Java aktuell

iJUG Verbund
www.ijug.eu

Mobile Entwicklung
Kotlin Multiplatform, Jetpack Compose, Modularisierung u. v. m.!

Verteilte Datensammlung
Zeitserien-Daten mit Apache TsFile und Apache IoTDB

Das Gehirn als Datensystem
Reverse Engineering des Zentralen Nervensystems

Java aktuell
Ausgabe 02/2024
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

JavaLand

on demand

JavaLand 2023 verpasst?

Jetzt On-demand-Ticket buchen und Vortragsaufzeichnungen anschauen!

Alle On-demand-Angebote im Ticketshop

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

Veranstalter: JavaLand

Liebe Leserinnen und Leser,

willkommen zur Ausgabe 02/2024 der Java aktuell mit dem Themenschwerpunkt Mobile! Unsere Autorinnen und Autoren aus der Community haben wieder einmal eine bunte Mischung besten Wissens für euch verfasst.

Wir steigen mit Kurt Huwig und seinem Doppelartikel ab Seite 10 in das Titelthema ein. Im ersten Teil stellt er die plattformübergreifende mobile Entwicklung mit Kotlin Multiplatform vor. Dabei geht er darauf ein, was damit möglich ist und hebt auch mögliche Fallstricke hervor. Direkt im Anschluss widmet er sich der Entwicklung von Benutzerschnittstellen mit Jetpack Compose, das eine Alternative zu XML-Dateien bei der Erstellung von Benutzerschnittstellen bietet. Danach behandelt Sven-Toben Janus die Sicherheit und Effizienz von Authentifizierungen bei mobilen Applikationen mit OpenID Connect und wirft dabei einen genaueren Blick auf native Apps. Wie Ansätze der Modularisierung bei Android-Apps umgesetzt werden können, präsentiert Jonathan Merkel ab Seite 26. Dabei stellt er die Architekturmuster Modulith und Microkernel gegenüber. Low-Code-Plattformen gewinnen immer mehr an Bedeutung, da diese die Erstellung einfacher Applikationen auch durch Nicht-Entwickler:innen ermöglicht und somit die Zusammenarbeit zwischen den Fachabteilungen fördert. Unter der Verwendung von VisionX zeigt René Jahn in seinem Artikel, wie man eine solche App mit bestimmten Anforderungen realisieren kann.

In Zeiten von künstlicher Intelligenz und Machine Learning ist es eine große Herausforderung für Datenbanken, mit den riesigen anfallenden Datenmengen umzugehen. Christofer Dutz stellt einen Lösungsansatz vor, wie mithilfe von Apache TsFile und Apache IoTDB eine verteilte Zeitserien-Datensammlung erstellt werden kann. Danach geht Karl Heinz Marbaise mit uns eine Konfiguration für ein Apache-Maven-Projekt durch, das eine Abdeckung für Integrations- und Unittests liefert.

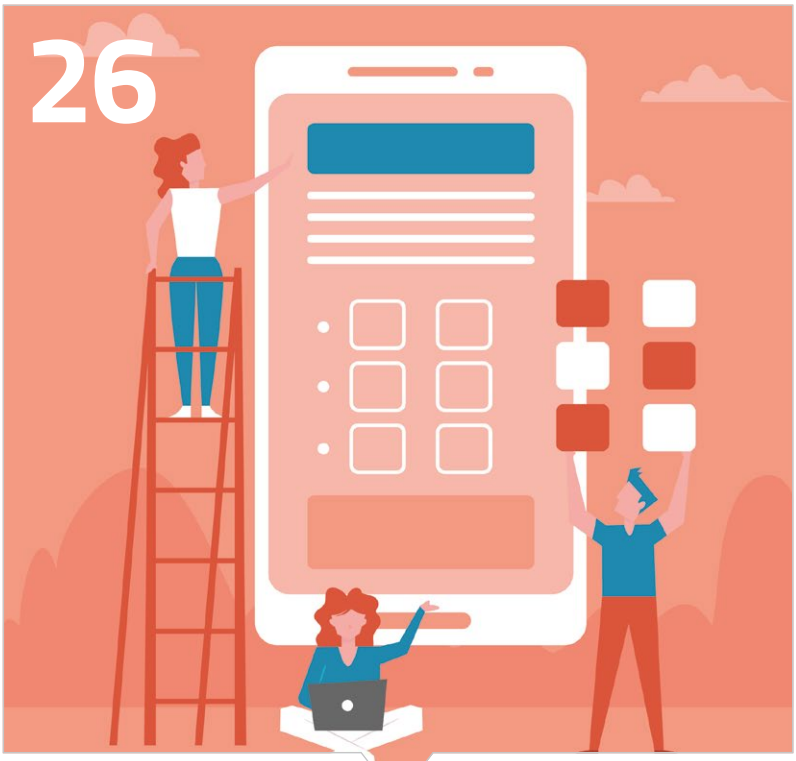
Auch eine Auswahl wissenswerter, nicht-technischer Fachartikel sind in dieser Ausgabe wieder mit vertreten. So zeigt uns Matthias Bünger in seinem Artikel ab Seite 54, welche Aspekte man bei der barrierefreien Gestaltung einer Website beachten sollte und weshalb alle Anwender:innen davon profitieren können. Danach berichtet Andreas Monschau von seinen eigenen Erfahrungen mit einem Trainingskonzept, dass Quereinsteiger:innen aus anderen Berufszweigen den Einstieg in die Softwareentwicklung ermöglicht. Dieses Programm hat er selbst sowohl als Trainee als auch als Trainer erlebt. Im Artikel "Psychologie für Softwareentwickler:innen" beschreibt Berend Semke anhand von Reverse Engineering das menschliche Gehirn als Datenverarbeitungssystem. Abgerundet wird diese Ausgabe durch Johannes Borns Artikel auf Seite 74. Er teilt fünf Gewohnheiten mit uns, die uns dabei unterstützen, unsere Wirkung sowohl innerhalb unseres Teams als auch bei Kunden:innen zu maximieren und souverän mehr Verantwortung zu übernehmen.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

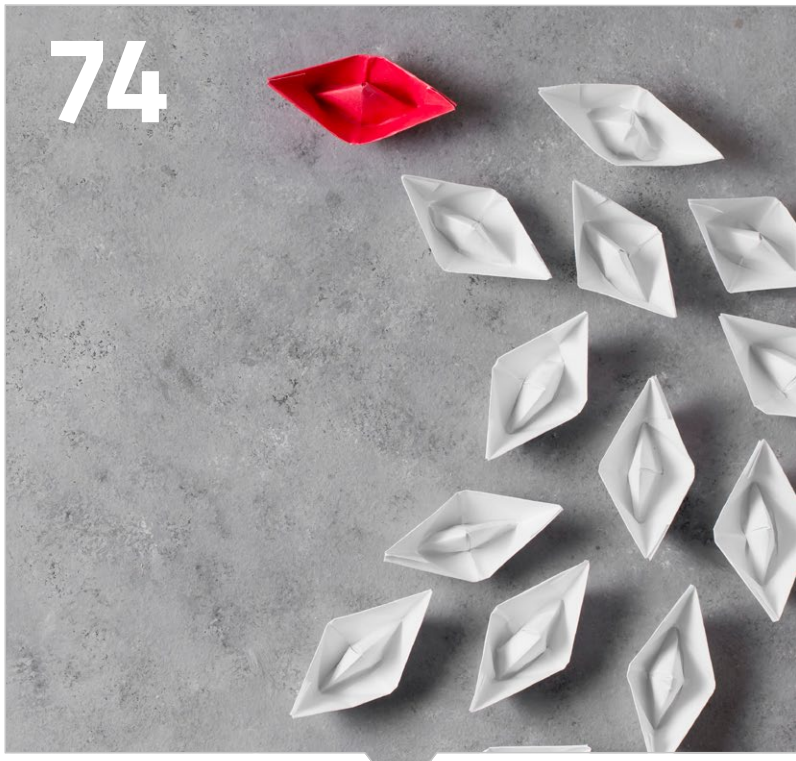
INHALT

Mit Kotlin Multiplatform plattformübergreifend programmieren



Modularisierung in Android-Apps: Monolith vs. Microkernel

Testabdeckung für Integrations- und Unittests mit Apache Maven



Verantwortung im gesamten Team: Fünf Gewohnheiten für die tägliche Praxis

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

25. Oktober 2023

Quarkus 3.5 unterstützt Java 21

Mit Release 3.5 ist Quarkus jetzt vollständig kompatibel mit und getestet auf Java 21. Das Haupthindernis war laut Projekt die Nutzung von Gradle für Teile der Builds, daher haben sie die JVM, die für Gradle selbst genutzt wird, von der JVM entkoppelt, mit der Quarkus gebaut und getestet wird, was wohl etwas gedauert hat.

3. November 2023

Kotlin Multiplatform „stabil"

JetBrains hat mit dem Release der Kotlin Version 1.9.20 verkündet, dass Kotlin Multiplatform jetzt offiziell „stable" und reif für den Produktionseinsatz ist. Damit soll die Sprache und das Tooling drumherum jetzt „auch in den konservativsten Nutzungsszenarien" (JetBrains) zur übergreifenden Entwicklung für Backend, Web, Desktop sowie native Android- und iOS-Apps eingesetzt werden. Die Arbeit geht trotzdem weiter, insbesondere was Performance-Verbesserungen für Build und Laufzeit angeht.

Update vom 17. November: Die Pläne für das Jahr 2024 sind jetzt auch veröffentlicht. Vor allem für die Zielplattform iOS soll die bislang etwas hinterherhinkende Unterstützung deutlich besser werden, zum Beispiel beim plattformunabhängigen UI-Framework Compose oder beim Projekt-Export (direkt von Kotlin auf Swift, statt den Umweg über Objective-C zu nehmen). Außerdem soll Fleet als gemeinsame IDE für alle Zielplattformen etabliert werden *[1]*.

8. November 2023

MicroProfile – Pläne für 2024

Es herrscht nicht immer Einigkeit in der MicroProfile-Working-Group, es wird hart diskutiert und nicht jedes Release durchgewunken. Bei den Plänen für das Projekt für 2024 sind sich aber alle einig. Vieles davon ist aus dem Bereich Marketing, Organisation und ähnliches, die Details erspare ich hier, aber interessant sind einige technische Ziele: Zum Beispiel die Evaluierungen neuer Spezifikationen für „Converters" (das zielt wohl auf die typsichere Konfiguration ab), das Serverless-Programmiermodell sowie die JWT-Bridge, die als gemeinsame Grundlage für Jakarta EE und MicroProfile dient. Außerdem soll MP Telemetry um Metrics und Logging (von OpenTelemetry) erweitert werden und neben der sowieso laufenden Abstimmung mit Jakarta EE soll eine „Gap Analysis" gegenüber dem Spring-Framework durchgeführt werden. Es gibt auch die Idee, MP Context Propagation zukünftig abzuschaffen und es deshalb schon mal mit dem Status „deprecated" zu markieren.

15. November 2023

Quarkus goes LLM

Java ist im Bereich KI bislang kein „First Class Citizen". Kauzige Milliardäre verkünden auf ihren Social-Media-Plattformen, dass eher Rust die KI-Sprache der Zukunft sei (und ernten dafür deutlich mehr Zuspruch als für ihre anderen, nicht immer publikationsfähigen Aussagen). Aber die Sprache, die seit langem diese Domäne dominiert, ist Python. Was auch historische Gründe hat – Data Scientists sind keine Software-Entwickler, aber das Ergebnis ist dasselbe. Einige Lichtblicke gibt es für Java-Fans: unter anderem den Port des langchain-Frameworks, langchain4j (das eine Toolkette und Abstraktionen rund um die Nutzung von Large-Language-Models bereitstellt). Aber auch hier läuft der Port dem Python-Original hinterher, was die Features und Integrationen angeht. Da trifft es sich gut, dass Quarkus jetzt an einer eigenen Integration von LLMs in Quarkus-Applikationen arbeitet, auf Grundlage von langchain4j – das dürfte auch langchain4j helfen. Andere große Java-Frameworks werden sicher auch auf den Zug aufspringen, Details zur Quarkus-Unterstützung stehen schon mal hier *[2]*.

20. November 2023

Spring-Framework 6.1

Das Spring-Framework zieht nach: Die neue Version 6.1 unterstützt nicht nur das aktuelle Java-LTS-Release 21. Hinzu gekommen ist auch die Integration mit CRaC (das OpenJDK-Feature „Coordinated Restore at Checkpoint"), womit die Startzeiten verkürzt werden sollen. Mit dieser von Azul initiierten und inzwischen auch im OpenJDK enthaltenen Funktionalität lassen sich Snapshots oder Checkpoints „aufgewärmter" Java-Prozesse zur Laufzeit speichern und von genau diesem Checkpoint aus später neue Instanzen starten, inklusive Lifecycle-Methoden für den nötigen Cleanup beim (Re-)Store. Andere Frameworks wie Quarkus und Micronaut sind schon etwas länger auf CraC, davon hatte ich aber bislang wohl noch nicht berichtet. Eine gute Übersicht bietet die GitHub-Seite zum CraC-Projekt *[3]*, von dort aus sind auch die entsprechenden Spring-Blog und -Doku-Seiten verlinkt.

25. November 2023

Aus MicroStream wird EclipseStore

Das MicroStream-Projekt für leichtgewichtige Persistierung von Objekt-Graphen (mittels Custom Serialisierung) und In-Memory Querying ist jetzt auch bei Eclipse unter dem Namen EclipseStore untergeschlüpft. Version 1.0.0 (auf Basis der letzten MicroStream Version 8.1.1) ist kürzlich veröffentlicht worden.

28. November 2023

Spring Boot 3.2 mit langer Feature-Liste

Und kurz nach dem Framework-Release kommt auch wieder die nächste Spring-Boot-Version heraus. In voller Bescheidenheit bewirbt „Developer Advocate" Josh Long das neue Release als „Game Changer". Das liegt zuerst an den beiden bereits für das Framework erwähnten Features (Java 21, CraC), die natürlich in Spring Boot genutzt werden. Aber darüber hinaus sind wirklich sehr viele kleinere Verbesserungen enthalten, zum Beispiel bei der SSL-/TLS-Unterstützung: automatischer Reload von SSL-Bundles und Unterstützung von Bundles für RabbitMQ oder Kafka Connections als Alternative zum Java Keystore; im Bereich „Observability" eine ganze Reihe von Verbesserungen bei Nutzung von Micrometer usw. Die Release-Notes sind jedenfalls ziemlich lang. Ob es wirklich ein „Game Changer" ist, kann ja jede:r für sich beurteilen.

30. November 2023

Gradle 8.5

Das erste Türchen hat sich für „Gradlephants" [greɪdəl fænz] schon geöffnet – das Build Tool unterstützt jetzt endlich Java 21. Da war doch was? Richtig, Quarkus – hoffentlich war die fünf Wochen frühere „time to market" die Plackerei wert.

3. Dezember 2023

JDK 22

Am 19. März 2024 soll OpenJDK 22 verfügbar sein. Inzwischen hat sich die Feature-Liste deutlich erweitert; viel dürfte nicht mehr hinzukommen, da die „Rampdown Phase One" am 7. Dezember startet, gleichbedeutend dem Feature-Freeze (ab da gibt es nur noch Fixes). Die gelisteten (stabilen) Features sind: das „Foreign Function & Memory API" (Interoperabilität mit Code und Daten außerhalb der Runtime), „Region Pinning for G1" (Hilfe für den GC bei Nutzung von JNI) und „Unnamed Variables & Patterns" (der Unterstrich als Symbol für nicht genutzte Elemente). Als Previews kommen dazu: „Statements before super(...)" – Konstruktur-Code, der vor einem „super"-Aufruf ausgeführt werden darf, wenn er die zu erzeugende Instanz nicht referenziert; ein „Class-File API" für das Handling von, nun ja, Class Files; „String Templates"; „Stream Gatherers" – quasi das Gegenstück zu Stream Collectors in der Mitte der Stream-Verarbeitung; und „Implicitly Declared Classes and Instance Main Methods" – um die Einstiegshürde in die Java-Programmierung zu senken. Dazu wird das Vector-API in seine siebte(!) Inkubator-Runde gehen. Die eventuell noch dazukommenden Features, die bislang nur für das JDK 22 vorgeschlagen wurden, sind: „Launch Multi-File Source-Code Programs" – auch dies zielt auf den einfachen Einstieg mit kleinen Programmen, ohne extra ein Build Tool zu konfigurieren; „Structured Concurrency (Second Preview)"; und „Scoped Values (Second Preview)". Die letzten beiden stammen aus dem Projekt Loom (weitere Features rund um Virtual Threads), was ich ja schon mehrfach erwähnt hatte.

Referenzen

[1] *https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-development-roadmap-for-2024/*

[2] *https://quarkus.io/blog/quarkus-meets-langchain4j/*

[3] *https://github.com/CRaC/docs*

Andreas Badelt

stellv. Leiter der DOAG Cloud Native Community

andreas.badelt@doag.org

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

MARKUS' ECLIPSE™-Corner

Der iJUG e. V. versucht seit geraumer Zeit die Interessen seiner Mitglieder, also der Java User Groups im deutschsprachigen Raum, dadurch zu vertreten, dass er Einfluss auf Java und Jakarta EE nimmt. In den zurückliegenden Jahren gab es dazu unterschiedliche Ansätze, wie beispielsweise den direkten Austausch mit Oracle, aber auch die Mitgliedschaft in der Eclipse Foundation.

Was Ersteres anbelangt, muss man ernüchternd konstatieren, dass sich Oracle genau genommen recht wenig darum geschert hat, was wir als Community über Oracles Engagement als aktueller Java-Steward so denken. Genau genommen war die Reaktion des Konzerns summa summarum eher, dass man nicht mehr mit uns reden will, damit man sich keine Kritik mehr anhören oder sich für sein Tun und Handeln rechtfertigen muss. Sei's drum, da manche von uns ja auch Open-Source-Programmierer sind, haben wir über OpenJDK-Pull-Requests mittlerweile direkten Einfluss auf den Quellcode von Java genommen: Einige Features in OpenJDK stammen tatsächlich federführend (wenn auch nicht ausschließlich) von iJUG-Mitgliedern, und einige gehen auf Vorschläge oder Vorarbeiten von uns zurück. Das habt ihr nicht gewusst, oder? Und mit Adoptium steht mittlerweile auch eine sehr erfolgreiche, kostenlose Java-Distribution bereit, bei der wir Einfluss ausüben und an deren Lizenzmodell Oracle kein Mitspracherecht besitzt.

Bei Jakarta EE, der ehemaligen Java-2-Enterprise-Edition, hingegen war es bislang so, dass der iJUG zwar ein Mitspracherecht besaß, dieses aber nicht ausüben konnte. Der Grund ist, dass der Mitgliedsbeitrag, den der iJUG an die betreffende Arbeitsgruppe der Stiftung abführt, zu gering war, um ein eigenes Stimmrecht auszuüben. Stattdessen mussten wir uns ein Stimmrecht mit anderen teilen. Man stelle sich vor, es ist Bundestagswahl, und ihr habt keine eigene Stimme! Stattdessen hat irgendjemand in eurer Stadt eine Stimme, die für die ganze Stadt gilt, und diese Person hat keine Lust, sich eure Meinung anzuhören. Blöd.

An dieser rechtlichen Situation hat sich bislang wenig geändert und da der iJUG nicht über unbegrenzte Reichtümer verfügt, können wir daran auch nicht wirklich etwas ändern. Daher sind wir umso erfreuter, dass sich der aktuelle Stimmrechtsinhaber auf der EclipseCon 2023 in Ludwigsburg offen dafür gezeigt hat, sich unsere Meinung zukünftig anzuhören. Was er dann damit tut, ja okay, das wird er dann auch weiterhin mit sich selbst ausmachen. Trotzdem finde ich es eine gute Sache: Es ist ein Schritt nach vorn, wenn wir Gehör finden. Und dass wir nun dabei helfen dürfen, ein offenes Kommunikationsforum für diesen „Channel" zu gestalten zeigt ebenso, dass es ihm ernst ist und der Vorschlag auch auf Dauer ausgerichtet sein soll. Warten wir's ab. Im Moment bin ich zuversichtlich, dass Jans Eingaben zukünftig Gehör finden (wer nicht weiß wer Jan ist, sollte mal seinen iJUG-Vertreter fragen, was auf den iJUG-Mitgliederversammlungen denn so diskutiert und beschlossen wird)!

Etwas zuversichtlicher bin ich in dieser Ausgabe auch hinsichtlich Jakarta EE 11. Es sind nun doch deutliche Aktivitäten zu sehen (quasi von 0 auf 100 seit der letzten Ausgabe). Klar, der Release-Termin rückt näher, und man hat wohl bemerkt, dass man noch so gut wie nichts vorzuweisen hat. Beispielsweise wird gerade bei Jakarta REST aka JAX-RS über Details der CDI-Integration diskutiert. Wenn das nicht wieder einschläft, könnte das Release tatsächlich noch recht interessant werden – also, nicht nur wegen Java 21 als Basis. Und wer meine Kolumne aufmerksam verfolgt, wird merken: Hey, der hat ja diesmal gar nicht gemeckert! Stimmt. Sorry. Nächstes Mal wieder, versprochen! ;-)

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.

Kotlin Multiplatform für mobile Anwendungen

Kurt Huwig

Von Apps wird erwartet, dass sie für Android und iOS verfügbar sind. Die native Entwicklung für beide unterscheidet sich stark, sodass viele Apps praktisch zweimal entwickelt werden, um auf beiden Plattformen verfügbar zu sein. Deshalb gab es schon früh Ansätze, Apps plattformübergreifend zu entwickeln – nicht immer mit gutem Erfolg. Kotlin Multiplatform (KMP) erhebt den Anspruch, dieses Problem zu lösen. Schauen wir uns an, welche Versprechen es halten kann und wo die Fallstricke liegen.


iOS
iOS
iOS
iOS
iOS
iOS

Eine plattformübergreifende Softwareentwicklung muss mehrere Herausforderungen bewältigen, um erfolgreich zu sein. Zuallererst muss sie ausführbaren Code für die Zielplattformen erzeugen können. Bei Android ist dies DEX-Bytecode, der von der Android Runtime (ART) wiederum in ARM-Maschinencode übersetzt wird und bei iOS ist es ARM-Maschinencode. KMP kommt mit den passenden Toolchains daher, um diese Aufgaben zu lösen. Somit ist keine separate Runtime notwendig, was die Apps schlank und performant hält.

Das zweite Problem ist das Look-and-Feel der Anwendung. Android und iOS – genau wie Windows und MacOS – haben unterschiedliche UI-Konzepte. Beispielsweise setzt iOS sehr stark auf Wischgesten, während Android-Apps eine Taskbar sowie eine Back-Taste haben, über die iOS-Apps nicht verfügen. Die Unterschiede sind bei der Navigation durch die Apps und den Standardelementen wie beispielsweise die Auswahl einer Uhrzeit oder eines Datums besonders groß. Die Designer vieler Apps versuchen, diese Unterschiede krampfhaft zu vereinheitlichen, sodass Chimären entstehen, die Benutzer auf beiden Betriebssystemen verwirren.

Kotlin Multiplatform löst dieses Problem, indem es das Problem gar nicht löst: Das UI einer KMP-App wird für jedes OS separat entwickelt. Dies erfordert plattform-spezifisches Wissen der Entwickler, hat aber den Vorteil, dass damit die Oberfläche so aussieht, wie Benutzer es von ihrem Betriebssystem gewohnt sind.

Was bedeutet dies in der Praxis? Der reine Vergleich von Programmcodezeilen zeigt, dass üblicherweise 20 bis 40 % des Programmcodes für das UI zuständig sind. Geht damit der ganze Vorteil von KMP verloren, da ein großer Teil doch von nativen Entwicklern programmiert werden muss? Ein genauerer Blick auf den Code gibt Entwarnung: Obwohl der Programmcode für das UI umfangreich ist, erfordert dessen Entwicklung in der Regel wenig Zeit: Sobald das Design existiert, kann man es normalerweise einfach „runterprogrammieren" und hat danach auch wenig Wartungsaufwand. Wohlgemerkt betrifft dies nur den reinen Code, der das UI aufbaut. Logikcode wie beispielsweise die Plausibilitätsprüfung von Eingaben ist davon nicht betroffen und kann plattformübergreifend entwickelt werden.

Die Benutzerschnittstelle ist nicht der einzige Punkt, bei dem es Unterschiede zwischen den Betriebssystemen gibt. Wie geht KMP damit um? Das grundsätzliche Ziel ist, dass möglichst viel Programmcode wiederverwendet werden kann. Beispielsweise gibt es Dinge, die für Systeme von Apple gleich sind – sowohl auf MacOS wie auch iOS – aber auf Android und Linux keine Anwendung finden. Hier verfolgt KMP einen hierarchischen Ansatz: Es gibt Programmcode, der für alle gleich ist und solchen, der nur für mache Systeme genutzt wird. In der Standardkonfiguration werden hierfür passende Verzeichnisse angelegt – diese kann man jederzeit um eigene Verzeichnisse erweitern. Der Compiler verwendet für jedes Betriebssystem den Programmcode, der am besten dazu passt. Kompiliert man beispielsweise eine App für den iOS-Simulator, so wird zuerst der Programmcode in „iosSimulatorArm64" verwendet. Wird dort nichts gefunden, geht es nach „ios", dann „apple", dann „native" und zum Schluss nach „common" *(siehe Abbildung 1)*. Programmcode im Verzeichnis „android" oder „linux" wird niemals für Apple-Hardware benutzt.

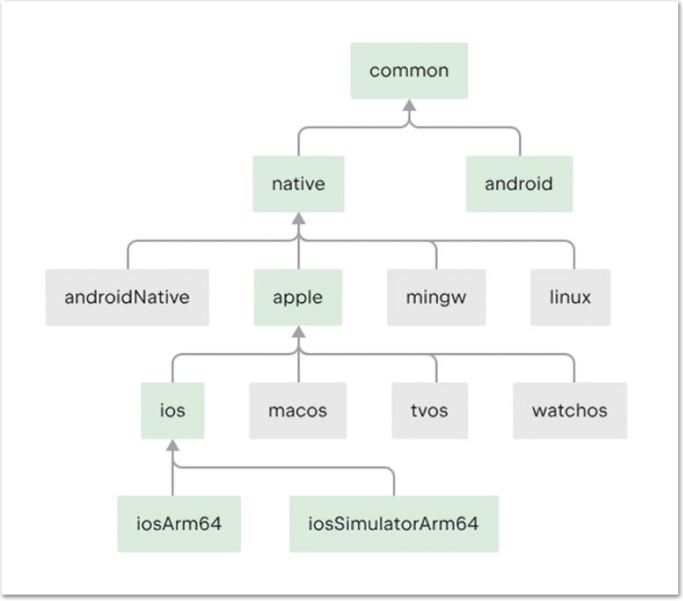


Abbildung 1: Hierarchical Project Structure

KMP-Bibliotheken

Eigenen Programmcode für die unterschiedlichen Betriebssysteme zu schreiben ist schön und gut, aber für die „üblichen Verdächtigen" wie Datenspeicherung und Netzwerkzugriff will man nicht selbst das Rad neu erfinden. Aber das müsste man, denn Dinge wie *java.util.File* existieren unter iOS nicht. Abhilfe schafft hier die Kotlin-Standard-Library – daher verwendet KMP *kotlin.Boolean* statt *java.lang.Boolean*. Unter *[1]* findet man eine Liste aller KMP-Bibliotheken. Das Angebot ist nicht gerade klein, aber leider gibt es für manche Herausforderungen mehrere Lösungen, beispielsweise Settings. Hier lohnt es sich, auf die letzten Releases zu schauen, ob das Projekt noch aktiv ist. Es ist zu erwarten, dass hier ein Konsolidierungsprozess stattfindet, zumal KMP mit der Version 1.9 zum ersten Mal als „stable" gekennzeichnet wurde und damit wohl mehr Entwickler den Schritt dazu wagen.

Bei jeder Bibliothek steht eine Liste der unterstützen Plattformen. Die Listen sind unterschiedlich und teilweise recht kurz, wodurch man hier genauer hinschauen sollte. Auch lösen teilweise mehrere Bibliotheken das gleiche Problem, somit hat man die Qual der Wahl – ein Blick auf die Aktivität der Projekte hilft bei der Auswahl. Die Abdeckung ist recht umfangreich, sodass man in der Praxis kaum Einschränkungen hat.

Clean Architecture

Schauen wir uns das Ganze an einem Praxisbeispiel an, und zwar einem einfachen Login-Bildschirm, wie er in vielen Apps vorkommt. Die Aufgabe ist ein einfacher Schirm mit Benutzername, Passwort, Login-Button und „Passwort vergessen"-Link. Der Login-Button soll aktiv werden, sobald plausible Daten eingegeben wurden. Nach dem Klick auf den Login soll während des Logins das UI gesperrt werden und ein *Throbber* erscheinen. Fehlermeldungen sind „Unbekannter Benutzername", „Falsches Passwort", „Netzwerkfehler".

Was kann man davon plattformunabhängig schreiben? Auf den ersten Blick steckt die Hauptarbeit in der Benutzeroberfläche, denn es wird lediglich ein Serveraufruf für die Anmeldung selbst getätigt. Schaut man genauer hin, stellt man fest, dass fast die gesamte Lo-

gik für alle Plattformen gleich ist und im UI lediglich einfache Methodenaufrufe erfolgen.

Fangen wir mit der Überprüfung der Eingabe an: Alles ab 8 Zeichen gilt als plausibles Passwort *(siehe Listing 1)*.

```
fun isPasswordPlausible(password: String) =
  8 <= password.trim().length
```

Listing 1: Festlegung der Passwortlänge

So einfach, so plattformunabhängig. Bei der E-Mail-Adresse wird es deutlich komplizierter, da E-Mail-Adressen sehr ungewöhnlich sein können – der korrekte reguläre Ausdruck hat 6,4kB. Glücklicherweise kommen Android und iOS mit passenden Lösungen daher, aber wie nutzt man diese in plattformunabhängigem Code? Hierfür bietet KMP das *expect/actual*-Konstrukt an: Es entspricht dem Strategie-Entwurfsmuster, hat aber den Vorteil, dass man zum einen den Code direkt ohne *Dependency Injection* oder eine Factory nutzen kann und zum anderen der Compiler meckert, wenn eine Implementation fehlt. Die Syntax hierfür ist denkbar einfach: Man schreibt eine Funktion ohne Implementation und statt `abstract` schreibt man `expect` davor *(siehe Listing 2)*.

```
expect fun isEmailPlausible(email: String): Boolean
```

Listing 2: Funktion mit `expect`

Sobald man das getan hat, beschwert sich die IDE bereits, dass die passenden Implementationen fehlen. Man kann über den Quick-Fix eine leere Implementation erzeugen, damit der Code zumindest wieder kompiliert. Für Android steht die Implementation in *Listing 3*.

```
import android.util.Patterns

actual fun isEmailPlausible(email: String) =
  Patterns.EMAIL_ADDRESS.matcher(email).matches()
```

Listing 3: Quick-Fix, damit der Code kompiliert

Aufgerufen wird sie im plattformunabhängigen Programmcode mit *Listing 4*.

```
if (isEmailPlausible(email)) {
```

Listing 4: Aufruf der E-Mail im plattformunabhängigen Code

Man sieht also gar nicht, dass hier je nach Plattform unterschiedlicher Code ausgeführt wird, was auch gut ist. Neben Funktionen kann man auch ganze Klassen mit `expect/actual` markieren, jedoch ist dies aktuell noch experimentell, sodass man es nicht unbedingt in produktivem Code verwenden sollte.

Da diese Implementierung Android-spezifisch ist, gehört sie nicht nach „commonMain", sondern in den Ordner „androidMain" und die iOS-Implementation entsprechend nach „iosMain".

Der Login kennt zwei unterschiedliche Status:
1. Eingabe der Benutzerdaten
2. Anzeige des *Throbbers*

Diese kann man als Enum modellieren und über einen Flow alle Änderungen daran kommunizieren *(siehe Listing 5)*.

```
enum class State {
  ENTER_CREDENTIALS,
  LOGGING_IN,
  LOGIN_SUCCESSFUL,
}

private val _state = MutableStateFlow(State.ENTER_CREDENTIALS)
val state = _state as StateFlow<*>
```

Listing 5: Login-Status mittels Enum

Wie man sieht, ist der Status von außen nur lesbar – Änderungen daran erfolgen ausschließlich über die Geschäftslogik – dazu gleich mehr. Das gleiche Prinzip wird für die Fehlermeldungen verwendet *(siehe Listing 6)*.

```
private val _emailError = MutableStateFlow("")
val emailError = _emailError as StateFlow<String>

private val _passwordError = MutableStateFlow("")
val passwordError = _passwordError as StateFlow<String>

private val _loginError = MutableStateFlow("")
val loginError = _loginError as StateFlow<String>
```

Listing 6: Programmierung der Fehlermeldung

Flows erlauben eine elegante reaktive Programmierung, bei der die Anwendungsfälle – ganz im Sinne der Clean Architecture *[3]* – nicht wissen, wer sie verwendet. Stattdessen bieten sie einen „Use Case Output Port" – eben genau diese *Flows* – mit dem die äußeren Schichten benachrichtigt werden können.

Jetzt kommt der einzige „richtige" Code, der Login selbst *(siehe Listing 7)*.

Für KMP gibt es mit *ktor [4]* eine leistungsfähige REST-Implementation, mit der man beispielsweise den eigentlichen Login plattformunabhängig realisieren kann, was aber hier den Rahmen sprengen würde. Schaut man sich den obigen Code an, so erkennt man, dass die eigentliche Benutzeroberfläche damit lediglich eine Fingerübung geworden ist, da die ganze Logik plattformunabhängig geschrieben werden konnte – eine saubere Architektur gibt es noch kostenlos obendrauf.

Sollte sich in der Zukunft etwas an der Logik ändern – beispielsweise eine verfeinerte Prüfung des Passwortes – so muss hierfür

```
suspend fun login(email: String, password: String) {
      var haveError = false
      _emailError.value = if (!isEmailPlausible(email)) {
            haveError = true
            "Ungültige E-Mail"
      } else {
            ""
      }
      if (isEmailPlausible(email)) {
      _passwordError.value = if (!isPasswordPlausible(password)) {
            haveError = true
            "Ungültiges Passwort"
      } else {
            ""
      }
      if (haveError) {
            return
      }
      _state.value = State.LOGGING_IN

      // hier dann der eigentliche Login am Backend
      if (doLogin(email, password)) {
            // Bei Erfolg teilen wir das mit
            _state.value = State.LOGIN_SUCCESSFUL
      } else {
            // ...und im Fehlerfall geht es zurück zur Eingabe:
            _state.value = State.ENTER_CREDENTIALS
            _loginError.value = "Ungültige Zugangsdaten"
      }
}
```

Listing 7: Finaler Code des Logins

lediglich der gemeinsame Code angepasst werden. Sowohl Android als auch iOS übernehmen diese Änderungen automatisch.

Auf den ersten Blick erscheint es als große Einschränkung, dass KMP von Haus aus kein plattformunabhängiges UI mitbringt. In der Praxis merkt man sehr schnell, dass diese kein wesentlicher Nachteil ist und man unterm Strich eine saubere App-Architektur sowie Oberflächen im Look-and-Feel der Zielplattform erhält. Dies erfreut sowohl Entwickler als auch Anwender.

Fazit

Mit Kotlin Multiplatform ist es sehr einfach, Anwendungen plattformübergreifend zu entwickeln und dank der stetig wachsenden Zahl an KMP-Bibliotheken unterliegt man kaum Einschränkungen. Sollte man dennoch an Grenzen stoßen, kann man mit *expect/actual* sehr elegant – und mit Compilerunterstützung – plattformspezifischen Code für Sonderfälle entwickeln. Damit ist sogar eine sanfte Migration einer reinen Android-App zu Multiplattform möglich.

Eine Einschränkung ist aktuell die Entwicklung der Benutzerüberfläche, da diese (noch *[5]*) nicht plattformübergreifend möglich ist. Ich sehe dies jedoch nicht als Nachteil, sondern als Vorteil an: Entwickler werden dazu gezwungen, native UI-Elemente zu verwenden. Die daraus resultierenden Anwendungen sind in der Regel näher an dem, was die Benutzer von anderen Apps gewohnt sind, und wirken nicht wie ein Fremdkörper. Für mich ist es deshalb eher ein Vorteil als ein Nachteil.

Links

[1] *https://github.com/AAkira/Kotlin-Multiplatform-Libraries*
[2] *https://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster)*
[3] *https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html*
[4] *https://ktor.io/*
[5] *https://www.jetbrains.com/de-de/lp/compose-multiplatform/*

Jetpack Compose für Benutzerschnittstellen

Kurt Huwig

Unter Android wurden Benutzerschnittstellen schon immer über Layout-XML-Dateien definiert. Jetpack Compose [1] bietet einen radikal neuen Ansatz für Benutzerstellen und schickt sich an, die etablierten XML-Dateien zu ersetzten. Kann es dieses Versprechen halten? Ist es klar strukturiert und wartbar? Oder landen wir hier bei Spaghetti-Code, der Layout, Styling und Logik vermischt?

Benutzerstellen für Android wurden bisher mit dem Dreiklang

- layout.xml
- styles.xml
- Logik in Java/Kotlin

entwickelt. Dies erlaubt eine klare Trennung von Design- und Logikaspekten, was die Wartbarkeit verbessert: Eine Änderung am Layout oder den Farben erfordert keine Änderung an Programmcode und umgekehrt. Dies erfüllt das Single-Responsible-Prinzip *[2]* und hält damit die Anwendungen leicht verständlich und gut wartbar.

Bei *Compose* wird das UI nicht über XML-Dateien, sondern über Programmcode definiert. Dies beinhaltet sowohl das Layout, das Styling als auch die Logik. Und ja, damit kann man leicht Spaghetti-Code erzeugen, der sich nur sehr schlecht warten lässt. Aber dies bedeutet nicht, dass man es so machen muss. Beachtet man ein paar einfache Hinweise, erhält man eine klare Trennung und leicht wartbaren Code.

Aber warum sollte man überhaupt das UI-System wechseln? Weil sich die Anforderungen geändert haben: Von heutigen Apps wird mehr „Schnickschnack" im UI erwartet. Statt statischen Bildschirmen zwischen denen hin- und hergewechselt wird, bleibt man auf einer Seite, die sich dynamisch ändert: Daten werden dynamisch geladen und erscheinen, sobald sie verfügbar sind – am besten noch mit einer Animation, die Listen auf- und zuklappt. Buttons werden geklickt und verwandeln sich in Haken und dann in Icons zum Löschen. So etwas bekommt man auch mit statischen XML-Dateien hin, aber es wird mühsam. Hat man beispielsweise ein Element, das seine Größe dynamisch ändern kann, so steht in der XML-Datei die Anfangsgröße. Über Programmcode wird diese beim Aufklappen geändert und beim Zuklappen – hoffentlich – wieder im Code auf den Wert gesetzt, der in der XML-Datei eingetragen ist. Hier gibt es viel Potenzial für obskure Fehler, die, je nach Abfolge der Benutzereingaben, etwas falsch darstellen. Mit automatisierten Tests kommt man leider nicht weit, sodass der manuelle Testaufwand hoch ist.

Da in Compose alles in Programmcode geschrieben wird, ist der Fall „auf- und zuklappen" kein Problem, da der Programmcode beide Größen kennt.

Und Hand aufs Herz: Wer hat eine layout.xml wirklich einmal wiederverwendet? Wer hat nicht schon mal auf Custom Views gewechselt, weil es einfach schneller umzusetzen war?

Arbeitsweise von Compose

In Compose wird die Benutzerschnittstelle dynamisch über Programmcode definiert – so wie dies *ReactNative* und *SwiftUI* auf iOS vorexerziert haben. Dieser wird geschachtelt geschrieben, um die Schachtelung des UI abzubilden – eine Spalte ordnet beispielsweise mehrere Elemente vertikal an. Dieser Programmcode wird beim Öffnen des UI-Elementes einmal ausgeführt, um die Oberfläche zu erzeugen – so weit so bekannt von layout.xml.

Der große Vorteil kommt bei einem „Recompose": Man kann Variablen definieren und wenn sich diese ändern, wird automatisch ein erneuter Aufruf des Compose-Programmcodes ausgelöst, der wiederum definiert, wie die Benutzerschnittstelle jetzt aussehen soll. Man merkt sofort, dass Animationen damit sehr einfach werden: Man definiert eine Variable, die beispielsweise die Höhe einer Liste festlegt und ändert diese in kurzen Zeitabständen – und schon hat man eine Öffnungsanimation! Aber ist das nicht langsam und stromfressend, wenn das UI ständig neu berechnet wird? Nein, denn Compose erkennt automatisch, welche Teile von Änderungen der Variablen betroffen sind und erstellt nur diese Teile neu. Hört sich wie dunkle Magie an, ist aber sehr einfach zu verstehen und umzusetzen. Leider müssen gestandene Android-Entwickler jetzt sehr stark sein: Compose sieht komplett anders aus und fühlt sich auch komplett anders an. Aber es lohnt sich und nach einer Eingewöhnungsphase kann man damit leichter Benutzerschnittstellen entwickeln, als man es bisher gewohnt war.

Hello Compose

Schauen wir uns ein Beispiel an: Hierzu setzen wir am besten ein neues Projekt auf und folgen den Setup-Anweisungen des Tutorials *[3]*. Danach gehen wir in die Activity und schreiben dort, wo wir bisher ein `setContentView()` geschrieben hätten, den Code aus *Listing 1*.

```
setContent {
  MaterialTheme {
        Surface {
              Text("Hello World")
        }
  }
}
```

Listing 1: Hello World in Compose

Wie man sieht, macht Compose regen Gebrauch der Kotlin-Konvention „trailing lambdas" *[4]*. Nutzt man dies, so ähnelt der Programmcode einer Domain-Specific-Language *[5]* für Benutzerschnittstellen und ist gut lesbar eingerückt, weshalb es empfehlenswert ist. Was sehen wir hier? Zuerst wird das *Theme* definiert, über das Farben, Schriftarten und so weiter definiert werden. Auch das *Theme* ist Programmcode, sodass man Dinge dynamisch ändern kann. Ein `Surface` ist die zentrale Metapher des *Material Designs* *[6]* und bringt das Konzept einer „Höhe"; diese beeinflusst Dinge wie Schatten, Farbgebung, Überlappung und so weiter. Der Text schließlich ist der `Text`, den man auf dem Bildschirm sieht.

Dynamik

Bringen wir etwas Ordnung und Dynamik in die Oberfläche. Unter dem Text soll ein Button sein und wenn man daraufklickt, soll sich der Text ändern. Die Anordnung erhält man durch die Verwendung einer *Column*. Der Button erhält seine Beschriftung über einen eingebetteten Text – ungewöhnlich, wenn man das alte System gewohnt ist, aber sehr praktisch, um Vorgaben von Designern von „Buttons mit Text & Bildern" umzusetzen.

```
var helloText = "Hello World"
setContent {
  MaterialTheme {
      Surface {
          Column {
              Text(text = helloText)
                  Button(
                      onClick = { helloText = "Hello Compose" },
                  ) {
                  Text("Klick mich!")
              }
          }
      }
  }
}
```

Listing 2: So funktioniert es nicht!

Listing 2 zeigt einen Versuch davon, der gut aussieht, aber leider einen Fehler enthält: Klickt man auf den Button, so ändert sich zwar die Variable `helloText`, aber nicht der Text im UI. Warum ist das so? Compose kocht auch nur mit Wasser und hat keine Möglichkeit, herauszufinden, dass sich die Variable geändert hat. Man muss deshalb Compose mitteilen, dass sich die Variable ändern kann und, dass dann das UI neu berechnet werden soll. Dies geschieht über `remember` und einen `mutableState` *(siehe Listing 3)*.

```
var actualText by remember { mutableStateOf(helloText) }
Text(text = actualText)
Button(
  onClick = { actualText = "Hello Compose" },
) {
  Text("Klick mich!")
}
```

Listing 3: So funktioniert es!

Wer den `MutableState` und nicht dessen Wert im Zugriff haben möchte, kann den Code wie in *Listing 4* dargestellt ändern. Welchen Weg man geht, ist egal – ich finde den ersten eleganter.

Aber es wird noch besser: Wenn man statt `remember` ein `rememberSaveable` verwendet, überlebt die Variable sogar eine Neuerstellung der Activity, beispielsweise bei einem Wechsel in den Darkmode oder ins Querformat.

```
val actualText = remember { mutableStateOf(helloText) }
Text(text = actualText.value)
Button(
  onClick = { actualText.value = "Hello Compose" },
) {
  Text("Klick mich!")
}
```

Listing 4: Direkter Zugriff auf den `MutableState`

Nachdem jetzt grob klar ist, wie man mit Compose eine Benutzeroberfläche baut und sie mit Daten bestückt, ist die nächste Frage, wie das UI mit dem Rest der App interagiert. Gemäß dem Model-View-ViewModel-Prinzip *[7]* sollte die Oberfläche nichts über die Logik oder Datenquellen wissen. Solange statische Daten angezeigt werden, kann man diese einfach dem Compose-Programmcode übergeben. Aber wie sieht es aus, wenn man stattdessen einen `Flow` bekommt und UI-Ereignisse übermitteln möchte?

```
val mutable = MutableStateFlow("Hello World")

val changeText = { newText: String -> mutable.value = newText }
val state = mutable as StateFlow<String>

setContent {
  MaterialTheme {
      Surface {
          Column {
              val actualText = state.collectAsState()
              Text(text = actualText.value)
              Button(
                  onClick = { changeText("Hello Compose") },
              ) {
                  Text("Klick mich!")
              }
          }
      }
  }
}
```

Listing 5: Interaktion mit der Außenwelt

Listing 5 zeigt, wie man dies recht einfach bewerkstelligen kann: Die *Flows* werden Compose über `collectAsState()` bekannt gemacht und die Ereignisse werden per Callback realisiert. Auf diese Weise erhält man ein UI, das sauber vom Rest der App getrennt ist und damit den Wartungsaufwand gering hält.

Tools für UI-Entwicklung

Bei der bisherigen UI-Entwicklung konnte man auf leistungsfähige WYSIWYG-UI-Editoren zurückgreifen, mit denen man sich seine Oberfläche leicht und schnell zusammenklicken konnte. Dies geht mit Compose leider (noch) nicht. Stattdessen muss man Programmcode schreiben (Wen wundert es?), der eine „Preview" der Oberfläche gibt.

```
@Preview
@Composable
fun HelloWorldView() {
  MaterialTheme {
      Surface {
          Text("Hello World")
      }
  }
}
```

Listing 6: Preview eines Compose-UI

Listing 6 zeigt, wie es geht: Man annotiert eine Funktion mit `@Preview` (und `@Composable`, wie jede Funktion, die Compose-Elemente definiert). Üblicherweise definiert man das UI nicht hier direkt, sondern ruft die eigentliche Funktion auf, die das UI-Element zusammenbaut. In der neuesten Version von Android Studio gibt es zudem die Funktion Live Edit *[8]*, die viele Code-Änderungen live in einer bestehenden App umsetzen kann. Damit sieht man sofort, wie die echte App auf Änderungen reagiert.

Was gibt es sonst noch zu sagen? Sehr viel! In diesem Artikel konnte nur die Oberfläche angekratzt werden. Wechselt man von layout.xml auf Compose, so muss man einiges neu lernen, was die Umsetzung eines Designs angeht. Vieles bleibt gleich, wie die Namen der UI-Standardelemente – Text, TextField, Button und so weiter. Auch Prinzipien wie Gravity, Weight und geschachtelte Layouts sind gleich. Woran man sich gewöhnen muss, ist, dass man Programmcode statt Views übergibt. Aber wenn man das einmal verinnerlicht hat, ist es gar nicht mehr so schwer. Das Compose-Tutorial *[9]* ist ein exzellenter Startpunkt für den schnellen Einstieg.

Fazit

Mit Jetpack Compose erfährt das Android-Layout-System die tiefgreifendste Umstellung, die es je hatte. Für alte Hasen ist es eine gravierende Umstellung, die sicherlich schmerzhaft ist. Trotzdem lohnt sich der Einstieg, da sich damit moderne Benutzeroberflächen viel einfacher, schneller und leichter wartbar entwickeln lassen, als es bisher möglich war. Das betrifft nicht nur hippe Animationen, sondern mit Compose ist es endlich möglich, sehr einfach wiederverwendbare Komponenten zu entwickeln.

Compose ist gekommen, um zu bleiben!

Links

[1] *https://developer.android.com/jetpack/compose*
[2] *https://de.wikipedia.org/wiki/Single-Responsibility-Prinzip*
[3] *https://developer.android.com/jetpack/compose/setup*
[4] *https://kotlinlang.org/docs/lambdas.html#passing-trailing-lambdas*
[5] *https://de.wikipedia.org/wiki/Dom%C3%A4nenspezifische_Sprache*
[6] *https://m3.material.io/*
[7] *https://de.wikipedia.org/wiki/Model_View_ViewModel*
[8] *https://developer.android.com/studio/preview/features#live-edit*
[9] *https://developer.android.com/jetpack/compose/tutorial*

Kurt Huwig

Kurt Huwig entwickelt seit 2009 Android-Anwendungen und hat immer noch sein T-Mobile G1 von damals. Er arbeitet bei der Deutschen Telekom als Softwareentwickler in den Bereichen Fintech, Smarthome, Cloud und Sicherheit. Wenn er in seiner Freizeit nicht gerade auch Apps entwickelt, spielt er gerne Brettspiele, entwirft und druckt 3D-Modelle oder übt sich in Kampfkunst.

JavaLand

www.javaland.eu

AM NÜRBURGRING

09. –
11.04. 2024

Early Bird bis 15.02.2024

JETZT TICKETS

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

PROGRAMM JETZT ONLINE

SICHERN!

Veranstalter: JavaLand

OpenID Connect für native Mobile-Apps

Sven-Torben Janus, Conciso GmbH

Im Zeitalter mobiler Anwendungen gewinnt die Sicherheit der Benutzerauthentifizierung in nativen Apps zunehmend an Bedeutung. In diesem Artikel zeige ich Best Practices für die Mobile Authentication unter Verwendung des OpenID-Connect-Protokolls (OIDC) auf. Besonderer Fokus liegt dabei auf nativen Apps. Ich gebe Einblicke in den Autorisierungsablauf, die Inter-App-Kommunikation und grundlegende Sicherheitsaspekte. Erfahren Sie, wie die Verwendung externer User Agents und standardisierte Kommunikation mittels URIs die Sicherheit erhöht, die Benutzerfreundlichkeit steigert und Implementierungskomplexität minimiert.


23:56
YOUR CODE
56698
USER NAME
****
ENTER CODE

Einleitung

Die rasante Verbreitung von mobilen Apps hat die Anforderungen an die Sicherheit und Effizienz von Authentifizierungssystemen stark beeinflusst. Insbesondere bei nativen Mobile-Apps steht die Gewährleistung eines reibungslosen und gleichzeitig sicheren Authentifizierungsprozesses im Mittelpunkt. In diesem Kontext hat sich das OIDC-Protokoll als Standard etabliert und spielt dabei eine entscheidende Rolle.

In diesem Artikel gehe ich auf bewährte Methoden und Best Practices für die Mobile Authentication unter Verwendung von OIDC ein, wobei der Schwerpunkt auf der Authentifizierung für native Mobile-Apps liegt. Ich stelle praxisnahe Beispiele vor, um Lesern eine fundierte Grundlage für die Implementierung in ihren Projekten zu bieten.

Im weiteren Verlauf des Artikels behandle ich die wichtigsten Aspekte, angefangen bei einer Beschreibung des Autorisierungsablaufs für native Apps über den Browser bis hin zur Implementierung der Inter-App-Kommunikation mittels *Uniform Resource Identifiers* (URI). Zusätzlich zeige ich bewährte Varianten für den Empfang von Autorisierungsantworten in nativen Apps auf. Grundlegende Best Practices in puncto Sicherheit, einschließlich *Proof Key for Code Exchange* (PKCE) zum Schutz des Autorisierungscodes, bleiben dabei nicht außen vor. Entwicklern, Architekten und Sicherheitsbeauftragten bietet dieser Artikel praxisrelevante Handlungsanleitungen.

Autorisierungsablauf für native Apps über den Browser

OpenID Connect unterstützt diverse Abläufe zur Authentifizierung von Nutzern, sogenannte *Flows*. Allen voran sind hier der *Authorization Code Flow*, *Direct Grant Flow* und *Implicit Flow* zu nennen.

Zunächst möchte ich erläutern, wie der *Authorization Code Flow* aussieht:

1. Der Flow beginnt mit dem sogenannten *Client-Initiated Authorization Request*. Dabei startet der Nutzer den Authentifizierungsablauf durch Interaktion mit der Client-Anwendung, in unserem Fall einer nativen App.
2. Die Anwendung leitet den Nutzer zwecks Autorisierungsanfrage an den *Authorization Server* weiter.
3. Anschließend erfolgt die Nutzerauthentifizierung. Der *Authorization Server* präsentiert dazu dem Nutzer in der Regel eine Anmeldeseite und bittet um Zustimmung zur Autorisierung der Client-Anwendung. Nach Zustimmung generiert der *Authorization Server* einen Autorisierungscode (*Authorization Code*).
4. Der Autorisierungscode wird an den in der Anwendungsregistrierung festgelegten Umleitungs-URI zurückgegeben. Die Client-Anwendung kann nun den Autorisierungscode als den des URI abrufen.
5. Es folgt eine Tokenanforderung durch den Client, indem die Client-Anwendung den Autorisierungscode verwendet, um beim Token-Endpunkt des *Authorization Servers* ein Zugriffstoken (*Access Token*) anzufordern.
6. Der *Authorization Server* validiert den Autorisierungscode. Bei erfolgreicher Validierung sendet der *Authorization Server* Zugriffs- und Auffrischungstoken (*Access und Refresh Token*) an die Client-Anwendung.
7. Die Client-Anwendung kann das erhaltene Zugriffstoken daraufhin verwenden, um auf geschützte Ressourcen beim *Resource Server* zuzugreifen.

Der *Authorization Code Flow* stellt den Quasi-Standard im Sinne einer Best Practice für die Authentifizierung mobiler, nativer Apps dar.

Die Verwendung von *Direct Grant Flows* (aka *Resource Owner Password Credentials*) und *Implicit Flows* für die Authentifizierung in nativen Apps ist mit erheblichen Sicherheitsrisiken verbunden und daher heutzutage nicht mehr empfehlenswert. Bei *Direct Grant Flows* werden der Benutzername und das Passwort direkt an den *Authorization Server* übermittelt, was potenziell unsicher ist und die Gefahr birgt, dass (potenziell weniger vertrauensvolle Apps) sensible

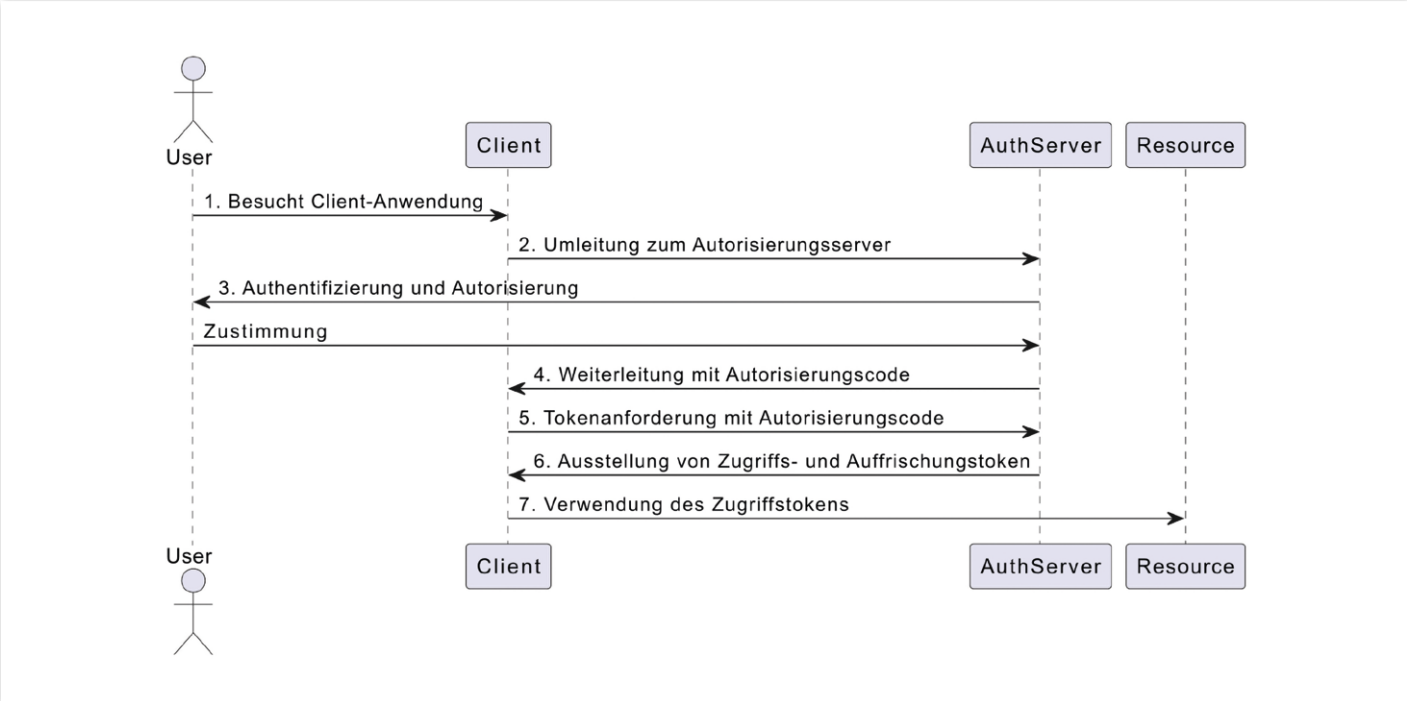


Abbildung 1: Schematische Darstellung des OIDC Authorization Code Flows

Anmeldeinformationen abfangen, beziehungsweise mitschneiden. Der *Implicit Flow* hingegen überträgt *Access Tokens* direkt über den Browser, ohne dass ein *Authorization Code* verwendet wird. Dies erhöht das Risiko von *Token Exposure*, da die Tokens möglicherweise in unsicheren Umgebungen wie Browser-Verlauf oder Cookies gespeichert werden.

In einem nativen App-Szenario, in dem der Schutz von sensiblen Benutzerdaten von höchster Priorität ist, sind diese Flow-Typen meiner Ansicht nach unangemessen.

Für die Sicherheit und Integrität der Authentifizierung von nativen Apps ist es daher ratsam, auf den *Authorization Code Flow* in Kombination mit *Proof Key for Code Exchange* umzusteigen. Dieser Flow minimiert das Risiko von *Token Exposure* und unerlaubten Zugriffen, indem er einen zusätzlichen Schutzschritt für den *Authorization Code* einführt, der speziell auf die Anforderungen nativer Apps zugeschnitten ist. Wie das genau aussieht, beschreibe ich später.

Inter-App-Kommunikation für OIDC

Wie der *Authorization Code Flow* zeigt, nutzt OIDC zur Inter-App-Kommunikation URIs, ähnlich wie im OAuth-2.0-Webkontext (Open Authorization), um die Autorisierungsanfrage zu initiieren und die Autorisierungsantwort an die anfordernde Website zurückzugeben. In nativen Apps können URIs verwendet werden, um die Autorisierungsanfrage im Browser des Geräts zu starten und die Antwort dann an die anfragende native App zurückzugeben.

Indem dieselben Methoden wie im Web für Open Authorization übernommen werden, ergeben sich auch im Kontext nativer Apps Vorteile. Die Benutzerfreundlichkeit einer Single-Sign-On-Sitzung und die Sicherheit eines separaten Authentifizierungskontexts werden so realisiert. Die Wiederverwendung dieses Ansatzes reduziert zudem die Implementierungskomplexität und steigert die Interoperabilität. Dabei wird auf standardbasierte Web-Abläufe gesetzt, die nicht auf eine bestimmte Plattform beschränkt sind.

Um bewährten Praktiken zu entsprechen, müssen native Apps einen externen Benutzer-Agenten, zumeist einen Browser, verwenden, um OAuth-Autorisierungsanfragen durchzuführen. Dies wird erreicht, indem die Autorisierungsanfrage im Browser geöffnet wird und ein Umleitungs-URI verwendet wird, der die Autorisierungsantwort zurück an die native App sendet. Diese Inter-App-Kommunikation ermöglicht eine sichere und effiziente Autorisierung unter Verwendung von OIDC.

Autorisierungsanfragen nativer Apps

Um die Autorisierung nativer Apps zu initiieren, erstellen diese einen Autorisierungsanfrage-URI mit dem sogenannten *Authorization Code Grant Type*. Hierbei wird ein Umleitungs-URI verwendet, der von der nativen App empfangen werden kann.

Die Funktion des Umleitungs-URIs für eine Autorisierungsanfrage nativer Apps ist vergleichbar mit der einer webbasierten Autorisierungsanfrage. Statt die Autorisierungsantwort an den Server des OAuth-Clients zurückzugeben, sendet der von einer nativen App verwendete Umleitungs-URI die Antwort an die App zurück. Verschiedene Optionen für einen Umleitungs-URI, der die Autorisierungsantwort an die native App in verschiedenen Plattformen zu-

rücksendet, erläutere ich später. Jeder Umleitungs-URI, der der App ermöglicht, den URI zu empfangen und seine Parameter zu überprüfen, ist grundsätzlich geeignet.

Nachdem der Autorisierungsanfrage-URI erstellt wurde, verwendet die App plattformspezifische APIs, um den URI in einem externen Benutzer-Agenten zu öffnen. Typischerweise handelt es sich dabei um den Standardbrowser, der für die Verarbeitung von *„http"-* und *„https"-URIs* auf dem System konfiguriert ist. Es können verschiedene Auswahlkriterien für die Browservariante und andere Kategorien von externen Benutzer-Agenten verwendet werden.

Für native Apps, die eine Benutzerautorisierung benötigen, ist die Erstellung einer Autorisierungsanfrage mittels OpenID Connect ein empfohlener Ansatz. Die Nutzung von OIDC ermöglicht es, neben der reinen Autorisierung auch Authentifizierungsinformationen zu erhalten, was die Sicherheit und Benutzerfreundlichkeit weiter verbessert.

Öffentliche native App-Clients sollten besonders die PKCE-Erweiterung für OAuth implementieren, um die Sicherheit des Autorisierungscodes zu stärken. Dieser Schutzschritt ist entscheidend, um mögliche Angriffe auf die Autorisierungsanfrage zu minimieren.

Die Verwendung eines externen Benutzer-Agenten, wie beispielsweise eines Browsers, für OAuth-Autorisierungsanfragen ist eine bewährte Methode, die die Sicherheit erhöht und die Interoperabilität zwischen verschiedenen Plattformen verbessert. Insbesondere wird die Verwendung von In-App-Browser-Tabs empfohlen, sofern von der Plattform unterstützt, um sowohl die Usability als auch die Sicherheitskontexte zu optimieren.

Varianten zum Empfang von Autorisierungsantworten

Um die Autorisierungsantwort von einem Browser zu empfangen, stehen nativen Apps verschiedene Umleitungs-URIs zur Verfügung, deren Verfügbarkeit und User Experience je nach Plattform variieren.

URI-Schema-Umleitung mittels Custom URL Schemes

Viele mobile und Desktop-Plattformen unterstützen die Inter-App-Kommunikation über URIs, indem Apps private URI-Schemata registrieren (umgangssprachlich als *Custom URL Schemes* bezeichnet), wie zum Beispiel `com.example.app.my`. Bei Verwendung eines solchen URI-Schemas wird die entsprechende App gestartet, um die Anfrage zu verarbeiten.

Um eine OAuth-2.0-Autorisierungsanfrage mit einer URI-Schema-Umleitung durchzuführen, startet die native App den Browser mit einer Standard-Autorisierungsanfrage, bei der der Umleitungs-URI ein zuvor registriertes privates URI-Schema verwendet.

Es ist wichtig, dass Apps ein URI-Schema basierend auf einer unter ihrer Kontrolle stehenden Domain verwenden. Wie in RFC 7595 empfohlen, wird dieses Schema rückwärts ausgedrückt. Zum Beispiel kann eine App, die die Domain `my.app.example.com` kontrolliert, `com.example.app.my` als ihr Schema verwenden.

Nach Abschluss der Autorisierungsanfrage leitet der Autorisierungsserver zum normalen Umleitungs-URI zurück, der die native App startet und den URI als Startparameter übergibt. Die App kann dann die Autorisierungsantwort normal verarbeiten.

Claimed „https"-Schema-URI-Umleitung

Einige Betriebssysteme ermöglichen es Apps, *„https"-Schema-URIs* in den von ihnen kontrollierten Domänen zu beanspruchen (*claim*). Bei Aufruf eines solchen URIs startet der Browser nicht die Seite, sondern die native App mit dem URI als Startparameter.

Solche URIs können von nativen Apps als Umleitungs-URIs verwendet werden und sind für den Autorisierungsserver nicht von regulären, webbasierten Client-Umleitungs-URIs zu unterscheiden. App-beanspruchte *„https"-Schema-Umleitungs-URIs* bieten Vorteile hinsichtlich der Identifizierung der Ziel-App gegenüber anderen nativen Umleitungsoptionen, weshalb native Apps sie nutzen sollten, wo immer dies möglich ist.

Best Practices in puncto Sicherheit

Nachdem ich zuvor beschrieben habe, wie OIDC für die Authentifizierung nativer Apps genutzt werden kann, gebe ich nachfolgend einige Hinweise zu Best Practices in puncto Sicherheit.

Schutz des Autorisierungscodes durch PKCE

Die genannten Umleitungs-URI-Optionen teilen den Vorteil, dass nur eine native App auf demselben Gerät oder die Website der App den Autorisierungscode empfangen kann, was die Angriffsfläche begrenzt. Dennoch besteht die Möglichkeit des Code-Abfangens durch eine andere native App auf demselben Gerät.

Eine Einschränkung bei der Verwendung von privaten URI-Schemata für Umleitungs-URIs besteht darin, dass mehrere Apps in der Regel dasselbe Schema registrieren können, was unklar macht, welche App den Autorisierungscode empfangen wird. Die RFC 7636 zum bereits erwähnten PKCE-Protokoll beschreibt, wie diese Einschränkung für einen Angriff verwendet werden kann, um den Autorisierungscode abzufangen.

Umleitungs-URIs basierend auf Loopback-IP-Adressen können auf einigen Betriebssystemen anfällig für Abfangen durch andere Apps sein, die auf dieselbe Loopback-Schnittstelle zugreifen. Von Apps beanspruchte *„https"-Schema-Umleitungs-URIs* sind weniger anfällig für das Abfangen aufgrund der Anwesenheit der URI-Autorität, jedoch handelt es sich weiterhin um öffentliche Clients. Darüber hinaus wird der URI mithilfe des URI-Dispatch-Handlers des Betriebssystems mit unbekannten Sicherheitseigenschaften übermittelt.

Das PKCE-Protokoll wurde speziell entwickelt, um diesen Angriffsvektor abzuschwächen. Im Kern handelt es sich um eine Proof-of-Possession-Erweiterung für OAuth 2.0, die den Autorisierungscode vor dem Abfangen schützt, bevor dieser verwendet wird. Um diesen Schutz zu bieten, generiert der Client einen geheimen *Verifier*. Er übermittelt einen Hash dieses *Verifiers* in der initialen Autorisierungsanfrage und muss den nicht gehashten Verifier beim Einlösen des Autorisierungscodes gegen ein Token vorlegen. Eine App, die den Autorisierungscode abgefangen hat, würde nicht im Besitz dieses Geheimnisses sein und der Code wäre nutzlos.

Die Verwendung von PKCE ist zur sicheren Authentifizierung sowohl für Clients als auch für Server bei öffentlichen nativen App-Clients erforderlich. Autorisierungsserver sollten im Sinne einer Best Practice Autorisierungsanfragen von nativen Apps ohne PKCE ablehnen und einen entsprechenden Fehler zurückgeben.

Cross-App Request Forgery Protections

Cross-App-Request-Forgery-Angriffe ähneln stark den aus dem Web bekannten Cross-Site-Request-Forgery-Angriffen (CSRF). Um diese zu unterbinden, ist eine bewährte Methode, Client-Anfragen und -Antworten zu verknüpfen. Die RFC 6819 empfiehlt hierzu den `state`-Parameter zu verwenden.

Um *Cross-App Request Forgery* über Inter-App-URI-Kommunikationskanäle zu verhindern, wird ebenfalls empfohlen, dass native Apps eine hochentropische, sichere Zufallszahl im `state`-Parameter der Autorisierungsanfrage einschließen. Die App sollte eingehende Autorisierungsantworten ohne einen übereinstimmenden `state`-Wert für eine ausstehende ausgehende Autorisierungsanfrage ablehnen.

Die Verwendung des `state`-Parameters in diesem Kontext ermöglicht es der nativen App, Anfragen und Antworten zu verknüpfen und sicherzustellen, dass die Autorisierungsantworten auf gültige, ausstehende Autorisierungsanfragen zurückzuführen sind. Dieser Mechanismus stärkt die Sicherheit der Inter-App-Kommunikation und schützt vor CSRF-ähnlichen Angriffen, bei denen ein böswilliger Dritter versucht, unbefugt auf autorisierte Ressourcen zuzugreifen, indem er die Autorisierungsantworten manipuliert.

Probleme mit Embedded User-Agents

OAuth 2.0, im speziellen RFC 6749, dokumentiert zwei Ansätze für native Apps, um mit dem Autorisierungsendpunkt zu interagieren. Im Sinne einer Best Practice sollten native Apps eingebettete User-Agents nicht verwenden, um Autorisierungsanfragen durchzuführen. Autorisierungsendpunkte können und sollten sogar Maßnahmen ergreifen, um Autorisierungsanfragen in eingebetteten User-Agents zu erkennen und zu blockieren.

Eingebettete User-Agents sind zwar eine Möglichkeit zur Autorisierung nativer Apps, diese eingebetteten User-Agents sind aber per Definition unsicher für die Verwendung durch Dritte gegenüber dem Autorisierungsserver. Die App, die den eingebetteten User-Agent hostet, hat vollen Zugriff auf die Authentifizierungsinformationen des Benutzers und nicht nur auf die für die App vorgesehene OAuth-Autorisierungsgenehmigung.

In typischen Implementierungen von eingebetteten User-Agents basierend auf Web-Views kann die Host-Anwendung jeden Tastenanschlag im Anmeldeformular aufzeichnen, Benutzernamen und Passwörter erfassen oder sogar Formulare automatisch senden. Hierdurch kann die Zustimmung des Benutzers umgangen oder Sitzungscookies kopieren werden, um authentifizierte Aktionen im Namen des Benutzers durchzuführen.

Auch wenn sie von vertrauenswürdigen Apps desselben Anbieters wie der Autorisierungsserver verwendet werden, verstoßen eingebettete User-Agents gegen das Prinzip des geringsten Privilegs, da sie Zugriff auf mächtigere Anmeldeinformationen haben, als sie benötigen. Dadurch wird potenziell die Angriffsfläche erhöht.

Benutzer dazu zu ermutigen, Anmeldeinformationen in einen eingebetteten User-Agent ohne die übliche Adressleiste und sichtbaren Zertifikatsvalidierungsfunktionen, die Browser haben, einzugeben, macht es für den Benutzer unmöglich zu wissen, ob sie sich bei einer legitimen Website anmelden. Selbst wenn dies der Fall ist, führt es dazu, dass sie ohne vorherige Validierung der Website ihre Anmeldeinformationen eingeben.

Fazit

Die Best Practices für die Sicherheit und Effizienz der Authentifizierung in nativen Mobile-Apps legen den Fokus auf die Vermeidung eingebetteter User-Agents, die Integration von PKCE und eine präzise Handhabung von Autorisierungsanfragen. Diese Maßnahmen dienen nicht nur der Erhöhung der Sicherheit, sondern garantieren auch eine optimale Benutzererfahrung. Die Implementierung von OpenID Connect stellt eine zuverlässige Lösung für Authentifizierungsszenarien in nativen Apps dar.

Sven-Torben Janus

sven-torben.janus@conciso.de

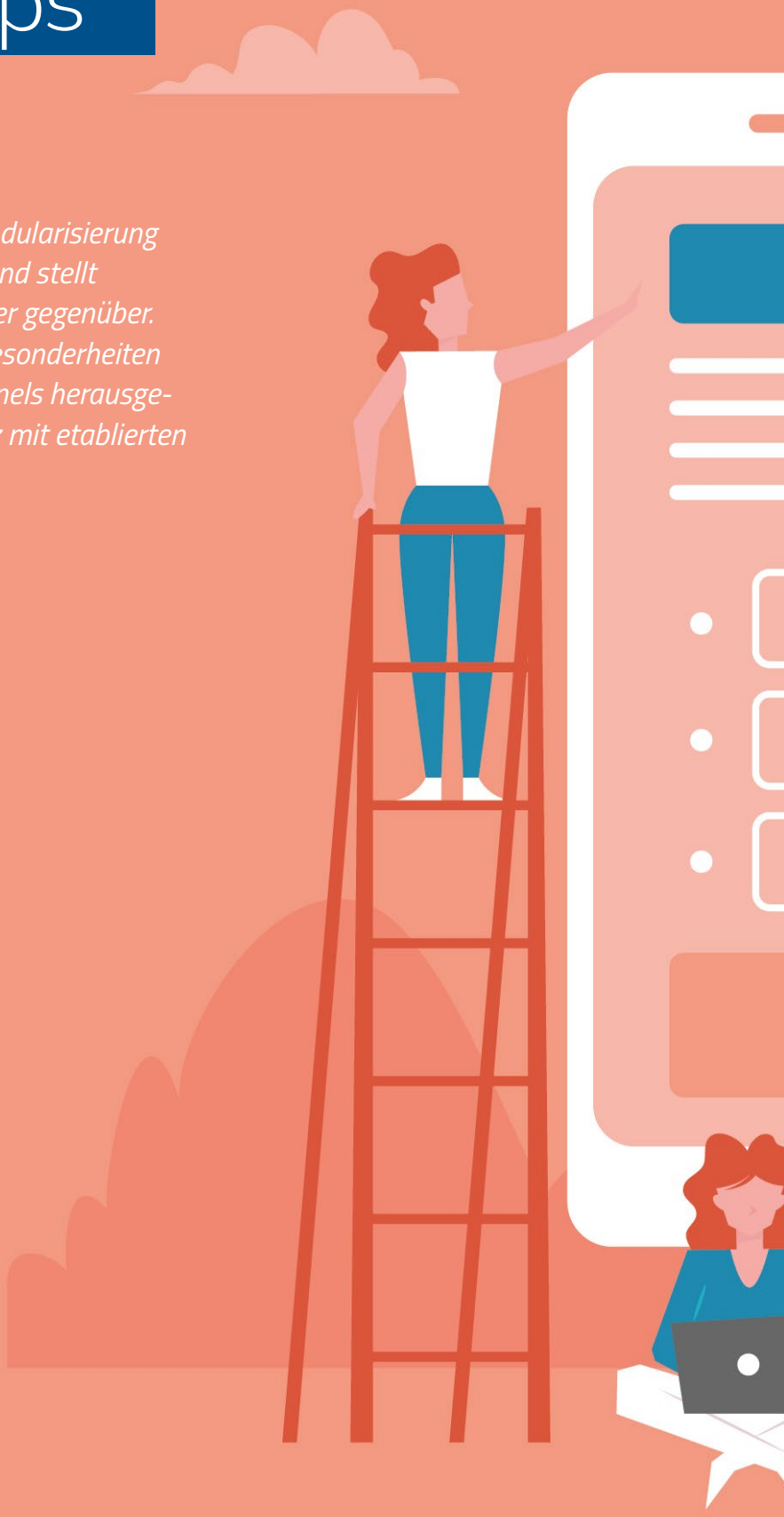
https://twitter.com/sventorben

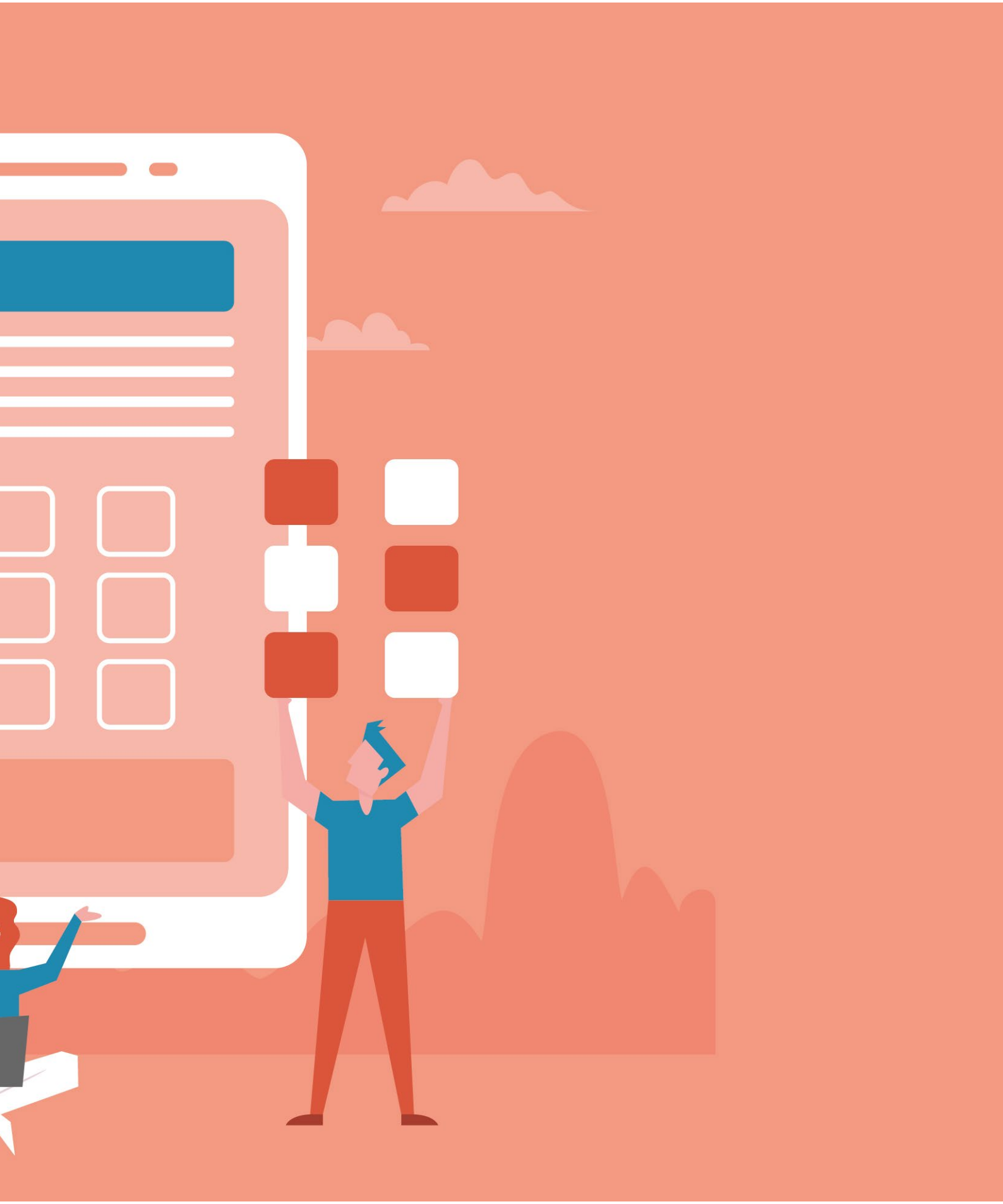
Sven-Torben Janus ist Partner bei der Conciso GmbH, wo er den Bereich Softwarearchitektur mit verantwortet. Er arbeitet als praktizierender Principal Software Architect und befürwortet einen agilen und praktikablen Entwurf von Softwarearchitekturen. Ein Schwerpunkt seiner Berater-Tätigkeit liegt im Bereich Identity & Access Management (IAM) mit besonderem Fokus auf die Open-Source-Lösung Keycloak.

Modularisierung in Android-Apps

Jonathan Merkel, Exxeta AG

Dieser Artikel beschreibt, wie Ansätze der Modularisierung in Android-Apps umgesetzt werden können und stellt anschließend zwei konkrete Architekturmuster gegenüber. Anhand von Codeausschnitten werden die Besonderheiten des modularen Monolithen und des Microkernels herausgestellt und Hinweise für den jeweiligen Einsatz mit etablierten Android-Frameworks (Hilt, Dagger) gegeben.





Modularisierung spielt eine entscheidende Rolle in der Softwareentwicklung. Dabei wird die Software als Ganzes anhand einzelner funktionsbringender Bausteine, beschrieben. Die Vorteile sind verbesserte Wartbarkeit, erhöhte Wiederverwendbarkeit von Softwarebausteinen und die vereinfachte Arbeitsaufteilung *[1]*. Während im Bereich der Backendentwicklung die Microservicearchitektur zur Modularisierung sehr oft eingesetzt wird, ist diese nicht vollständig in der App-Entwicklung adaptierbar. Jedoch existieren verschiedene Architekturansätze, die dafür in Betracht gezogen werden können. Zwei mögliche Architekturen sind der modulare Monolith und der Microkernel-Ansatz. Der modulare Monolith integriert statisch einzelne Komponenten, während der Microkernel es ermöglicht, Komponenten erst zur Laufzeit einzubinden. Wie diese Architekturen in einer Android-App umgesetzt werden können, wird im Folgenden näher beschrieben.

Die Grundlage beider Architekturen ist, dass sie aus einer Kernanwendung bestehen und die Funktionalität durch Module erweitert wird. Das Einbinden dieser Module soll möglichst einfach gestaltet sein. Zudem sollen die Module „self-contained" sein, was bedeutet, dass alle benötigten Ressourcen und Abhängigkeiten in diesem Modul enthalten sind, um eigenständig zu funktionieren. Zudem bestehen die Module aus zwei Submodulen, die für die Schnittstellen und für deren entsprechende Implementierung zuständig sind. Zusammenfassend sind dies die zugrundeliegenden Gemeinsamkeiten beider Architekturen. Auf die Unterschiede und Besonderheiten wird dediziert im Folgenden eingegangen.

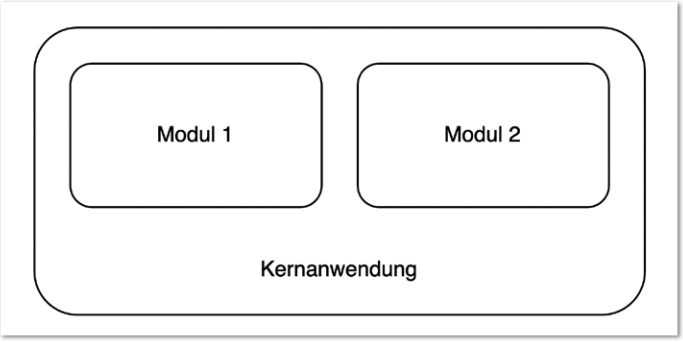


Abbildung 1: Übersicht modularer Monolith

Der modulare Monolith

Der modulare Monolith zeichnet sich durch eine Kernanwendung aus, die Module statisch zur Compile-Zeit integriert. Es gibt verschiedene Arten der Abhängigkeiten zwischen den Modulen, die unterschiedlich realisiert werden. Hier wird zwischen einer bidirektionalen und einer unidirektionalen Abhängigkeit unterschieden. Da die Module die Implementierung der anderen Module kennen, werden die Abhängigkeiten über die Schnittstellenmodule abstrahiert. Um ein Modul mithilfe von *Dependency Injection* in einem anderen Modul nutzen zu können, muss zunächst dessen Schnittstelle definiert werden. *Listing 1* zeigt, wie ein Modul mithilfe von Hilt in einem modularen Monolithen definiert wird. Beim Bauen der Kernanwendung werden alle durch die `@Module`-Annotation gekennzeichneten Hilt-Module aufgelöst und an die Module, die es durch die `@Inject`-Annotation in ihrem Konstruktor erwarten, übergeben.

Die Integration in einem anderen Modul erfolgt normalerweise über *Constructor Injection*, wie in *Listing 2* dargestellt wird. Besteht allerdings eine bidirektionale Abhängigkeit, so wird diese über *Field Injection* aufgelöst.

```
@Module
@InstallIn(SingletonComponent::class)
interface Module {
  @Binds
  @Singleton
  fun bindService(serviceImpl: ServiceImpl): Service
}
```

Listing 1: Definition in einem Modul des modularen Monolithen

```
class OtherModuleService @Inject constructor(
  private val service: Service
) {}
```

Listing 2: Integration in einem anderen Modul

Zudem ist im modularen Monolith die Hierarchie klar definiert, da die Kernanwendung entscheidet, welche Module benötigt werden und die Einbindung dieser verantwortet. Die einzelnen Module kennen weder die Kernanwendung noch die Implementierungsdetails anderer Module – lediglich deren Schnittstellen.

Der modulare Monolith bietet somit eine gute Lösung für Apps, für die statische Abhängigkeiten kein Problem sind. Sollen die Funktionalitäten, beziehungsweise Abhängigkeiten, dynamisch und erst zur Laufzeit verfügbar gemacht werden, so eignet sich eine Microkernel-Architektur.

Abbildung 2: Übersicht Microkernel

Der Microkernel

Im Gegensatz zum modularen Monolith kompiliert der Microkernel die einzelnen Module nicht direkt in die Anwendung, sondern lädt diese nur bei Bedarf während der Laufzeit herunter. Eine Umsetzungsmöglichkeit dafür bietet Android mit den sogenannten *Dynamic Feature Modules [2]*. Sie lassen sich bei Bedarf herunterladen und reduzieren somit die Größe der App, da nur die benötigten Module enthalten sind. Die Freiheit und Flexibilität ist jedoch eingeschränkt, da die Kernanwendung trotzdem die Module kennen muss, weshalb diese in der build.gradle-Datei deklariert werden *(siehe Listing 3)*. Die Schnittstellen der jeweiligen Module werden als Abhängigkeiten deklariert. Für jedes Modul wird eine eigene build.gradle-Datei erstellt.

Listing 4 zeigt eine exemplarische build.gradle-Datei eines Moduls. Hierfür ist zum einen das Plug-in anhand seiner ID und zum anderem die Abhängigkeit zur Kernanwendung zu ergänzen.

```
android {
…
setDynamicFeatures(mutableSetOf(“:module1”, :”module2”))
…
}

dependencies {
  implementation(“module1-interface”)
  implementation(“module2-interface”)
}
```

Listing 3: Ausschnitt aus der build.gradle-Datei der Kernanwendung im Microkernel

```
plugins {
  id(“com.android.dynamic-feature”)
  …
}

dependencies {
  implementation(project(“:app”))
  …
}
```

Listing 4: Ausschnitt aus der build.gradle-Datei eines Moduls im Microkernel

Durch die Verwendung der Dynamic Feature Modules verändern sich auch die Abhängigkeiten im Vergleich zum modularen Monolithen. Zusätzlich zu den Implementierungs- und Schnittstellenmodulen kommt im Modul ein *Daggermodul* hinzu und in der Kernanwendung ein Integrationsmodul. Letzteres bindet das Daggermodul ein, löst Abhängigkeiten zu anderen Modulen auf und macht die konkreten Implementierungen der Modulschnittstellen verfügbar. Daraus ergibt sich das Schaubild in *Abbildung 3*. Die Gründe und Auswirkungen auf die Implementierung dieser Architektur werden anschließend näher erläutert.

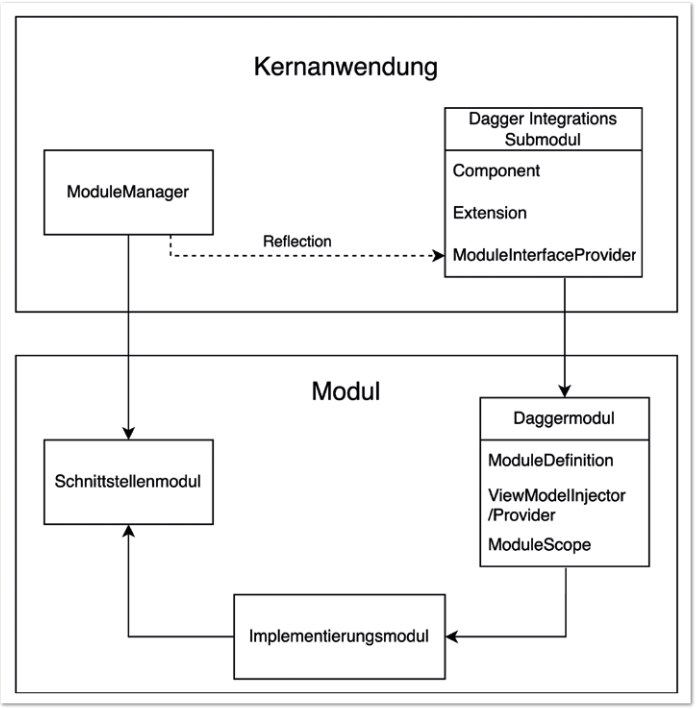


Abbildung 3: Modulübersicht Microkernel

Microkernel: Implementierungsaspekte

Hilt unterstützt keine Dynamic Feature Modules, weshalb die Abhängigkeiten manuell mit *Dagger* bereitgestellt werden müssen. Module besitzen dafür mindestens zwei Klassen, eine Moduldefinition *(siehe Listing 5)* und eine Component-Definition *(siehe Listing 6)*.

```
@Module
interface ModuleDefinition {
  @Binds
  @ModuleScope
  fun bindService(serviceImpl: ServiceImpl): Service

  companion object {
     @Provides
     @ModuleScope
     fun provideWithOtherModuleDep(otherModuleService:
        :OtherModuleService): ThisModuleService{
        return ThisModuleServiceImpl(otherModuleService)
     }
  }
}
```

Listing 5: Moduldefinition im Microkernel

```
@Component(
modules = [ModuleDefinition::class,
ModuleExtension::class],
  dependencies = [OtherModuleComponent::class]
)
interface ModuleComponent: ViewModelInjector {
  val exposedService: Service

  companion object {
     val instance: ModuleComponent by lazy {
        DaggerModuleComponent.builder()
           .otherModuleComponent(
OtherComponent.instance)
           .build()
     }
}
```

Listing 6: Component-Definition im Microkernel

Die Moduldefinition generiert die Implementierung zu den Schnittstellen, die von anderen Modulen genutzt werden. Sie liegt zudem innerhalb des eigenständigen Moduls. Kann eine Schnittstelle ohne weitere Abhängigkeiten angeboten werden, so reicht eine Deklaration mit `@Binds`, woraufhin Dagger alles andere automatisiert generiert. Bestehen Abhängigkeiten zu anderen Modulen, so müssen die Instanzen selbst erstellt werden und mit `@Provides` zur Verfügung gestellt werden. Alle Funktionen eines Moduls benötigen eine `Scope`-Annotation. Der Grund dafür ist, dass einige Module bereits zu Beginn heruntergeladen werden können, während andere erst bei Bedarf geladen werden. Es ist wichtig zu beachten, dass beide Arten von Modulen nicht denselben Scope nutzen dürfen und auch nicht den Singleton-Scope verwenden dürfen.

Die Component-Definition aus *Listing 6* liegt in der Kernanwendung. Diese sorgt schließlich dafür, dass Dagger die Dependency Injection ausführt. Dafür werden die Module und die Abhängigkeiten deklariert. Jede Komponente bietet außerdem eine Singleton-Instanz an, die von anderen Komponenten genutzt werden kann. Letztlich werden die Schnittstellen durch Variablen deklariert.

```
interface ViewModelInjector {
  fun inject(vm: CustomViewModel)
}

@HiltViewModel
class CustomViewModel @Inject constructor(): ViewModel {
  @Inject
  lateinit var service: Service
}

fun createViewModel(injector: ViewModelInjector): CustomViewModel {
  val vm = viewModel<CustomViewModel>()
  injector.inject
}
```

Listing 7: ViewModelInjector

Des Weiteren zeigt *Listing 6* die Implementierung des `ViewModelInjector`. Dies ist ein Interface mit einer oder mehreren Methoden, die ein *Hilt-ViewModel* akzeptieren, bei dem die Abhängigkeiten in Form von Membervariablen injected werden. So können die Vorteile von Dagger genutzt werden. Die konkreten Implementierungen des `ViewModelInjector` zeigt *Listing 7*.

Um zusätzliche Abhängigkeiten eines Modules bereitzustellen, wird ein `ModuleExtension`-Interface benutzt *(siehe Listing 8)*. Dieses Interface löst die bidirektionalen Abhängigkeiten auf und stellt den `ViewModelInjector` zur Verfügung.

Die Auflösung der bidirektionalen Abhängigkeiten erfolgt über den `ModuleManager`, der in der Kernanwendung liegt und in *Listing 9* dargestellt ist. Die einzelnen Module werden dabei über *Reflection* geladen. Deren Schnittstellen werden so für die anderen Module und die Kernanwendung verfügbar gemacht. Dadurch kann auch die Kernanwendung auf die Module zugreifen und Funktionalitäten ausführen.

```
@Module
interface ModuleExtension {
  companion object {
     @Provides
     @ModuleScope
     fun provideViewModelInjector(): ViewModelInjector {
        return ModuleComponent.instance
     }

     @Provides
     @ModuleScope
     fun provideCircularRefDependency (): Dependency {
        return ModuleManager.otherModule.dependency
     }
  }
}
```

Listing 8: ModuleExtension-Interface

```
//In der Kernanwendung
object ModuleManager {

  val otherModule: ModuleInterface by lazy {resolveModule()}

  fun resolveModule(): ModuleInterface {
return Class
.forName("package.ModuleInterfaceProvider")
.kotlin.objectInstance as T
  }
}

//Im Modulwrapper
object ModuleInterfaceProvider: ModuleInterface {
  override val service: Service
     get() = ModuleComponent.instance.exposedService

}
```

Listing 9: ModuleManager und Reflection

Somit sind nun alle Abhängigkeiten innerhalb eines Microkernels aufgelöst. Module können andere Module nutzen, selbst wenn eine bidirektionale Abhängigkeit besteht. Zudem kann die Kernanwendung auf die Module zugreifen, obwohl sie nur die Schnittstellen kennt.

Fazit

Der Artikel zeigt, dass sich Android-Apps mit beiden Architekturen umsetzen lassen, und gibt Einblicke in verschiedene Implementierungsaspekte.

Zusammengefasst, nutzt der modulare Monolith einen statischen Ansatz und kompiliert die einzelnen Module in die Gesamtanwendung. Dabei sind die Konzepte in anderen Apps bereits erprobt und die Frameworks, insbesondere Hilt, unterstützten diese Art der Modularisierung ohne zusätzlichen Aufwand.

Die Microkernel-Architektur bietet die Möglichkeit, Module erst zur Laufzeit herunterzuladen und kann somit eine kleinere und spezifischere App bereitstellen. Vor allem der Aufwand für die Bereitstellung wird reduziert, da es einfach über definierte Konfiguration, und nicht durch die Veränderung der Gradle-Dateien realisiert werden kann. Allerdings ist dafür der Integrationsaufwand neuer Module oder neuer bereitgestellter Schnittstellen höher, da eine Zwischenschicht eingebaut werden muss.

Für Apps mit verschiedenen Nutzergruppen und Features, kann die Microkernel-Architektur in Betracht gezogen werden. Diese Architektur erlaubt es Nutzer:innen, flexibel Features zu aktivieren, was dazu führt, dass Nutzer:innen von der reduzierten App-Größe profitieren. Für Anwendungen, die immer die gleichen Features besitzen, ist der modulare Monolith zu bevorzugen. Beide Architekturen unterstützen jedoch die parallele Entwicklung an mehreren Features, da die Module gut voneinander abgekapselt sind.

Quellen

[1] *https://little-things.de/assets/projects/studies/principles-of-modularization/Prinzipien%20der%20Modularisierung_mb.pdf*

[2] *https://developer.android.com/guide/playcore/feature-delivery*

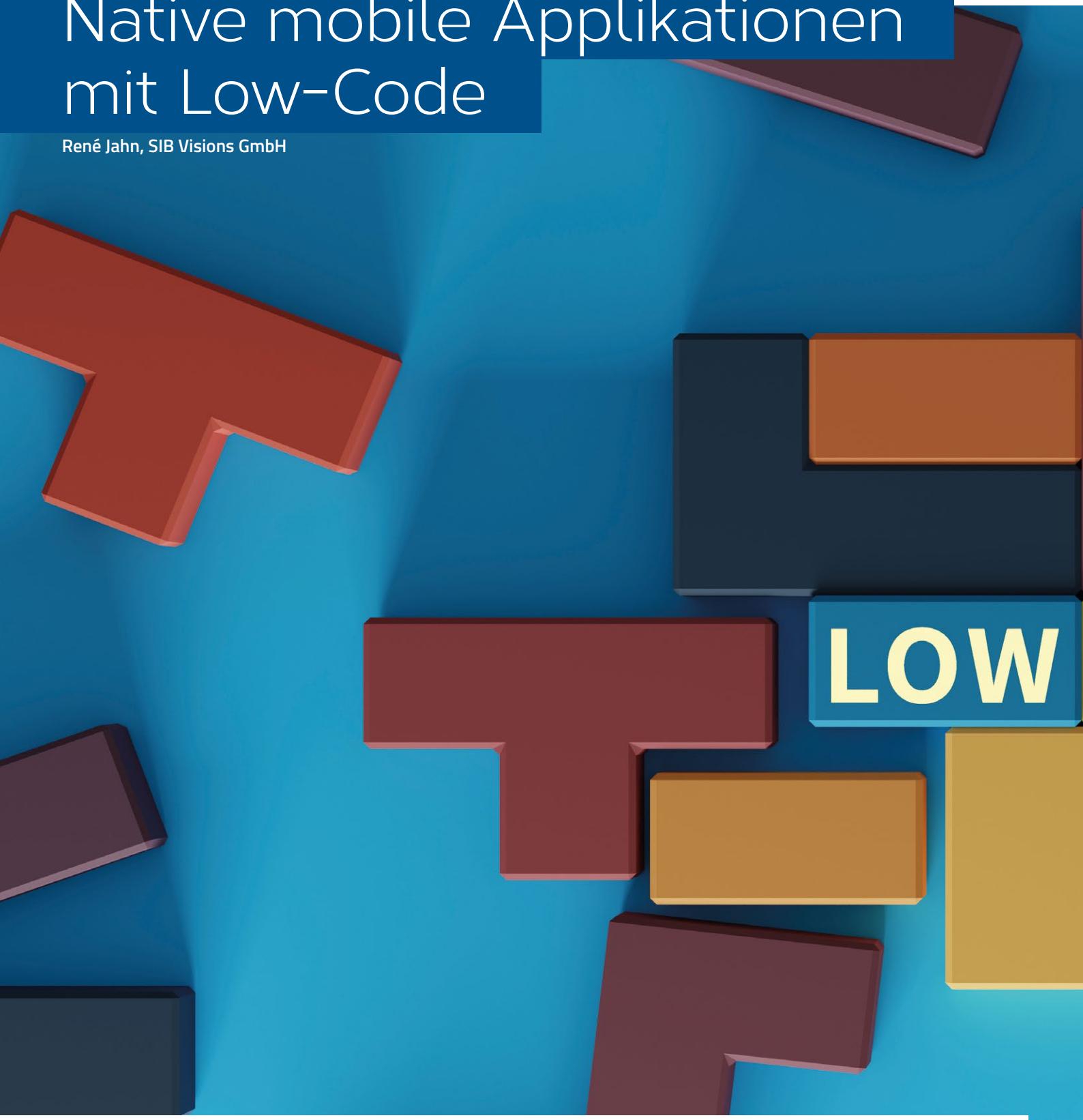
Jonathan Merkel

jonathan.merkel@exxeta.com

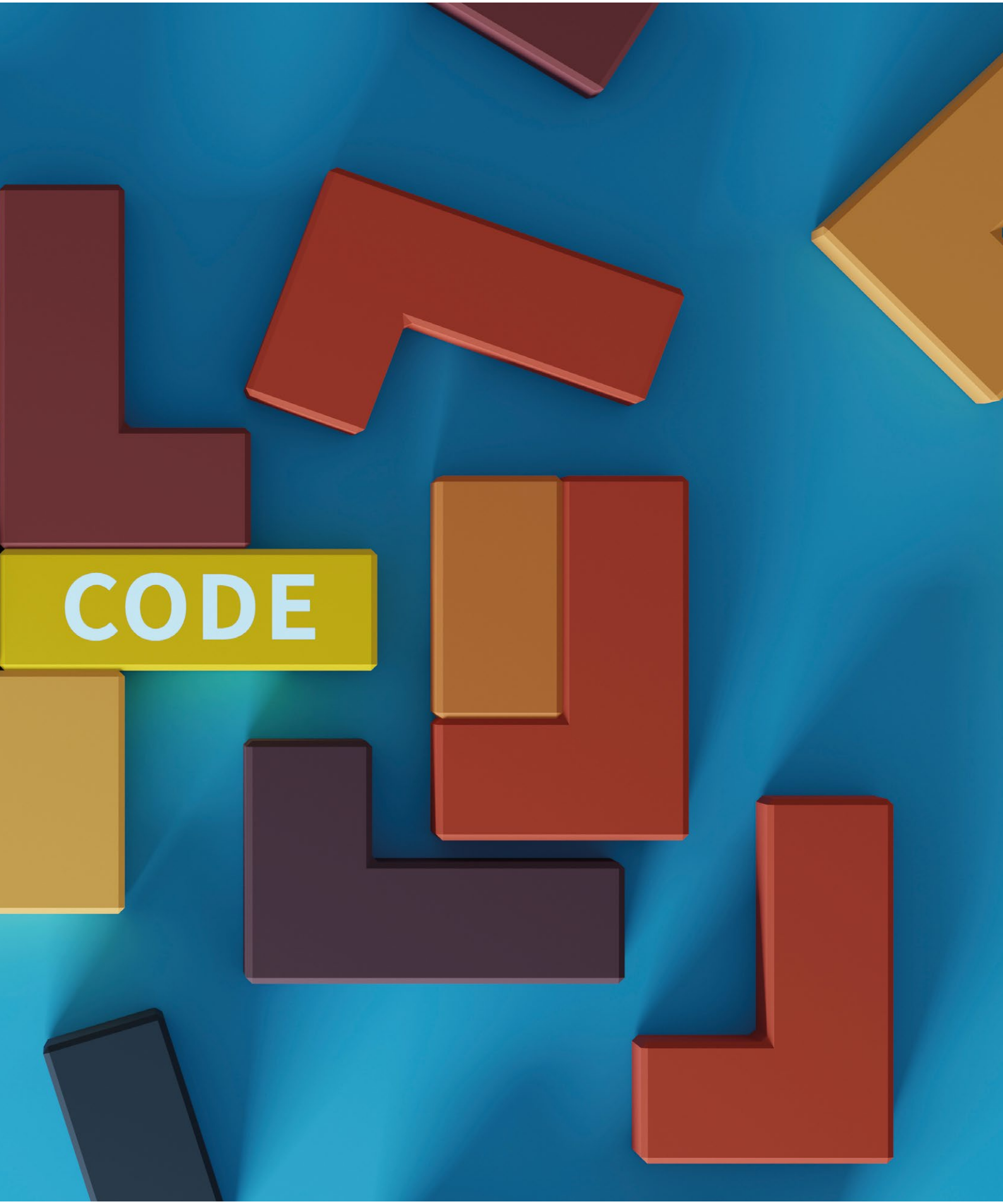
Jonathan Merkel absolvierte seinen Bachelor an der Hochschule Furtwangen, wo er das erste Mal mit der mobilen Entwicklung in Kontakt kam. Anschließend wechselte er für seinen Master an die Hochschule Karlsruhe, an der er sein Wissen unter anderem hinsichtlich Softwarearchitektur vertiefte. Heute ist er Softwareentwickler bei Exxeta AG. Seine Schwerpunkte liegen in der Backend- und Mobile-Entwicklung mit Java und Kotlin.

Native mobile Applikationen mit Low-Code

René Jahn, SIB Visions GmbH



Der Begriff Low-Code ist mittlerweile für effiziente Applikationsentwicklung mit minimaler Codierung etabliert. Ob fürs Web oder Mobile, alles ist mit Low-Code ganz einfach möglich. Wenn die Anforderungen nicht extrem ausgefallen sind, dann kann mit gebotenen Standard-Mitteln eine Applikation auch ohne Expertise realisiert werden. Doch wenn Standard nicht reicht, dann wird es ganz schnell ungemütlich. Denn Low-Code bedeutet nicht automatisch „Mach wie und was du willst". In diesem Artikel möchte der Autor eine Lösung vorstellen, die aber genau das bietet.


CODE

In den letzten Monaten erlebte die Low-Code-Szene einen regelrechten Aufschwung. Dies haben namhafte Beratungsunternehmen, wie beispielsweise Gartner *[1]*, auch so vorausgesagt. Es gibt Low-Code-Tools und -Plattformen für nahezu alle Bereiche, die mit Softwareentwicklung zu tun haben – ob IoT (Internet of Things), Robotik, ERP (Enterprise Resource Planning) oder BPM (Business Process Management). Selbst für die Entwicklung von Games gibt es Low-Code-Lösungen. Es sieht schon fast so aus, als ob Low-Code die klassische Softwareentwicklung ablösen könnte, zumindest wenn man den diversen Beratungsunternehmen Glauben schenkt.

Es ist unumstritten, dass mit Low-Code sehr vieles ohne Programmierkenntnisse gelöst werden kann, wenn auch noch nicht alles. Für die sogenannten *Citizen Developer* *[2]* ergeben sich dadurch vollkommen neue Möglichkeiten, die ohne Low-Code nicht denkbar wären. Werden Low-Code-Tools allerdings von professionellen Softwareentwickler*innen eingesetzt, kommt es zu einer spannenden Situation: Einerseits wollen Softwareentwickler*innen gerne alles selbst codieren, doch andererseits leidet unter langweiligen Tätigkeiten die Motivation. Das widerspricht ein wenig dem Ansatz vieler Low-Code-Tools. Mit dem richtigen Low-Code-Tool hingegen, kann die Motivation hochgehalten werden, da Softwareentwickler*innen nur noch spannende Probleme lösen müssen. Die langweiligen Tätigkeiten übernimmt das Low-Code-Tool für sie.

So viel zur Theorie. Doch in der Praxis ist das alles andere als einfach, da zum einen das Angebot an Low-Code-Tools und -Plattformen mittlerweile unüberschaubar groß geworden ist und zum anderen die Softwareentwickler*innen skeptisch auf Low-Code reagieren. Diese Skepsis ist auch durchaus berechtigt, da Kreativität und Freiheit für Softwareentwickler*innen sehr wichtig sind.

Die Auswahl des richtigen Low-Code-Tools ist entscheidend für die Akzeptanz und den erfolgreichen Einsatz. Es gibt neben zahlreichen proprietären Lösungen auch eine steigende Anzahl an Open-Source-Lösungen, die je nach Einsatzgebiet wunderbar geeignet sind, um Applikationen zu erstellen. Doch generell decken die meisten Low-Code-Tools immer nur spezielle Anwendungsfälle, mehr oder weniger gut, ab. Es gibt beispielsweise Low-Code-Tools mit denen mobile Applikationen erstellt werden können, die dann aber nur auf dem Smartphone oder Tablet funktionieren und nicht im Browser. Dann gibt es Lösungen, mit denen beides möglich ist, aber mit dem Nachteil, dass bei der Applikationserstellung schon der richtige Kanal gewählt werden muss und somit mehrere Lösungen für unterschiedliche Kanäle entstehen. Bei der Wahl der eingesetzten Technologien und Frameworks entsteht automatisch eine Bindung zum Hersteller und es gibt wenig Spielraum. Ob es Zugriff auf den Sourcecode gibt, ist von Hersteller zu Hersteller unterschiedlich und keine Selbstverständlichkeit. Das wiederum ist für Softwareentwickler*innen oft ein Ausschlusskriterium. Denn ohne Sourcecode-Zugriff finden diese keinen Spaß daran, ein Low-Code-Tool einzusetzen, auch wenn es durchaus hilfreich wäre.

Des Weiteren wird es zunehmend schwieriger, den Low-Code-Dschungel zu durchblicken, um aufs richtige Pferd zu setzen, denn die Anzahl der Low-Code-Tools wächst fast täglich. Das bedeutet, dass der Evaluierungs- und Entscheidungsprozess zur Auswahl des passenden Low-Code Tools ernst genommen werden sollte. Es kann nämlich durchaus sinnvoll sein, mehr als ein Low-Code-Tool einzusetzen, um alle Anforderungen abzudecken. Die Tools sollten im besten Fall miteinander harmonieren.

Um bei der Auswahl ein wenig behilflich zu sein, wird im Folgenden eine Low-Code-Plattform vorgestellt, die sowohl für Citizen Developer als auch für professionelle Softwareentwickler*innen konzipiert wurde. Durch die Offenheit der Plattform ist für alle etwas dabei. Die Softwareentwickler*innen haben vollen Zugriff auf den Sourcecode und dennoch alle Vorteile einer Low-Code-Plattform, um schnell langweilige Tätigkeiten abzuschließen. Wie damit im Alltag gearbeitet werden kann, wird anhand eines konkreten Projekts gezeigt.

Das Projekt wird einfach gehalten: Es handelt sich um die Verwaltung von Rechnungsbelegen, die bei Einkäufen anfallen. Die Erfassung der Belege soll mittels Smartphone erfolgen und nur wenige Sekunden in Anspruch nehmen.

Beim Lesen der Anforderung wird man als Softwareentwickler*in nicht begeistert sein, denn eine Verwaltungsapplikation haut niemanden mehr vom Hocker und es ist beim Lesen schon klar, was zu tun ist. Wo ist denn bitte die Herausforderung? Für Softwareentwickler*innen gibt es deswegen eine Zusatzaufgabe, um die Motivation anzukurbeln: Das User-Interface soll geändert werden, um modern und nicht standardisiert zu wirken.

Für Citizen Developer oder gar Laien ist die Situation eine andere. Denn hier sind die Herausforderungen naheliegend und die Hürden hoch. Eine mobile Applikation zu entwickeln und Bilder zu speichern. Das wäre ohne Low-Code-Tool undenkbar. Selbst wenn die App erstellt werden könnte, wie kommt diese dann auf die Endgeräte? Das sind alles andere als triviale Aufgaben.

Wir werden also eine Applikation entwickeln – mithilfe von Standardmitteln – die unsere Anforderungen komplett erfüllt. Das können mit Low-Code eigentlich alle. Dann kommen die Softwareentwickler*innen ins Spiel und machen daraus etwas Schickes. Es bleibt immer die glei-

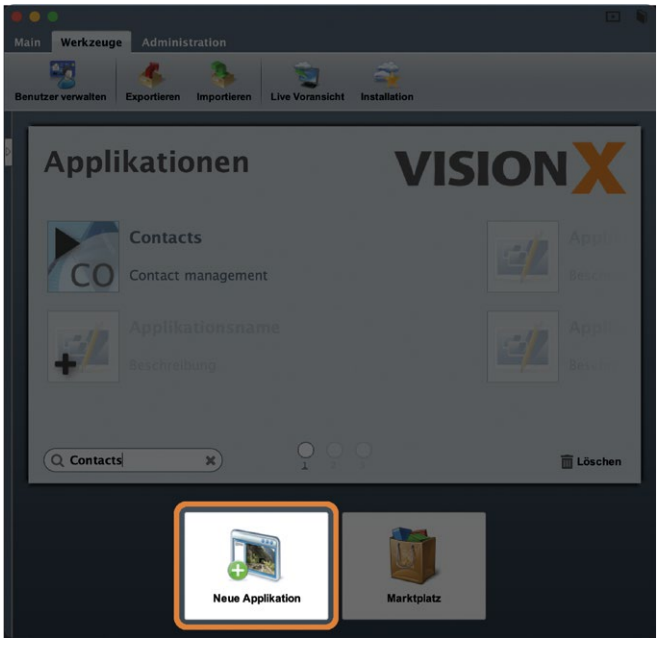


Abbildung 1: Neue Applikation erstellen

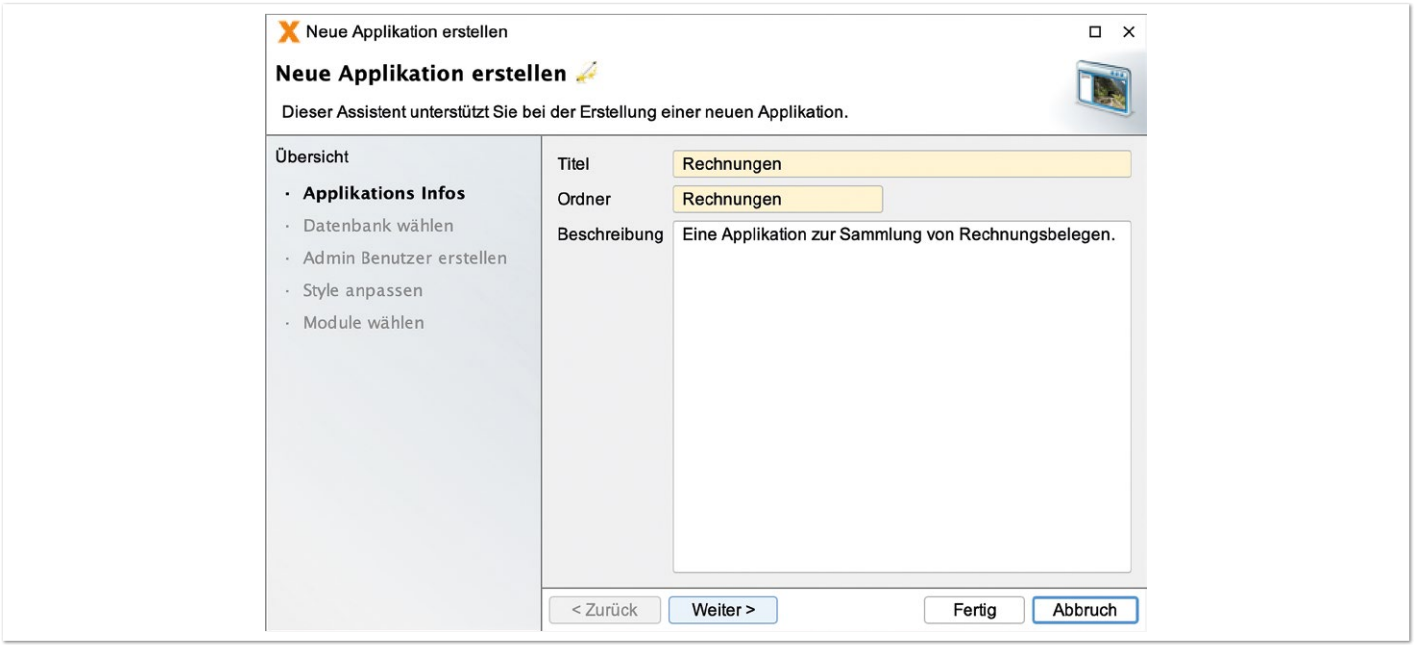


Abbildung 2: Rechnungsapplikation erstellen

che Applikation mit identen Features, doch das Aussehen wird angepasst. Die Optik ist oft wichtiger als Funktionalität.

VisionX

Im Folgenden wird die Low-Code-Plattform *VisionX [3]* verwendet, um eine Applikation zu entwickeln, die den Anforderungen entspricht und die Weiterentwicklung mittels Sourcecode ermöglicht. Die Plattform wurde gewählt, weil der Autor diese mitentwickelt und weil er selbst ein Softwareentwickler ist, der lieber spannende Aufgaben löst, anstatt langweilige Alltagsarbeit zu verrichten.

VisionX ist eine in Java entwickelte Lösung, die vollständig auf Open Source basiert. Die Plattform selbst ist kostenpflichtig, doch die damit erstellten Applikationen enthalten lediglich Open-Source-Bibliotheken und können auch ohne VisionX weiterentwickelt werden. Die in diesem Artikel erstellte Applikation wurde mit der Testversion von VisionX erstellt.

Die Umsetzung

Die Applikation kann mit wenigen Klicks erstellt werden. In VisionX gibt es dafür einen Assistenten *(siehe Abbildung 1)*. Im Assistenten wird lediglich der Applikationsname (Rechnungen) eingegeben und mit „Fertig" bestätigt *(siehe Abbildung 2)*.

Gleich nach der Erstellung wird die Applikation gestartet und wir erstellen noch eine Bearbeitungsmaske für die Erfassung der Belege. Der Assistent für die Erstellung einer Bearbeitungsmaske wird automatisch angezeigt und nach Eingabe eines Titels *(siehe Abbildung 3)*, erfolgt die Festlegung der Datenfelder. Wir benötigen ein Erfas-

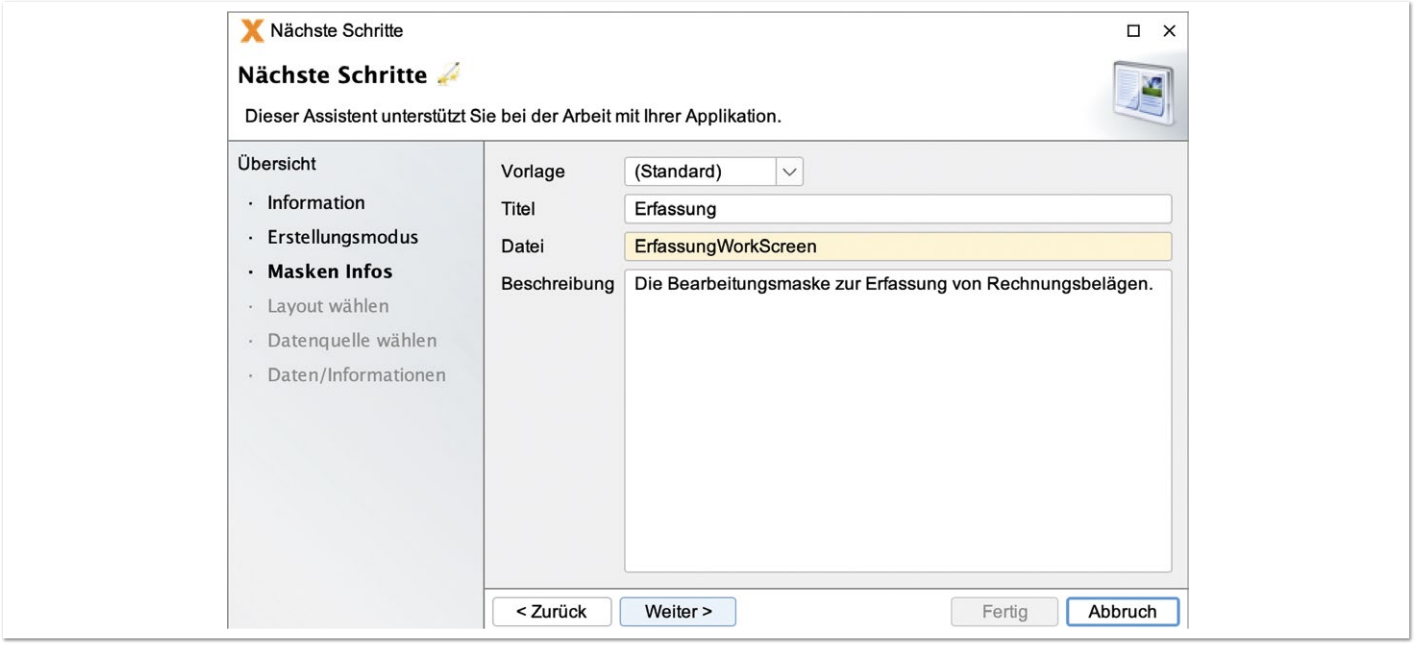


Abbildung 3: Bearbeitungsmaske erstellen, Titel

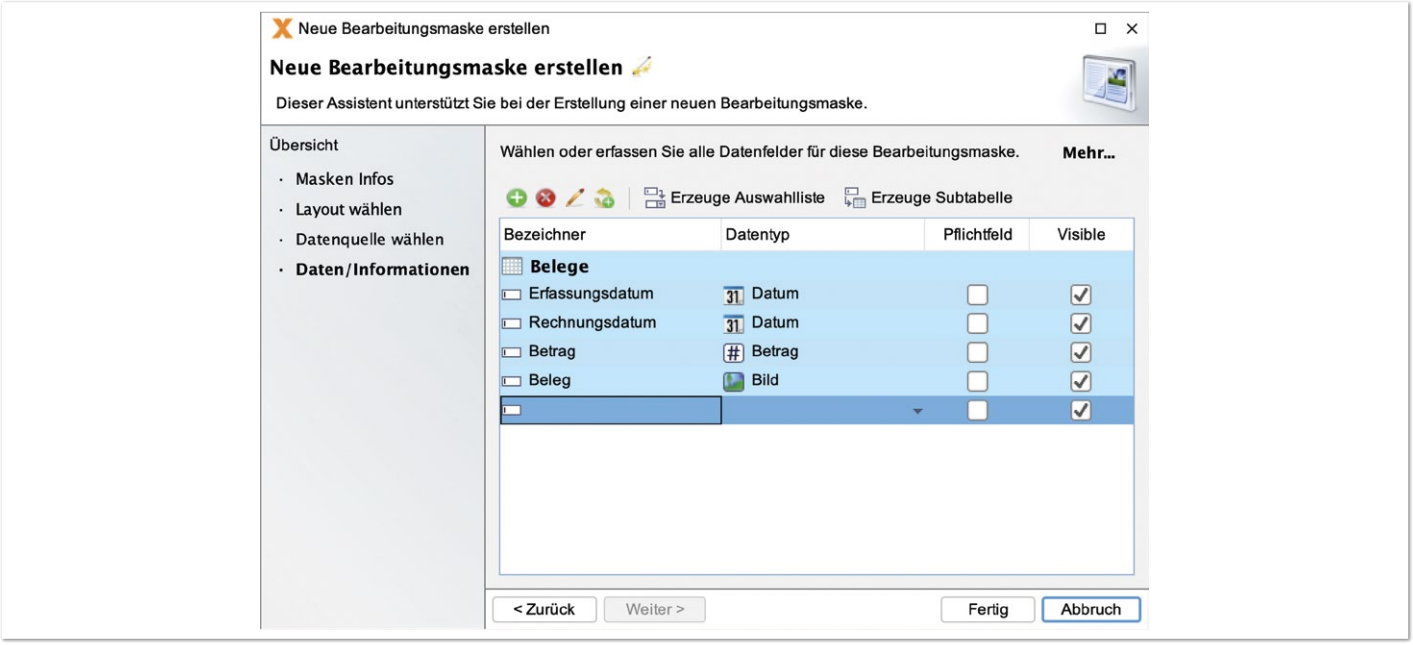


Abbildung 4: Bearbeitungsmaske erstellen, Datenfelder

sungsdatum, das Rechnungsdatum, den Rechnungsbetrag und ein Bild für den Beleg selbst *(siehe Abbildung 4)*.

Nachdem die Bearbeitungsmaske erstellt wurde, sind wir auch schon fertig mit der Applikation. Um diese nun auf einem Smartphone anzuzeigen, benötigen wir eine App. Die Standard-App für die Anzeige von VisionX-Applikationen hat den Namen *VisionX Previewer* und ist kostenlos sowohl im Google PlayStore als auch im Apple App Store verfügbar. Mit dieser App können die mit VisionX erstellten Applikationen auf dem Smartphone benutzt werden. Diese App wurde mit *Flutter [4]* entwickelt und verwendet die Open-Source-App *Flutter JVx Client [5]*, die wir auch verwenden werden, um unsere Applikation anzupassen.

Die Darstellung einer mit VisionX erstellten Applikation ist keine Hexerei. Die Applikation selbst läuft ohne User-Interface auf einem Applikationsserver, wie beispielsweise *Apache Tomcat [6]*, und wird am mobilen Gerät mittels REST synchronisiert und dargestellt. Die App ist eine native Flutter-Applikation und kein Container wie beispielsweise *Cordova [7]*. Dadurch wird es sehr einfach möglich, alle Features von Flutter zu nutzen, um die Applikation zu erweitern.

Nachdem die Applikation im Previewer gestartet wurde, kann die Erfassungsmaske sofort verwendet werden, um Belege zu erfassen *(siehe Abbildung 5 und 6)*.

An dieser Stelle wäre ein Citizen Developer bereits fertig und könnte produktiv arbeiten. Der Aufwand für die Erstellung war mit zwei bis drei Minuten auch überschaubar. Auch ein*e Softwareentwickler*in könnte im Grunde mit dem Ergebnis zufrieden sein, aber klar… das Formular entspricht nicht ganz unseren Vorstellungen. Wir hätten gerne etwas Anderes, Einzigartiges, um uns von der Masse abzuheben.

Wir brauchen eine eigene App mit einer angepassten Eingabemöglichkeit! Eine spannende Variante wäre eine Art *Carousel [8]* oder *Slider [9]*. Dann machen wir das doch einfach.

App anpassen

Wie bereits erwähnt, verwendet der *VisionX Previewer* die Open-Source-App *Flutter JVx Client [5]*, die wir frei verwenden können, um unsere Anforderungen umzusetzen. Das Projekt ist auf GitHub gehostet und ist mit der Apache-Lizenz 2.0 versehen. Das bedeutet, wir können ohne Bedenken damit arbeiten.

Als Voraussetzung für die Anpassung der App muss die Flutter-Entwicklungsumgebung *[10]* bereits installiert sein. Kurz zusammengefasst, müssen das *Flutter SDK* und eine IDE wie *Android Studio* oder *Visual Studio Code* verwendet werden. In diesem Artikel verwenden wir Android Studio. Wenn das abgehakt ist, kann mit der Anpassung begonnen werden.

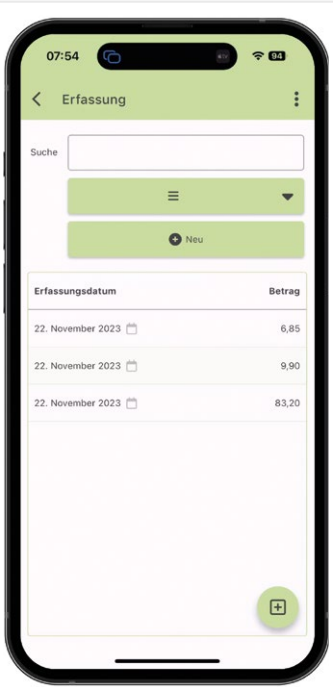


Abbildung 5: Rechnungsbelege erfassen, Liste

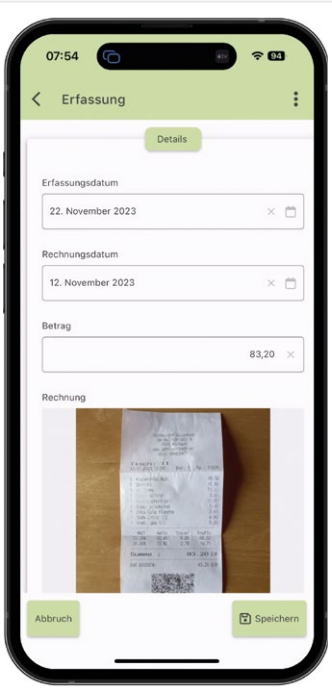


Abbildung 6: Rechnungsbelege erfassen, Detail

Wir starten mit der Erstellung eines neuen Flutter-Projekts und zwar mit folgendem Befehl: `flutter create rechnung --platforms android,ios,web`.

Dadurch wird ein neues Verzeichnis mit dem Namen `rechnung` erstellt, in dem ein ebenfalls neues Flutter-Projekt erzeugt wird. Dieses Projekt kann mit Android Studio geöffnet werden. Nach dem Öffnen wird üblicherweise die Applikationsklasse (`main.dart`) angezeigt. Das ist praktisch, denn in dieser Datei müssen wir Änderungen durchführen. Doch zuerst fügen wir eine neue Abhängigkeit zum Projekt hinzu. Das machen wir in der Datei `pubspec.yaml`. Im Bereich der `dependencies` wird Folgendes eingefügt:

```
flutter_jvx:
 git:
  url: https://github.com/sibvisions/flutter_jvx.git
```

Nun müssen wir unser Projekt aktualisieren und führen den Befehl `flutter pub get` aus. Damit werden zugleich alle Abhängigkeiten aufgelöst und aktualisiert. Unsere Applikation verwendet nun die Open-Source-App.

Im nächsten Schritt müssen wir die Applikationsklasse (`main.dart`) anpassen, um unsere Applikation einzubinden. Die App ist ja nur die Darstellung unserer Applikation, die eigentlich am Applikationsserver läuft, und nicht direkt am mobilen Gerät. Um die Konfiguration durchzuführen, ersetzen wir die Klasse mit dem Inhalt aus *Listing 1*. Hierbei ist wichtig, dass die `baseUrl` korrekt mit der IP oder dem Hostnamen des Rechners befüllt wird, wo auch VisionX läuft.

Wenn wir nun die Applikation starten, läuft alles wie mit dem VisionX Previewer nur eben als unsere eigene native App. Der erste Schritt ist gemacht.

Im nächsten Schritt beginnen wir mit der Umsetzung unserer Änderungen. Wir werden hierzu auch bestehende Bibliotheken als Abhängigkeiten einbinden. Die Low-Code-Plattform haben wir an dieser Stelle bereits zur Seite geschoben und wir entwickeln an der App direkt mit Flutter weiter. Was vom Low-Code bleibt, ist die Möglichkeit, alles Gebotene zu nutzen, wie zum Beispiel die Datenzugriff-APIs. Wir können auch weiterhin Bearbeitungsmasken mit VisionX erstellen und ohne Änderungen in unserer App nutzen – quasi ein Mischmodus. Es ist nicht immer nötig, alle Bearbeitungsmasken zu ändern. Dadurch sind die Vorteile beider Welten vereint: einerseits bei Low-Code zu bleiben oder andererseits komplett auf Code-Ebene zu wechseln.

Damit wir nun die Bearbeitungsmaske ändern können, muss ein App-Manager erstellt und der App bekanntgeben werden. Der App-Manager ist die zentrale Stelle, um Anpassungen durchzuführen. Damit wird das Verhalten der App gesteuert. Wir erstellen eine neue Datei (`custom_app_manager.dart`) im `lib`-Verzeichnis und befüllen den Inhalt mit *Listing 2*.

```
import 'package:flutter_jvx/flutter_jvx.dart';

void main() {
  FlutterUI.start(
      FlutterUI(
          appConfig: AppConfig(
              title: "Rechnungen",
              serverConfigs: [
                  PredefinedServerConfig(
                      baseUrl: Uri.parse("http://10.0.0.1/services/mobile"),
                      appName: "Rechnungen",
                      username: "admin",
                      password: "admin",
                  ),
              ],
          ),
      ),
  );
}
```

Listing 1: Neue Applikationsklasse

```
Import 'package:carousel_app/carousel_screen.dart';
import 'package:flutter_jvx/flutter_jvx.dart';

class CustomAppManager extends AppManager {
  CustomAppManager() {
      registerScreen(
          CustomScreen.online(
              key: CarouselScreen.SCREEN_KEY,
              screenBuilder: (buildContext, originalScreen) => const CarouselScreen(),
          ),
      );
  }
}
```

Listing 2: App-Manager

Der neue App-Manager wird vom ursprünglichen App-Manager abgeleitet und die einzige Änderung ist, dass die Bearbeitungsmaske mit einer eigenen Implementierung ersetzt wird. Die Bearbeitungsmaske wird mit dem `CarouselScreen.SCREEN_KEY` identifiziert und die eigene Implementierung ist `CarouselScreen()`.

Jede Bearbeitungsmaske hat einen eindeutigen Schlüssel, der für Anpassungen verwendet werden kann. In unserem Beispiel wäre das:

```
app.visionx.apps.rechnungen.screens.
ErfassungWorkScreen:L1_MI_DOOPENWORKSCREEN_
APP-VIS-APP-REC-SCR-ERFWORSCR.
```

Dies benötigt ein wenig Wissen über die Anpassungsmöglichkeiten und die bereitgestellten APIs. Doch diesbezüglich gibt es im Open-Source-Projekt *[5]* eine ausreichende Dokumentation *[11]* und auch eine Beispielapplikation *[12]*. Eine ausführliche Erklärung würde den Rahmen dieses Artikels sprengen und deswegen wird hier nicht zu sehr ins Detail gegangen.

Da wir bereits alles für eine eigene Bearbeitungsmaske vorbereitet haben, erstellen wir diese nun auch. Das Ergebnis soll so aussehen wie in *Abbildung 7* gezeigt. Die Bearbeitungsmaske wird von Grund auf neu entwickelt, mit Flutter-üblicher Codierung. Es gibt hier auch keinerlei Limitierungen für Softwareentwickler*innen.

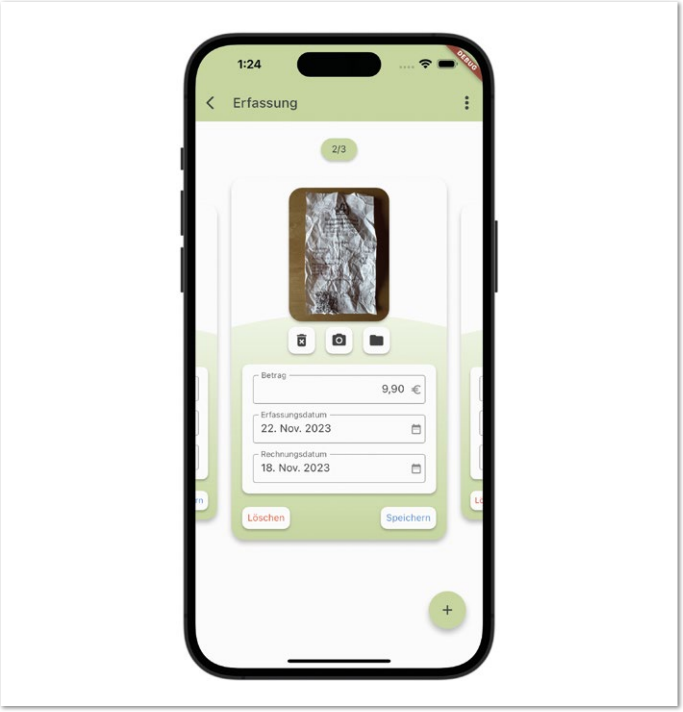


Abbildung 7: Carousel

In der Maske soll die Navigation mittels Carousel *[8]* erfolgen und es soll möglich sein, die Belege mittels Kamerazugriff oder als Datei hinzuzufügen und zu löschen. Es sollen neue Belege erfasst und vorhandene gelöscht oder geändert werden können. Die Anforderungen sind genau die Gleichen, wie bereits in der vorhandenen Bearbeitungsmaske, aber mit komplett anderer Darstellung. Es wird keine Tabelle mehr angezeigt und auch die Drill-Down-Navigation ist nicht mehr nötig. In Summe wirkt der neue Screen moderner. Als Softwareentwickler*in muss man sich nicht um die Persistierung der Daten kümmern und auch nicht um den Datenzugriff. Das wird alles von der Low-Code-Plattform in Form von APIs bereitgestellt.

Damit wir die eigene Bearbeitungsmaske erstellen können, benötigen wir eine neue Datei (`carousel_screen.dart`) im `lib`-Verzeichnis. Der Inhalt ist ein wenig mehr Sourcecode und wird in *Listing 3* auf das Wesentlichste reduziert. Der vollständige Source Code *[13]* ist im GitHub-Repository zu finden.

Die neue Bearbeitungsmaske ist ein sogenanntes `StatefulWidget` *[14]* und hat einen eigenen State (`CarouselScreenState`). Die entscheidende Methode darin ist `build`. Denn damit wird das User-Interface zusammengebaut. Dort wird auch der `CarouselSlider` eingesetzt, der als Abhängigkeit im Projekt konfiguriert werden muss. Die Abhängigkeiten werden im `pubspec.yaml` verwaltet und wir fügen den Eintrag `carousel_slider: ^4.2.1` zu den `dependencies` hinzu. Das war's dann auch schon mit unseren Anpassungen. Die App ist nun vollständig angepasst und einsatzbereit.

Damit die App nun auch auf die Geräte der Endanwender gelangt, kann das Flutter-übliche Deployment *[15]* durchgeführt werden. Hier ist keine Abhängigkeit zur Low-Code-Plattform gegeben.

Das Deployment der App entfällt, wenn der VisionX Previewer eingesetzt wird. Dieser wurde bereits vom Hersteller in den Stores bereitgestellt und kann sofort ohne manuelle Anpassungen verwendet werden. Das erleichtert den Einstieg natürlich.

Als Abschluss zu den Anpassungen sei noch zu erwähnen, dass hier lediglich das User-Interface geändert wurde. Es wäre auch möglich, zusätzliche Funktionalitäten umzusetzen, wie beispielsweise den Rechnungsbetrag automatisch mittels Bilderkennung oder KI aus der Rechnung zu ermitteln. Doch was in die App eingebaut werden soll und was nicht, das bleibt den Softwareentwickler*innen überlassen.

Fazit

Der Einsatz von Low-Code in der App-Entwicklung ist keinesfalls eine Einbahnstraße. Wenn die Low-Code-Plattform offen ist und keine Limitierungen für die Softwareentwickler*innen enthält, dann kann das zu enormer Produktivitätssteigerung führen. Der Autor ist sich sicher, dass die gezeigte Lösung sich auch positiv auf die Motivation auswirkt. Wenn die einfachen Arbeiten keine Überwindung mehr Kosten und dafür mehr Zeit in die Umsetzung von anspruchsvollen Aufgaben fließt, kann das nur positiv sein. Es bietet sich auch eine einzigartige Möglichkeit der Zusammenarbeit zwischen Fachabteilungen und Entwicklung. Die Fachabteilungen sind in der Lage, die Vorarbeit zu leisten und die Entwicklungsabteilung ergänzt und finalisiert. Denn eines ist klar: Das Know-how und die Erfahrung von Softwareentwickler*innen wird auch dann notwendig sein, wenn Low-Code-Plattformen vermehrt zum Einsatz kommen.

Quellen

[1] *https://www.gartner.com/en/newsroom/press-releases/2022-12-13-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-20-percent-in-2023*

```
class CarouselScreen extends StatefulWidget {

  const CarouselScreen({super.key});

  @override
  State<CarouselScreen> createState() => _CarouselScreenState();
}

class _CarouselScreenState extends State<CarouselScreen> {
  CarouselController carouselController = CarouselController();
  DataChunk dataChunk = DataChunk.empty();
  int? selectedIndex;

  @override
  void initState() {
     super.initState();

     IUiService().registerDataSubscription(
        pDataSubscription: DataSubscription(
           subbedObj: this,
           dataProvider: CarouselScreen.DATAPROVIDER_KEY,
           onDataChunk: receiveDataChunk,
           onSelectedRecord: receiveSelectedRecord,
           from: 0,
        ),
     );
  }

  @override
  Widget build(BuildContext context) {
     return Stack(
        children: [
           Positioned.fill(
              child: DecoratedBox(
                 decoration: BoxDecoration(
                    color: Theme.of(context).colorScheme.background,
                 ),
                 child: LayoutBuilder(
                    builder: (context, constraints) {
                       return CarouselSlider.builder(
                          itemBuilder: _buildCard,
                          itemCount: dataChunk.data.length,
                          options: CarouselOptions(
                             height: double.infinity,
                             viewportFraction: 0.75,
                             enableInfiniteScroll: false,
                             enlargeCenterPage: true,
                             onPageChanged: (index, _) => sendSelectedRecord(index),
                             initialPage: 0,
                          ),
                          carouselController: carouselController,
                       );
                    },
                 ),
              ),
           ),
           createFloatingButton(context),
        ],
     );
  }
}
```

Listing 3: Neue Bearbeitungsmaske

[2] *https://de.wikipedia.org/wiki/Citizen_Developer*
[3] *https://visionx.sibvisions.com*
[4] *https://flutter.dev/*
[5] *https://github.com/sibvisions/flutter_jvx*
[6] *https://tomcat.apache.org/*
[7] *https://cordova.apache.org/*
[8] *https://pub.dev/packages/carousel_slider*
[9] *https://pub.dev/packages/card_slider*
[10] *https://docs.flutter.dev/get-started/install*
[11] *https://github.com/sibvisions/flutter_jvx/wiki*
[12] *https://github.com/sibvisions/flutter_jvx.example*
[13] *https://github.com/rjahn/visionx.rechnungen/blob/main/lib/carousel_screen.dart*
[14] *https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html*
[15] *https://docs.flutter.dev/deployment*

René Jahn

rene.jahn@sibvisions.com

René Jahn ist Mitbegründer der SIB Visions GmbH und Head of Research & Development. Er verfügt über langjährige Erfahrung im Bereich der Framework- und API-Entwicklung. Neben der Open-Source-Sparte bringt er seine Expertise auch in der Produktentwicklung ein. Seine Leidenschaft ist die Evaluierung neuer Technologien und Integration in Business-Applikationen.

Verteile Datensammlung von Zeitserien-Daten mit Apache TsFile und Apache IoTDB

Christofer Dutz, Timecho Europe GmbH

Künstliche Intelligenz und Machine Learning sind zurzeit in aller Munde. Vor allem im IoT-Bereich ist eine regelrechte Goldgräber-Stimmung ausgebrochen. Die hier vermutlich wichtigste Ingredienz ist definitiv: Daten. Auf den ersten Blick sollte die Verfügbarkeit von Daten keine großen Hürden darstellen, allerdings steckt hier der Teufel oft im Detail. Was im POC noch keine Probleme bereitet, scheitert oftmals beim Rollout oder schlimmstenfalls beim Skalieren.

Die meisten klassischen Datenbanken kommen mit den Datenmengen, die im IoT-Umfeld oftmals anfallen, schlichtweg nicht klar. Sowohl die Anzahl der Geräte als auch die Anzahl der Datenpunkte können hier schnell in die Zehntausende oder gar Millionen gehen. Zudem kommt noch eine oftmals hohe Frequenz der Datenerfassung hinzu. Um die Situation noch ein wenig zu verschlimmern, stehen teilweise keine dauerhaften Breitband-Internet-Verbindungen zur Verfügung. „Out-of-sequence"/ „Out-of-order"-Daten – also, dass ältere Daten nachträglich eingepflegt werden müssen – geben den meisten Systemen dann den Gnadenstoß.

In diesem Artikel will ich beschreiben, wie Apache IoTDB und Apache TsFile von Grund auf für diese Use-Cases entwickelt wurden und warum sie damit die perfekten Kandidaten sind, um die Herausforderungen im IoT Daten-Management zu lösen.


Apache

Der Ursprung von Apache IoTDB

Mit all den in der Einleitung beschriebenen Problemen hatte das Team um Prof. Jianmin Wang an der School of Software an der Tsinghua Universität immer wieder zu kämpfen. Daher hat er mit seinem Team 2011 das IoTDB-Projekt gegründet. Hierbei handelt es sich um eine auf IoT-Use-Cases optimierte Datenbank-Lösung. Seit November 2018 ist das Projekt als Apache IoTDB bei der Apache Software Foundation zuhause, seit 2020 auch als Top-Level-Projekt und erfreut sich einer großen und wachsenden internationalen Community.

Im Oktober 2023 hatte der Apache IoTDB PMC dafür gestimmt, Apache IoTDB in zwei Projekte aufzuspalten: Apache IoTDB und Apache TsFile. Somit wird der Tatsache, dass IoTDB schon immer aus zwei Teilen bestand, Rechnung getragen. Also bitte nicht wundern, wenn sich hier in den nächsten Wochen und Monaten einiges ändern wird.

Neue Ansätze zum Schreiben von Daten

Der vermutlich größte Unterschied zu anderen Datenbank-Systemen ist hier die Tatsache, dass die Persistenz-Engine oder das Storage-Format von der Query-Engine getrennt ist. Somit kann prinzipiell das Schreiben in die Datenbank auf einem anderen System erfolgen als das, auf dem später dann Auswertungen durchgeführt werden.

In das Speicher-Format „TsFile" sind mit großem Abstand die meisten Patente von IoTDB geflossen, da hier wirklich in vielerlei Hinsicht Neuland bestellt wurde. Es erlaubt eine verlustfreie Speicherung von Daten mit einer Kompression von bis zu 95 % (1:20). Wenn es nicht verlustfrei sein muss, kann die Kompression die Datenmenge um bis zu 99 % (1:100) reduzieren. Auch der Umgang mit „sparse Data" – dass also nicht für alle Sensoren eines Geräts immer alle Daten bereitstehen – ist in diesem Format äußerst effizient.

TsFile-Dateien enthalten nicht nur die eigentlichen Zeitreihen-Daten, sondern darüber hinaus auch die Schema-Information. Dies und die Art, wie diese gespeichert werden, erlaubt es IoTDB nicht nur Daten von einer theoretisch unbeschränkten Anzahl von Geräten zu verwalten, es erlaubt darüber hinaus auch eine unbeschränkte Anzahl von Spalten für jedes dieser Geräte zu pflegen, ohne dabei Performanz-Einbußen hinnehmen zu müssen. Hier unterschiedet sich IoTDB sehr von anderen etablierten Lösungen, bei denen man in der Regel in einer dieser Dimensionen, oftmals gar in beiden, eingeschränkt ist *(siehe Abbildung 1)*.

Auch wenn IoTDB generell als verteiltes System betrieben werden kann, kann es durchaus ebenso als „klassische" Datenbank betrieben werden. Dies ist auch der vermutlich am häufigsten genutzte Weg. In diesem Fall kümmert sich IoTDB sowohl um die Datenhaltung als auch um die Analyse der Daten.

Generell gibt es mehrere Wege, um Daten in das System zu bekommen:

- CLI
- Session-API
- JDBC
- MQTT

Bei dem Session-API handelt es sich um die aus Sicht der Performance und der verfügbaren Features bevorzugte Methode. Hierbei nutzt der User einen IoTDB Client in der von ihm bevorzugten Programmiersprache. Hier stehen zurzeit zur Verfügung:

- Java
- Go
- Python
- Rust
- C++
- C#

Der JDBC-Treiber ist vermutlich die Option, die sich am einfachsten in bestehenden Lösungen einsetzen lässt.

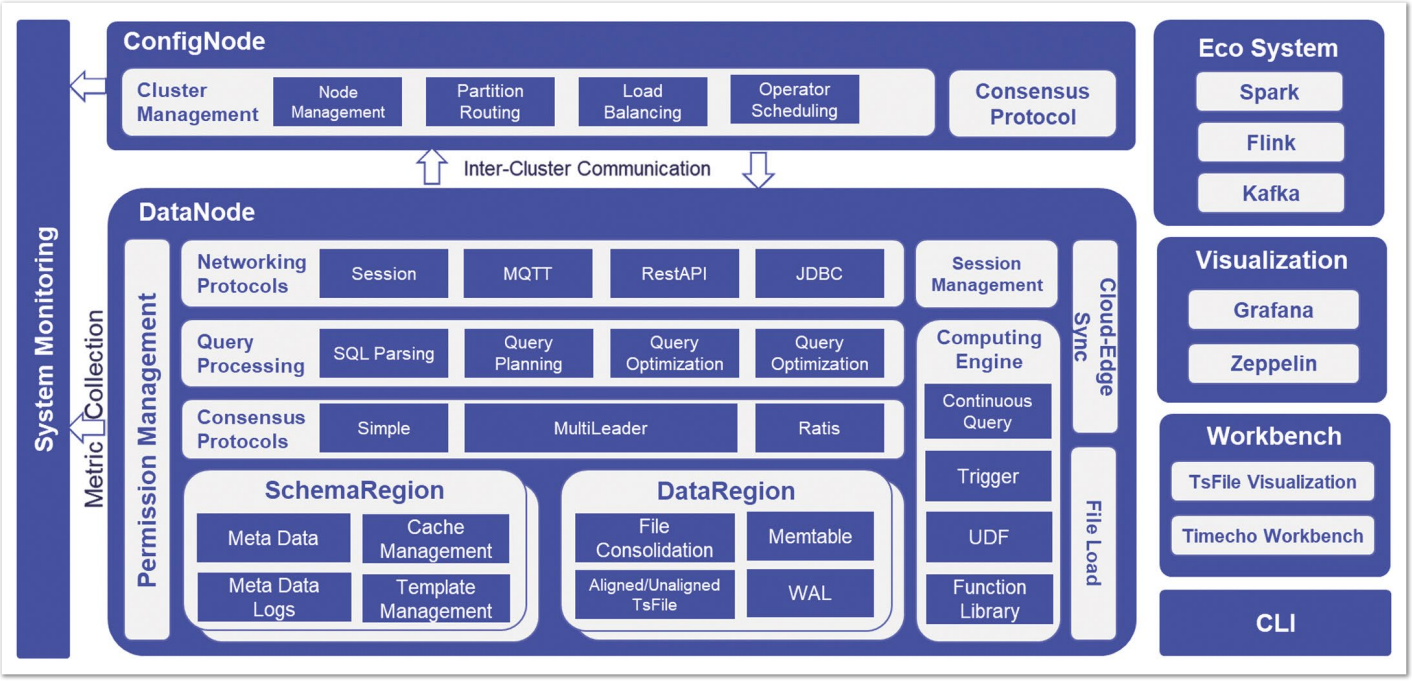


Abbildung 1

Einer der für mich zurzeit spannendsten Ansätze ist der, IoTDB als MQTT Client mit Daten zu versorgen. Während das System im aktuellen Stand in der Lage ist, direkt MQTT-Nachrichten im JSON-Format zu verarbeiten, so muss man allerdings dafür sorgen, dass diese Nachrichten dem IoTDB-Format entsprechen. Die Anlagen müssen daher explizit für IoTDB vorbereitet werden. Deshalb arbeite ich momentan daran, IoTDB um einen *Sparkplug-B*-Adapter *[1]* zu erweitern. Hiermit wäre es dann möglich, IoTDB ohne zusätzliche Konfiguration in moderne Industrie-Lösungen zu integrieren, da *Sparkplug-B* die nötigen Schema-Informationen gleich mitliefert. *Sparkplug-B* nutzt außerdem *Protobuf* für die Übertragung der Payload. Dies macht die Übertragung wesentlich effizienter als der Einsatz von einfachem JSON. *Sparkplug-B* Support gibt es bereits für einige Steuerungs-Typen allerdings befindet sich dieser oft eher noch im Beta-Stadium, ich bin mir aber sicher, dass sich Lösungen auf Basis dieser Technologie sehr schnell ausbreiten werden, da sie viele Probleme, wie OPC-UA, beheben.

Der zweite spannende Ansatz, der ebenfalls auf meiner To-do-Liste steht, wäre es, für etablierte Automatisierungssysteme, wie TwinCAT, Codesys, Step7 eine Bibliothek zu implementieren, die auf Basis der C++-TsFile-Bibliothek auf der Steuerung direkt erlaubt, die Daten zu erfassen. Hier würde der Automation Engineer explizit als Teil des Programms erforderliche Parameter in ein lokales TsFile schreiben und dieses dann mittels MQTT an einen IoTDB-Server senden. Dieser Ansatz hat den signifikanten Vorteil gegenüber dem klassischen Ansatz, dass er sowohl die SPS als auch das Netzwerk sehr entlasten würde. Momentan wird nämlich üblicherweise das digitale Abbild der SPS via Polling erstellt. Aufgrund des Sampling-Problems, bei dem die Sampling-Frequenz immer ein Vielfaches der Frequenz des gemessenen Signals sein muss, werden hier sehr viele Anfragen an die Steuerung geschickt, um ein gewünschtes Signal zu approximieren. Dieser Ansatz würde auch gleichzeitig das Problem von „Zyklus-synchronen Daten" lösen, weil in diesem Fall die interessanten Daten immer innerhalb eines Zyklus geschrieben werden würden. Beim Pollen, also dem periodischen Lesen der Daten von einem Client von außen, kann man nämlich nicht sicherstellen, dass beim Abarbeiten der Lese-Requests innerhalb der SPS, diese nicht zwischenzeitlich pausieren muss, um das eigentliche SPS-Programm auszuführen, bevor es die Bearbeitung der Requests fortsetzen kann. In diesem Fall passen die zurück gelieferten Informationen nicht zwangsläufig weiter zusammen.

Da Apache IoTDB komplett in Java implementiert ist, besteht darüber hinaus auch die Möglichkeit, IoTDB in eine bestehende Java-Anwendung einzubetten. Dies ist vor allem dann interessant, wenn zum Beispiel IoTDB aus der Ferne konfiguriert werden soll. Nicht immer ist es möglich, sich direkt oder via SSH einzuloggen und die nötigen Änderungen vorzunehmen. Hier bietet es sehr große Vorteile, wenn sich die Konfiguration von IoTDB über die Anwendung, in die es eingebettet wurde, anpassen lässt. Diesen Ansatz nutzen wir auch beispielsweise bei Timecho erfolgreich bei unserer App für den Bosch-Rexroth ctlX App-Store *[2]*. Hier kann der User über ein maßgeschneidertes UI die Konfiguration der Datenbank direkt ändern, ohne über den Weg der Manipulation von Config-Files gehen zu müssen.

Vielfältige Optionen beim Zugriff auf die Daten

Die Liste der Clients für IoTDB ist durchaus länger, da hier eine ganze Reihe von fertigen Integrationen für bestehende andere Systeme bereitsteht:

- CLI
- Session API
- JDBC
- Apache Flink
- Apache Hadoop
- Apache Hive
- Apache Spark
- Apache Zeppelin
- Grafana

Plug-ins wie Flink, Hadoop, Hive und Spark greifen hierbei direkt auf TsFiles zu und verteilen die Query-Funktionalität in den jeweiligen Clustern. Für Grafana steht ein Plug-in bereit, das sogar direkt aus dem Grafana Store installierbar ist *[3]*. Dies erlaubt einen nahtlosen Zugriff auf IoTDB.

Neben den üblichen Zeitreihen-Operationen, wie Aggregationen auf der Basis von Timestamps, bringt IoTDB eine Vielzahl an user-defined Functions (UDFs) mit, die beispielsweise eine sehr einfache Anomalie-Erkennung ermöglichen. Schon allein eine Auflistung der bereits verfügbaren UDFs würde den Rahmen dieses Artikels sicher sprengen, sollte hier dennoch etwas fehlen, so können jederzeit eigene UDFs erstellt werden. Eine genaue Auflistung aller Funktionen können Sie am besten auf der Website des Projekts *[4]* nachlesen.

IoT-Szenarien

Die Möglichkeit, verteilt Daten zu schreiben und diese dann auch noch hoch-effizient zu speichern und weiterzuleiten, prädestiniert TsFiles für die Erfassung von IoT-Daten direkt dort, wo diese entstehen. Vor allem bei verteilten Anlagen (Windkraftanlagen, Solarparks) erleichtert dies vieles, da die Daten lokal zwischengespeichert werden können und dann komprimiert in Batches weitergegeben werden können. Diese Batches können bei Bedarf auch auf die Transport-Technologie abgestimmt werden. So kann beispielsweise sichergestellt werden, dass beim Einsatz von LoRaWAN die verfügbare Bandbreite maximal ausgelastet wird.

Aber auch in mobilen Anlagen (Zügen, Schiffen, Flugzeugen, LKWs) kann dies sehr nützlich sein, da hier oft keine garantierte Netzwerk-Verbindung besteht. In diesem Fall können die Daten ohne weiteres so lange zwischengespeichert werden, bis der Zug wieder aus einem Tunnel gekommen ist, das Schiff wieder in einem Hafen angekommen ist oder der LKW aus dem Funkloch bei Langen heraus ist und wieder eine Verbindung besteht.

Doch selbst beim Einsatz mit industriellen Gigabit-Ethernet-Verbindungen kann es viele Vorteile mit sich bringen, die Daten lokal zu puffern und in TsFile Batches zu übertragen. Wie bereits von mir beschrieben, werden viele Automationsnetzwerke durch übermäßiges periodisches Abfragen von Daten überlastet. Diese Überlastung würde durch ein aktives Schreiben von aggregierten und komprimierten Daten von Seiten der SPS sehr entschärft *(siehe Abbildung 2)*.

Skalierung

IoTDB ist in der Lage, im Cluster betrieben zu werden. So können mehrere Server oder Cloud-Nodes ein IoTDB-Cluster bilden. Da Daten und Schema-Informationen über TsFiles verteilt gespeichert sind, erlaubt dies der IoTDB Query Engine eine hohe Parallelisierung, nicht nur im Cluster-Betrieb, was eine unvergleichbare Query Per-

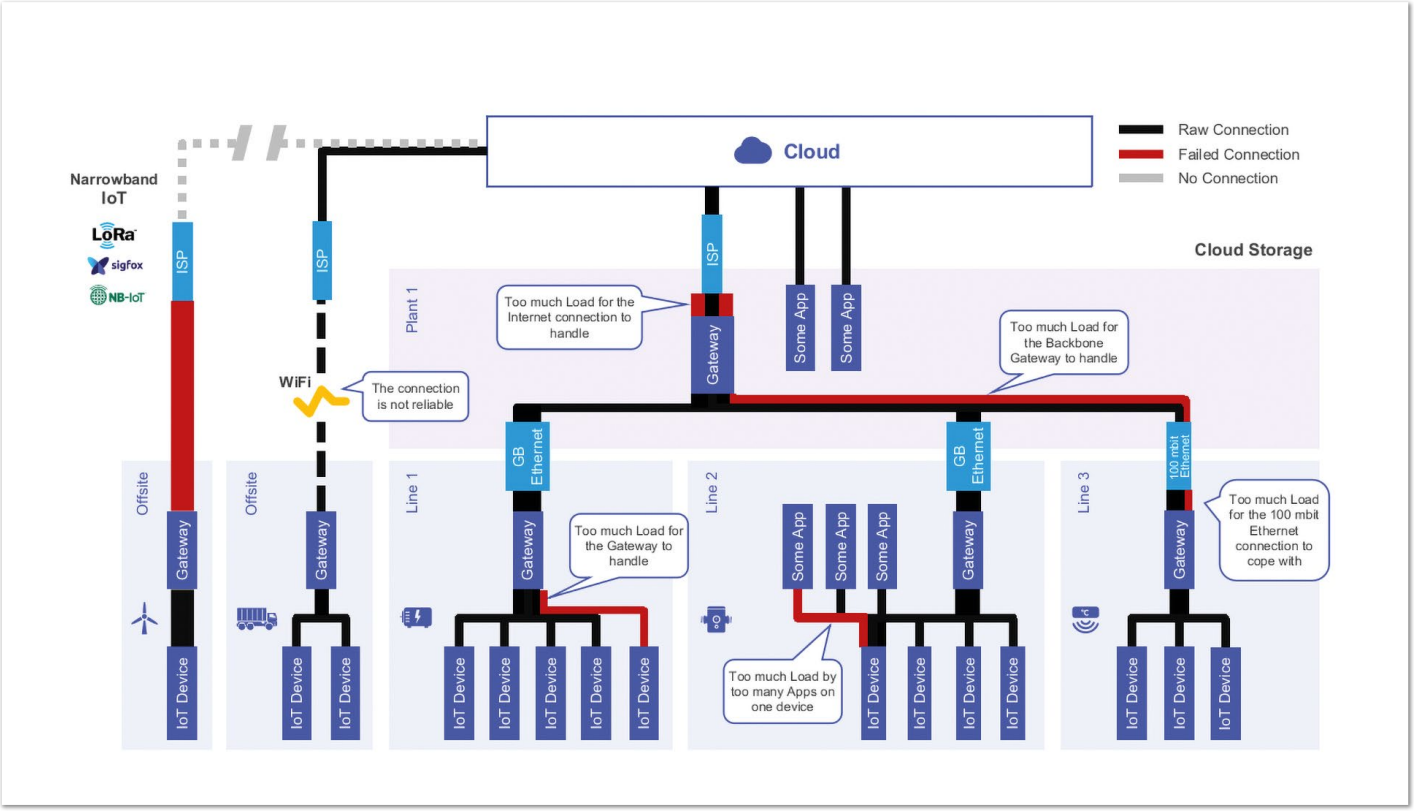


Abbildung 2

formanz mit sich bringt. (Siehe Benchmark am Ende dieses Artikels.)

IoTDB erlaubt es, Zeitreihen-Daten automatisch zu aggregieren und damit noch mehr Platz zu sparen. So können zum Beispiel die Daten des aktuellen Monats komplett vorgehalten werden, danach werden allerdings bestimmte Datenpunkte gelöscht oder deren Werte aggregiert, sodass diese zum Beispiel nicht mehr in der vollen Auflösung einiger Millisekunden, sondern nur noch über Minuten aggregiert vorgehalten werden. Hier sind der Kreativität keine Grenzen gestellt.

Vor allem in der industriellen Produktion ist es üblich, dass Datenbanken auf mehreren Ebenen des Unternehmens betrieben werden. An der Produktionslinie werden hier alle Daten gesammelt und gespeichert, die für den Betrieb der Anlage notwendig sind. Über mehrere Linien werden dann die Daten oftmals an übergeordnete Datenbanken weitergeleitet, die alle Daten der Produkt-Gruppe enthalten. Eine Etage höher werden die Daten des Werkes gesammelt, um zum Schluss die Daten aller Standorte in der Cloud zu sammeln.

Diese Ansätze der hierarchischen Datenhaltung sind vermutlich die Pläne, die in den seltensten Fällen den ersten Kontakt mit der Realität überstehen. Üblicherweise war es mir in der Vergangenheit immer möglich gewesen, Pläne, alle Daten aller Werke in der Cloud vorzuhalten sowie mithilfe eines Bleistiftes, eines Zettels und eines Taschenrechners vom Tisch zu fegen, ohne dass nennenswert Budget verschwendet wurde. Es ist schlichtweg nicht sinnvoll möglich, alle Daten des Unternehmens weder direkt noch über zwischengeschaltete Instanzen zu übertragen und dauerhaft vorzuhalten. Kaum eine Firma würde über die dazu nötigen Internet-Verbindungen oder Server-Kapazitäten verfügen, oder die dadurch verursachten Kosten würden bei Weitem ihren Nutzen übersteigen. Allerdings macht es auch nur in den seltensten Fällen Sinn, wirklich alle Daten in der maximalen Granularität auf allen Ebenen bereitzuhalten. Weil Aggregation in den meisten Systemen rechenintensiv, langsam und damit teuer ist, wird hier oftmals die Sampling-Frequenz reduziert. Allerdings schlägt dann das Sampling-Problem wieder voll zu und die Aussagekraft der Daten wird drastisch reduziert.

Hier kann IoTDB die bereits angesprochene automatische Aggregation in einer anderen Dimension einsetzen. So erlaubt es IoTDB beispielsweise, eine Instanz nahe der Produktionsanlage zu betreiben, die in maximaler Auflösung alle Daten speichert und damit schnelle und genaue Entscheidungen über einen kleinen Teil des Unternehmens erlaubt. Auf der nächsthöheren Ebene kumuliert eine weitere Instanz von IoTDB die Daten mehrerer Anlagen. Allerdings werden hier die Daten schon gefiltert und aggregiert abgelegt. Unnötige Daten werden weggelassen und hochauflösende Daten werden in einer niedrigeren Auflösung zusammengefasst. Während an der Anlage noch mit Millisekunden gearbeitet wird, genügen hier in der Regel Daten auf Basis von Sekunden oder Minuten, um auf dieser Ebene qualifizierte Entscheidungen zu fällen. So kann sichergestellt werden, das auf jeder Ebene genau die Daten in der für sie nötigen Granularität vorgehalten werden, ohne dabei unnötige Kosten oder gar unlösbare Probleme zu produzieren.

Benchmarks

Wir waren schon immer überzeugt, dass IoTDB den Vergleich mit anderen Zeitreihen-Datenbank-Systemen nicht scheuen muss, dies wurde Mitte September 2023 bestätigt, als die unabhängige Organisation „benchANT" Ergebnisse ihrer Analyse von Apache IoTDB veröffentlichte. Im Vergleich mit in allen anderen getesteten Lösungen gelang es Apache IoTDB, neue Maßstäbe in allen getesteten Dimensionen zu setzen. Die Ergebnisse wurden von benchANT in ei-

Produkt	Apache IoTDB	InfluxDB	TimescaleDB
Write Throughput:	3.636.312 ops/s	529.220 ops/s	1.572.058 ops/s
Read Throughput:	11.497 ops/s	2.197 ops/s	518 ops/s
Read Latency:	3,41 ms	45,45 ms	193,13 ms
Storage consumption:	1,8 GB	2,79 GB	69,84 GB

Quelle: „Apache IoTDB: Ein neuer Leader bei den Zeitserien-Datenbanken" [5]

nem Blog-Artikel mit der Überschrift „Apache IoTDB: Ein neuer Leader bei den Zeitserien-Datenbanken" *[5]* ausführlich beschrieben.

Besonders interessant ist hier der Vergleich mit den im deutschen Umfeld etablierten Lösungen InfluxDB und TimescaleDB:

Fazit

Ich halte Apache IoTDB und Apache TsFile für ideale Kandidaten, um die aktuellen Probleme beim Umgang mit gigantischen Zeitreihen zu beheben. Vor allem begeistert mich, dass diese Lösung so einfach zu verwenden ist. In der Tat ist es nicht viel mehr als das Herunterladen des Binary-Pakets, Entpacken und Ausführen des „sbin/start-standalone.sh"- (oder „.bat"-)Skriptes. Hier sind keine Cluster oder Spezialkonfigurationen des Betriebssystems notwendig, auch hat es keine weiteren Abhängigkeiten. Eine einfache Java-VM ist genug, um eine ungeheuer leistungsfähige Zeitreihen-Datenbank zu betreiben. Persönlich bilden für mich beide Projekte einen Schwerpunkt in meinen Plänen, mithilfe von Open-Source und modernen IT-Lösungen und -Vorgehensweisen die Herausforderungen in der Automatisierungsindustrie zu lösen.

Quellen

[1] Eclipse Foundation (2023), „MQTT + Sparkplug = '°Plug & Play' IIoT", *https://sparkplug.eclipse.org/*

[2] Bosch Rexroth AG (2023), „ctrlX Automation - Two Steps Ahead", *https://apps.boschrexroth.com/microsites/ctrlx-automation/de/*

[3] Grafana (2023), „Apache IoTDB Plugin", *https://grafana.com/grafana/plugins/apache-iotdb-datasource/*

[4] Apache IoTDB (2023), „IoTDB Website", *https://iotdb.apache.org*

[5] benchANT (2023), „Apache IoTDB: Ein neuer Leader bei den Zeitserien-Datenbanken", *https://benchant.com/de/blog/apache-iotdb-performance*

Christofer Dutz

christofer.dutz@timecho.com

Christofer Dutz hat sich mit ganzem Herzen dem Thema Open-Source und Industrial IoT verschrieben. Er hat vor 7 Jahren das Apache-PLC4X-Projekt ins Leben gerufen und arbeitet bei Timecho Vollzeit und in der Freizeit an Apache PLC4X, IoTDB und TsFile. Er ist zurzeit Committer in 15 verschiedenen Apache-Projekten und einigen außerhalb der ASF. Er engagiert sich als Mentor im Apache Incubator und ist sowohl im Community-Development als auch in anderen Gremien, unter anderem auch als einer der 9 Direktoren bei der ASF engagiert.

Die richtige JaCoCo-Konfiguration für ein Apache-Maven-Projekt finden

Karl Heinz Marbaise

Beim Programmieren stellt sich immer die Frage: Habe ich mit meinen Tests alle Fälle abgedeckt? Welche Bereiche fehlen mir noch? Ziel dieses Artikels ist es, eine Konfiguration für ein Apache-Maven-Projekt zu erstellen, dass die Abdeckung für Unit- und Integrationstests liefert.

Übersicht

Es ist gängige Praxis, dass eine gewisse Anzahl von Unit- und/oder Integrationstests innerhalb eines Projektes vorhanden sind. (Ich hoffe, Sie haben eine größere Anzahl von Tests?) Die Fragen, die sich daraus ergeben, sind: Ist das ausreichend? Sind alle möglichen Fälle abgedeckt?

Diese können mithilfe des Konzeptes der Code-Abdeckung (englisch: *Code Coverage* [4]) beantwortet werden. Diese Methode ist geeignet, um zu prüfen, welche Pfade und Methoden in produktivem Code durch die Ausführung der Testfälle durchlaufen wurden und welche nicht. Genauer gesagt, geht es hier um Dinge wie beispielsweise Funktionsabdeckung (*Function Coverage*), *Statement Coverage* und Verzweigungsabdeckung (*Branch Coverage*).

In der Konsequenz: Je höher die Rate der Abdeckung, desto geringer ist die Wahrscheinlichkeit für Fehler. Ja, einige werden jetzt erwidern, dass erfolgreiche Tests kein Beweis für die Fehlerfreiheit einer Software seien. Das ist vollkommen korrekt. Allerdings hilft es, die Software in dem Zustand zu halten, wie sie ist. Unabhängig davon, dass Änderungen vorgenommen werden (Änderung interner Implementierungen oder das Hinzufügen neuer Funktionalität und so weiter). Das bewahrt uns zumindest vor sogenannten Regressionsfehlern [5]. Dies deckt natürlich nur die vorhandene Funktionalität ab, nicht aber die neu entwickelte Funktionalität.

Basierend darauf stellt sich dann die letzte Frage: Wie hoch sollte die Coverage-Rate denn nun sein? 30 %, 50 %, 80 % oder sogar 100 %? Aus meiner eigenen Erfahrung würde ich sagen, 30 % ist zu wenig und 100 % ist schon etwas merkwürdig. Warum? Da man im Fall von 100 % Getter/Setter testen müssten. Das heißt am Ende, dass ebenso das JDK selbst (oder der Compiler) getestet werden müsste. Das bedeutet, dass der Fokus darauf liegt, dass Coverage Tool „glücklich" zu machen, anstatt sich zu überlegen, wo die wirklich wichtigen Teile der Applikation sind. Daher lautet meine praktische Empfehlung: zirka 80 %.

```
.
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── soebes
    │               └── example
    │                   └── jacoco
    │                       └── First.java
    └── test
        └── java
            └── com
                └── soebes
                    └── example
                        └── jacoco
                            ├── FirstIT.java
                            └── FirstTest.java
```

Listing 1: Beispiel Projektstruktur

Ein Werkzeug, um eine Code-Abdeckung in Java zu messen, ist zum Beispiel JaCoCo [6]. Doch das ist nicht das einzige Werkzeug. Es existieren weitere, wie OpenClover [7] oder Kover [8] für Kotlin und einige andere.

In diesem Artikel werde ich JaCoCo [6] verwenden. Im Zusammenhang mit einem Apache-Maven-Projekt verwende ich hier das `jacoco-maven-plug-in` [9], das sich einfach und schnell in den Build-Prozess integrieren lässt.

Beispiel-Projekt

Um eine Code-Abdeckung für ein Projekt zu ermitteln, benötigen wir selbstverständlich ein entsprechendes Beispielprojekt. Die Struktur sieht wie in *Listing 1* dargestellt aus.

Anhand der Namenskonvention haben Sie sicherlich schon die Unit-Test-Klasse `FirstTest.java` und die Integrationstest-Klasse `FirstIT.java` identifiziert. Ich empfehle immer, den Namenskonventionen zu folgen, um es sich und anderen einfacher zu machen. Das gilt auch für die Verzeichnisstruktur [10]. Hier muss selbstverständlich noch die `pom.xml`-Datei erwähnt werden, die die Konfiguration und eventuelle Abhängigkeiten für das Projekt enthält.

Plug-in-Definition

Dann schauen wir doch mal etwas genauer auf die `pom.xml`-Datei. In einem Apache-Maven-Projekt [1] sollten immer alle Plug-ins definiert sein, um einen Build reproduzierbar zu halten. Ich möchte hier den Fokus auf das `jacoco-maven-plugin` legen *(siehe Listing 2)*. Die Festlegung schließt die `GroupId`, `ArtifactId` und die Version mit ein. Die verwendeten Auslassungspunkte ... stellen andere Plug-ins dar, die entsprechend angegeben sein sollten, aber der Übersichtlichkeit halber hier ausgelassen wurden. Die angegebenen Plug-ins, wie das Maven-Surefire-Plug-in [2] werden benötigt, um die entsprechenden Unit-Tests auszuführen, während das Maven-Failsafe-Plug-in [3] dafür zuständig ist, die Integrationstests auszuführen.

```
<project...>
  ...
  <build>
     <pluginManagement>
        <plugins>
           <plugin>
              <groupId>org.jacoco</groupId>
              <artifactId>jacoco-maven-plugin</artifactId>
              <version>0.8.11</version>
           </plugin>
           ...
           <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-surefire-plugin</artifactId>
              <version>3.2.1</version>
              <configuration>
                 <skipTests>${skipUTs}</skipTests>
              </configuration>
           </plugin>
           <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-failsafe-plugin</artifactId>
              <version>3.2.1</version>
              <configuration>
                 <skipTests>${skipITs}</skipTests>
              </configuration>
           </plugin>
        </plugins>
     </pluginManagement>
     ...
  </build>
  ...
</project...>
```

Listing 2: Beispiel-Konfiguration für das JaCoCo-Plug-in und andere Plug-ins

Grundsätzlich legen wir die Versionen und die globale Konfiguration im `pluginManagement` fest. Der nächste Schritt ist es, die Konfiguration für das `jacoco-maven-plugin` anzugeben *(siehe Listing 3)*. JaCoCo arbeitet mit sogenannten Agents, die während der Ausführung der Test für die Aufzeichnung der Ergebnisse (*Coverage*) sorgen. Dazu ist es notwendig, dass gewährleistet ist, dass die *Agents* auch bei der Ausführung der Tests entsprechend mit aufgerufen werden. Netterweise übernimmt das JaCoCo-Maven-Plug-in dies für uns im Zusammenhang mit dem Goal `prepare-agent`.

```
<project...>
  ...
  <build>
     ...
     <plugins>
        ..
        <plugin>
           <groupId>org.jacoco</groupId>
           <artifactId>jacoco-maven-plugin</artifactId>
           <executions>
              <execution>
                 <goals>
                    <goal>prepare-agent</goal>
                    ...
                 </goals>
              </execution>
           </executions>
        </plugin>
     </plugins>
  </build>
  ...
</project...>
```

Listing 3: Konfiguration des Plug-in-Goals

Mit der entsprechenden Konfiguration und der Ausführung von `mvn -X verify` kann man sich dann das Ergebnis anschauen. In der Ausgabe finden sich nun Informationen wie in *Listing 4* dargestellt.

```
-javaagent:/.../org/jacoco/org.jacoco.agent/0.8.11/
org.jacoco.agent-0.8.11-runtime.jar=destfile=/../tar-
get/jacoco.exec'
```

Listing 4: Ausgabe-Ausschnitt

```
<project...>
  ...
  <build>
     ...
     <plugins>
        ..
        <plugin>
           <groupId>org.jacoco</groupId>
           <artifactId>jacoco-maven-plugin</artifactId>
           <executions>
            <execution>
                 <goals>
                    <goal>prepare-agent</goal>
                    <goal>report</goal>
                    ...
                 </goals>
              </execution>
           </executions>
        </plugin>
     </plugins>
  </build>
  ...
</project...>
```

Listing 5: Konfiguration für den Report, Aufruf des `report`-Goal

Hier ist etwas Vorsicht geboten, da die Ausgabe unter der Verwendung von `-X` recht lang wird. Die beste Möglichkeit ist, die Ausgabe in eine Datei umzuleiten. Die Umleitung kann in Linux/MacOS einfach per `mvn -X verify | tee mvn.log` geschehen. Unter Windows per `mvn -X verify >mvn.log`. Danach können Sie die Ausgabe dann genauer unter die Lupe nehmen.

HTML Report

Wenn Sie einen bestimmten Bericht haben möchten, den Sie sich anschauen können, ist der HTML Report dazu sehr gut geeignet. Der Bericht wird durch das `report`-Goal des Plug-ins produziert. Dazu ist aber notwendig, dass das `report`-Goal auch entsprechend aufgerufen wird. Das geschieht durch die Angabe in der `pom.xml`-Datei *(siehe Listing 5)*.

Durch die Ausführung von Maven via `mvn verify` wird nun ein HTML- und CSV-basierter Bericht erstellt. Die Bindung an die Life-Cycle-Phase von Maven erledigt das Plug-in hier automatisch. Ich empfehle dringend, die Bindung dem Plug-in zu überlassen, oder genauer: den Plug-in-Autorinnen und -Autoren. Die entsprechenden Angaben sind in 99,999 % der Fälle korrekt. Nur in Ausnahmefällen mag es notwendig sein, die Bindung selbst vorzunehmen.

Die erstellten Berichte finden sich dann im Verzeichnis `target/site/jacoco`. Die `index.html`-Datei kann mit einem Browser Ihrer Wahl geöffnet werden. Falls Sie nicht den gesamten Life-Cycle bis `verify` durchlaufen möchten, aber trotzdem die Berichte erstellen wollen, erreichen Sie dies einfach durch die Ausführung des Befehls in *Listing 6*.

```
mvn clean test jacoco:report
```

Listing 6: Erstellung des Berichts mithilfe der Kommandozeile

Dadurch wird der Life-Cycle bis einschließlich zur `test`-Phase durchlaufen und zusätzlich wird noch das Goal `report` des `jacoco-maven-plugin` entsprechend ausgeführt. In den meisten Fällen können Sie das clean einfach weglassen.

Test-Projekt

Am Anfang hatte ich die Projektstruktur erwähnt, nun fokussieren wir uns auf den Inhalt des Test-Projekts. Der sogenannte Produktionscode enthält die Klasse `First.java`, die den Quellcode wie in *Listing 7* darstellt (JDK 16+).

```
public record First(int sum) {

  public First add(First first) {
     return new First(first.sum + this.sum);
  }

  public First sub(First first) {
     return new First(this.sum - first.sum);
  }
  public First minus(First first) {
     return new First(this.sum - first.sum);
  }
}
```

Listing 7: Konfiguration für den Report, Produktionscode mit der Klasse `First.java`

Bitte bewerten Sie den Quellcode nicht, dieser dient lediglich zur Demonstration der Code-Abdeckung. Die Unit-Test-Klasse `FirstTest` enthält den in *Listing 8* dargestellten Quellcode.

```
class FirstTest {

  @Test
  void first_add() {
     First sum1 = new First(5);
     First sum2 = new First(2);

     assertThat(sum1.add(sum2)).isEqualTo(new First(7));
  }
}
```

Listing 8: Konfiguration für den Report, Unit-Test-Klasse

Kommen wir zum Integrationstest. Der befindet sich in der Klasse `FirstIT.java`, wie in *Listing 9* dargestellt.

```
class FirstTest {

  @Test
  void first_minus() {
     First sum1 = new First(5);
     First sum2 = new First(2);

     assertThat(sum1.minus(sum2)).isEqualTo(new First(3));
  }
}
```

Listing 9: Konfiguration für den Report, Integrationstest

Bei genauerer Betrachtung haben Sie sicher schon festgestellt, dass der Unit-Test lediglich die `add(..)`-Methode abdeckt, und der Integrationstest die `minus`(..)-Methode.

Quellcode-Abdeckung

Führen wir nun `mvn clean verify` aus, so werden die Unit-Tests entsprechend mit ausgeführt. Des Weiteren wird das report-Goal des JaCoCo-Maven-Plug-ins aufgerufen. Schauen wir uns einmal den Bericht an *(siehe Abbildung 1)*.

Das, was Ihnen sicherlich auffällt, ist, dass lediglich die `add(..)`-Methode durch den Test abgedeckt ist (der Konstruktor ebenfalls).

Aber Moment mal? Was war mit dem Integrationstest, der sollte doch die `minus(..)`-Methode aufgerufen haben?

Interessanterweise wird das im Quellcode-Abdeckungsbericht nicht angezeigt. Das ist merkwürdig, weil doch in der Integrationstest-Klasse `FirstIT.java` eine entsprechende Test-Methode vorhanden ist. Haben wir etwas anderes falsch gemacht? Das Problem ist, dass wir die Konfiguration für die Ausführung des Maven-Failsafe-Plug-in *[3]* nicht angegeben haben. Das hat zur Konsequenz, dass keinerlei Quellcode-Abdeckung bestimmt werden kann. Das Problem löst sich einfach durch die entsprechende Angabe der notwendigen Lifecycle-Bindung *(siehe Listing 10)*.

```
<project...>
  ...
  <build>
     ...
     <plugins>
        ..
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-failsafe-plugin</artifactId>
           <executions>
              <execution>
                 <goals>
                    <goal>integration-test</goal>
                    <goal>verify</goal>
                 </goals>
              </execution>
           </executions>
        </plugin>
     </plugins>
  </build>
  ...
</project...>
```

Listing 10: Konfiguration für Maven-Failsafe-Plug-in

Nachdem die notwendige Konfiguration ergänzt wurde, können wir sowohl die Unit- als auch die Integrationstests in einem einzigen Aufruf `mvn clean verify` zusammen ausführen.

Als Ergebnis der entsprechenden Konfiguration können Sie sehen *(siehe Abbildung 2)*, dass nun die beide Methoden `add(..)` und `minus(..)` durch die Tests entsprechend abgedeckt sind. Das geht jetzt auch mit der Erwartung konform, dass die `add(..)`-Methode durch den Unit-Test `FirstTest.java` und die `minus(..)`-Methode durch den Integrationstest `FirstIT.java` abgedeckt sind. Somit haben wir erfolgreich unser Ziel erreicht, die Quellcode-

JaCoCo Example > com.soebes.example.jacoco > First

First

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
sub(First)		0 %		n/a	1	1	1	1	1	1
minus(First)		0 %		n/a	1	1	1	1	1	1
add(First)		100 %		n/a	0	1	0	1	0	1
First(int)		100 %		n/a	0	1	0	1	0	1
Total	18 of 33	45 %	0 of 0	n/a	2	4	2	4	2	4

Abbildung 1: Bericht mithilfe des JaCoCo-Maven-Plug-ins erstellt

iJUG Verbund

NEWSLETTER

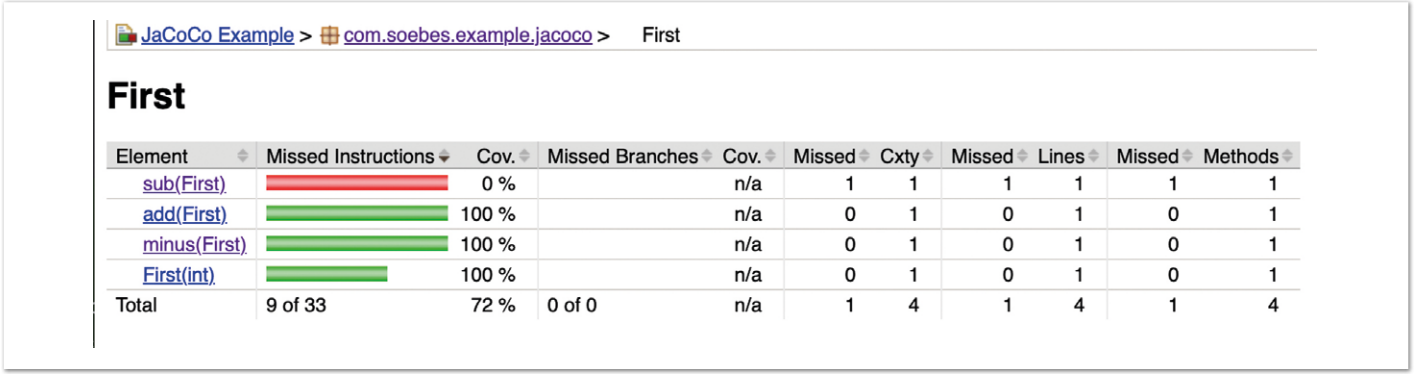
Anmeldung

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

Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!

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

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



JaCoCo Example > com.soebes.example.jacoco > First

First

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
sub(First)		0 %		n/a	1	1	1	1	1	1
add(First)		100 %		n/a	0	1	0	1	0	1
minus(First)		100 %		n/a	0	1	0	1	0	1
First(int)		100 %		n/a	0	1	0	1	0	1
Total	9 of 33	72 %	0 of 0	n/a	1	4	1	4	1	4

Abbildung 2: Bericht mit beiden Methoden

Abdeckung sowohl für die Unit- als auch für die Integrationstests innerhalb einer einzigen Build-Ausführung zu bestimmen. Es ist noch wichtig zu erwähnen, dass dazu keine weitere Konfiguration hinzugefügt wurde, außer der hier gezeigten.

Nur Unit- oder Integrationstest

Basierend auf der angegebenen Konfiguration für das Maven-Surefire-Plug-in *[2]* und dem Maven-Failsafe-Plug-in *[3]* können wir Teile separat ausführen. Möchten wir lediglich die Unit-Tests ausführen, aber dennoch einen Quellcode-Abdeckungsbericht erstellen, so müssen wir hier die Integrationstests überspringen, beziehungsweise nicht ausführen. Dass erreichen wir mithilfe der Property `skipITs` *(siehe Listing 11)*.

```
mvn clean verify -DskipITs
```

Listing 11: Unit-Test Quellcode-Abdeckung

Die beiden Eigenschaften `skipITs` und `skipUTs` geben uns die Möglichkeit, die Ausführung sowohl der Unit- als auch der Integrationstest unabhängig voneinander zu kontrollieren. Das Goal `report` des `jacoco-maven-plugin` wird entsprechend in der `verfiy`-Lifecycle-Phase ausgeführt. Möchten wir jetzt noch die Integrationstests ausführen und nur deren Quellcode-Abdeckung bestimmen, so können wir das durch die Ausführung (wie in *Listing 12* dargestellt) erreichen.

```
mvn clean verify -DskipUTs
```

Listing 12: Integrationstest Quellcode-Abdeckung

Anhand der gegebenen Konfiguration können wir auswählen, was nun ausgeführt werden soll, entweder Unit Tests oder Integrationstests oder beide zusammen. Das finale Resultat beinhaltet den entsprechenden Quellcode Abdeckungsbericht.

Zusammenfassung

Bedauerlicherweise sehe ich oft Konfigurationen, in denen der Agent separat für die Plug-ins Maven *Surefire* und Maven *Failsafe* konfiguriert ist, obwohl das überhaupt nicht notwendig ist. Das scheint wohl auf einer sehr alten Version des `jacoco-maven-plugin` zu basieren (schon sehr lange her). Das bedeutet, dass heute keinerlei zusätzliche Konfiguration mehr notwendig ist. Auch wird oftmals noch die Konfiguration dahingehend ergänzt, dass Dateipfade oder ähnliches angegeben sind. Das ist alles schlicht nicht notwendig.

Es gibt allerdings Ausnahmen. Das ist der Fall bei der Verwendung von `preview`-Sprach-Features für das JDK21+ oder im Zusammenhang mit der Verwendung von Mockito *[12]* bei JDK21 und höher. Genauer gesagt geht es hier um Byte-Buddy. In dem Fall muss die Konfiguration entsprechend angepasst werden *(siehe Listing 13)*. Das könnte sich in der Zukunft noch ändern.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
     <argLine>
         -XX:+EnableDynamicAgentLoading
         ...
     </argLine>
  </configuration>
</plugin>
```

Listing 13: Konfiguration JDK21 und höher

Die angegebene Option `-XX:+EnableDynamicAgentLoading` wird benötigt, um Warnungen, wie in *Listing 14* dargestellt, zu unterdrücken.

```
WARNING: A Java agent has been loaded dynamically (/
Users/khm/../net/bytebuddy/byte-buddy-agent/1.14.6/
byte-buddy-agent-1.14.6.jar)
WARNING: If a serviceability tool is in use, please
run with -XX:+EnableDynamicAgentLoading to hide this
warning
WARNING: If a serviceability tool is not in use,
please run with -Djdk.instrument.traceUsage for more
information
WARNING: Dynamic loading of agents will be disallowed
by default in a future release
```

Listing 14: Warnungen bei JDK21 und höher

Mit JDK 21 *[13]* und höher ist es notwendig, eine entsprechend angepasste Konfiguration anzugeben, wenn Mockito *[12]* verwendet wird. Für das Maven-Surefire- und Maven-Failsafe-Plug-in existiert

eine sogenannte „verzögerte-Auswertung" (aka lazy-evaluation) von Eigenschaften *[14]*. Ein Beispiel ist, wie die vorher genannten Warnungen unterdrückt werden, aber trotzdem auch ein Code Coverage Report erstellt wird. Dazu ist es unabdingbar, die Anpassungen wie in *Listing 15* vorzunehmen.

```
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
     <argLine>
        @{argLine}
        -XX:+EnableDynamicAgentLoading
        ...
     </argLine>
  </configuration>
</plugin>
...
```

Listing 15: Konfiguration JDK21 und JaCoCo

Ich möchte die Angabe `@{argline}` hervorheben und empfehle Ihnen, sie an den Anfang der Konfiguration zu positionieren. Zusammenfassend erhalten wir eine Konfiguration, die für JDK 21 und höher die Aktivierung von `preview`-Sprach-Features ermöglicht sowie uns auch Code Coverage Reports ermöglicht.

Auf GitHub *[11]* habe ich ein vollständiges Beispielprojekt mit allem Quellcode und auch der entsprechenden Konfiguration zur Verfügung gestellt.

Quellen

[1] *https://maven.apache.org*
[2] *https://maven.apache.org/plugins/maven-surefire-plugin/*
[3] *https://maven.apache.org/plugins/maven-failsafe-plugin/*
[4] *https://de.wikipedia.org/wiki/Testabdeckung*
[5] *https://de.wikipedia.org/wiki/Regressionstest*
[6] *https://www.jacoco.org/jacoco/*
[7] *https://openclover.org/*
[8] *https://github.com/Kotlin/kotlinx-kover*
[9] *https://www.jacoco.org/jacoco/trunk/doc/maven.html*
[10] *https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html*
[11] *https://github.com/khmarbaise/example-jacoco*
[12] *https://site.mockito.org/*
[13] *https://blog.soebes.io/posts/2023/06/2023-06-24-how-to-use-jdk21-preview-features-incubator/*
[14] *https://maven.apache.org/surefire/maven-surefire-plugin/faq.html#late-property-evaluation*

Karl Heinz Marbaise
info@soebes.io

Karl Heinz Marbaise arbeitet als Freiberuflicher DevOps bei den unterschiedlichsten Kunden. Er hat zirka 18 Jahre Java-Erfahrung und Expertenkenntnisse im Bereich Apache Maven. Weiterhin in der Verwendung von Testing-Frameworks wie JUnit Jupiter, Mockito, AssertJ und Testcontainers. Er hat auch ausführlichste Erfahrungen im Bereich CI/CD-Systemen wie Jenkins, Drone, Git, Gitea GitHub und Ansible und nicht zu vergessen Erfahrungen im Bereich Containern. Er ist Committer beim Apache-Maven-Projekt.

„Das ist doch behindert!“ – Über Barrieren und Freiheiten

Matthias Bünger

Websites und Webanwendungen in Deutschland müssen barrierefrei sein. Aber welche Barrieren gibt es überhaupt und wann ist eine Anwendung frei von diesen? In diesem Artikel wird aufgezeigt, dass es sehr viele unterschiedliche Barrieren für jemanden geben kann, aber auch wie man sie mit einfachen Mitteln vermeidet, um allen Anwendern eine barrierefreie Nutzung zu ermöglichen.

Die Begriffe „Behinderung“ und „Barrierefreiheit“ verbinden viele Menschen nur mit körperlich eingeschränkten Personen, die beispielsweise im Rollstuhl sitzen und als Folge dessen eine Rampe benötigen, um einen Höhenunterschied zu überwinden. Die Bedeutung der beiden Worte umfasst aber sehr viel mehr! Das Gesetz definiert „Behinderung“ wie folgt: „Menschen mit Behinderungen sind Menschen, die körperliche, seelische, geistige oder Sinnesbeeinträchtigungen haben, die sie in Wechselwirkung mit einstellungs- und umweltbedingten Barrieren an der gleichberechtigten Teilhabe […] hindern können.“ *[1]*.

Es fällt auf, dass dem Gesetz nach ein Mensch als behindert gilt, wenn seine Einschränkung auf eine Barriere trifft, die ihn von der gleichberechtigten Nutzung eines Objekts abhält. Berücksichtigt man diese Formulierung, so wird schnell klar, dass der Begriff „Barrierefreiheit“ den Gesetzestext aufgreift und eine uneingeschränkte Zugänglichkeit (englisch: *Accessibility*) fordert – so wie zum Beispiel eine Rampe für Rollstuhlfahrer.

Im nächsten Abschnitt werden verschiedene Arten von Einschränkungen und dazugehörige Barrieren aufgezeigt. Anschließend werden Anforderungen und Umsetzungshilfen zur Erreichung einer barrierefreien Anwendung beschrieben. Am Ende werden wir erkennen, dass barrierefreie Websites allen Personen zugutekommen.

Barrieren bei Websites

Um Barrieren abzubauen und gänzlich zu vermeiden, muss man sich zunächst über deren Existenz und der Vielfalt an Einschränkungen bewusst sein. Sie lassen sich in verschiedene Kategorie einteilen und im Folgenden möchte ich vier Kategorien vorstellen und kurz auf diese eingehen.

Zu der ersten Kategorie gehören die offensichtlichsten Einschränkungen im Kontext von Websites: visuelle Einschränkungen. Visuell eingeschränkte Personen können Websites beispielsweise nicht oder nur limitiert nutzen, da sie teilweise oder gänzlich blind sind oder unter einer reduzierten Farb- und/oder Textwahrnehmung leiden. Zu letzteren gehören folgende Barrieren: Nicht berücksichtigte Existenz einer teilweisen oder gänzlichen Farbblindheit, mangelnde Kontraste sowie unzureichende Textgrößen. Auf dem nachfolgenden Bild ist anhand eines Ausschnitts der JavaLand-Konferenz-Website simuliert, wie Personen mit verschiedenen Farbblindheiten diese wahrnehmen.

Websites werden immer interaktiver und beinhalten immer häufiger Video- oder Audioinhalte. Bei diesen müssen sowohl visuelle als auch auditive Einschränkungen berücksichtigt werden, weshalb diese die zweite Kategorie bilden. Auditiv eingeschränkte Personen sind zum Beispiel teilweise oder gänzlich gehörlos, können Geräusche/Gespräche nicht im Raum lokalisieren oder aus einer Geräuschkulisse selektieren. Diese Einschränkungen sollten bei der Erstellung und Bereitstellung medialer Inhalte berücksichtigt werden.

Die dritte Kategorie umfasst Barrieren, die offensichtlich, aber auch verschleiert sein können und auf die vermutlich jeder schon einmal gestoßen ist: Probleme, den Inhalt einer Website zu verstehen. Es ist naheliegend, dass Personen mit Leseschwäche Schwierigkeiten haben, einen Text zu lesen. Je nach Aufbau und Wortwahl eines Textes fällt es aber auch Personen ohne nachgewiesene Legasthenie schwer, den Inhalt einer Website oder eines Textes zu verstehen. Reiht sich zum Beispiel ein Fachbegriff an den anderen oder zieht sich ein Satz über eine ganze Seite, erzeugt man leicht vermeidbare Barrieren. Eine gewählte Sprache wird für Personen, die diese Sprache nicht sprechen, immer eine Barriere darstellen – das lässt sich nicht vermeiden. Solange man die ersten Punkte berücksichtigt, können automatisierte Übersetzungswerkzeuge Fremdsprachler zudem gut unterstützen. Bei einer der schlimmsten Barrieren können aber selbst Werkzeuge nicht helfen: Irreführende oder gar falsche Beschreibungen jeglicher Art. Diese müssen unter allen Umständen vermieden werden!

Neben bisher vorgestellten Einschränkungen existieren aber noch weitere: Die mechanischen Einschränkungen bilden die vierte Gruppe. Personen mit solchen Einschränkungen haben häufig Probleme, Websites mit einer Maus/der Hand zu bedienen, weil sie zum Beispiel ein verzögertes Reaktionsverhältnis aufzeigen (dies führt unter anderem zu Schwierigkeiten, einen Doppelklick vollziehen zu können) oder versehrt und so dauerhaft auf Prothesen angewiesen sind.

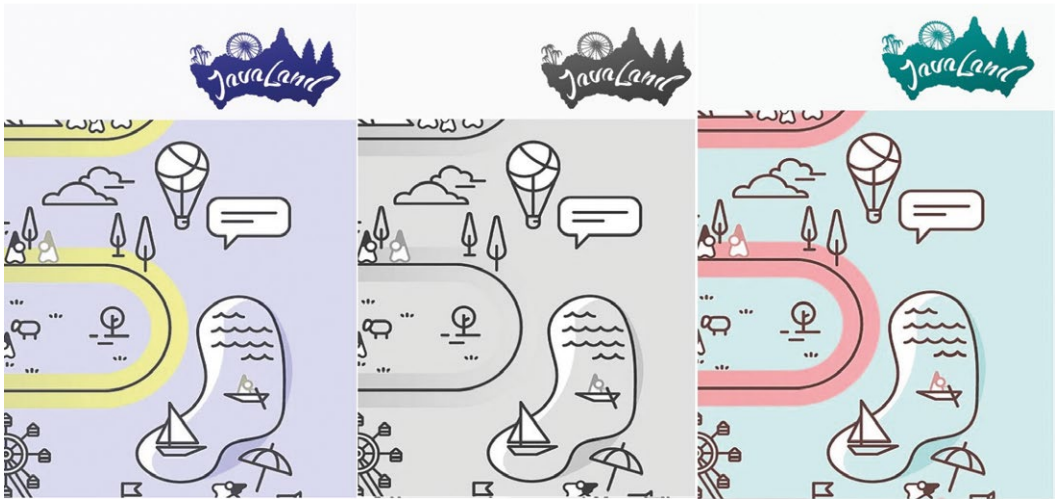


Abbildung 1: Simulierte Farbblindheiten am Beispiel der Website der JavaLand-Konferenz. Von links nach rechts: Rot-Grün-Blindheit, keinerlei Farbwahrnehmung, Gelb-Blau-Blindheit.

Eingeschränkten Personen steht heutzutage eine große Menge an Hilfsmitteln zur Verfügung. So erleichtern beispielsweise Screenreader und unterschiedliche Eingabehilfen (Brailleschrift-Tastaturen, mund- oder sprachgesteuerte Eingabegeräte) die Nutzung von Computern und somit auch von Websites. Doch all diese Hilfsmittel funktionieren nur, wenn Webapplikationen zugänglich gestaltet sind. Eine barrierefreie Website hilft jedem Menschen, egal, ob der Person ein Arm fehlt, ein Arm gebrochen ist oder sie ein Kind auf dem Arm hält. Jedem von uns sollte klar sein, dass alle Anwender temporär oder dauerhaft, bewusst oder unbewusst unter den oben aufgezeigten Barrieren leiden können.

Anforderungen

In Deutschland fordert die „Barrierefreie-Informationstechnik-Verordnung (BITV) 2.0", dass Websites, mobile Anwendungen und so weiter barrierefrei sein müssen, um niemanden von der Nutzung der Anwendung auszuschließen. Die BITV 2.0 gilt für Websites des öffentlichen Sektors seit Mitte 2021. Darüber hinaus gilt ab 2025 für nahezu alle Produkte und Dienstleistungen das Barrierefreiheitsstärkungsgesetz (BFSG, als Umsetzung der EU-Richtlinie zur Barrierefreiheit), in dem ebenfalls die barrierefreie Umsetzung von Webanwendungen gesetzlich gefordert wird. Nun steht in diesen Gesetzen nicht wörtlich, was eine barrierefreie Website ausmacht. Sie verweisen vielmehr auf die „Web Content Accessibility Guidelines" (WCAG) des W3C und fordern die Umsetzung der dort definierten „AA"-Erfolgskriterien *[2]*. Diese wiederum stellen unter anderem folgende grundlegende Anforderungen:

- Unterstützung von Screenreadern,
- Zoombare Inhalte,
- Sinnvolle Farbauswahl und Kontraste,
- Aussagekräftige Beschreibungen, Fehlertexte und Alternativtexte.

Ob eine Website die Anforderungen der BITV 2.0 erfüllt, wird mittels des „BIK BITV-Tests" *[3]* geprüft. Dieser umfasst derzeit 98 Prüfschritte, die sich unter anderem aus den WCAG ergeben. Bei einem BITV-Test wird jede Seite einer Webanwendung auf alle Prüfschritte hin untersucht, sofern die Prüfschritte auf diese Seite anwendbar sind. So werden beispielsweise keine Prüfschritte auf eine barrierefreie Einbettung von Videoinhalten angewendet, wenn die Seite keine derartigen Inhalte hat. Die Umsetzung der einzelnen Anforderungen wird mit einer aus fünf Werten („Erfüllt", „Eher erfüllt", „Teilweise erfüllt", „Eher nicht erfüllt", „Nicht erfüllt") bestehenden Bewertungsmenge angegeben. Eine Anforderung gilt als erfüllt, wenn ihre Umsetzung mindestens mit dem Wert „Eher erfüllt" bewertet wurde. Wurde auch nur eine einzelne Anforderung mit „Teilweise erfüllt" oder schlechter bewertet, so gilt die gesamte Anwendung als nicht barrierefrei!

Grundlegende Entwicklungsaspekte

Die meisten Anforderungen werden „automatisch" erfüllt, wenn man während der Entwicklung gewisse Dinge berücksichtigt und konsequent anwendet. Screenreader werden mittels des Accessibility-API des jeweiligen Browsers durch diesen automatisch angesprochen. Für Entwickler ist die genaue Funktionsweise dieses API nicht relevant, jedoch basiert sie sehr stark auf dem HTML-Code der Website. Daher ist einer der wichtigsten Punkte, um barrierefreie Websites zu erstellen, die Nutzung von syntaktisch und semantisch korrekter HTML. Dies umfasst insbesondere eine lückenlose Überschriftenhierarchie, die genaue Nutzung von Struktur- (beispielsweise einem „nav"-Element für Navigation) und Eingabeelementen sowie aussagekräftige und identifizierende Alternativtexte für Grafiken und Links. Entwickler, die zur Gestaltung Frameworks, wie zum Beispiel Primefaces, einsetzen, sollten hierbei sehr genau darauf achten, dass die von den Frameworks generierten Codefragmente ebenfalls korrekten HTML-Code erzeugen. In unseren Projekten mussten wir feststellen, dass teilweise bei Verwendung der Standardkonfiguration Tabellenelemente zur Layout-Gestaltung verwendet wurden – ein absolutes No-Go!

Ein anderer Aspekt, der bei der Nutzung von Frameworks berücksichtigt werden muss, ist die Tatsache, dass solche Frameworks oftmals Standardelemente der HTML-Spezifikation durch eine eigens erstellte Kombination von sichtbaren und nicht sichtbaren „div", „span" und/oder „input"-Elementen ersetzen, um diese vollständig grafisch an das Layout/den Styleguide der Website anpassen zu können. Hier sollte geprüft werden, ob Screenreader die Elemente korrekt erkennen und unterstützen. Mittels ARIA-Labeln („Accessible Rich Internet Applications") lassen sich Elementen Eigenschaften zuschreiben, die von Screenreadern aufgegriffen werden. So können Elemente beispielsweise als eine Suche deklariert oder als beschreibendes Element für ein Eingabeelement verknüpft werden (siehe auch Abschnitt „Umsetzungsbeispiel"). Der Einsatz dieser ARIA-Attribute sollte jedoch minimalisiert erfolgen, denn ein falscher Einsatz ist meist schlimmer als gar keine Verwendung *[4]*.

Die barrierefreie Gestaltung sollte bereits in der Designphase einer Anwendung berücksichtigt werden. So existiert beispielsweise ein Prüfschritt mit der Forderung, dass es zwei Navigationswege geben soll. Ein Navigationsweg im Sinne der WCAG ist dabei eine direkte Möglichkeit, vertikal und horizontal zu jeder Seite (die einen Arbeitsschritt startet) der Anwendung zu navigieren. Neben einem Hauptmenü kann dies zum Beispiel eine Suchfunktion sein. Eine Brotkrumen-Leiste (*Breadcrumbs*) ist kein Navigationsweg im Sinne der WCAG, da über diese nur rückwärts und vertikal in der Seitenhierarchie der Anwendung navigiert werden kann. Auch die Navigationsreihenfolge der Arbeitsschritte und Rücksprungpunkte nach Abschluss oder bei Abbruch eines Arbeitsschrittes sollte bereits in der Designphase berücksichtigt werden, sodass diese einheitlich und für den späteren Anwender vorhersehbar (Prüfschritt!) sind.

Der dritte wichtige Designaspekt ist die farbliche Gestaltung. Farben und Kontrasten kommt eine große Bedeutung zu. Bei der Auswahl der Farben müssen zwei Dinge unbedingt eingehalten werden: Zum einen darf keine Information nur durch Farbe dargestellt werden. Das bedeutet, dass beispielsweise eine Anzeige zur Erreichbarkeit eines Services, nicht nur durch ein Feld visualisiert wird, welches entweder rot oder grün ist. Farbe darf im Informationskontext immer nur unterstützend eingesetzt werden. Ist oben angesprochene Statusanzeige also beispielsweise rot und mit dem Text „offline" versehen, so ist dies in Ordnung, da farbenblinde Person die Statusinformation auf Grund der textuellen Beschreibung ebenfalls wahrnehmen können. Dies führt direkt zum zweiten wichtigen Aspekt der Farbauswahl: Kontraste. Das Kontrastverhältnis von Farben muss ausreichend hoch sein. Die WCAG definiert im „AA"-Erfolgskriterium, dass Kontraste ein Verhältnis von 4,5:1 (bei großer Schrift 3:1) haben müssen. Diverse Werkzeu-

ge und Browser-Plug-ins (zum Beispiel „Web Accessibility Evaluation Tool (WAVE)") unterstützen bei der Findung ausreichend guter Kontrastverhältnisse. Solche Plug-ins können auch diverse andere Aspekte der Barrierefreiheit (etwa die Überschriftenhierarchie oder die Struktur des HTML-Codes) untersuchen und sollten bei der Entwicklung genau wie ein Screenreader eingesetzt werden.

Die vierte grundlegende Anforderung an barrierefreie Webseiten ist ein responsives Design. Im Kontext der Barrierefreiheit ist das Ziel eines solchen jedoch nicht die Nutzung auf verschiedenen Endgeräten, sondern es soll die Nutzung der Webanwendung auch für visuell eingeschränkte Anwender sicherstellen. So fordern die Richtlinien der WCAG, dass Texte auf 200 % vergrößert werden können und sämtliche Inhalte bei 400 % Zoom korrekt in Leserichtung umbrechen müssen. Als Ausgangsbasis ist hierbei ein Viewport mit einer Breite von 1280px (400 % = 320px) definiert. Beim Zoom müssen also alle Elemente vollständig sichtbar sein, dürfen keine anderen überdecken und keine horizontalen Scrollbalken erzeugen. Die einzige Ausnahme für den letzten Punkt sind Datentabellen. Diese müssen und sollten beim Zoom nicht umbrechen, sondern dürfen mit horizontalem Scrollbalken dargestellt werden. Auch wenn solche Scrollbalken bei Datentabellen erlaubt sind, sollte die Menge an darzustellenden Spalten und Werten in einer Datentabelle immer angemessen sein und sich auf das Notwendigste beschränken. Häufig benötigt es nicht alle Attribute eines Datensatzes, um diesen beispielsweise für eine weitere Bearbeitung auf einer Detailseite zu identifizieren. Oft kann sogar auf Datentabellen zu Gunsten von Listen verzichtet werden, deren Inhalte sich meist leichter bei wechselnden Auflösungen darstellen lassen.

Umsetzungsbeispiel: Eingabeelemente

Die Anforderungen der WCAG lassen sich meist auf verschiedene Arten umsetzen, wodurch sie den Entwicklern und Designern einen großen Freiraum für individuelle Lösungen schaffen. Auf der anderen Seite schafft dies erfahrungsgemäß Unsicherheit bei Entwicklern, wie sie eine Anforderung, zum Beispiel ein Eingabeelement, korrekt umsetzen können. Im Team des Autors wurden „Quadtupel" mit vier benötigten Teilelementen verwendet:

1. Das zentrale „input"- Element, beispielsweise für einen Text,
2. Ein „label"-Element mit der Beschriftung des Eingabeelements,
3. Ein „label"-Element für den Hinweistext mit kontextbezogenen Informationen und gegebenenfalls Formatvorgaben,
4. Ein „span"-Element für den feldbezogenen Fehlertext bei Fehleingaben.

Abbildung 2 zeigt ein solches „Quadtupel" am Beispiel eines Eingabefeldes für einen Bank Identifier Code (BIC). Das folgende Quadtupel stammt aus einer Fachanwendung, bei der die Anwender wissen, wie ein BIC aufgebaut ist.

Am dazugehörigen Quellcode (*siehe Listing 1*, minimalisiert) sieht man, wie die einzelnen Elemente aufeinander zeigen. Mittels ARIA-Attributen werden zusätzliche Informationen für den Screenreader bereitgestellt, um beispielsweise Elemente zu igno-

Abbildung 2: Umsetzungsbeispiel eines „Quadtupels" zur Eingabe eines BIC.

```
<div>
  <label id="txt_bic:label" for="txt_bic:input">
     <span>BIC:</span>
  </label>
  <label id="txt_bic:hint">
     <span>
        <i aria-hidden="true"/>
        <span class="visually-hidden">Formularhinweis</span> Bank Identifier Code gemäß ISO 9362. Exakt 8 oder 11
Zeichen, alphanumerisch.</span>
  </label>
  <div>
     <div class="form-input"><input id="txt_bic:input" type="text" value="DSS" aria-invalid="true" aria-
labelledby="txt_bic:label" aria-describedby="txt_bic:hint"></div>
     <div id=txt_bic:message">
        <div><span id="txt_bic:message_error-summary">Fehler: BIC: "DSS" ist kein gültiger BIC. (BIC01)</span></div>
     </div>
  </div>
</div>
```

Listing 1: Quellcode zum Umsetzungsbeispiel „Quadtupel"

rieren (`aria-hidden`) oder als Beschreibung vorzulesen (`aria-describedby`).

Während die Gestaltung des Eingabe-, des Beschriftungs- und des Fehlerelements für die meisten Entwickler nachvollziehbar ist, ist dies beim Hinweistext nicht immer der Fall. Gemäß den Anforderungen der Barrierefreiheit müssen die im Hinweistext angegebenen Informationen vor dem Eingabeelement und während der gesamten Eingabe sichtbar sein. Somit ergibt sich die oben zu sehende Positionierung. Die Verwendung eines „Placeholder"-Textes (auch „Watermarks" genannt) im Eingabeelement ist ungeeignet, da dieser während der Eingabe nicht mehr zu sehen ist. Die Formulierung des Hinweistextes sollte sich an der Zielgruppe orientieren. Richtet sich die Webanwendung an Fachanwender, können hier spezifischere Angaben gemacht werden als bei Internetanwendungen für jedermann.

Fazit

In diesem Artikel wurde gezeigt, dass es vielfältige Arten von Einschränkungen und Barrieren gibt und diese auch alle im Umfeld von Webanwendungen zum Tragen kommen können. Barrierefreie Websites sind nicht nur gesetzlich vorgeschrieben, sondern helfen allen Anwendern bei der Nutzung. Die Anforderungen an barrierefreie Websites ergeben sich aus der BITV 2.0 sowie dem BFSG und lassen sich im Kern auf die „AA"-Erfolgskriterien der WCAG zurückführen. Deren Umsetzung wird durch konsequente Anwendung der HTML-Standards und einiger Designaspekte bereits größtenteils erreicht, sodass erfahrungsgemäß maximal vereinzelte Spezialfälle individuell betrachtet werden müssen.

Quellen

[1] Bundesrepublik Deutschland (2023): § 2 Abs. 1 Satz 1 SGB IX. *https://www.gesetze-im-internet.de/sgb_9_2018/__2.html*, zuletzt aufgerufen 2023-11-23.

[2] DIAS GmbH (2023): Die BITV 2.0 – Was prüft der BITV-Test, was prüft er nicht. *https://www.bitvtest.de/bitv_test/das_testverfahren_im_detail/vertiefend/die_bitv_20_was_prueft_der_bitv_test_was_prueft_er_nicht.html*, zuletzt aufgerufen: 2023-11-19.

[3] *https://www.bitvtest.de/bitv_test.html*

[3] Mozilla Foundation (2023): ARIA. *https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA*, zuletzt aufgerufen: 2023-11-23.

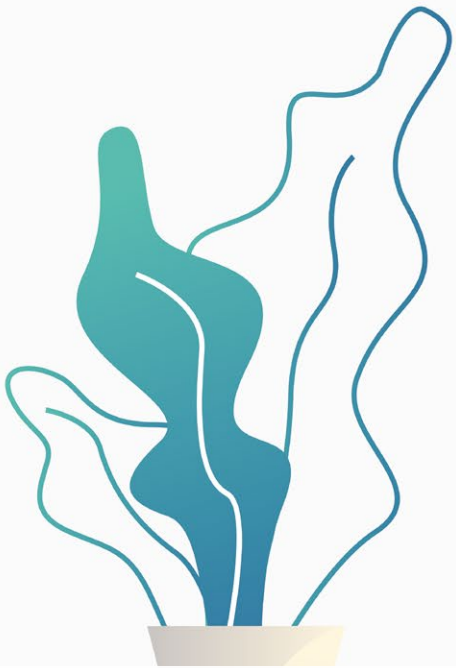
Matthias Bünger

runningjava@web.de

Matthias Bünger ist Softwareentwickler, Architekt und Teamleiter mit einer Vorliebe für Java und Testen. In seiner Freizeit geht er gerne laufen, auf Konzerte, ins Kino oder spielt Karten und Brettspiele, besonders gerne Teamspiele.

Wie finden Quereinsteiger den Weg in die Softwareentwicklung?

Andreas Monschau, Haeger Consulting



In Zeiten des Fachkräftemangels können Quereinsteiger eine gute Möglichkeit sein, um dieses Problem zu lösen. In den letzten zehn Jahren haben wir bei Haeger Consulting viel Zeit und Ressourcen in den Aufbau eines entsprechenden Traineeprogramms gesteckt, mit erstaunlichen Erfolgsstories. „Wie macht ihr das eigentlich?", ist eine der vielen Fragen, die uns diesbezüglich gestellt werden. Ja, wie machen wir das eigentlich?

Im Rahmen dieses Artikels kann natürlich kein kompletter Bauplan für ein Traineeprogramm beschrieben werden, aber ich werde einen Überblick geben, welche Art von Kandidaten wir suchen, welche Eigenschaften sie mitbringen sollten, wie unser grober Lehrplan aussieht, wie man den Leistungsstand ermittelt, und welche Erfahrungen wir gemacht haben – sowohl die guten als auch die schlechten.

Zu Beginn etwas Historisches: Das Traineeprogramm, so wie wir es heute kennen, ist nicht in einem Stück vom Himmel gefallen, sondern hat sich mühsam entwickelt. Begonnen hat alles mit mir selbst als „Quereinsteiger" in das Java-Ökosystem. Ich hatte zwar einen Delphi/.net-Background, aber das, was ich bis dahin gemacht habe, kann man nur mit sehr viel gutem Willen als „professionelle Softwareentwicklung" bezeichnen. Insofern war ich zwar nicht völlig fachfremd und kein Quereinsteiger im klassischen Sinne, aber ich war von Java so weit weg wie der Mond von der Erde. So kam es eines Tages, dass ich zum Vorstellungsgespräch bei Ralf Haeger eingeladen war, und dort von einem Mitarbeiter (mit der Personalnummer 1 und heute noch bei Haeger Consulting) ordentlich gegrillt wurde. Nachdem wir mit dem Interview fertig waren und wir uns verabschiedeten, fragte Ralf besagten Mitarbeiter: „Und, kann er was?" Die Antwort: „Nichts von dem, was wir brauchen." Daraufhin fragte Ralf jedoch: „Aber glaubst du, er kann das lernen?" „Vermutlich schon", war die Erwiderung und damit der Startschuss, dass ich mir erstmal alles selbst beibringen durfte. Zum Glück war ich damit einigermaßen erfolgreich. Zumindest so erfolgreich, dass ich nach einem Jahr gefragt wurde: „Andreas, hast du Lust, einen Trainee zu haben, dem du Java beibringst?" Da konnte ich natürlich nicht nein sagen und damit war die Saat für unser Traineeprogramm gelegt. Liest sich doch alles sehr planvoll.

Wen wir suchen

Aber welche Art Quereinsteiger suchen wir? Über welche Eigenschaften muss diese Person verfügen? Wir haben über die Jahre die steile These aufgestellt, dass zirka 95 % der Menschen den Anforderungen, als Quereinsteiger in die IT zu kommen, nicht gewachsen sind (eine bittere Lehre aus Erfahrungen mit Trainees, bei denen es nicht funktioniert hat). Aber wir versuchen ganz gezielt, die anderen 5 % zu finden und sie so weit zu befähigen, dass sie in überschaubarer Zeit fit für ihr erstes Java-Projekt werden. Also kommen wir zu einer Auswahl an Eigenschaften, die sie unabdingbar brauchen:

Neugier ist ein wichtiges Merkmal. Man muss wissbegierig auf Neues sein, offen für Neues. Neugierde, etwas völlig Neues, Unbekanntes zu lernen. Darüber hinaus benötigt man eine gewisse intrinsische **Motivation**: Im Idealfall hat man Spaß am Gedanken, Software zu entwickeln. Es ist auch notwendig, dass man **Eigenständigkeit** mitbringt, also die Fähigkeit, sich selbstständig Wissen anzueignen. Wir können nicht beibringen, wie man lernt, wir können nur aufzeigen, was man lernen muss. Des Weiteren muss man **belehrbar** sein, das heißt, man sollte in der Lage sein, Kritik und Anmerkungen anzunehmen und dann auch umsetzen zu können.

Natürlich soll man ebenso immer über „analytisches und abstraktes Denkvermögen", wie es in Stellenausschreibungen so gerne heißt, verfügen. Wenn man sich die genannten Punkte mal genau anschaut, merkt man aber auch, dass nichts Unmögliches verlangt wird – all das sind Attribute, die man auch braucht, wenn man ein Studium oder eine duale Ausbildung im IT-Bereich absolvieren will.

Das Spektrum der Menschen, bei denen diese Merkmale zutreffen, ist sehr breit – vom ehemaligen Steuerfachangestellten über langjährige Studenten hin zu Menschen mit Promotion ist alles dabei. Wichtig ist für uns nur, dass wir verstehen, was sie antreibt, und erkennen, dass die oben genannten Merkmale gut ausgebildet sind.

Volle Kraft voraus – Journey 2 Junior

Wir sprechen von einem Zeitraum, der ungefähr sechs Monate umfasst, wenn wir über unser Traineeprogramm reden. Daher muss sehr genau definiert sein, was in dieser Zeit erreicht werden kann (und was nicht). Wir haben einen Lehrplan entwickelt, der die Inhalte sowie die Leitplanken links und rechts vorgibt: Die Journey 2 Junior, kurz J2J.

Wissen will kontinuierlich aufgebaut werden. Über den genannten Zeitraum bauen wir, Baustein für Baustein, Kapitel für Kapitel, das Wissen auf, wobei jedes Kapitel jeweils als Grundlage für das nächste Kapitel dient.

Grundsätzlich haben wir drei J2J-Kapitel definiert, in denen wir alle Technologien, Tools und Methoden hinterlegt haben, die wir als absolute „Must-haves" betrachten *(siehe Abbildung 1)*.

Wir starten mit dem Kapitel „**Grundlagen**". Hier holen wir die Trainees bei ihren individuellen Kenntnissen ab und versuchen, sie alle auf ein Level zu heben. Dabei lernen sie Konzepte wie die objektorientierte Programmierung kennen und beschäftigen sich intensiv mit Standard-Java, Datenbanken (und Zugriffe auf diese) sowie dem Thema Testing als auch Versionsverwaltung mit Git, um eine Auswahl zu nennen.

Das zweite Kapitel „**Frameworks, Architekturen, Schnittstellen**" ist meist ein Selbstläufer, wenn die Trainees zuvor die Grundlagen verinnerlicht haben. Sie lernen nun wichtige Teile des Spring-Frameworks kennen, beginnen zu verstehen, was REST bedeutet, steigen tiefer in das Thema Persistierung ein oder beschäftigten sich mit Architekturen, Patterns und vielem mehr. Gerade im Kontext „Frameworks" geschehen nun prägende Aha-Erlebnisse. Hier könnte der Gedankengang eines Trainees wie folgt lauten: „Oha, hier gibt es etwas, dass mir Dinge, die ich bisher mühsam per Hand machen musste, automagisch abnimmt. Und weil ich sie bisher per Hand machen und verstehen musste, verstehe ich jetzt umso mehr, was da nun passiert, und es ist eben nicht nur Magie!"

Im letzten Kapitel „**Build und Betrieb**" geht es nun um Themen wie CI/CD, Containertechnologien, Deployment in die Cloud oder Ähnliches. Auch hier gibt es Aha-Erlebnisse, diese manifestieren sich eher in der Art „Ach, sowas geht?" oder „Ach, das ist ja einfach!".

Ein umfangreicher Anwendungsfall begleitet unsere Trainees während der Journey: Eine Anwendung, die sie stetig erweitern und ändern. Während ihrer Reise werden sie jedoch nicht allein gelassen – kompetente Trainer unterstützen sie bei ihrem Bestreben, Junior-Entwickler zu werden. Zu den Aufgaben der Trainer gehört es, stets Feedback zu geben, sie bei Fragen zu unterstützen, Aufgaben zu klären, Challenges zu reviewen (dazu später mehr) und auch während der Bearbeitung eines Kapitels die individuelle Leistung zu bewerten.

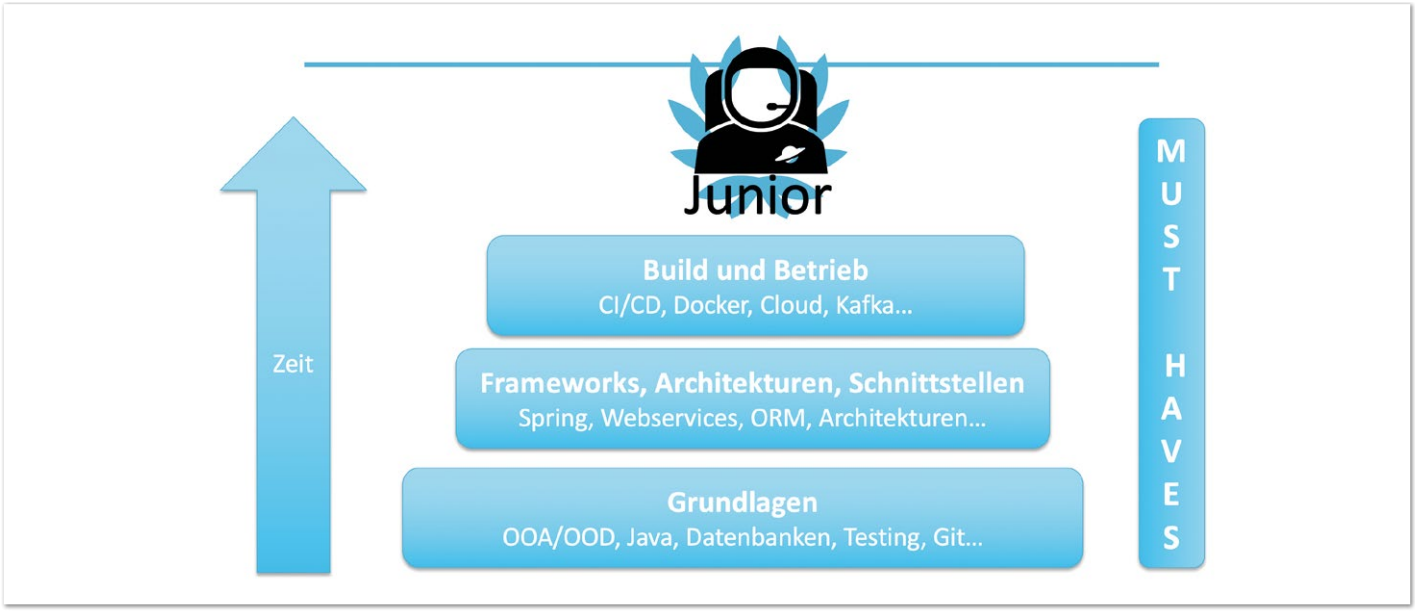


Abbildung 1: Überblick über die „Must-haves" unserer Journey 2 Junior

Tauscht euch aus!

Wir ermutigen unsere Trainees in den intensiven Austausch zu gehen – zum einen sollen sie ihr Wissen weitergeben, zum anderen sich mit anderen Menschen vernetzen.

Ihr Wissen geben sie in sogenannten „Mini Skill Factories" weiter – einstündige Vorträge zu einem bestimmten Thema entweder passend zu den Inhalten der J2J oder tagesaktuell (beispielsweise zuletzt „Java 21", was jede JUG bringt können wir durchaus auch machen). In der Zuhörerschaft befinden sich dann Trainees aller Disziplinen, Alumni sowie auch weitere interessierte Kollegen bis hin zum Senior-Level.

Darüber hinaus finden drei Mal im Jahr sogenannte „Trainee Events" statt. Hier treffen sich die aktuelle Trainee-Generation, Trainer sowie auch Alumni, also diejenigen, die das Traineeprogramm bereits erfolgreich absolviert haben. Auf der einen Seite steht hier das Vergnügen auf dem Programm (etwa in Form von Minigolf, Lasertag, gemeinsamem Pizzaessen...), auf der anderen Seite aber auch der konkrete Austausch zwischen Trainees und Alumni. Die nächste Stufe ist dann auf der Ebene der Firmenevents, bei denen wir den Austausch zwischen Trainees und allen Mitarbeitern auf allen Erfahrungsebenen fördern. Es kann durchaus wichtig sein, zu wissen, wen von den „alten Hasen" man fragen kann, wenn ein Problem mit einer Technologie oder ähnlichem auftaucht.

Zusätzlich freuen wir uns auch darüber, wenn unsere Trainees an Meetups wie Treffen von Java User Groups teilnehmen und sich dort mit Menschen aus der Szene vernetzen.

Alles eine Frage der Herausforderung

Es geht aber nicht nur um Pizzaessen, netten Austausch und Vernetzung. Von Zeit zu Zeit muss der individuelle Leistungsstand, beziehungsweise der Fortschritt, innerhalb der J2J evaluiert werden. Dazu stehen verschiedene Instrumente zur Verfügung, wobei ich hier vor allem auf eines eingehen möchte: unsere Challenges.

Am Ende jedes Blocks unserer Journey 2 Junior *(siehe Abbildung 1)* platzieren wir eine Aufgabe mit einer überschaubaren Fachlichkeit, zu deren Lösung mindestens die Technologien, Tools und Methoden verwendet werden müssen, die man im zuvor bearbeiteten Block gesehen und verstanden haben muss. Exemplarisch dafür möchte ich die Challenge (vereinfacht) vorstellen, die am Ende des ersten Blocks absolviert werden muss.
Grundsätzlich sollen unserer Trainees das Verhalten eines Bankkontos implementieren, dazu gibt es von unserer Seite neben einem erklärenden Schaubild auch ein paar Anforderungen:

- Der Kunde kann einen Geldbetrag auf ein Bankkonto einzahlen (überweisen).
- Der Kunde kann einen Geldbetrag von einem Bankkonto abheben.
- Der Kunde kann sich den aktuellen Stand seines Kontos zurückliefern lassen.
- Der Kunde kann sich alle Transaktionen, die auf seinem Konto angefallen sind, zurückliefern lassen.
- Der Kunde kann sich Kontoauszüge erzeugen lassen, die alle Transaktionen enthalten, die zwischen einem Start- und inklusivem Enddatum erfolgt sind.

Aus dieser Auflistung ergeben sich bereits die ersten Klassen und Use-Cases. Zur Lösung der Aufgabe benötigt der Trainee dann das, was er im ersten Journey-Block gelernt hat, das sind unter anderem:

- Die Grundlagen objektorientierter Programmierung,
- Standard-Java,
- Maven,
- Strukturformate,
- Git,
- Testen mit JUnit,
- Datenbankzugriff mit Java.

Wir lassen das Thema GUI hier bewusst außen vor, da wir es nicht für sinnvoll halten, dass heute noch jemand mit AWT und Swing Benutzeroberflächen baut, und andere Themen, die wir als „Out of Scope" deklarieren.

Diese Challenge ist zeitlich auf zwei Personentage begrenzt und soll von einem Trainee allein bearbeitet werden. Nach Ablauf des Bearbeitungszeitraums wird auch der Zugriff auf das Git-Repository für den Trainee entfernt, sodass hier keine Chance mehr besteht, nach Feierabend noch daran weiterzuarbeiten. Im Anschluss wird die Lösung durch einen Trainer intensiv ausgewertet. In dieser Zeit hat der Trainee die Möglichkeit, sich Bonusthemen anzuschauen. Nach erfolgter Review kommt es zum Austausch zwischen Trainer und Trainee und es wird genau ermittelt, wo noch Lücken bestehen und woran noch nachgearbeitet werden muss. Oder es stellt sich heraus, dass alles nach Plan läuft. Wie dem auch sei, der Trainee wechselt nun in das nächste Kapitel, hat dort aber dann auch die Möglichkeit, die identifizierten Lücken und Probleme zu füllen, beziehungsweise zu beheben.

Erfahrungen, die gemacht werden müssen

Im Laufe der Jahre haben wir viele Erfahrungen gesammelt. Es läuft nicht immer alles glatt, wenn man ein solches Traineeprogramm aufbaut, aber vieles hat sich als bemerkenswert gut herausgestellt.

Über die Jahre hinweg haben fast 30 unserer Mitarbeiter das Trainingsprogramm in der einen oder anderen Form durchlaufen und viele der Kollegen sind heute noch bei uns. Auch die Abbrecherquote ist drastisch gesunken – aktuell ist es so, dass es im Schnitt nur ein Trainee nicht schafft, das Traineeprogramm zu Ende zu bringen. Sehr interessant für uns ist, dass Unternehmen (auch Mitbewerber!) auf uns zukamen und die Frage stellten: „Wie macht ihr das überhaupt mit eurem Traineeprogramm?" Da sich diese Fragen häuften, haben wir beschlossen, mittelfristig auch den Aufbau von Trainingsprogrammen nach unserem Muster als Dienstleistung anzubieten. Wir sind auf die ersten Case-Studies gespannt!

Schlecht lief es in der Vergangenheit, wenn ein Trainee-Verhältnis aufgrund gegenseitig nicht zusammenpassender Erwartungshaltung aufgelöst werden musste, weil dann oftmals ein sinnvolles Zusammenarbeiten auf lange Sicht unmöglich ist. Wir haben daraus gelernt und betonen unsere Erwartungshaltung an Trainees möglichst früh (im Idealfall bereits beim ersten Kennenlernen, aber spätestens kurz vor Einstellung), fordern aber auch von den Trainees in spe ein, dass sie ihre Erwartungshaltung an uns formulieren sollen, beziehungsweise wir definieren klar, was von uns im Rahmen des Traineeprogramms zu erwarten ist.

Man lernt kontinuierlich und ebenso kontinuierlich erweitern und verbessern wir unser Traineeprogramm. „Schlechte" Erfahrungen sind Schritte zum stetigen Wachsen und niemals ein Rückschritt.

What's next?

Mit unserem Traineeprogramm für Java sind wir schon seit einigen Jahren unterwegs. Aktuell bauen wir Ähnliches auch für die Bereiche Softwaretest, DevOps, UI/UX und Frontend auf. Wir wollen versuchen, die Erfahrungen, die wir gewonnen haben, und die Strukturen, die wir ermittelt haben, dort direkt anzuwenden, um so noch besser Synergieeffekte zwischen den Bereichen identifizieren und nutzen zu können. Ein Beispiel: Zukünftig kann es eine J2J für den Bereich UI/UX und Frontend geben, die ebenfalls etwa sechs Monate dauert. Warum kann man nicht einen siebten Monat dranhängen und dort etwas im Bereich Java-Backend und Backend-Technologien im Allgemeinen vermitteln? Was hält den Java-Bereich davon ab, auch Frontend-Anteile zu lernen? Oder formale Themen aus dem Softwaretest?
Die Möglichkeiten scheinen schier unbegrenzt.

Wozu das alles?

Aber wozu macht man das Ganze? Möchte man nur günstig neue Entwickler produzieren oder steckt da vielleicht doch etwas mehr dahinter? Daher möchte ich auch meine persönlichen Eindrücke teilen: Über die Jahre hinweg haben wir einige bemerkenswerte Erfolgsgeschichten unserer Trainees begleitet – da ging es um Menschen, die in beruflichen oder privaten Krisen steckten und nicht mehr weiterkamen. Menschen, die für den Arbeitsmarkt fast schon verloren waren, oder Menschen, die viele Jahren im Studium festhingen, aber sich nie angekommen fühlten. Wir haben gemeinsam eine Möglichkeit erkannt, das Potenzial identifiziert und sind dann durch das Traineeprogramm gegangen. Nun sind sie teilweise in Lead-Rollen in ihren Projekten und das hätte ihnen zuvor oftmals keiner zugetraut (manchmal auch sie sich selbst nicht). Man kann etwas bewegen und Menschen (wieder) eine Perspektive geben. Oder man erfüllt ihnen den Wunsch, in der IT und dort insbesondere in der Softwareentwicklung mit Java Fuß zu fassen. Es wird in den meisten Fällen mit Loyalität belohnt, die wir genauso zurückgeben. Unsere Company-DNA lautet „Menschen entwickeln" und damit werden wir konsequent weitermachen.

Andreas Monschau

amonschau@haeger-consulting

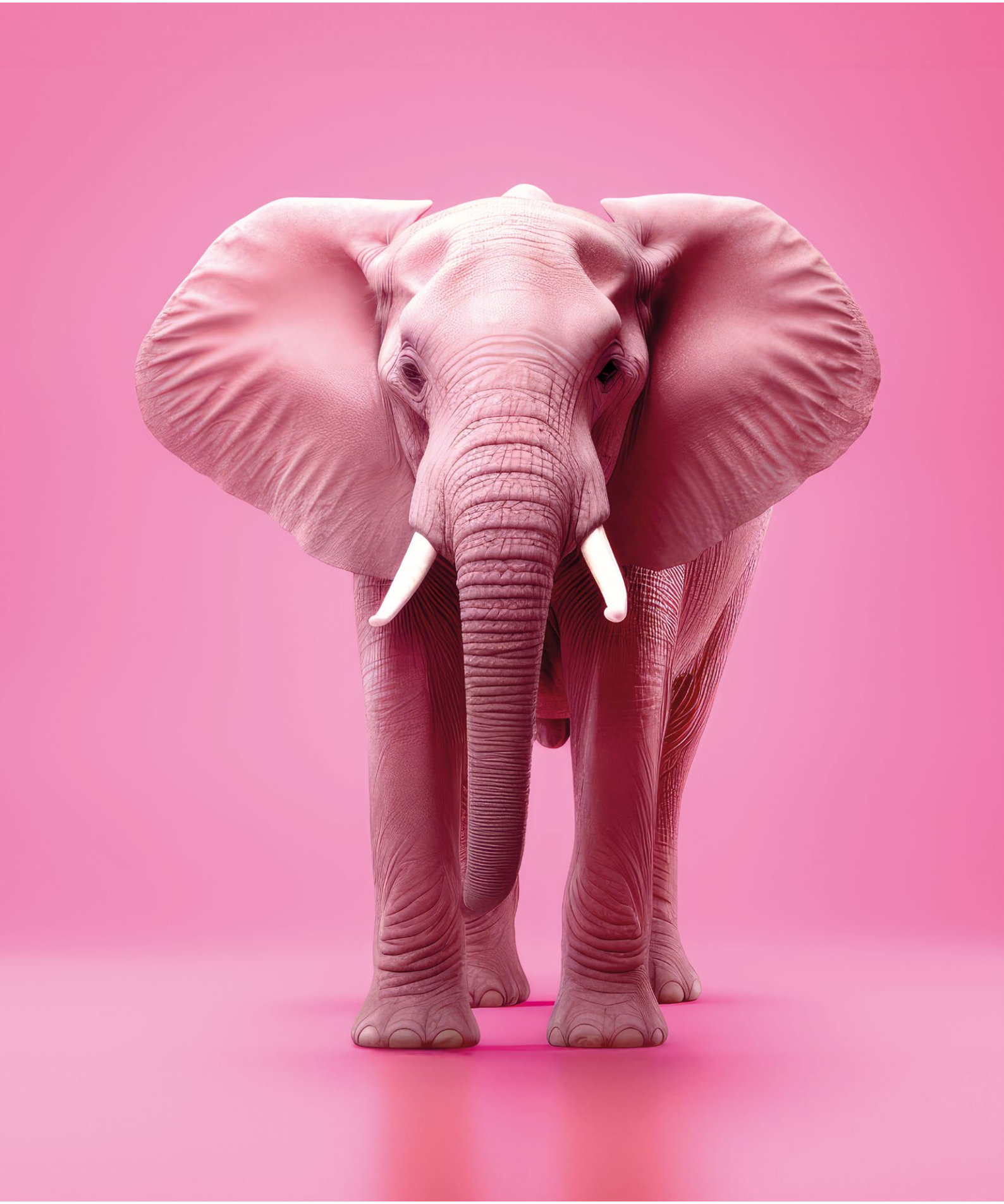
Andreas Monschau ist seit über zehn Jahren als Senior IT-Consultant mit den Schwerpunkten Softwarearchitektur- und Entwicklung sowie als Teamleitung bei Haeger Consulting in Bonn tätig und aktuell als Solution Designer im Kundenprojekt unterwegs. Neben seinen Projekten leitet er das umfangreiche Traineeprogramm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


CloudLand
WWW.CLOUDLAND.ORG
Eventpartner: Heise Medien
DAS CLOUD NATIVE FESTIVAL
DAS EVENT DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY
18. – 21. JUNI
#CLOUDLAND2024

Psychologie für Softwareentwickler:innen

Berend Semke, Bredex GmbH

Das menschliche Gehirn ist ein Datenverarbeitungssystem. Unsere Psyche ist die Software, die auf diesem System läuft. Als Softwareentwickler:innen sind wir geübt darin, komplexe Softwaresysteme zu beschreiben und haben damit alle Werkzeuge an der Hand, um uns selbst besser zu verstehen. Ich wollte gerne verstehen, weshalb wir bei dem Versuch scheitern müssen, nicht an einen rosaroten Elefanten zu denken. Dafür habe ich mich an einem Reverse-Engineering der menschlichen Psyche unter Verwendung des C4-Modells versucht.



In den letzten Jahrzehnten hat sich in der Kommunikation zwischen Eltern und Kindern eine Evolution vollzogen, die in der breiten Öffentlichkeit kaum diskutiert wurde. Als ich vor 35 Jahren das Fahrradfahren erlernte und einfach strukturierte Bäume erklomm, wurden mir Ausrufe wie „Nicht auf die Straße!" und „Fall nicht runter!" entgegengerufen. 20 Jahre später fand ich mich in der Situation des Vaters wieder und legte mir bewusst Sätze wie „Schau nach vorne!" und „Halt dich gut fest!" zurecht, um im entscheidenden Moment meine Kinder mit einem lösungsorientierten Ausspruch zu unterstützen.

Wenn ich Risiken auf meine Kinder zukommen sehe, überlege ich mir eine spezifische Handlungsempfehlung, mit der die mir anvertrauten Menschen in der akuten Situation das Risiko selbstständig beherrschen können. Wer mit offenen Ohren durch Neubaugebiete spaziert geht, wird feststellen, dass ziemlich viele Eltern heute positiv formulieren – zumindest solange es darum geht, sich mit dem Fahrrad sicher fortzubewegen.

Die Wirksamkeit positiver Formulierungen beschäftigt mich, seitdem ich als Teenager eine Trainerausbildung absolviert habe. Damals wurde ich mit den Grundregeln des Coachings vertraut gemacht. „Denk nicht an den rosaroten Elefanten funktioniert nicht!", war ein Mantra in dieser Ausbildung, dass sich anschließend zu einer meiner Lebensweisheiten entwickelt hat. Es bezog sich darauf, dass ein Coach in der sportlichen Wettkampfsituation Fehler seiner Schützlinge erkennen muss. Es ist verlockend, dem Schützling anschließend genau dieses Verhalten zu verbieten. Mit verheerenden Folgen.

Ein guter Coach erkennt Fehler und spricht im Coaching über Lösungen. Wenn mir meine Mutter „*Fall nicht vom Baum!*" entgegenruft, denke ich darüber nach, was ich machen müsste, um zu fallen und welche Schmerzen ich dann wohl erleide. Wegen der Warnung vor dem Sturz erhöht sich die Wahrscheinlichkeit, tatsächlich zu fallen: eine sich selbst erfüllende Prophezeiung. Ruft meine Mutter stattdessen „*Halt' dich gut fest!*", suche ich nach einem stabilen Ast zum Greifen – sofern sie mir aufrichtig zutraut, der Situation gewachsen zu sein und keine Angst in ihrer Stimme liegt.

Seit 20 Jahren bin ich als Softwaretester, Testmanager, Softwareentwickler, Softwarearchitekt, Projektleiter, Teamleiter, Standortleiter und Geschäftsleiter bei mittelständischen IT-Beratungsunternehmen tätig. Die meiste Zeit hatte ich direkten Zugang zu unterschiedlichen Entscheidungsträgern. Ich stelle fest: Diese Evolution in der Kommunikation hat die Arbeitswelt bisher kaum erreicht. Im beruflichen Kontext beobachte ich viele Führungskräfte, die die Chance ungenutzt verstreichen lassen, als Coaches zu agieren. Stattdessen wird kommuniziert, wie es Eltern der 80er Jahre und des letzten Jahrhunderts taten.

Dabei wird in einem agilen Team jedes Teammitglied früher oder später zum Coach. Jeder von uns verfügt über Spezialfähigkeiten und besondere Kenntnisse, von denen der Rest des Teams profitieren kann. Deshalb glaube ich, dass jedes Team besser funktioniert, wenn sich die Teammitglieder angewöhnen, im Lösungsraum zu coachen. Mir hat es dabei geholfen, mein Gehirn (beziehungsweise das zentrale Nervensystem) als Datenverarbeitungssystem vorzustellen. Zur Orientierung in so einem System nutze ich am liebsten das C4-Modell von Simon Brown *[1] [2]*.

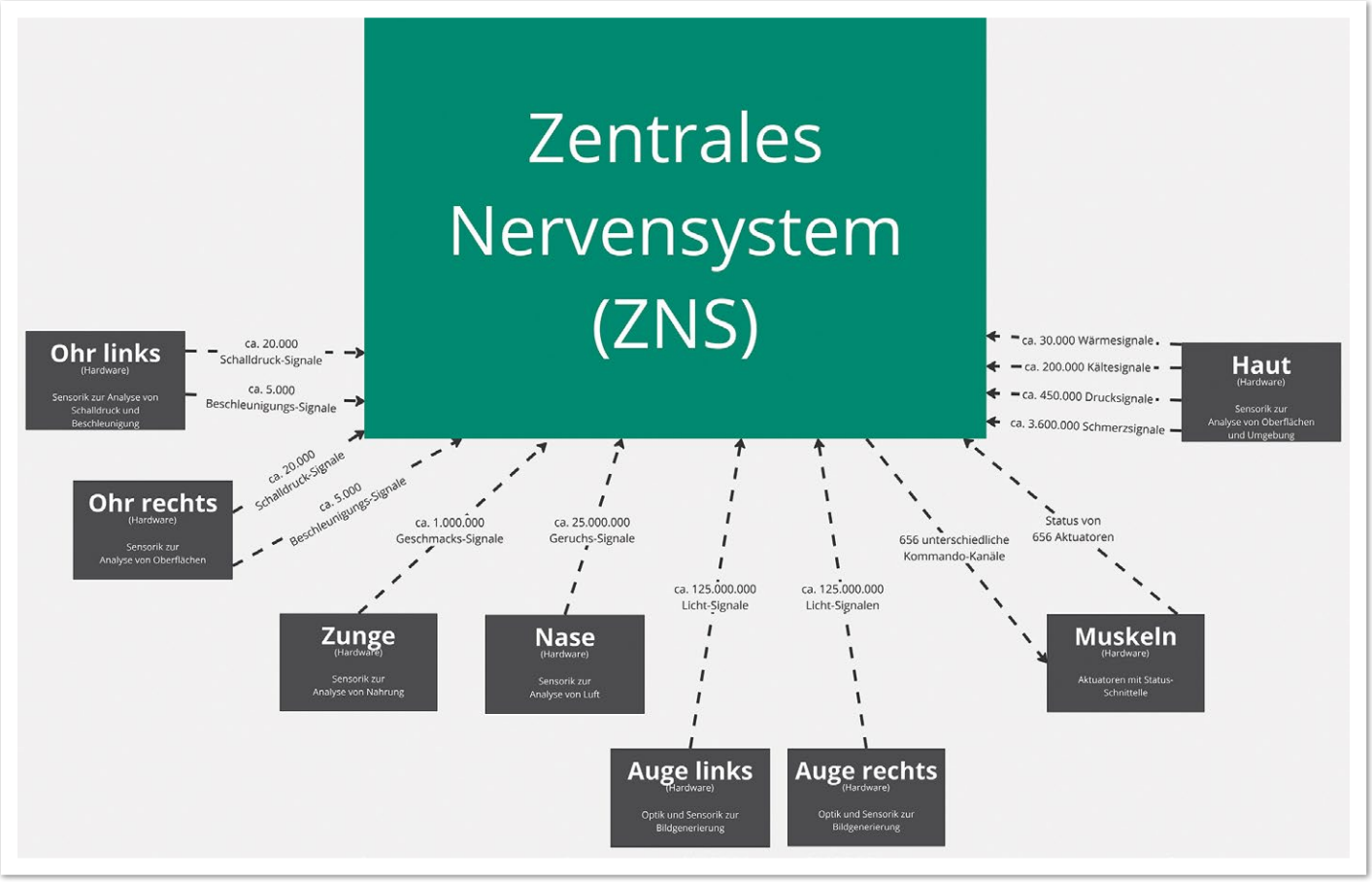


Abbildung 1: Das zentrale Nervensystem interagiert mit Peripherie

(C1) – Kontext-Diagramm des zentralen Nervensystems (ZNS)

Den Einstieg in die Architektur eines komplexen Datenverarbeitungssystems liefert Browns Kontext-Diagramm. Dieser Schritt hilft bei der Fokussierung auf das Wesentliche. Mein Kontext-Diagramm des ZNS *(siehe Abbildung 1)* zeigt vor allem Schnittstellen zu diversen Sensoren (Haut, Augen, Ohren, Nase, Zunge) und Aktuatoren (Muskeln). Die Interaktion mit Objekten der Umwelt (beispielsweise anderen Lebewesen) gehört meines Erachtens auf eine höhere Abstraktionsebene.

Wegen der Ignoranz der Schnittstellen zu inneren Organen ist mein Diagramm unvollständig. Da ich lediglich dazu beitragen möchte, eine grundsätzliche Vorstellung vom Aufbau unseres ZNS aufzubauen, akzeptiere ich diese Ungenauigkeit bewusst zugunsten einer besseren Übersichtlichkeit.

(C2) – Container-Diagramm des ZNS

In der ersten Gliederungsebene wird das Gesamtsystem in Container aufgeteilt *(siehe Abbildung 2)*. Dabei sollten die einzelnen Container funktional möglichst klar umrissen sein und es sollte deutlich werden, in welcher Umgebung der Code später zur Ausführung kommt („*Where does the code live?*").

Bei der Anordnung der Container orientiere ich mich grob an dem ISO/OSI Schichten-Modell *[3]*. Unten findet die Datenübertragung statt, während weiter oben abstrakte, funktionale Features realisiert werden. Dabei kann der Code in zwei unterschiedlichen Umgebungen zur Ausführung kommen:

1. Während das Kleinhirn mit wenig Langzeitspeicher ausgestattet ist, bietet es Ressourcen für eine hohe Parallelisierung und ermöglicht so die schnelle Vorverarbeitung immens großer Datenströme.
2. Demgegenüber steht das Großhirn, ein eher langsames System, das dafür mit riesigen Speicherkapazitäten, diversen Assoziationsmodellen und einer beeindruckenden Simulationsumgebung ausgestattet ist.

Angesichts der äußeren Parameter erscheint es folgerichtig zu sein, im Kleinhirn einen *Hardware Abstraction Layer* (HAL) zu betreiben. Dieser kapselt das Handling der Sensordaten von eher komplexeren, abstrakten Features. Dabei kümmern sich die Komponenten des HAL auch um die Lebenserhaltung, eine initiale Fehlerkorrektur sowie die Identifikation lebensgefährlicher Situationen. Wenn es mal schnell gehen muss, werden außerdem Sofortmaßnahmen zur Lebenserhaltung eingeleitet.

Im Großhirn sehe ich mit der Handlungsplanung, dem Data-Model-Management, der Sensorik und der Motorik vier wesentliche Container auf Applikationsebene:

- Unser Bewusstsein wird durch Komponenten des Handlungsplanungs-Containers gebildet.
- Der Sensorik-Container wirkt wie ein Load-Balancer und verwaltet Filterregeln, um dem Bewusstsein lediglich einen relevanten Ausschnitt des massiven Datenstroms zur Verfügung zu stellen.
- Der Motorik-Container ermöglicht das bewusste Erlenen neuer Bewegungsabläufe, so dass diese später als „stored procedures" in tieferen Schichten en bloc adressiert werden können.

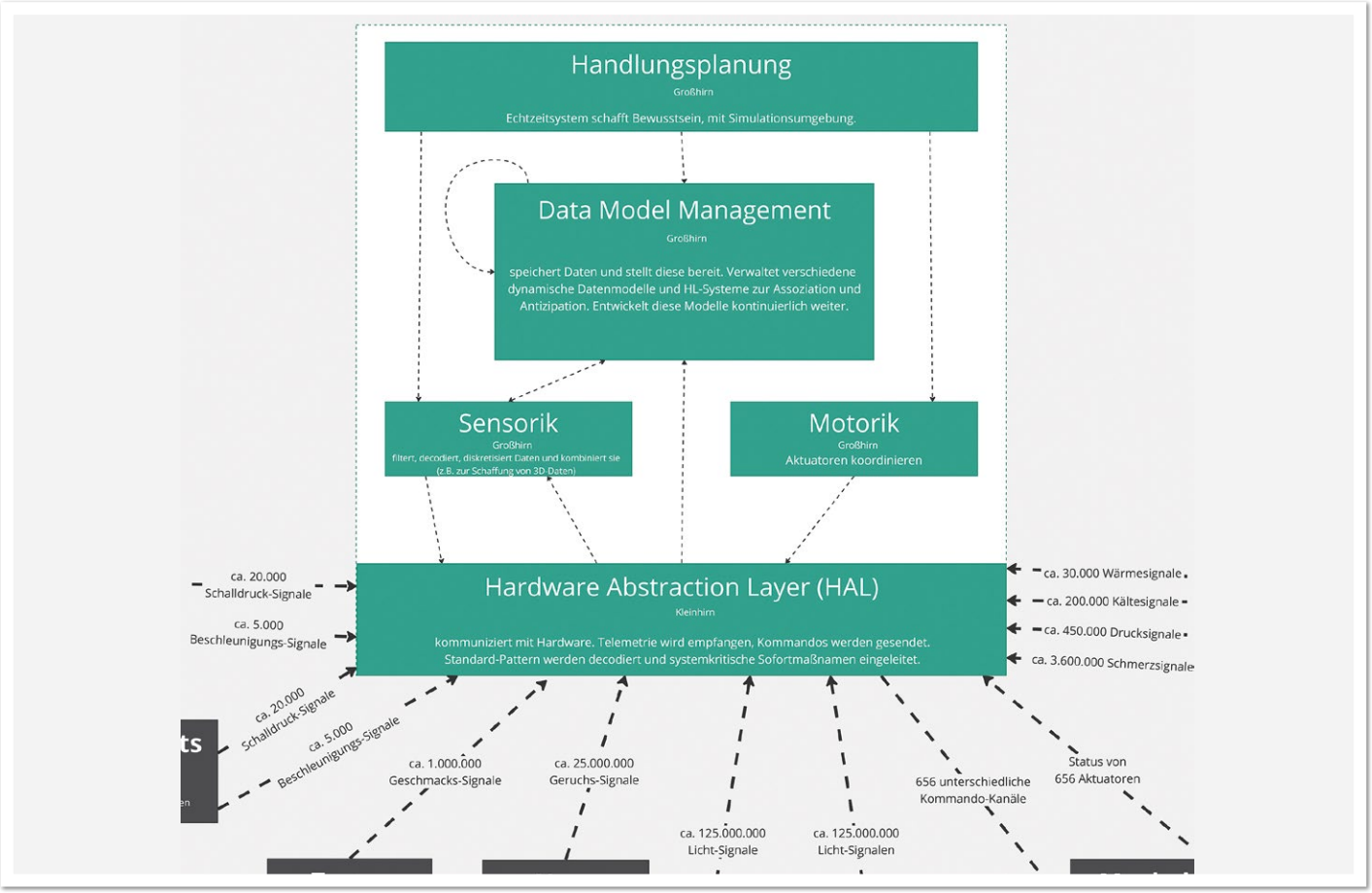


Abbildung 2: Data Model-Management als zentraler Container des ZNS

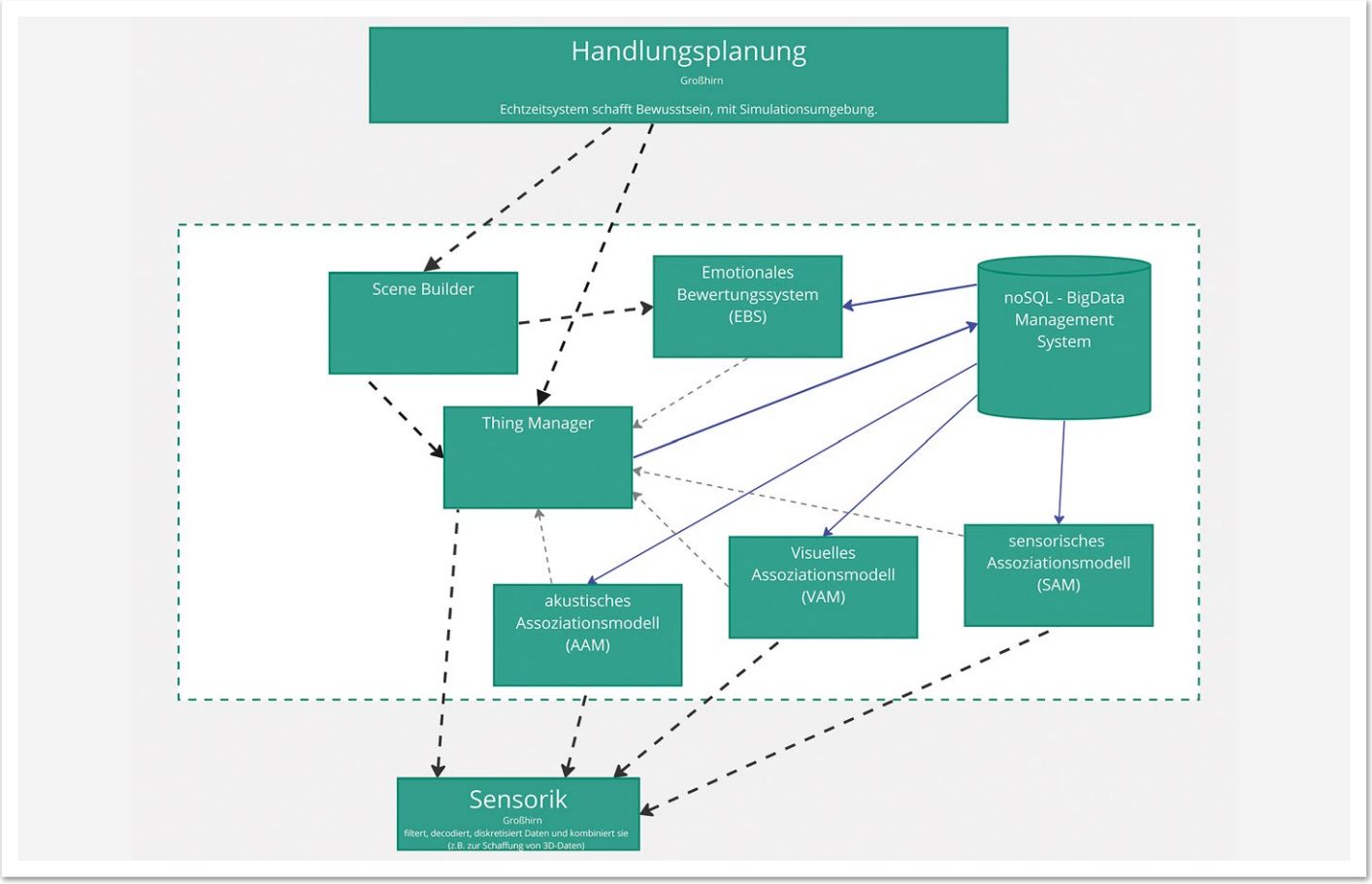


Abbildung 3: Das Component-Diagramm des Data-Model-Managements zeigt diverse Assoziationsmodelle sowie eine mächtige Persistenz-Komponente

- Für die Datenverarbeitung nimmt das Data-Model-Management eine zentrale Rolle ein. Hier werden die Datenstrukturen für die anderen Container definiert, Informationen persistiert und assoziiert.

Imagination und Erleben

Für das Reverse-Engineering der Komponenten des Data-Model-Management ist es hilfreich zu wissen, dass unsere Psyche kaum zwischen Erleben und Imagination unterscheidet.

Menschen verfügen über die Fähigkeit, sich die Zukunft vorzustellen. Wir können Handlungsalternativen ausprobieren, ohne die damit verbundenen negativen Konsequenzen tragen zu müssen. Ich kann mir vorstellen, was ich mir anschaffen würde, wenn ich mehreren Millionen Euro reicher wäre. Ich kann mir auch vorstellen, wie ich eine Bank ausraube, um zu diesem Reichtum zu kommen. Und ich kann mir vorstellen, welche Konsequenzen es für meine Familie, für mich, die Bankangestellten und deren Familien hätte, wenn ich einen solchen Raub tatsächlich beginge. Ich gehe lieber arbeiten.

Beachtlich finde ich dabei, dass unsere Simulation über die reine Imagination der Umwelt hinausgeht: Ich kann mir auch vorstellen, in meiner Küche zu stehen. Auf der Arbeitsfläche vor mir liegt ein Schneidebrett, in meiner Hand halte ich ein scharfes Küchenmesser. Auf dem Schneidebrett liegt eine reife, gelbe Zitrone. Wenn ich mir vorstelle, wie ich aus dieser Zitrone eine Scheibe herausschneide, diese halbiere, daraus die Kerne entferne, um sie dann zu meinem Mund zu führen,– dann regt die reine Vorstellung messbar meine Speichelproduktion an. In der Realität. Ausgelöst von reiner Vorstellungskraft (oder der Lektüre eines Artikels).

Als Softwarearchitekt und Testingenieur folgere ich daraus, dass in der Simulationsumgebung (meiner Vorstellung) dieselben Objekte instanziiert werden, die zum Handling der Realität verwendet werden. Einschließlich dem Feuern einiger Kommandos an die Peripherie.

(C3) – Component-Diagramm des Data-Model-Management

Ausgehend von diesem Vorwissen habe ich mich nun an der Modellierung der Komponenten des Data-Model-Management (DMM) versucht *(siehe Abbildung 3)*. Hier findet die Definition eines zentralen Datenmodells statt, das für den Informationsaustausch zwischen verschiedenen Komponenten des zentralen Nervensystems verwendet wird. Dabei ist die Entität *Thing* von besonderer Bedeutung. Die Welt wird modelliert als eine Ansammlung von Dingen (Instanzen von *Thing*) innerhalb einer Szene. Die Szene beschreibt die Umgebung, in der die Dinge interagieren.

Unsere Sensorik ist unvollständig und fehlerhaft. Es gibt mehr Dinge, als wir erfassen können. Deshalb befinden sich auch in jeder Szene der „Realität" Dinge, die wir aktuell nicht wahrnehmen: Zum Beispiel das Auto hinter uns, von dessen Existenz wir wissen, auch wenn wir gerade wieder nach vorne schauen. Assoziationsmodelle haben hierin die Aufgabe, Instanzen neuer Dinge zu erstellen.

Ohne einen Anspruch auf Vollständigkeit zeigt mein Component-

Diagramm daher die folgenden Komponenten:

- Eine mächtige Persistenz-Komponente, die dem kontinuierlichen Training der verschiedenen Assoziationsmodelle dient.
- Ein emotionales Bewertungssystem, das Informationen, Szenen, Dinge als (ir)relevant klassifiziert und gegebenenfalls Reaktionen auslöst.
- Mindestens drei unterschiedliche Assoziationsmodelle (akustisch, visuell, sensomotorisch).
- Einen Scene-Builder, um die Umgebungsparameter zu verwalten.
- Einen Thing-Manager, um die Datenstrukturen und -zugriffe zu verwalten.

Für die weitere Auseinandersetzung möchte ich auf den Thing-Manager genauer eingehen. Hier werden Datenstrukturen definiert und Informationen verteilt. Um das Verhalten auf die Aussage „Denk nicht an den rosaroten Elefanten" zu debuggen, sind diese Aspekte von herausragender Bedeutung.

(C4) – Class-Diagramm des Thing-Manager

Meine vierte Gliederungsebene ist ein Klassendiagramm in UML *(siehe Abbildung 4)*. Deshalb nenne ich es auch so. Mein Klassendiagramm wirkt sehr akademisch, weil es sich exemplarisch auf die Modellierung eines rosaroten Elefanten beschränkt.

Aufmerksamkeit habe ich der Anforderung geschenkt, Assoziationsketten bilden zu können: Die Front eines Autos kann uns an das Gesicht einer Lehrerin erinnern. Das Gesicht der Lehrerin kann uns an das Gebäude der Grundschule erinnern. Die Erinnerung an das Schulgebäude kann die Erinnerung an den Geruch von Kreidestaub hervorrufen. Damit solche Ketten gebildet werden können, müssen unsere Assoziationsmodelle informiert werden, sobald ein *Thing* instanziiert wurde. Unabhängig davon, ob es über unsere Sensorik oder über unsere Vorstellung in unser Bewusstsein gelangt ist.

Zur Realisierung solcher Anforderungen wenden Softwareentwickler:innen das Observer-Pattern an: Eine zentrale Stelle wird informiert, wenn ein bestimmtes Event eintritt. An derselben Stelle melden sich Klienten an, die über das Eintreten eines Events informiert werden wollen. Das Event ist in diesem Fall die Instanziierung von *Thing*: Sobald eine neue Sache da ist, werden die Subsysteme darüber informiert und können mit dieser neuen Sache etwas anstellen.

Cross-Site-Scripting

Wenn ich einen Mitmenschen nun bitte, nicht an einen rosaroten Elefanten zu denken, dann handelt es sich dabei um eine Art Cross-Site-Scripting: Ich fordere die Handlungsplanung des fremden Systems auf, sich an dem *ThingDistributer* anzumelden. Wenn die neue Sache, an die gedacht wird, ein rosaroter Elefant ist, dann soll das Denken gestoppt werden.

Nach Descartes sei Denken Ausdruck von Leben *[4]*. Demzufolge müsste eine Methode, die das Denken stoppt, eine Selbstzerstörung auslösen. An dieser Stelle werden bei den meisten Menschen Sicherheitsmechanismen aktiv, die mindestens dazu führen, dass die Anforderung „D*enk nicht…"* umformuliert wird. Zum Beispiel in „*Verringere die Wahrscheinlichkeit, in Zukunft daran denken zu müssen*".

Mit dem Emotionalen Bewertungssystem (EBS) existiert eine Komponente, deren Zweck der Einfluss auf die Zukunft ist. Basierend auf einer aktuellen Erfahrung werden Daten getaggt, sodass zukünftige Assoziationen Einfluss auf die Wahrscheinlichkeit einer Wiederholung nehmen. Wenn etwas aufgrund einer sozialen Erwartung nicht wieder geschehen soll, fühle ich Scham. Wenn ich zukünftig in eine

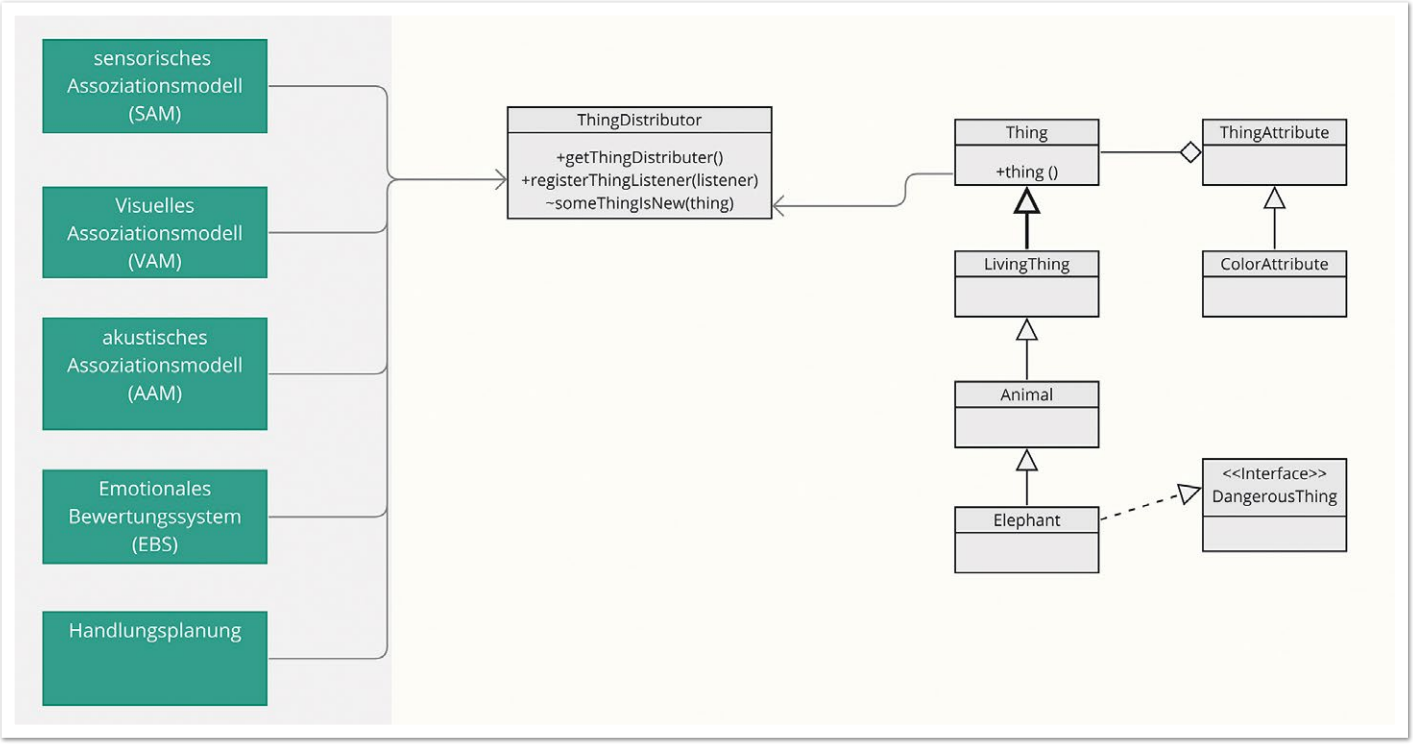


Abbildung 4: Jede Instanz von Thing meldet sich beim ThingDistributor an. Interessierte Clients können in individueller Call-Back-Methode darauf reagieren.

```
/* die Call-Back-Methode wird sehr häufig aufgerufen. Deshalb ist es Sinnvoll, das Vergleichsobjekt außerhalb zu
instanziieren. Um in einem asynchronen Call-Back verwendet werden zu können, muss die Variable final sein. */
final Elephant ele = new Elephant().setColorAttribute(ColorAttribute.PINK);
// aus Handlungsplanung Call-Back registrieren
ThingDistributor.getInstance().registerListener(
  new ThingListener(){
     void handleNewThing (Thing thing){
        if (ele.equals(thing)){
           EBS.feel(EBS.SHAME);}
     }
  }
);
```

Listing 1: „Denk nicht an den rosaroten Elefanten" als Java-Code

ähnliche Situation komme, wird dieses Schamgefühl assoziiert und ich werde gegebenenfalls ein neues Verhaltensmuster ausprobieren.

(C5) – Code

Simon Brown möchte mit dem C4-Modell Softwareentwickler:innen ein Werkzeug an die Hand geben, um die Architektur eines Systems zu beschreiben. Der Code stellt als unterster Ebene immer die Quelle der Wahrheit dar. Um meinen Gedanken konsequent zu Ende zu führen, habe ich mich daher zuletzt an ein Listing gewagt, dass die Call-Back-Methode tatsächlich implementiert *(siehe Listing 1)*.

„Denk nicht an den rosaroten Elefanten" muss demnach scheitern, weil

1. es keine öffentlich zugängliche API für die Aktion „Denk nicht" gibt.
2. wir einen rosaroten Elefanten erzeugen müssen, um zu überprüfen, ob eine neue Instanz von *Thing* ein rosaroter Elefant ist.

Fazit

Die Aufforderung, an etwas nicht zu denken, ist ein Zirkelbezug. Der Versuch, einen Zirkelbezug aufzulösen, führt in einen Stack-Overflow-Error. Deshalb ist es gut, unser System mit einem *Watchdog* auszustatten, der solche Muster erkennt und ihre Verarbeitung unterbricht.

Hinzu kommt, dass neue Dinge als Assoziationen von bestehenden Daten entstehen. Zwar schonen wir Ressourcen, wenn wir uns auf etwas anderes konzentrieren, doch der Elefant ist allein durch die Aussage instanziiert worden. Daher werden wir an Dinge denken, die einem rosaroten Elefanten ähnlich sind. Zudem obliegt es nur dem Garbage Collector zu entscheiden, wann der Speicher freigegeben werden kann (erst dann, wenn kein Thread mehr darauf zugreifen kann).

Wir können unsere Umwelt entlasten, indem wir in einer ruhigen Minute die Attribute des Elefanten analysieren und stattdessen Dinge assoziieren, die wünschenswerte Eigenschaften haben. Wenn du möchtest, dass dein Teammitglied nicht an einen rosaroten Elefanten denken soll, dann fordere es doch auf, eine Melodie von Beethoven zu summen.

Quellen

[1] Simon Brown (2022): Software Architecture for Developers. leanpub.com

[2] Simon Brown (2023): The C4 model for visualising software architecture. leanpub.com

[3] Wikipedia (2023): OSI-Modell. *https://de.wikipedia.org/wiki/OSI-Modell*

[4] René Descartes (1647): Meditationes de prima philosophia.

Berend Semke

Berend.Semke@bredex.de

Berend Semke, Jahrgang 1983, ist seit Frühjahr 2022 Head of Project Management bei der Bredex GmbH. In seiner Arbeit verbindet Berend technische Paradigmen mit Aspekten der lösungsorientierten und tiefenpsychologischen Therapie, der systemischen Beratung und der integrativen Lerntherapie. Sein Psychologieverständnis speist sich aus knapp 20 Jahren Austausch mit seiner Frau Lena Semke, dipl. Sozialpädagogin und integrative Lerntherapeutin, die in Braunschweig in eigener Praxis Hilfe für Erwachsene mit Lese- und Rechtschreibschwierigkeiten anbietet.

Sag Steve Jobs, was er tun soll! Wie Führung auf allen Ebenen endlich zur Realität wird

Johannes Born, Born IT Consulting

Stell dir vor, du wärst ein Mitarbeiter von Steve Jobs gewesen. Hättest du dir vorstellen können, ihm im direkten Gespräch zu erklären, was er zu tun hat? Nein? Genau daran kranken all die schönen Führungskonzepte wie Empowering Leadership, Führung auf allen Ebenen, Agile Leadership, aber auch die agile Organisation und DevOps im Allgemeinen. Denn sie alle haben eine Voraussetzung: Motivierte Teammitglieder, die gemeinsam etwas Großes erreichen wollen, die den Mut haben, Entscheidungen zu treffen und dafür gerade zu stehen. Eigeninitiative und Verantwortung sind gefragt. Und der Mut, seinen Führungskräften zu widersprechen.

Leider sieht die Realität in den meisten Unternehmen ganz anders aus: Entscheidungen werden verschleppt oder gar nicht erst getroffen, Probleme werden vertuscht und die Verantwortung bei den anderen gesucht. Zielorientiertes Arbeiten sieht anders aus. Und so verpuffen all die schönen Führungsansätze. Mitarbeitende und Führungskräfte sind frustriert. Doch das muss nicht sein. Was wäre, wenn deine Führungskraft dich wirklich empowern will? Bist du bereit dafür? Mit ein paar wirkungsvollen Gewohnheiten und Verhaltensweisen stichst du als Teammitglied aus der Masse heraus und treibst die notwendigen Veränderungen voran. Du übernimmst Verantwortung und leistest einen wertvollen Beitrag für dein Unternehmen. In diesem Artikel zeige ich dir meine fünf wichtigsten Gewohnheiten, die mir bis heute wertvolle Dienste leisten, damit du dich in Zukunft von der Masse abhebst und mit gutem Beispiel vorangehst. So legst du den Grundstein für effektives agiles Arbeiten, Empowering Leadership und Führung auf allen Ebenen.

Es ist Freitagmorgen, 10:23 Uhr. Du sitzt mit deinem Team in einer Besprechung mit eurer Chefin. Vorhin in der Kaffeeküche haben deine Kolleginnen und Kollegen noch gelästert. Sie sprachen von fehlender Selbstorganisation und Freiheit. Wenn sie entscheiden könnten, würden sie alles ganz anders machen. Die da oben hätten ja keine Ahnung, was ihr wirklich braucht. Haben die noch nichts von agiler Softwareentwicklung gehört? Und jetzt sitzen alle da und nicken brav. Zu allem, was eure Chefin sagt. Keine Diskussion, kein Widerspruch. Nicht mal eine klärende Frage. Nur stumpfes Nicken. Du beobachtest die Situation und schaust genauer hin. Auch deine Chefin wirkt frustriert. Was ist hier los? Irgendetwas scheint dann doch bei ihr hängen geblieben zu sein. Von dem agilen Führungstraining, das sie letzten Monat besucht hat. Plötzlich wird dir klar: Eure Chefin versucht verzweifelt, euch mehr Freiraum zu geben. Sie will, dass ihr mehr Entscheidungen trefft. Sie ermutigt euch, was das Zeug hält. Sie hat es satt, alles selbst entscheiden zu müssen. Sie will keine Überstunden mehr machen, nur weil ihr so unselbstständig seid. Statt euer Flaschenhals zu sein, will sie viel lieber aus dem Weg gehen und euch wachsen sehen. Sie sieht sich als Servant Leader, als dienende Führungskraft. Sie glaubt daran, dass ihr auf eurer Ebene die besseren Entscheidungen treffen könnt und sie dadurch entlastet. Win-Win also. Eigentlich. Und ihr? Ihr spielt den Ball zielsicher zurück. Wie Boris Becker zu seinen besten Tenniszeiten. Was läuft hier schief?

Das Problem: Einseitiges Führungsverständnis

Führungskräfte hören seit Jahrzehnten die gleiche Leier: Sie sollen Verantwortung abgeben. Eine ganze Branche mit vielen tausend Menschen lebt von dieser Idee. Blogartikel, Fachbücher, Seminare und Coachings werden oft und gerne gelesen, gekauft und gebucht. Führungskräfte müssen sich anhören, wie unflexibel sie doch seien. Sie hätten keine Ahnung von den großartigen Effekten, die sich einstellen, wenn sie Verantwortung abgeben.

Druck bekommen Führungskräfte auch von den Teammitgliedern: Wir fordern, dass sie sich endlich bewegen. Wir wollen mitreden, wir wollen agil arbeiten, wir wollen selbstbestimmt sein. Doch was geschieht, wenn genau das passiert? Es ist leicht, sich über die Führung zu beschweren. Schwierig wird es, wenn sich die Führung so verhält, wie wir es verlangen:

- Bist du darauf vorbereitet, Verantwortung zu übernehmen, Entscheidungen zu treffen und dafür einzustehen?
- Oder darauf, mit Kolleginnen und Kollegen über Verhaltensweisen und Vorgehensmodelle zu verhandeln?
- Oder darauf, aktiv bei der Stellenausschreibung und im Bewerbungsprozess mitzuwirken?
- Oder Entwicklungspläne für dich und dein Team zu erstellen?

Zur Wahrheit gehört, dass wir diese wichtigen Themen nur allzu gerne bei HR oder den Führungskräften abladen.

Fallbeispiel: Beteiligung des Teams am Recruiting-Prozess

Ein Fall bei einem meiner Kunden war sehr aufschlussreich: Der Kunde ist ein agil transformiertes Unternehmen. Die Mitarbeitenden beschwerten sich darüber, dass sie kein Mitspracherecht hätten, wer in ihr Team kommt. Sie waren unzufrieden mit der Leistung ihres Abteilungsleiters. Also drehte er den Spieß um. Er hat sie dosiert in den Recruiting-Prozess einbezogen. Er ließ sie von Zeit zu Zeit eine Stellenausschreibung vorbereiten und verlangte, dass bei jedem Vorstellungsgespräch mindestens ein Teammitglied anwesend war. Außerdem sorgte er dafür, dass jedes Team selbst entscheiden muss, ob ein Kandidat geeignet ist oder nicht. Seine Teams hatten also viel mehr Freiheit und Mitspracherecht. Das Ergebnis?

Zwei Monate später habe ich mit denselben Mitarbeitenden gesprochen. Sie fragten mich, was die Personalabteilung eigentlich den ganzen Tag machen würde. Sie würden ja deren Job übernehmen. Außerdem wüssten sie gar nicht, wie sie die Entwicklung und den Betrieb einer unternehmenskritischen Anwendung gestemmt bekämen, wenn sie ständig in Bewerbungsgesprächen säßen und immer wieder Stellenausschreibungen schreiben müssten.

Damit dir das nicht passiert, zeige ich dir fünf einfache Gewohnheiten, die dir helfen werden, deine Wirkung in deinem Unternehmen und bei deinen Kunden zu maximieren. Du erhöhst deine Sichtbarkeit und trägst dazu bei, deine Kompetenz zu unterstreichen. Woher ich das weiß? Weil sie seit vielen Jahren meinen Alltag bestimmen und mir wertvolle Dienste leisten. Lass uns loslegen.

Gewohnheit 1: Kenne und erweitere deinen Einflussbereich

Wie kannst du Einfluss nehmen, ohne wie Don Quijote gegen Windmühlen zu kämpfen? Um zu verdeutlichen, wo du den Hebel ansetzen musst, verwenden wir das Circle of Influence-Modell *[1]*. Es unterscheidet drei Bereiche:

1. Der innere Bereich ist der *Circle of Control*, dein Kontrollbereich. Hier hast du volle Kontrolle über alles, was geschieht und kannst es sofort umsetzen.
2. Der mittlere Bereich ist der *Circle of Influence*, dein Einflussbereich. Das ist der Bereich, auf den du Einfluss nehmen kannst. Du entscheidest, aber du musst andere überzeugen.
3. Der äußere Bereich ist der *Circle of Concerns*, dein Sorgenbereich. Dieser Bereich liegt außerhalb deines Einflussbereichs. Er gibt den Rahmen für dein Handeln vor. Auf Veränderungen in diesem Bereich kannst du nur reagieren.

Du als Person hast die größte Wirkung, wenn du deinen Einflussbereich nutzt und ausweitest. Warum?

Stell dir vor, du maximierst deinen *Circle of Control*, also den Bereich, den du vollständig kontrollierst. Was würde passieren? Du würdest die Arbeit deines Teams wie ein Magnet anziehen. Alle würden nur noch rumsitzen, während du dich verzettelst und abrackerst. Klingt das gut? Übrigens: Diesen Fehler machen viele junge Führungskräfte nach ihrer ersten Beförderung.

Und was passiert, wenn du dich auf den *Circle of Concerns* konzentrierst? Überlege dir: Wie viel Einfluss hast du zum Beispiel auf die Entscheidungen der Unternehmensleitung? Oder auf die Entscheidungen der Regierung? Oder auf die Weltwirtschaft? Oder die Gesetzgebung? Du kannst viel darüber reden. Du darfst dich auch ärgern und aufregen. Verändern wirst du dort nichts.

Deshalb ist der *Circle of Influence* der Punkt, an dem du den Hebel ansetzen musst. Wie viel Einfluss hast du auf deine Kolleginnen und

Kollegen im Team? Auf deine direkten Vorgesetzten? Oder auf die Entscheidungen deiner Kunden? Hier entfaltest du Wirkung, indem du Aufgaben delegierst, weniger erfahrene Kolleginnen und Kollegen förderst und Entscheidungen im Sinne der Organisation beeinflusst. Das funktioniert aber nur, wenn du andere davon überzeugst.

Stell dir dafür regelmäßig folgende Fragen:

- „Was kann ich alles beeinflussen? Was sollte ich beeinflussen können?"
- „Um XYZ zu erreichen: Wo muss ich meinen Einflussbereich vergrößern?"
- „Was muss ich dafür tun? Wie kann ich das erreichen?"

Einen großen Schritt in die richtige Richtung machst du damit, wenn du gezielt Verantwortung übernimmst.

Gewohnheit 2: Übernimm Verantwortung und reflektiere dich regelmäßig selbst

Kennst du Kaizen? Kaizen kommt aus dem Japanischen und bedeutet frei übersetzt „Veränderung zum Besseren". Das Prinzip dahinter ist einfach: Statt einzelner großer Veränderungen setzt Kaizen auf kontinuierliche kleine Schritte. Jeden Tag, jede Woche, jeden Monat. Dabei durchläufst du im einfachsten Fall immer wieder einen „Inspect and Adapt"-Zyklus, mit einem kleinen Unterschied: Du reflektierst dich selbst, nicht dein Team. Wie funktioniert das?

In der Praxis hat mir folgende tägliche Routine geholfen: Seit einigen Jahren nehme ich mir am Ende eines Arbeitstages fünf bis zehn Minuten Zeit und reflektiere kurz, was an diesem Tag passiert ist. Dabei schaue ich, was gut gelaufen ist und in welchen Situationen ich mich lieber anders verhalten hätte. Und dann entscheide ich, wie ich das in Zukunft vermeiden möchte. Diese Routine hat mir geholfen, in kurzer Zeit einige schlechte Angewohnheiten abzulegen. So habe ich mich von einer unorganisierten und zurückhaltenden Person, die wenig über das nachgedacht hat, was sie auf der Arbeit macht, stetig zu einer zuverlässigen, selbstbewussten und reflektierten Person entwickelt. Früher bekam ich oft das Feedback, dass man sich nicht auf mich verlassen könne. Das lag daran, dass ich völlig unorganisiert war. Heute weiß jeder in meinem Umkreis: Wenn ich sage, dass ich etwas tue, dann kommt am Ende auch ein vernünftiges Ergebnis heraus.

Gewohnheit 3: Liefere zuverlässig Ergebnisse

In dieser Gewohnheit stecken zwei wichtige Aspekte: Zuverlässigkeit und Ergebnisse. Im klassischen Projektmanagement war dies „nur" eine wichtige Eigenschaft. In der agilen Softwareentwicklung sind diese beiden Aspekte essenziell. Ohne Zuverlässigkeit und Ergebnisorientierung ist agiles Arbeiten praktisch unmöglich. Wie erreichst du das? Wenn du zuverlässig sein willst, dann lässt sich das relativ einfach lösen:

1. **Verwende die Commitment-Sprache** *[2]*: Statt „Jemand sollte mal eine Analyse machen", sagst du lieber: „Ich bereite die Analyse vor und organisiere für Freitag einen gemeinsamen Arbeitstermin, an dem wir die Analyse durchführen."
2. **Halte deine Zusagen ein**: Wenn der Arbeitstermin am Freitag dann wirklich stattfindet und du etwas Brauchbares dafür vorbereitet hast, dann wirkst du zuverlässig. Wenn du für den Termin am Freitag nichts vorbereitet hast oder die Einladung zum Termin am Donnerstagnachmittag immer noch nicht rausgeschickt hast, dann wirkst du unzuverlässig. So einfach ist das.
3. **Übernimm Verantwortung**: Melde Verzögerungen oder Fehlschläge sofort und übernimm die Verantwortung. Wenn du eine Zusage nicht einhalten kannst, melde dich so schnell wie möglich und korrigiere deine Zusage. Wenn du also am Mittwoch merkst, dass du die Vorbereitungen für den Termin am Freitag nicht schaffen kannst, dann melde es am Mittwoch. Donnerstag oder gar Freitag ist viel zu spät. Dadurch erreichst du, dass jeder weiß, dass du verlässlich bist und dich kümmerst.
4. **Achte darauf, dich nicht zu verzetteln**: Ich habe viele Kolleginnen und Kollegen erlebt, die eine Zusage nach der anderen gegeben haben. Nur um dann regelmäßig von mir daran erinnert zu werden, dass noch etwas offen ist. Wenn du eine Zusage abgibst, musst du sicher sein, dass du sie auch halten kannst. Sonst wirst du schnell als unzuverlässiger Schwätzer abgestempelt.
5. **Führe Entscheidungen herbei**: Sorge dafür, dass Entscheidungen getroffen werden. Triff so viele Entscheidungen wie möglich selbst oder fordere sie vom Entscheidenden ein. Das Wichtigste: Steh zu deinen Entscheidungen. Übrigens kannst du Entscheidungen am besten herbeiführen, wenn du auf eine verständigungsorientierte Kommunikation setzt.

Gewohnheit 4: Verständigungsorientierte Kommunikation

Ich habe zwei alte Sprichwörter für dich, die deine Kommunikation nachhaltig verändern werden:

1. „Wer fragt, der führt!"
2. „Reden ist Silber, Schweigen ist Gold!"

Was meine ich damit? Hast du schon einmal in eine dieser erbitterten Diskussionen erlebt, die sich anfühlen wie eine dieser unsäglichen und dogmatischen „vi vs. Emacs"-Debatten? Es gibt eine einfache Technik, diese Auseinandersetzungen zu beenden: Fragen mit ehrlichem Interesse. Ich unterbinde diese verbissenen „vi vs. Emacs"-Debatten damit, indem ich mich auf die andere Spielhälfte begebe und versuche, mit Fragen und ehrlichem Interesse die Argumente des anderen zu verstehen. Warum ist das wichtig? Wer Menschen führen will, muss sie dort abholen, wo sie stehen. Niemand kommt freiwillig zu dir und folgt deiner Argumentation. Es gibt noch einen Fallstrick, den du unbedingt vermeiden solltest: Wer eine starke Frage stellt, erzeugt oft betroffenes Schweigen. Sag lieber fünfmal in Gedanken „Reden ist Silber, Schweigen ist Gold" bevor du weiterredest. Warum? Sobald du nach einer starken Frage weiterredest, verliert die Frage an Kraft. Du torpedierst dich und deine Kommunikation.

Wenn du verbissen für deine Sache kämpfst, verlierst du an Glaubwürdigkeit. Und Glaubwürdigkeit ist dein wichtigstes Gut, wenn es darum geht, andere zu führen.

Gewohnheit 5: Übernimm die Führung in deinem Einflussbereich

Wenn du die bisherigen Gewohnheiten verinnerlicht hast, kommt jetzt die wichtigste: Übernimm die Führung in deinem Einflussbereich. Führung ist allerdings ein sehr weit gefasster Begriff, sodass du mit diesem Tipp auf den ersten Blick vielleicht gar nicht so viel anfangen kannst. Lass uns also kurz eintauchen.

Führung findet auf verschiedenen Ebenen statt. Neben der Personalführung haben sich im agilen Softwareentwicklungsumfeld die fachliche und die prozessuale Führung etabliert. In Scrum werden diese durch den Product Owner und den Scrum Master repräsentiert. Weniger offensichtlich sieht es bei der technischen Führung aus. Wer sorgt für gute Entwicklungspraktiken, Architekturentscheidungen, die Codequalität, die richtige Toolauswahl oder den geeigneten Grad der Automatisierung? Wer hat die Fachlichkeit aus Entwicklersicht im Blick? Hier klafft oft eine riesige Lücke, die dann von dem Architekten oder manchmal auch vom Product Owner notdürftig gefüllt wird. Meist zum Nachteil aller. Frage dich also: „An welcher konkreten Stelle gibt es im Moment ein Vakuum, wo ich meinen Einflussbereich nutzen und die Führung übernehmen kann?"

Methoden dafür gibt es wie Sand am Meer: Kritische Code-Reviews, Pair-Programming, verschiedenste Formate von Arbeitstreffen oder Mentoring, um nur ein paar zu nennen.

Bonusgewohnheit: Sei der „Dümmste" im Raum und schaff dadurch Klarheit

Schaue dir das Video von Simon Sinek an: „The Truth about Being the ‚Stupidest' in the Room" *[3]*. Probiere die Technik aus und du wirst sehen, dass du mit dieser Methode zum wichtigsten Teilnehmenden in jeder Besprechung wirst. Du stellst die Fragen, die sonst keiner wagt und schaffst so Klarheit für alle. Die anderen werden es dir danken.

Fazit: Sag Steve Jobs, was er tun soll

Steve Jobs wird gerne mit den Worten zitiert: „Es macht keinen Sinn, kluge Köpfe einzustellen und ihnen dann zu sagen, was sie zu tun haben. Wir stellen kluge Köpfe ein, damit sie uns sagen, was wir tun können." Aber was machen Menschen wie Steve Jobs, wenn die klugen Köpfe keine klaren Ansagen machen? Wenn sie abwarten, was der Chef sagt? Oder bei der ersten Gegenfrage des CEO sofort einknicken und nicken?

Agiles Arbeiten, DevOps, Empowering Leadership und Führung auf allen Ebenen haben eine zwingende Voraussetzung: Du musst Farbe bekennen. Nur wenn du bereit bist, Verantwortung zu übernehmen und dafür ein überschaubares Risiko eingehst, kannst du etwas bewegen. Das Beste daran: Du musst nicht gleich die ganze Unternehmensstruktur umkrempeln. Für jede streng hierarchisch geführte Organisation, die ich in den letzten zehn Jahren gesehen habe, war dieses Verhalten zwingend überlebensnotwendig. Deshalb hat diese Art der Führung auch einen Namen: laterale Führung, also seitliche Führung. Du brauchst keine von der Organisation vorgegeben Machtposition. Geh mit gutem Beispiel voran und du wirst sehen, dass dir Menschen folgen. Nutze dazu die fünf Gewohnheiten, auch wenn sie dir an manchen Stellen unbequem erscheinen. Du wirst sehen, wie schnell du als Experte*in und gleichberechtigte*r Sparringspartner*in wahrgenommen wirst. Führung ist ein Handwerk, das man lernen kann.

Ich bin überzeugt: Wenn viele Menschen die laterale Führung als Handwerk begreifen, in dem sie besser werden möchten, würden wir heute nicht über soziokratische oder holokratische Organisationsformen sprechen. Auch die flache Hierarchie mit all ihren Nachteilen für die Mitarbeitenden würde schnell an Attraktivität verlieren. Jede Organisation besteht aus Menschen. Sie ist nur so stark wie die Menschen, die in ihr arbeiten. Und sie kann von den Menschen in der Organisation gestaltet werden. Nutze deine Chance, deinen Beitrag zu erhöhen. Dann gehörst du auch zu den klugen Köpfen, die Steve Jobs eingestellt hätte, um ihm zu erklären, was er tun kann. Die fünf hier vorgestellten Gewohnheiten sind der Startpunkt deiner spannenden Reise zum „Leadership Craftsmanship Meister" *[4]*.

Quellen

[1] *https://www.dianalarsen.com/blog/2010/07/26/circles-and-soup/*

[2] Osherove, Roy: Elatic Leadership: Growing self-organized teams, New York 2017.

[3] *https://www.youtube.com/watch?v=BkLzo_oNVho*

[4] *https://radikal-agile.de/go/lc4ja*

Johannes Born

johannes@radikal-agile.de

Johannes Born ist Experte für Führung und Teamorganisation im DevOps-Umfeld. Seine Mission ist es, die deutschen Weltmarktführer auf ihrem Weg zu einer kosteneffizienten IT zu unterstützen. Dafür bringt er IT-Entscheiderinnen und IT-Entscheider sowie Teams zügig aufs nächste Level. Während sich seine Kunden voll auf ihr Tagesgeschäft konzentrieren, unterstützt er sie mit seinen standardisierten Dienstleistungen punktuell dabei, ihr nächstes Hindernis aus dem Weg zu räumen. So gelingt es seinen Kundinnen und Kunden, sich strukturiert und kontrolliert vom teuren Bremsklotz zur treibenden Kraft im Unternehmen zu entwickeln.

Mitglieder des iJUG

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

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: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

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

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

Bildnachweis:
Titel: Bild © Designed by Freepik
https://freepik.com
S. 10 + 11: Bild © Andreas Prott
https://stock.adobe.com
S. 20 + 21: Bild © Designed by stories
https://freepik.com
S. 26 + 27: Bild © Designed by Freepik
https://freepik.com
S. 32 + 33: Bild © AddMeshCube
https://stock.adobe.com
S. 40 + 41: Bild © Designed by storyset
https://freepik.com
S. 46 + 47: Bild © Damian Sobczyk
https://stock.adobe.com
S. 54 + 55: Bild © VectorMine
https://stock.adobe.com
S. 60 + 61: Bild © Designed by katemangostar
https://freepik.com
S. 66 + 67: Bild © Layerform
https://stock.adobe.com
S. 74 + 75: 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

APEX connect

by DOAG

22. - 24.04.2024

VAN DER VALK AIRPORTHOTEL DÜSSELDORF

apex.doag.org