnt ist. Sie dient dazu, die Aktivität des präfrontalen Cortex zu senken. Diese Region im Hirn ist unter anderem für unseren kritischen, wissenschaftlichen Verstand zuständig. Ohne diesen kann der Geist tiefer in Erinnerungen und Gedanken eintauchen. Sie wirken realer und unser Unterbewusstsein kann besser aktiv werden.

Jetzt sollten die Teilnehmer/innen an eine frühere Erinnerung denken, bei der ihnen das Programmieren so richtig Freude bereitet hatte. Das erste eigene Programm, die ersten erfolgreichen Skripte oder frühe Lernerfolge. Diese Erinnerungen sollten nun ausgemalt, Details, spontane Empfindungen und Bilder zugelassen werden.

Ich hörte nicht nur das gleichmäßige Atmen der Teilnehmer/innen, sondern sah auch, wie sich die Gesichter entspannten und bemerkte sogar das eine oder andere Lächeln.

Nach der Reise wurde die erlebte Situation erkundet. Warum genau empfand man in der Situation Freude? Was motivierte einen damals? Welche Fähigkeiten wurden in der Situation gefordert? Was machte daran stolz? Was entzündet das innere Feuer für Softwareentwicklung in uns und lässt es lodern?

Die Teilnehmer/innen machten Notizen, malten Bilder oder schrieben Texte. Wer wollte, konnte sein Erlebnis mitteilen. Es entwickel-

```
Weisheit und Wissen
Kreativität, Neugier, Liebe zum Lernen, Aufgeschlossenheit, Weisheit

Mut
Authentizität, Tapferkeit, Lebenskraft, Ausdauer

Humanität
Freundlichkeit, Bindungsfähigkeit, soziale Intelligenz

Gerechtigkeit
Fairness, Führungsvermögen, Teamfähigkeit / Loyalität

Mäßigung
Vergebungsbereitschaft, Bescheidenheit, Umsicht, Selbstregulation

Transzendenz
Sinn für das Schöne und Gute, Dankbarkeit, Hoffnung, Humor/Verspieltheit, Spiritualität
```

Listing 1: Die sechs Kerntugenden mit zugehörigen 24 Charakterstärken

ten sich schöne Gespräche über Leidenschaft und darüber, was wir so großartig an der Softwareentwicklung finden. Jede/r hatte prägende Erfahrungen gemacht und war mit Begeisterung zum Programmieren gekommen.

Charakterstärken identifizieren

Aufbauend auf dieser Erfahrung erkundeten die Teilnehmer/innen nach einer längeren Pause ihre Stärken. Stärken sind nach Biswas-Diener *[3]* „persönliche, überdauernde Muster von Gedanken, Gefühlen und Verhaltensweisen. Sie sind individuell, geben Energie und ermöglichen beste Leistung". Wer in Kontakt mit ihnen ist, „spricht klarer, schneller, zeigt mehr Ausdruck in der Mimik und wird als charismatischer erlebt". Nicht schlecht.

Ich teilte eine Liste mit den 24 Charakterstärken aus *(siehe Listing 1)* – ein zentrales Konzept der Positiven Psychologie. Es wurde von Christopher Peterson und Martin Seligman *[4]* beschrieben. Jede dieser Stärken gehört einer der sechs Kerntugenden an. Tugenden tauchen in Schriften verschiedenster Kulturen immer wieder auf und werden als Kerneigenschaften des Menschen angesehen. Die Stärken stellen Wege dar, die zugeordneten Tugenden zu leben und erleben. Mut kann beispielsweise sowohl als Tapferkeit als auch als Authentizität Ausdruck finden.

Wer seine Stärken lebt, führt ein mit sich selbst stimmiges, authentisches Leben. Ziele, deren Erreichen die Anwendung der eigenen Stärke benötigen, werden stimmige Ziele genannt. Wir benötigen also das Wissen über unsere Stärken, um stimmige Ziele zu finden.

Daher war es nun an den Teilnehmer/innen, fünf bis acht Charakterstärken auf der Liste zu umkreisen, die sie selbst charakterisieren. Zu jeder dieser Stärken sollte sich in Ruhe die Frage gestellt werden: „Wie wäre es für mich, wenn ich diese Stärke zwei Wochen nicht einsetzen könnte?"

Die Frage stellte Ryan Niemiec auf dem Weltkongress Positive Psychologie 2017 in Montreal vor. Durch sie lassen sich unsere Signaturstärken „erfühlen". Die Stärken, mit denen wir uns sehr stark identifizieren, deren Anwendung uns Freude bereitet und die einen wesentlichen Bestandteil unseres Selbstbildes ausmachen. Wer bei der Frage ein ungutes Gefühl hat oder ganz klar mit „auf keinen Fall!" antwortet, erkennt eine Signaturstärke.

Im Anschluss dieser Übung konnten Teilnehmer/innen sich und ihre Stärken vorstellen. Ich fragte auch in die Runde, ob neue Erkenntnisse gemacht wurden. Einige Teilnehmer/innen hatten sich noch nie so konkret mit ihren Stärken und Tugenden auseinandergesetzt. Interessant.

Der perfekte Tag

Um ein stimmiges Ziel zu entwickeln, braucht es eine attraktive Vision, die uns anzieht. Ziele können in zwei Kategorien eingeteilt werden. Mit Weg-von-Ziele versuchen wir, Dinge oder Situationen zu vermeiden. Ich möchte keinen Kundenkontakt mehr oder ich möchte nicht mehr faul sein. Solche Ziele sind viel weniger attraktiv und motivierend als Hin-zu-Ziele. Ich möchte viel Zeit haben, um mich intensiv mit dem Backend-Code zu beschäftigen, oder, mein aktuelles Hin-zu-Ziel, ich möchte jeden Monat mindestens eine Nacht im Zelt verbringen.

Um Hin-zu-Ziele zu finden, eignet sich eine wunderbare Übung – „Der perfekte Tag". Hierbei geht es darum, sich einen perfekten Tag in der Zukunft vorzustellen. Am besten einen Tag, in dem alle Signaturstärken zur Geltung kommen. Quasi Tagträumen – mit der Erlaubnis, groß zu träumen und zu übertreiben. Die Teilnehmer/innen sollten sich einen schönen Ort suchen, ein frisches Heißgetränk zubereiten und den perfekten Tag zu Papier bringen. In welchem Stil, war vollkommen ihnen überlassen. Auch, ob es sich um einen Tag in einem Jahr oder in zehn Jahren handelt. Ob als Text, Bild oder in Stichpunkten. Hierzu hatten sie eine Stunde Zeit.

Zur Inspiration gab ich den Teilnehmer/innen einige Fragen mit auf den Weg *(siehe Listing 2)* – sie sollten sich jedoch unbedingt weitere, eigene Fragen stellen.

Bei dieser Übung wird eine Art Vision vom zufriedenen, zukünftigen Leben aufgezeigt. Genau das, was wir brauchen, um attraktive Hin-zu-Ziele zu finden, die unsere Stärken fordern und für uns authentisch sind.

Durch das Wissen, was unser Feuer zum Lodern bringt, unsere Signaturstärken und die klare Vision eines perfekten Tages waren die Teilnehmer/innen bereit, konkrete, stimmige Ziele für sich zu beschreiben.

Stimmige Ziele konkretisieren

Die letzte Aufgabe des Tages war, mindestens drei konkrete Ziele zu bestimmen und zu verschriftlichen. Ziele, die realistisch, konkret, messbar und attraktiv sind. Mindestens ein Ziel sollte in den nächsten Tagen angegangen und erreicht werden. Mindestens eins sollte sich auf neue Fähigkeiten oder Verhaltensweisen beziehen und mindestens ein weiteres Ziel sollte ein klares Zwischenziel auf dem Weg zum perfekten Tag sein.

Die Ziele wurden auf Karteikarten festgehalten und einige in der Runde vorgestellt und besprochen. Es herrschte eine großartige Atmosphäre, als wir eine kleine Retrospektive begannen. Davon schreibe ich hier nichts – denn „what happens in Vegas stays in Vegas". Nur so viel, wir haben alle viel gelernt und gelacht. Einige Teilnehmer/innen wollten sich direkt nach dem Seminar noch auf ein Getränk treffen, um den Tag ausklingen zu lassen.

```
Wo wachst du morgens auf?
Wie verbringst du den Vormittag?
Wie verbringst du den Nachmittag?
Wie verbringst du den Abend?
Mit wem verbringst du den Tag?
Wie ist die Atmosphäre bei der Arbeit?
Wie sieht dein Arbeitsplatz aus?
Welche deiner Stärken setzt du ein?
Welchen Menschen begegnest du?
Wie ernährst du dich?
Wie geht es deinem Körper?
Wie ist dein Lebensgefühl?
Wie ist deine Ausstrahlung?
Woran arbeitest du?
Was ist das Beste an diesem Tag?
Was inspiriert dich?
Welche Kleidung trägst du?
```

Listing 2: Fragen an den perfekten Tag

Ich fuhr noch am Abend mit dem ICE zurück nach Berlin. Von den Teilnehmer/innen bekam ich leider nie eine Rückmeldung, ob ihre stimmigen Ziele wirklich erreicht wurden. Wieso nicht? Nun, das Seminar hat es so nie gegeben. Als ich mich im türkischen Café trocknete und mir Gedanken über Zufriedenheit und gute Softwareentwicklung machte, entschloss ich mich, kein Risiko einzugehen und lieber nichts Neues zu wagen. Den Tag über erklärte ich mittelmäßig motivierten Teilnehmer/innen die Vorzüge von JUnit – immer mit dem Gefühl, dass es nur ein wenig mehr Zufriedenheit in ihrem Leben bedarf, um aus den Teilnehmer/innen ein wirklich gutes Entwicklungsteam zu machen.

Sie sind dran

Sie, werte Leser/innen, haben es sicher schon erkannt. Sie halten meinen perfekten Tag in den Händen – wenn auch ohne die zahlreichen Übertreibungen, mit denen ich ihn mir ausgeschmückt hatte. Entstanden ist der perfekte Seminartag auf dem Weg zurück nach Berlin – aufbauend auf meinen Signaturstärken: Liebe zum Lernen, Aufgeschlossenheit, Authentizität und Humor.

Dass Sie diese Zeilen hier lesen, bedeutet, dass ich eines meiner stimmigen Ziele erreicht habe. Vielleicht helfen Sie mir bei der Erfüllung eines weiteren aktuellen Ziels: Machen Sie doch die vorgestellten Übungen, erreichen Sie einige Ihrer stimmigen Ziele und geben mir Rückmeldung, ob mehr Zufriedenheit auch bei Ihnen zu besserer Software führt.

Seit zwei Jahren bin ich diesem Zusammenhang bereits auf der Spur. Die Hinweise verdichten sich, dass meine Hypothese stimmt. Bei mir jedenfalls hat es funktioniert!

Quellen

[1] Prof. Dr. Michaela Brohm-Badry (2019), „Das Gute Glück", Ecowin Verlag

[2] Mehr über Flow zu erfahren lohnt sich: *https://de.wikipedia.org/wiki/Flow_(Psychologie)*

[3] Robert Biswas-Diener (2010), „Practicing Positive Psychology Coaching", Wiley

[4] Christopher Peterson und Martin Seligman (2004), „Character strength and virtues", Oxford University Press

Thorsten Diekhof

Abenteuer Softwareentwicklung

kontakt@thorsten-diekhof.de

Thorsten Diekhof ist Trainer für gute Softwareentwicklung, modernes Arbeiten und agiles Denken. Diese Themen behandelt er auch in vielen Video-Kursen und als Mentor. Unter dem Label COACHING.CARDS vertreibt Thorsten Kartensets für Coding, Business und Persönlichkeit.

Mitglieder des iJUG

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

iJUG Verbund

www.ijug.eu

Impressum

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

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

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

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

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

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

Bildnachweis:
Titel: Bild © Ramcreative
https://stock.adobe.com
S. 14: Bild © royyimzy
https://stock.adobe.com
S. 22: Bild © s.salvador
https://freepik.com
S. 26: Bild © AKrasov
https://stock.adobe.com
S. 33: Bild © Gorodenkoff
https://stock.adobe.com
S. 38: Bild © AndSus
https://stock.adobe.com
S. 42 + 43: Bild © Visual Generation
https://stock.adobe.com
S. 48: Bild © Danussa
https://stock.adobe.com
S. 55: Bild © Kraphix
https://freepik.com
S. 58: Bild © Golden Sikorka
https://stock.adobe.com
S. 64: Bild © Designed by Freepik
https://freepik.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

CALL FOR PAPERS #JSD2022*

* Weinberge, Innovation, Community & Eure Themen
auf der IT-Konferenz der JUG Saxony
am 23. September 2022 in Radebeul/Dresden