Ausgabe 04/2024

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwicklerinnen und Entwickler

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

4 191978 304903 04

Java aktuell

iJUG Verbund
www.ijug.eu

Java 22
Details zum aktuellen Release

Microfrontends
Praxisbericht zur Integration in bestehende Anwendungen

Soft Skills
Die wichtigsten Kompetenzen für Entwickler:innen

KI Navigator 2024

Konferenz zur Praxis der KI in IT, Wirtschaft und Gesellschaft

20. + 21. November in Nürnberg

Jetzt anmelden

Early-Bird-Tarif sichern

ki-navigator.doag.org

Veranstalter: DOAG Heise Medien dgepol

Liebe Leserinnen und Leser,

willkommen zur fünften Ausgabe des Jahres und dem neuen, aktuellen Major-Release Java 22, mit dem wir uns beschäftigen wollen. Zu Beginn verschafft uns Falk Sippach in seinem Artikel ab Seite 12 einen detaillierten Überblick über alle Neuheiten und Änderungen des Release. Im Anschluss geht Michael Inden in seinem Doppelartikel näher auf die neuen Features Structured Concurrency sowie String Templates ein. Auch bei Maven gibt es Neues zu berichten: Karl Heinz Marbaise stellt uns im ersten Teil seiner Artikelreihe das Versionshandling in Maven 4 vor. Alle Neuigkeiten rund um Java, Eclipse und die Community findet ihr im Java-Tagebuch und der Eclipse Corner wie gewohnt am Anfang der Ausgabe. Dort ist übrigens auch wieder eine neue Edition der "Goldenen Regeln" von Andreas Monschau zu finden!

Dynamisch geht es weiter: Michael Schweiker zeigt uns ab Seite 44, wie Java, Kotlin und Scala sinnvoll gemeinsam verwendet werden können. Danach folgt ein Erfahrungsbericht von Christian Siebmanns, dessen Team Microfrontends erfolgreich in eine bestehende Anwendung integriert hat. Er berichtet vom Ablauf und den bewältigten Herausforderungen. Ab Seite 60 beginnt der erste Teil der Artikelreihe von Richard Gross, in der er eine Lösung zur Vermeidung strukturzementierender Tests aufzeigt. In diesem Teil zeigt er uns, wie eine TestDsl, die Grundlage seiner Lösung ist, gebaut wird. Wie man durch die Integration von CQRS und DOP ein robusteres, skalierbareres und leichter wartbares System erhält, zeigt uns Simon Martinelli. Das Thema nachhaltige Enterprise-Architekturen behandelt Prof. Dr.-Ing. Stefan Wagenpfeil ab Seite 76. Darin beleuchtet er die wichtigsten Thematiken rund um das Thema und zeigt mögliche Lösungsansätze. Zum Abschluss dieser Ausgabe erläutert Eldar Sultanow die relevantesten Soft-Skills-Kompetenzen für Softwareentwicklerinnen und -entwickler näher.

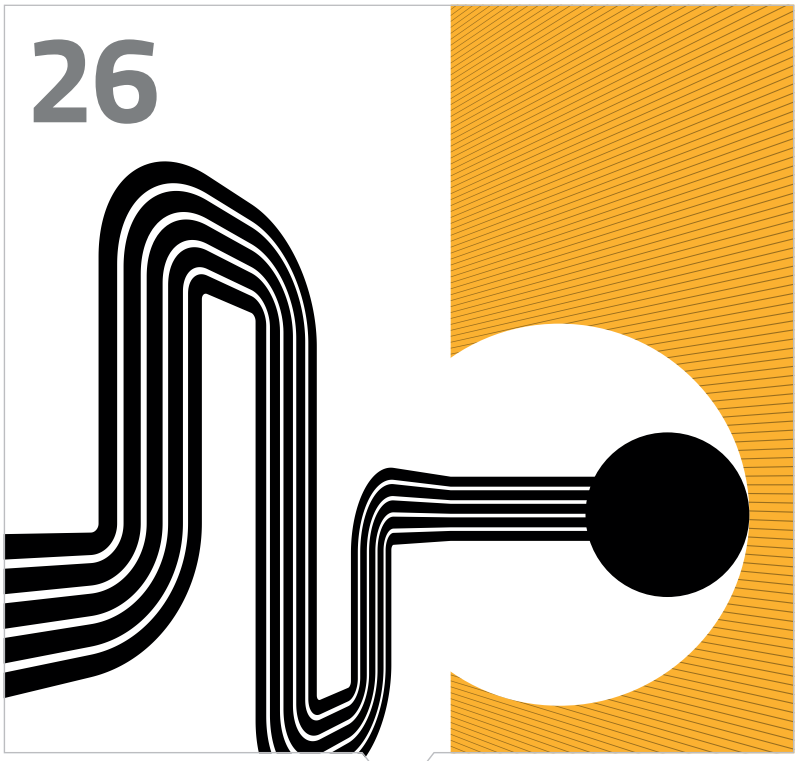
Falls ihr auch einmal (oder natürlich auch mehrfach) euer Wissen mit der Java-Community teilen möchtet, meldet euch bei uns! Wir sind stets auf der Suche nach Autorinnen und Autoren, die ihr Know-how und ihre Erfahrung weitergeben möchten. Wenn ihr ein interessantes Thema habt, freuen wir uns über eure Kontaktaufnahme und euren Artikelvorschlag an *redaktion@ijug.eu*. Noch einfacher und bequemer geht es über unseren Newsletter: Abonniert über euer Online-Konto das Themengebiet "iJUG Autoren/Speaker", so erhaltet ihr unsere Artikelaufrufe mit allen wichtigen Eckdaten bequem per E-Mail.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

Java 22: Das aktuelle Release im Überblick



Unter der Lupe:
Structured Concurrency und String Templates

Herausforderungen der Integration von Microfrontends in bestehende Anwendungen

Die relevantesten Soft-Skill-Kompetenzen für Entwicklerinnen und Entwickler

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

4. Februar 2024

Die „1 Milliarde Zeilen Challenge“

Spannend, zu welchen Höchstleistungen das „ach so langsame“ Java fähig ist: Gunnar Morling hatte im Januar in seinem Blog und Social Media zur „1 Billion Row Challenge“ (1BRC) aufgerufen: Wer kann am schnellsten mittels Java-Code eine CSV-Datei mit einer Milliarde Zeilen einlesen und ein paar einfache statistische Berechnungen darauf ausführen? Die Einreichungen wurden auf derselben Hardware getestet, um eine einheitliche Bewertung sicherzustellen. Und die Ergebnisse sind wirklich beeindruckend: Die besten 5 eingereichten Programme schaffen das in unter 2 Sekunden! Den ersten Platz hat sich wohl nicht ganz zufällig ein Dreier-Team aus den Züricher Oracle Labs (GraalVM!) gesichert. Allein aus den Top 10 sind 8 Einreichungen GraalVM-basiert, da hier schon die Startup-Zeiten einen entscheiden Unterschied machen. Die Ergebnisse kann man online bestaunen *[1]*.

5. Februar 2024

MP API-Tutorial

Das MicroProfile API-Tutorial entsteht gerade (basierend auf Version 6.1), die Eclipse Foundation hatte dafür ein kleines Budget bereitgestellt und die wesentlichen Arbeiten als Projekt ausgeschrieben. Der neueste Stand ist online einsehbar *[2]*. Das Ergebnis wird dann zukünftig auf der microprofile.io-Seite zu sehen sein.

18. Februar 2024

MP Major/Minor-Release-Diskussion

MicroProfile 7 ist unterwegs, jedoch begleitet von vielen Diskussionen rund um die korrekte semantische Versionierung und die Kompatibilität mit Java SE und Jakarta EE. Ein zentraler Diskussionspunkt: Soll das Release auf Jakarta EE 9.1 basieren oder dem EE 10 Core Profile?

Ein anderer Punkt: Sollten die Versionen der Einzelspezifikationen auf eine neue Major-Version gehoben werden, wenn sich die API selbst nicht ändert, wohl aber Abhängigkeiten (wie die Minimum Java-Version)? Aus der API-/Consumer-Perspektive ist das nicht nötig, aber die API-Provider (Implementierer) sind da vielleicht anderer Ansicht, weil das für sie sehr wohl ein „breaking change“ ist, beziehungsweise sein kann. Also doch ein „Ja“? So wie es aussieht, hat sich eher die reine API-Sicht durchgesetzt – eine neue Minor-Version reicht aus.

Was neue Features angeht, sind die Änderungen sehr überschaubar. Viel dreht sich um das weitere Alignment mit Jakarta EE. Bei Telemetry 2.0 hat sich aber doch einiges getan: Das neueste OpenTelemetry-Release wird unterstützt, inklusive Metrics und Logging (statt wie bislang nur Tracing).

19. März 2024

JDK 22 und die JavaOne

Java 22 ist mit einer ganzen Reihe von Features (nicht nur Preview- oder Inkubator-Features) freigegeben worden, alle schon im (vor-) letzten Tagebuch erwähnt und auch auf der OpenJDK-Website aufgelistet *[3]*.

Oracles Sharat Chander (kurze Rollen-Bezeichnung: Senior Director, Product Management and Developer Engagement) setzt auf diese erwartete Nachricht noch mal eine ganz andere Ankündigung drauf: Die JavaOne wird 2025 wieder aufleben, statt in Las Vegas dann wieder in oder zumindest bei San Francisco. Der Termin ist ungewohnter Weise schon im Frühjahr: 17. bis 20. März 2025. Der Ort sieht auch nach einer Verkleinerung aus: Nicht mehr das riesige Moscone Center in San Francisco, sondern das Oracle Conference Center in den Oracle Headquarters in Redwood Shores (neben den bekannten „Datentopf“-Türmen). Mehr Details gibt es noch nicht, aber die Ankündigung ist online *[4]*.

19. März 2024

GraalVM 22

Das korrespondierende GraalVM-Release wird traditionellerweise am gleichen Tag wie das Java-SE-Release freigegeben. Die Graal VM 22 bietet unter anderem die aus anderen Profilern bekannten bunten FlameGraph-Visualisierungen – sie sind jetzt Teil der Native Image Build Reports. „Profile-guided optimizations“ bieten hilfreiche Statistiken darüber, wie gut ein Profil zum ausgeführten Code beziehungsweise der Ausführung an sich passt. Außerdem ist eine experimentelle Unterstützung für die „Foreign Function & Memory API“ in Java 22 enthalten.

20. März 2024

JDK 23

Nach dem Release ist vor dem Release. Die Termine für das JDK 23 stehen fest: Am 6. Juni startet der „Rampdown Phase One“, zu dem

die Entwicklung im Wesentlichen abgeschlossen sein muss, und der Release-Termin ist für den 17. September geplant. Viel Inhalt ist aber noch nicht „gesetzt", bislang nur eine zweite Preview für die Class-File API, und eine (erste) Preview von „Primitive Types in Patterns, instanceof, and switch". Aber da geht bestimmt noch was, möglich wären beispielsweise Updates der bisherigen Preview-/Inkubator-Versionen „statements before super()" (JEP 447) oder „Vector API" (JEP 448) sowie eventuell Markdown in JavaDoc (JEP 467). Erst vor ein paar Tagen ist auch ein neuer JEP-Entwurf namens „Hot Code Heap" angelegt worden *[5]*. Damit soll „hot" (also häufig genutzter und damit auch für die Optimierung besonders spannender) Code in einem eigenen Heap-Bereich abgelegt werden, um Fragmentierung zu reduzieren und am Ende die Execution Time zu reduzieren.

26. März 2024

„AI" in MicroProfile

Nach langchain4j und Spring AI will jetzt auch MicroProfile ein Projekt starten, um Large Language Models einfach und generisch in eigene Java-Services einzubetten. Das Projekt formiert sich gerade erst, sodass es auf jeden Fall eine Weile dauern wird, bis erste Resultate zu sehen sind (wenn sich genügend Freiwillige finden, aber das ist bei dem Thema vermutlich nicht das größte Problem). Andererseits heißt das, dass Interessenten auch bei Erscheinen dieses Tagebuchs sicher immer noch ein Projekt vorfinden, das offen für weitere Ideen und Mitarbeit ist. Dazu am besten diesen Chat hier aufrufen *[6]*, der enthält zumindest bislang den aktuellen Stand der Diskussion.

1. April 2024

MicroProfile 7

Die Abstimmung über MP 7.0 ist heute beendet worden. Jan Westerkamp hatte als Repräsentant des iJUG zwar dagegen gestimmt, unter anderem aufgrund der nicht erfolgten Fokussierung auf das Jakarta EE 10 Core Profile (stattdessen dienen noch EE 9.1 Spezifikationen als Basis), und wegen nicht adressierter CVEs im Maven Parent der API-Projekte (es geht eher um Grundsätzlichkeit in der Adressierung von Abhängigkeiten mit bekannten Sicherheitslücken, die konkreten Risiken hier beziehen sich „nur" auf die Builds, die gebauten API Jars sind nicht betroffen). Auswirkungen hat die Gegenstimme aber nicht, da alle anderen Stimmberechtigten dafür stimmen (8), bei zwei Enthaltungen.

2. April 2024

Jakarta 11 ist trotz Diskussionen auf dem Weg

Jakarta EE 11 soll im Juni oder Juli freigegeben werden, die Abstimmung über den Release-Plan ist bereits im Februar erfolgt. Es gab allerdings einige Gegenstimmen durch Firmenvertreter im Spezifikationskomitee, namentlich von Oracle, Fujitsu und Payara. Stein des Anstoßes war die Entscheidung, neben Java 21 auch noch Version 17 zu unterstützen, wohl insbesondere um niemanden zu „verschrecken", der oder die noch nicht bereit ist, auf Java 21 zu migrieren. Was aber auf der anderen Seite nicht nur höhere Testaufwände bedeutet, sondern auch die vergebene Chance, in den Spezifikationen neue Java 21-Features zu nutzen, was besonders dem Oracle-Vertreter nicht gefiel (wobei ja eh nur ein kleiner Teil der Features wirklich in einer API „sichtbar" werden).

Jakarta Data 1.0 ist aber, Stand heute, im Release enthalten. Die neue Spezifikation, die eine Abstraktion über verschiedenen Datenzugriffsformen (SQL, NoSQL, Web-Services …) bieten soll, und damit diesen Zugriff von den technischen Details lösen und besser in die (fachliche) Domänen-Ebene einbinden soll, wäre ein echter Zugewinn für Jakarta.

3. April 2024

Finalisierung

Java-Blogger Billy Korando listet in einem Blog-Eintrag die „Deprecated Features" von Java 18 bis 21 auf *[7]*. Der Eintrag ist schon gute drei Monate alt, aber er passte begrifflich ganz gut, da ich gerade das Tagebuch für den Druck finalisiere, und die „finalization" das wohl prominenteste Feature ist, das in jüngerer Vergangenheit abgekündigt wurde. Dazu hat er gleich noch den Tipp, dass sich nicht nur mit „--finalization=disabled" prüfen lässt, ob eine Applikation in einem zukünftigen Release ohne Finalization noch laufen würde, sondern dass der FlightRecorder mit den FinalizerStatistics Hilfestellung gibt, wenn's tatsächlich Probleme gibt.

Verweise

[1] *https://www.morling.dev/blog/1brc-results-are-in/*
[2] *https://github.com/eclipse/microprofile-tutorial*
[3] *https://mail.openjdk.org/pipermail/jdk-dev/2024-March/008827.html*
[4] *https://jugs.groups.io/g/jug-leaders/message/1731*
[5] *https://openjdk.org/jeps/8328186*
[6] *https://groups.google.com/g/microprofile/c/9lXZG_BnUcs*
[7] *https://inside.java/2023/12/17/sip093/*

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

Ich programmiere seit den 80er Jahren aus Leidenschaft, seit 1997 hauptberuflich und seit etwa einem Vierteljahrhundert ist die Entwicklung von Open-Source-Contributions zudem mein größtes Hobby. Die GitHub-Statistik hat mir kürzlich verraten, dass ich offensichtlich an mindestens 63 Open-Source-Projekten beteiligt war (so genau lässt sich das nicht mehr nachvollziehen, da es GitHub noch nicht so lange gibt) und, dass ich anscheinend in den letzten Jahren mehr oder minder regelmäßig an mindestens fünf regelmäßig mitprogrammierte (wen's interessiert: OpenJDK, Jersey, JCommander, das Maven-Universum und Liquibase). Für dieses private Zeitinvestment wurde ich auch schonmal „nützlicher Idiot" genannt. Ich bin jedoch überzeugt, dass die Welt eine bessere wird, wenn jeder ein klein wenig seiner Freizeit in die Gemeinschaft und deren Infrastruktur investiert. Bei mir ist es eben – wie bei so vielen im iJUG – ein bisschen mehr.

Viele Projekte haben sich prächtig entwickelt, wie zum Beispiel OpenJDK und Maven, einige köcheln eher auf Sparflamme vor sich hin, wie beispielsweise JCommander, und viele von den 63 gibt es längst nicht mehr. Spaß gemacht haben sie aber alle – bis zu einem gewissen Grad.

Der Spaß endet nämlich dann, wenn man nur noch „der Dumme" ist, der in der Wartungsphase eines längst ausgereiften Projekts als Einzelkämpfer die gesamte Wartung und Weiterentwicklung stemmt, ohne irgendeine Hilfe oder einen Dank zu erhalten – oder (im Unterschied zu hauptberuflichen Marktschreiern aka Developer Advocates) gar eine Bezahlung. So geht es nicht nur mir. Viele (die meisten?) Open-Source-Projekte stecken in derselben Misere, und wie die xz-Backdoor gezeigt hat, führt dieser Zustand zu massiven Problemen für die gesamte Gesellschaft.

Bereits in der Corona-Pandemie zeigte sich, dass die Gesellschaft sich sehr gerne auf die Hilfe des Gesundheitssektors (aka der gesellschaftlichen Infrastruktur) verlassen hat, dem Personal der Intensivstationen gerne großherzig Standing Ovations gespendet hat (naja, mindestens einmal haben wir doch alle am Balkon mitgeklatscht, und sei es, weil das so chic und trendy war), um selbigen bei nächster Gelegenheit sowohl personelle Entlastung als auch einen angemessenen Inflationsausgleich zu verwehren. Und – schlimmer noch – nun gar im Nachgang ebenso ernsthaft wie unbegründet fordert, für die damalige „Entlastung des Gesundheitssystems" sich gefälligst bei den Generationen Z und Alpha auch noch zu entschuldigen. Man fühlt sich angesichts dieser sich schnell wandelnden Haltung fast wie in einem surrealen Stück à la Kafka oder Dürrenmatt (oder einfacher gesagt, „im falschen Film")! Kündigungen und Burn-outs sind die Folge dieser gesamtgesellschaftlichen verzerrten Realitätswahrnehmung und das Gesundheitssystem steht schlechter da als je zuvor, ist weitgehend marode und je nach Kommune erodiert bis non-existent. Und zwar, weil die Gesellschaft als Menge der Individuen das offensichtlich so will (sonst würde man es ja ändern, wir sind ja angeblich alles soziale Wesen und des logischen Denkens mächtig).

Gleiches droht uns nun auf dem Feld der Softwareentwicklung mit unserer Infrastruktur, die weitgehend auf Open Source basiert. Die Gesellschaft als Menge der Entwickler verlässt sich darauf, dass bald wieder ganz viel neuer „heißer Scheiß" kommt, zum Beispiel in Form von Jakarta EE 11. Und gleichzeitig sind immer weniger Individuen bereit, in dessen Entwicklung zu investieren – weder Personen noch Unternehmen. Seit Jahren betteln wir mal lauter, mal leiser um Hilfe, sei es hier in dieser Kolumne, sei es auf Konferenzen oder bei JUGs. Und von euch kommt: Nichts. Der Rest ist Schweigen (Hamlet, 5. Aufzug, 2. Szene). Vorhang!

Epilog: Demnächst erscheint als neueste Inkarnation des laufenden Trauerspiels die Version 11 von Jakarta EE. Aufgrund der soeben dargestellten Milieustudie bleibt es der Fantasie des geneigten Lesers überlassen, wie viel toller, neuer „heißer Scheiß" darin wohl zu finden sein wird, und eigene Rückschlüsse zu ziehen, wer daran Schuld trägt und in wessen Macht es wohl stünde, dies im unvermeidlichen Sequel endlich zu ändern.

Markus Karg
markus@headcrashing.eu

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

Die goldenen Regeln: Teil 2

Andreas Monschau, Haeger Consulting

Stetig bin ich auf der abenteuerlichen Suche nach ihnen – den goldenen Regeln, mit denen man Neulingen, Junioren oder Quereinsteigern den Start möglichst vermiesen kann. Softwareentwicklung ist hart und unfair. So soll es auch bleiben. Du hast gelitten, also sollen alle leiden. Diese Regeln entspringen nicht meiner bunten Fantasie, sie finden sich noch in vielen Projekten, Unternehmen und Organisationen wieder – mit dieser Sammlung möchte ich zum einen erheitern und zum anderen mahnen. Mahnen, diese Regeln nicht mehr anzuwenden und Menschen, die darunter leiden ermutigen, sich zu wehren. Hast du weitere Beispiele für Regeln und möchtest sie mit mir und anderen teilen? Dann kontaktiere mich gerne!

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

Regel 2: Die Ungewissheit der ersten Wochen

In den späten siebziger Jahren des 20. Jahrhunderts gab es einen kleinen Science-Fiction-Film, der Horrorelemente in ungewohntem Ausmaß in die Handlung integrierte. An den Film wird sich heute kaum einer erinnern, er ist mit Sicherheit in Vergessenheit geraten und hat mit Sicherheit auch keine Fortsetzung mit stark schwankender Qualität nach sich gezogen. Die Tagline des Films lautete seinerzeit: „IN SPACE, NO ONE CAN HEAR YOU SCREAM".

Und damit sind wir auch schon genau im richtigen Mindset gelandet – der Neuling kommt an, und dann ist er erstmal allein. Mit sich und seinen Gedanken, denn Hardware hat er keine, und sollte er Hardware haben, dann hat er keine Zugänge zu wichtigen Systemen. Niemand bemerkt ihn, niemand hört auf ihn. Natürlich hat er viele Fragen, aber niemand ist da, der sie ihm beantwortet. Insbesondere du nicht, denn du bist für ihn verantwortlich, aber eigentlich ist er dir egal. Anrufe ignorierst du, ebenso wie E-Mails oder Chat-Nachrichten. Das höchste Maß an Erwiderung ist der Hinweis, dass du dich „später mal melden wirst, wenn du Zeit hast". Das geht auch so lange gut, bis der Neuling dir in Persona begegnet, vor der Kaffeemaschine oder er sucht dein Büro auf. In diesem Fall musst du stets eine gute Ausrede parat haben: Das in wenigen Minuten ein wichtiges Meeting mit ungewisser Länge anstehe, das zieht beispielweise immer. Remote-Arbeit sei Dank, wird es aber nicht oft so sein, dass man sich in der „Realität" begegnet.

Sorge auch dafür, dass deine Vertretungen, wenn du welche hast, entsprechend eingeweiht sind. Daher formuliere ich die oben genannte Tagline um, passe sie etwas an und lasse dies den Neuling einfach spüren: „IN YOUR PROJECT, NO ONE WANTS TO HEAR YOU".

Irgendwann bist du der Einsamkeit des Neulings aber sicher überdrüssig – schließlich muss er langsam auch mal mit Aufgaben versorgt werden, und das sollte wohlüberlegt sein. Am besten sprichst du dich mit deinem Team ab. Nehmen wir hierbei einfach mal an, dass das Team, inklusive dir, fünf Personen umfasst. Das heißt, hier kann man schon mal ordentlich Verwirrung stiften, denn so viel ist klar: Kommt jemand Neues in dein Team oder ins Projekt, dann sind alle unvorbereitet und agieren erstmal völlig aktionistisch. Was könnte nun von dir und deinen Leuten so gesagt werden? Hier ein paar Beispiele für Vorschläge zur Beschäftigung, direkt mit Bewertung, ob es sinnvoll ist oder nicht.

DEV 1: „Fix ein paar Bugs!"
Na klar, das ist absolut sinnvoll. Der Neuling hat weder Ahnung von der Fachlichkeit, noch von der Implementierung.

DEV 2: „Auf keinen Fall Bugfixing!"
Auch das ist absolut sinnvoll. Der Neuling hat weder Ahnung von der Fachlichkeit, noch von der Implementierung.

DEV 3: „Der Neue kann doch ein paar 1-Punkt-Stories abarbeiten!"
Auch klar, das ist ebenfalls sinnvoll. Der Neuling hat weder Ahnung von der Fachlichkeit, noch von der Implementierung.

DEV 4: „Brauchen wir wirklich einen weiteren Developer?"
Das ist sehr gut, denn nach dem initialen Alleinlassen fühlt sich der Neuling nach dieser Aussage noch mehr fehl am Platze.

DEV 5: „Lies' lieber erstmal die Doku!"
Der Klassiker! Damit macht man doch nichts verkehrt! Oder vielleicht doch?

Über das Thema Dokumentation wird es im dritten Teil dieser Reihe noch einiges zu lesen geben…

Andreas Monschau
Haeger Consulting
amonschau@haeger-consulting

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

Java 22 – Was gibt es Neues?

Falk Sippach, embarc Software Consulting GmbH

Scheinbar gerade erst, ist im September 2023 mit dem OpenJDK 21 eine Long-Term-Support-Version (LTS) erschienen. Und da geht es auch schon mit dem nächsten Major-Release weiter. Wie jedes halbe Jahr lohnt sich erneut der Blick auf die Neuerungen und die Änderungen an den bereits in früheren Versionen als Inkubator oder Preview erschienenen Features. Es sind wieder sowohl kleine, für uns Entwickler aber sehr nützliche Funktionen dabei und es werden natürlich die großen Themenblöcke weiterhin verfolgt.

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK *[1]* ein guter Startpunkt. Dort sind diesmal immerhin 12 JEPs (JDK Enhancement Proposals) gelistet:

- 423: Region Pinning for G1
- 447: Statements before super(...) (Preview)
- 454: Foreign Function & Memory API
- 456: Unnamed Variables & Patterns
- 457: Class-File API (Preview)
- 458: Launch Multi-File Source-Code Programs
- 459: String Templates (Second Preview)
- 460: Vector API (Seventh Incubator)
- 461: Stream Gatherers (Preview)
- 462: Structured Concurrency (Second Preview)
- 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)
- 464: Scoped Values (Second Preview)

Es sind zwar ein paar Themen weniger als beim letzten Release, aber mit den Statements before super(), der Class File API, dem Launch Multi-File Source-Code Programs und den Stream Gatherers sind auch vier ganz neue dabei.

Vereinfachungen beim Starten von Java-Anwendungen

Programmierneulingen fällt der der Einstieg in Java häufig schwer. Seit einiger Zeit gibt es aber immer wieder kleine Verbesserungen, die die ersten Schritte erleichtern. Bereits seit Java 9 wird die *JShell* mitgeliefert. In dieser REPL (*Read Eval Print Loop*) lassen sich sehr schnell kleine Programmierbeispiele ausprobieren, ohne sich mit dem Thema Kompilieren und Build-Management befassen zu müssen.

In Java 11 wurden *Launch Single-File Source-Code Programs* eingeführt, wodurch einzelne Java-Dateien (mit Klassendeklaration und `main`-Methode) ohne separaten Kompilierungsschritt einfach von der Konsole gestartet werden können.

In Java 21 kam mit dem JEP 445 (*Unnamed Classes and Instance main Methods*) ein Preview hinzu, um Java-Main-Anwendungen viel schlanker und ohne überflüssigen Boilerplate-Code zu definieren. Somit lässt sich die `main`-Methode viel kompakter schreiben (ohne `public`, `static`, `String[] args`, ...). Außerdem muss sie auch nicht mal mehr in eine Klassendefinition eingebettet sein. Gerade für Programmieranfänger, die in Java mit einem einfachen Hello-World-Programm starten, stellte die bisherige Vorgehensweise eine übermäßig große Hürde dar. Sie müssen bereits zu Beginn gleich mehrere, in dem Moment nicht relevante Konzepte (Klassen, statische Methoden, String-Arrays und so weiter) verstehen. Im Zusammenhang mit dem oben erwähnten, im OpenJDK 11 als JEP 330 eingeführten L*aunch Single-File Source-Code Programs*, sinkt der Aufwand weiter, da man in der `*.java`-Datei nur noch eine schlanke `main`-Methode benötigt und sogar auf eine Klassendefinition verzichten kann.

Für Java 22 wurde das Konzept nochmal überarbeitet und mit dem JEP 463 (*Implicitly Declared Classes and Instance Main Methods*) ein zweiter Preview vorgelegt. Dabei gibt es jetzt keine *Unnamed Classes* mehr, vielmehr wird von der Laufzeitumgebung ein Name gewählt. Bei ersten Tests der neuen Funktion wurde der Name der Java-Datei (`ImplicitMain.java` => Klasse `ImplicitMain`) verwendet. Das scheint für das OpenJDK der Default zu sein, kann aber bei anderen Distributionen abweichen. Diese implizit deklarierten Klassen verhalten sich wie normale Top-Level-Klassen und benötigen auch keine zusätzliche Unterstützung mehr beim Tooling, der Verwendung von Bibliotheken oder in der Laufzeitumgebung. Die einzige Bedingung ist, dass die Datei im Root-Package liegen muss.

Ein einfaches Beispiel inklusive des Kommandozeilenbefehls zeigt *Listing 1*. Da es sich noch um ein Preview-Feature handelt, muss die Angabe zur aktuellen Java-Version erfolgen und die Preview-Funktion aktiviert werden.

```
// > java --source 22 --enable-preview Main.java
void main() {
     System.out.println("Hello, World!");
}
```

Listing 1: Instance Main Method

Die implizit deklarierte Klasse darf sogar noch weitere Attribute/Felder und Methoden enthalten *(siehe Listing 2)*. Dadurch fühlt sich Java jetzt fast schon wie eine Skriptsprache an und es werden ganz neue Anwendungsfälle möglich. Wir können nun Shell-Skripte schreiben und profitieren dabei von dem mächtigen Funktionsumfang und der Typsicherheit Javas.

```
// > java --source 22 --enable-preview Main.java
final String greeting = "Hello";

void main() {
  System.out.println(greet("World"));
}

String greet(String name) {
  return STR."\{greeting}, \{name}!";
}
```

Listing 2: Zusätzliche Felder und Methoden mit der Instance-Main-Methode

```
// Gewinner, da mit String[] args
void main(String[] args) {
     System.out.println("instance main with String[] args");
}

// nicht erlaubt, da es schon eine nicht-statische main-Methode mit der gleichen Signatur gibt
/*
static void main(String[] args) {
     System.out.println("static main with String[] args");
}
*/

// wird beim Starten nicht aufgerufen
void main() {
     System.out.println("instance main without String[] args");
}
```

Listing 3: Aufrufreihenfolge von main-Methoden in unbenannten Klassen

Da es theoretisch mehrere `main`-Methoden in einer implizit deklarierten Klasse geben kann, wird eine Priorisierung beim Aufruf der Start-Prozedur benötigt. Die Logik dahinter hat sich im zweiten Preview deutlich vereinfacht. Es wird nur noch unterschieden, ob es einen Parameter vom Typ String-Array gibt oder nicht. Wenn ja, dann hat diese Methode Vorrang. In *Listing 3* wird die oberste `main`-Methode aufgerufen. Der Compiler verbietet, dass es sowohl eine statische als auch eine Instanz-Methode mit der gleichen Methodensignatur gibt. Die `main`-Methode mit dem String-Array dürfte also auch *static* sein. Sichtbarkeitsattribute wie `public`, `protected` und so weiter werden an dieser Stelle ignoriert und der Compiler würde wiederum verhindern, dass es eine private und eine öffentliche Methode mit der gleichen Methoden-Signatur gibt.

Die weiter oben erwähnten *Launch Single-File Source-Code Programs* haben in Java 22 auch ein Update bekommen. Bisher konnte nur der Code in genau einer Datei direkt ohne vorherige Kompilierung ausgeführt werden. In einer Datei sind zwar auch mehrere Klassen erlaubt, das wird aber schnell unübersichtlich. Die Klassen sollten in separate Dateien aufgeteilt werden, da funktioniert der *Single-File Source-Code* aber nicht mehr. Mit dem JEP 458 (Launch Multi-File Source-Code Programs) darf der Code nun in beliebig vielen Java-Dateien strukturiert sein. Zusätzlich zu den Multi-Files kann auch Code aus JARs (die im Unterverzeichnis `libs` liegen) mittels `--class-path 'libs/*'` eingebunden werden.

Zwei Besonderheiten gilt es noch zu beachten. Liegt eine der Klassen in einem Package, zum Beispiel in „de.embarc", dann muss auch die Klasse im entsprechenden Unterverzeichnis „de/embarc" liegen und das Java-Kommando muss um den vollständigen Pfad erweitert werden: „java de/embarc/Klasse.java". Weitere Informationen hält die Beschreibung des JEP 458 bereit, unter anderem wie dieses Feature bei der Verwendung von Modulen (JPMS) angewendet wird.

```
class Person {
  private final String firstname;
  private final String lastname;
  private final LocalDate birthdate;

  public Person(String firstname, String lastname, LocalDate birthdate) {
     this.firstname = firstname;
     this.lastname = lastname;
     this.birthdate = birthdate;
  }
}

class Employee extends Person {
  private final String company;

  public Employee(String name, LocalDate birthdate, String company) {
     if (company == null || company.isEmpty()) {
        throw new IllegalArgumentException("company is null or empty");
     }
     String[] names = name.split("\\s" );
     super(names[0], names[1], birthdate);
     this.company = company;
  }
}

System.out.println(
  new Employee("Dieter Develop", LocalDate.now(), "embarc"));
```

Listing 4: Statements before super

Statements before super()

Die *Statements before super()* ermöglichen in Konstruktoren das Einfügen von Codezeilen vor dem eigentlich in der ersten Zeile obligatorischen `super()`- oder `this()`-Aufruf. Der Grund für die bisherige Einschränkung ist die Sicherstellung durch den Compiler, dass zunächst potenzielle Oberklassen vollständig initialisiert sein müssen, bevor der Zustand der Sub-Klasse hergestellt oder abgefragt werden kann. Bei `this()` geht es zwar um eine Delegation an einen überladenen Konstruktor in der gleichen Klasse, aber dieser wird letztlich auch explizit oder implizit den `super`-Konstruktor der Oberklasse aufrufen.

Genau genommen gelten die strengen Anforderungen auch weiterhin. Die Statements, die man jetzt vor dem `super()`-Aufruf verwenden darf, dürfen nicht schreibend oder lesend auf die aktuelle Instanz zugreifen. Aber die Validierung oder die Transformation von Eingabeparametern sowie das Aufspalten eines Parameters in Einzelteile kann nun vor dem Aufruf von `super()` erfolgen *(siehe Listing 4)*.

Bisher wurde dafür typischerweise ein überladener Hilfskonstruktor oder eine private Methode als Workaround benötigt. Das hat den Lesefluss gestört und die Implementierung unnötig komplex gemacht. Im Falle von Berechnungen auf den Input-Parametern werden Werte zudem unnötigerweise mehrfach ermittelt. Beim Beispiel in *Listing 4* müsste das Aufspalten in Vor- und Nachname zweimal erfolgen, um einmal das erste Element des `split()`-Aufrufs als ersten Parameter und dann das zweite Element aus der zweiten Berechnung als zweiten Parameter zu übergeben.

Alle Zeilen vor dem `super()`- oder `this()`-Aufruf werden *Prolog* genannt. Codezeilen nach beziehungsweise ohne `super()` oder `this()` werden *Epilog* genannt. Im Prolog darf nicht lesend auf Felder der Klasse oder der Oberklasse zugegriffen werden, da diese möglicherweise noch nicht initialisiert sein könnten. Außerdem dürfen keine Instanz-Methoden der Klasse aufgerufen und keine Instanzen von nicht-statischen inneren Klassen erzeugt werden, da diese potenziell eine Referenz auf das noch nicht fertig initialisierte Elternobjekt halten können.

Im Gegensatz dazu darf im Prolog des Konstruktors einer nicht-statischen inneren Klasse auf Felder und Methoden der äußeren Klassen zugegriffen werden.

Auch bei Records und Enums sind Codezeilen vor `this()` nach den oben genannten Regeln erlaubt. Da sie nicht von anderen Klassen

```
// Quelle
var number = Arrays.asList("abc1", "abc2", "abc3").stream()
  .skip(1) // 1. Intermediate Operation
  .map(element -> element.substring(0, 3)) // 2. Intermediate Operation
  .sorted() // 3. Intermediate Operation
  .count(); // Terminal Operation
System.out.println(number);
```

Listing 5: Verschiedene Stufen der Stream-Pipeline

```
1
// will contain: Optional["12345"]
Optional<String> numberString = Stream.of(1, 2, 3, 4, 5)
  .gather(Gatherers.fold(
     () -> "", (string, number) -> string + number))
  .findFirst();
System.out.println(numberString);

// will contain: ["1", "12", "123"]
List<String> numberStrings = Stream.of(1, 2, 3)
  .gather(Gatherers.scan(
     () -> "", (string, number) -> string + number))
  .toList();
System.out.println(numberStrings);

// will contain: [[1, 2, 3], [4, 5, 6], [7]]
List<List<Integer>> windows = Stream.of(1, 2, 3, 4, 5, 6, 7)
  .gather(Gatherers.windowFixed(3)).toList();
System.out.println(windows);

// will contain: [[1, 2], [2, 3], [3, 4], [4, 5]]
List<List<Integer>> windows2 = Stream.of(1, 2, 3, 4, 5)
  .gather(Gatherers.windowSliding(2))
  .toList();
System.out.println(windows2);

// will contain: [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
List<List<Integer>> windows3 = Stream.of(1, 2, 3, 4, 5)
  .gather(Gatherers.windowSliding(3))
  .toList();
System.out.println(windows3);
```

Listing 6: Mitgelieferte Gatherers

ableiten können, verwenden wir in dem Umfeld keine Aufrufe von `super()`.

Benutzerdefinierte Zwischenoperationen bei Streams

Die Stream-API wurde in Java 8 eingeführt. Ein Stream wird erst bei Bedarf ausgewertet und kann potenziell eine unbeschränkte Anzahl von Werten enthalten. Sie können sequenziell oder parallel verarbeitet werden. Eine Stream-Pipeline besteht typischerweise aus drei Teilen, der Quelle, ein oder mehreren *Intermediate Operations* und einer abschließenden Operation. *Listing 5* zeigt ein Beispiel.

Im JDK gibt es eine begrenzte Anzahl an vordefinierten *Intermediate Operations* wie `filter`, `map`, `flatMap`, `mapMulti`, `distinct`, `sorted`, `peak`, `limit`, `skip`, `takeWhile` und `dropWhile`. Es kommt immer wieder der Wunsch nach weiteren Methoden wie `window` oder `fold` auf. Aber anstatt nur genau die geforderten Operationen bereitzustellen, wurde jetzt eine API (JEP 461: Stream Gatherers) als Preview entwickelt. Diese ermöglicht es sowohl den JDK-Entwicklern als auch den normalen Anwendern, beliebige *Intermediate Operations* selbst zu implementieren.

Es werden die folgenden *Gatherer* mitgeliefert, erreichbar über die Klasse `java.util.stream.Gatherers` (*Listing 6* zeigt einige Beispiele):

- fold: zustandsbehafteter N-1-Gatherer, baut ein Aggregat inkrementell auf und gibt dieses Aggregat am Ende aus
- mapConcurrent: zustandsbehafteter 1-1-Gatherer, der die übergebene Funktion nebenläufig für jedes Input-Element aufruft
- scan: zustandsbehafteter 1-1-Gatherer, wendet eine Funktion auf dem aktuellen Zustand und dem aktuellen Element an, um das nächste Element zu erzeugen
- windowFixed: zustandsbehafteter N-N-Gatherer, der Eingabeelemente in Listen vorgegebener Größe gruppiert
- windowSliding: ähnlich zu windowFixed, nach dem ersten Rahmen wird der nächste Rahmen erzeugt, in dem das erste Element gelöscht wird und alle weiteren Werte nachrutschen

Werfen wir zunächst einen Blick auf den Aufbau eines Stream-Gatherers. Er kann einen Status besitzen, sodass Elemente abhängig von den vorherigen Aktionen unterschiedlich transformiert werden können. Er kann einen Stream auch vorzeitig terminieren, wie beispielsweise `limit()` oder `takeWhile()`. Der *Gatherer* startet mit einem optionalen *Initializer*, der den Status bereitstellen kann. Dann folgt der Integrator, der jedes Element des Streams verarbeitet und gegebenenfalls den Status aktualisiert. Dann folgt der optionale *Finisher*, der nach der Verarbeitung des letzten Elements aufgerufen wird und gegebenenfalls je nach Status weitere Elemente an die nächste Stufe der Stream-Pipeline sendet. Zu guter Letzt kombiniert ein optionaler *Combiner* den Status von parallel ausgeführten Transformationen.

Um eine eigene Implementierung zu schreiben, muss man vom Interface-Gatherer ableiten und mindestens die Methode `integrator()` implementieren. Die anderen bringen eine Default-Implementierung mit und sind daher optional *(siehe Listing 7)*.

Wenn man den *Gatherer* `windowFixed` nochmal implementieren möchte, würde es wie in *Listing 8* aussehen. Das Beispiel ist aus der Beschreibung des JEP 461 übernommen *[2]*.

Analysieren und Manipulieren von Java-Bytecode

Die Class-File API (JEP 457) ermöglicht das Lesen und Schreiben von `class`-Dateien, also von kompiliertem Bytecode. Sowohl das JDK selbst als auch viele Bibliotheken und Frameworks haben für diese Aufgabe bisher auf ASM gesetzt, ein universelles Java-Bytecode-Manipulations- und Analyse-Framework *[3]*. Es kann sowohl zum Modifizieren existierender Klassen als auch zum dynamischen Generieren von Klassen im Binärformat verwendet werden. Neben dem OpenJDK kommt ASM unter anderem auch beim *Groovy*- und *Kotlin*-Compiler, einigen Test-Coverage-Tools (*Cobertura*, *JaCoCo*) und Build-Management-Werkzeugen (*Gradle*) zum Einsatz. *Mockito* verwendet es indirekt, um per *Byte Buddy* Mock-Klassen zu generieren. Aufgrund der kürzeren Release-Zyklen des OpenJDK fällt es nicht leicht, dass ASM mit den Änderungen des Bytecodes mithält. Das führt dann wiederum zu Abhängigkeiten und somit können die oben genannten Tools und Frameworks nicht schnell genug mit neuen OpenJDK-Releases umgehen. Mit der Entwicklung der JDK-internen Class-File API sollen solche Abhängigkeiten minimiert werden.

ASM hat bestehenden Bytecode auf Basis des Visitor-Patterns analysiert und modifiziert. Das Visitor-Pattern ist sperrig und unflexibel. Es ist ein Workaround für die fehlende Unterstützung von Pattern

```
interface Gatherer<T, A, R> {
  default Supplier<A> initializer() {
   return defaultInitializer();
  };

  Integrator<A, T, R> integrator();

  default BinaryOperator<A> combiner() {
     return defaultCombiner();
  }

  default BiConsumer<A, Downstream<? super R>> finisher() {
     return defaultFinisher();
  };
  [...]
}
```

Listing 7: Interface Gatherer

Matching. Da Java Pattern Matching mittlerweile unterstützt, kann der notwendige Code in der neuen Class-File API direkter und konsistenter ausgedrückt werden. *Listing 9* zeigt ein Beispiel, welches auch im JEP 457 beschrieben ist.

Das Erzeugen von Klassen erfolgt im Gegensatz zum Visitor-Ansatz von ASM mit Buildern. Um beispielsweise eine Methode `foobar` *(siehe Listing 10)* zu erzeugen, kann das ebenfalls aus dem JEP 457 entnommene Code-Beispiel verwendet werden *(siehe Listing 11)*.

Es kann auch existierender Code verändert werden. *Listing 12* zeigt beispielhaft, wie bei einer bestehenden Klasse alle Methoden gelöscht werden, die mit „debug" beginnen.

Wiederkehrende Features

Mit der *Foreign Function & Memory API* wurde diesmal ein echter Langläufer abgeschlossen. Die beiden Teile sind bereits seit Version 14 beziehungsweise 16 im JDK zunächst einzeln und ab 17 als gemeinsamer Inkubator-JEP enthalten. Wer schon sehr lange in der

```
record WindowFixed<TR>(int windowSize)
  implements Gatherer<TR, ArrayList<TR>, List<TR>> {

  public WindowFixed {
     // Validate input
     if (windowSize < 1)
        throw new IllegalArgumentException("window size must be positive" );
  }

  @Override
  public Supplier<ArrayList<TR>> initializer() {
     // Create an ArrayList to hold the current open window
     return () -> new ArrayList<>(windowSize);
  }

  @Override
  public Integrator<ArrayList<TR>, TR, List<TR>> integrator() {
     // The integrator is invoked for each element consumed
     return Gatherer.Integrator.ofGreedy((window, element, downstream) -> {

        // Add the element to the current open window
        window.add(element);

        // Until we reach our desired window size,
        // return true to signal that more elements are desired
        if (window.size() < windowSize)
           return true;

        // When the window is full, close it by creating a copy
        var result = new ArrayList<TR>(window);

        // Clear the window so the next can be started
        window.clear();

        // Send the closed window downstream
        return downstream.push(result);

     });
  }

  // The combiner is omitted since this operation is intrinsically sequential,
  // and thus cannot be parallelized

  @Override
  public BiConsumer<ArrayList<TR>, Downstream<? super List<TR>>> finisher() {
     // The finisher runs when there are no more elements to pass from
     // the upstream
     return (window, downstream) -> {
        // If the downstream still accepts more elements and the current
        // open window is non-empty, then send a copy of it downstream
        if (!downstream.isRejecting() && !window.isEmpty()) {
           downstream.push(new ArrayList<TR>(window));
           window.clear();
        }
     };
  }
}

// [[1, 2], [3, 4], [5]]
System.out.println(
  Stream.of(1,2,3,4,5)
  .gather(new WindowFixed(2))
  .toList());
```

Listing 8: Verschiedene Stufen der Stream-Pipeline

```
CodeModel code = ...
Set<ClassDesc> deps = new HashSet<>();
for(CodeElement e : code) {
  switch (e) {
     case FieldInstruction f   -> deps.add(f.owner());
     case InvokeInstruction i -> deps.add(i.owner());
     ... and so on for instanceof, cast, etc ...
  }
}
```

Listing 9: Parsen von Klassen mit Pattern Matching

```
void fooBar(boolean z, int x) {
  if (z)
     foo(x);
  else
     bar(x);
}
```

Listing 10: Zu erzeugende Methode

Java-Welt unterwegs ist, wird das *Java Native Interface* (JNI) kennen. Damit kann nativer C-Code aus Java heraus aufgerufen werden. Der Ansatz ist aber relativ aufwendig und fragil. Die *Foreign Function API* bietet einen statisch typisierten, rein Java-basierten Zugriff auf nativen Code (C-Bibliotheken). Diese Schnittstelle kann den bisher fehleranfälligen und langsamen Prozess der Anbindung einer nativen Bibliothek beträchtlich vereinfachen. Mit der *Foreign Memory Access API* bekommen Java-Anwendungen die Möglichkeit, außerhalb des Heap zusätzlichen Speicher zu allokieren. Ziel der neuen APIs ist es, den Implementierungsaufwand um 90 % zu reduzieren und die Leistung um den Faktor 4 bis 5 zu beschleunigen. In diesem Release wurde die FFM-API finalisiert. Dabei gab es nur noch ein paar kleinere Verbesserungen basierend auf dem Feedback der vergangenen Releases.

Auch die *Vector API* ist nun schon das siebte Mal als Inkubator enthalten und taucht seit Java 16 regelmäßig in den Releases auf. Es geht dabei um die Unterstützung der modernen Möglichkeiten von *Single-Instruction-Multiple-Data*-Rechnerarchitekturen (SIMD) mit Vektorprozessoren. SIMD lässt viele Prozessoren gleichzeitig unterschiedliche Daten verarbeiten. Durch die Parallelisierung auf Hardware-Ebene verringert sich beim SIMD-Prinzip der Aufwand für rechenintensive Schleifen.

Der Grund für die lange Inkubationsphase der *Vector API* wird in den Zielen des JEP 460 erklärt *[4]*:

Alignment with Project Valhalla – The long-term goal of the Vector API is to leverage Project Valhalla's enhancements to the Java object model. Primarily this will mean changing the Vector API's current value-based classes to be value classes so that programs can work with value objects, i.e., class instances that lack object identity. Accordingly, the Vector API will incubate over multiple releases until the necessary features of Project Valhalla become available as preview features.

Man wartet also auf die Reformen am Typsystem. Java hat aktuell ein zweigeteiltes Typsystem mit den primitiven Datentypen sowie Referenztypen (Klassen). Die primitiven Datentypen wurden ursprünglich aus Performanceoptimierungsgründen eingeführt, haben aber im Handling entscheidende Nachteile. Referenztypen sind auch nicht immer die beste Wahl, insbesondere was die Effizienz und den Speicherverbrauch angeht. Es braucht etwas dazwischen, was sich so schlank und performant wie primitive Datentypen verhält, aber auch die Vorzüge von selbst zu erstellenden Referenztypen in Form von Klassen kennt. Schon bald könnten daher aus dem Inkubator-Projekt *Valhalla Value Types* (haben keine Identität) und *Universal Generics* (List<int>) ins JDK übernommen werden. Für Java 22 hat es der JEP 401 (*Value Classes and Objects*) leider noch nicht geschafft. Dementsprechend werden wir die *Vector API* wohl auch noch einige Releases als Inkubator-Feature wiedersehen. Diesmal gab es nur wenige Änderungen an der API im Vergleich zum JDK 21, hauptsächlich Bugfixes und kleinere Performance-Verbesserungen.

```
CodeBuilder classBuilder = ...;
classBuilder.withMethod("fooBar",
  MethodTypeDesc.of(CD_void, CD_boolean, CD_int),
  flags,
  methodBuilder -> methodBuilder
     .withCode(codeBuilder -> {
        codeBuilder
           .iload(codeBuilder.parameterSlot(0))
           .ifThenElse(
              b1 -> b1.aload(codeBuilder.receiverSlot())
                      .iload(codeBuilder.parameterSlot(1))
                      .invokevirtual(
                         ClassDesc.of("Foo"),
                         "foo",
                         MethodTypeDesc.of(CD_void, CD_int)),
              b2 -> b2.aload(codeBuilder.receiverSlot())
                      .iload(codeBuilder.parameterSlot(1))
                      .invokevirtual(
                          ClassDesc.of("Foo"),
                             "bar",
                             MethodTypeDesc.of(
                                 CD_void, CD_int))
           .return_();
});
```

Listing 11: Erzeugen der Methode aus Listing 10

```
ClassFile cf = ClassFile.of();
ClassModel classModel = cf.parse(bytes);
byte[] newBytes =
  cf.build(
     classModel.thisClass().asSymbol(),
     classBuilder -> {
        for (ClassElement ce : classModel) {
           if (!(ce instanceof MethodModel mm
                   && mm.methodName().stringValue()
                      .startsWith("debug"))) {
                         classBuilder.with(ce);
           }
        }
     });
```

Listing 12: Bestehende Klasse transformieren

Weiterer Baustein beim *Pattern Matching* finalisiert

Ebenfalls abgeschlossen wurden die im JDK 21 zunächst als Preview eingeführten *Unnamed Variables & Patterns* (JEP 456). Seit der letzten Version gab es keine Änderungen. Die *Unnamed Patterns* sind Teil des *Pattern Matchings*, das vor vielen Jahren in Java eingeführt worden ist. Erst in Java 21 wurden mit *Pattern Matching for switch* und *Record Patterns* zwei größere Teile abgeschlossen.

Beim *Pattern Matching* geht es darum, bestehende Strukturen mit Mustern abzugleichen, um komplizierte Fallunterscheidungen effizient und wartbar implementieren zu können. Ein Pattern ist dabei eine Kombination aus einem Prädikat, das auf die Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert.

Die Intention des *Pattern Matching* ist die Destrukturierung von Datenobjekten, also das Aufspalten in die Bestandteile und das Zuweisen in einzelne Variablen zur weiteren Bearbeitung.

Je nach Anwendungsfall sind die Deklarationen von Variablen oder geschachtelten Patterns zwar für den Compiler notwendig, werden aber nie verwendet. Um unnötige Warnungen vom Compiler und Lint-Tools zu vermeiden und die Intention, dass diese Variable nicht verwendet wird, besser ausdrücken zu können, kann sie einfach durch einen Unterstrich ersetzt werden. Da der Unterstrich kein Variablenbezeichner ist, kann er in Java-Anweisungen in einer Zeile auch mehrfach eingesetzt werden. In *Listing 13* wird im Zweig 1 der Wert auf die Empty-Instanz nicht benötigt. Auch in den Zweigen 2 und 3 wird jeweils nur eine der beiden Record-Properties verwendet. Die anderen Werte lassen sich einfach mit dem „_“ ignorieren.

```
static <T> boolean contains(T value, LinkedList<T> list) {
  return switch (list) {
     case Empty _ -> false;
     case Element<T>(T v, _)
        when Objects.equals(v, value) -> true;
     case Element<T>(_, var tail) ->
        contains(value, tail);
  };
}
```

Listing 13: Unnamed Patterns

Früher war der Unterstrich ein legaler Bezeichner. Ab Java 9 darf der einzelne Unterstrich nicht mehr verwendet werden. Variablen dürfen aber weiter mit einem Unterstrich beginnen, auch zwei Unterstriche („__“) sind möglich. Wurde im Quellcode vor Java 9 mit einem Unterstrich als Symbol für eine unbenutzte Variable gearbeitet, lässt sich dies mit dem Anhängen eines zweiten Unterstrichs leicht auf höhere Java-Versionen migrieren.

Beim *Pattern Matching* wird es in Zukunft noch weitere Neuerungen geben. Aktuell ist für das OpenJDK 23 (Veröffentlichung im September 2024) bereits der JEP 455 (*Primitive Types in Patterns, instanceof,* and *switch*) eingeplant. Dabei werden *instanceof* und *switch* so erweitert, dass auch primitive Datentypen (`int`, `long`, …) verwendet werden können. In *Listing 14* wird zur Laufzeit geprüft, ob der Wert von „i“ in den Wertebereich von `byte` passt. Wenn ja, kann direkt mit der neuen Variable „b“ weitergearbeitet werden. Weitere Themen werden in den nächsten Jahren vermutlich neue Pattern-Typen zum Matchen von Arrays, Maps, POJOs und so weiter sein.

```
int i = 1000;
if (i instanceof byte b) {
     ...
}
```

Listing 14: Primitive Types in Patterns

Das Prinzip für *Unnamed Patterns* gilt auch für deklarierte, aber nie benutzte Variablen bei Exceptions in `catch`-Blöcken, bei Lambda-Parametern oder in `for`-Schleifen *(siehe Listing 15)*. Mit hoher Wahrscheinlichkeit wird dieses kleine, unscheinbare Feature in der Zukunft sehr präsent in unserer täglichen Arbeit als Java-Entwickler werden.

```
try {
  var result = 5 / 0;
} catch(ArithmeticException _) {
  System.out.println("Division nicht möglich");
}

System.out.println(new ArrayList<>(List.of(1, 2, 3))
     .stream()
     .map(_ -> 42)
     .toList()); // [42, 42, 42]

int total = 0;
  for (Order _ : orders) {
     total++;
}
```

Listing 15: Unnamed Variables

StringTemplates erneut als Preview dabei

Bereits in Java 12 sollte mit den *Raw String Literals* eine neue Art bei der Erzeugung von Zeichenketten eingeführt werden. Da ging es um mehrzeilige Strings und das Ignorieren von Escape-Sequenzen. Der zugehörige JEP 326 *[5]* wurde allerdings aus diversen Gründen (verwirrende Syntax, anderes Verhalten als normale Strings in Bezug auf Mehrzeiligkeit, …) zurückgezogen. Aber bereits im OpenJDK 15 wurde durch *Text Blocks* die einfache Definition von mehrzeiligen Strings erlaubt. Schon damals wurden erste Stimmen laut, dass Java aber auch einen String-Interpolationsmechanismus benötige, wie das viele andere Sprachen bereits seit längerem anbieten. Bei der String-Interpolation dürfen Zeichenketten Platzhalter enthalten, deren Code-Ausdrücke automatisch aufgelöst werden.

Um Zeichenketten zur Laufzeit aus statischen Texten, dem Inhalt von Variablen und berechneten Werten zusammenzufügen, gibt es in Java schon immer viele verschiedene Möglichkeiten: String-Verkettung, StringBuilder/-Buffer, String.format(), MessageFormat. Die verschiedenen Ansätze haben aber ganz unterschiedliche Herausforderungen. Durch den unveränderlichen Charakter (Immutability) von String-Instanzen kann die Performance bei String-Verkettungen ein Problem sein, außerdem leidet nahezu bei allen Varianten die Lesbarkeit und es kann leicht zu Fehlern kommen (zum Beispiel

beim Zusammenbauen von Datenbank-Abfragen).

Ab Java 21 lassen sich *Text Templates* definieren (JEP 430). Im OpenJDK 22 wurde nun der zweite Preview veröffentlicht. Es gab aus Anwendersicht keine relevanten Änderungen und die Entwickler möchten zunächst weitere Erkenntnisse sammeln und wünschen sich Feedback.

Innerhalb eines *String Template* dürfen die durch \{..} gekennzeichneten Platzhalter Variablen oder auch beliebige Java-Ausdrücke (Berechnungen, Methodenaufrufe, ...) enthalten. Ein einfaches Beispiel zeigt *Listing 16*. Die Template Expression besteht aus dem Template Prozessor (STR), einem Punkt (.) und dem eigentlichen Template, das Platzhalter mit eingebetteten Ausdrücken enthalten darf.

Die Syntax scheint auf den ersten Blick gewöhnungsbedürftig. Das Template wird der Prozessor-Instanz hinter einem Punkt (ähnlich einem Instanz-Methoden-Aufruf) übergeben. Diese Schreibweise ist der Abwärtskompatibilität geschuldet, damit bestehender Code sich weiter korrekt verhält. Die Zeichenfolge \{ ist keine gültige Escape-Sequenz, dementsprechend kann es im bestehenden Java-Quellcode keine Strings geben, die \{ enthalten. Der Compiler hätte in Versionen vor 21 einen Fehler geworfen. Dadurch ist sichergestellt, dass kein alter Code fälschlicherweise als String-Template erkannt wird. In anderen Programmiersprachen mag die Interpolation einfacher gelöst sein, die Syntax in Java bietet aber sehr flexible und vielseitige Möglichkeiten, gerade durch die Entwicklung von eigenen Prozessoren.

Innerhalb eines Platzhalters sind Zeilenumbrüche erlaubt (um zum Beispiel Code-Kommentare einzufügen) und es dürfen innen wiederum Anführungsstriche und sogar verschachtelte *Listing 16*: Einfache String-Interpolation verwendet werden. Die *String Templates* lassen sich auch mit mehrzeiligen Strings kombinieren, um beispielsweise längere SQL-Abfragen oder XML/HTML- beziehungsweise JSON-Dokumente zu erzeugen. *Listing 17* zeigt ein Template mit einem mehrzeiligen String und mehreren Zeilen beim Platzhalter `title`, der zudem noch mit einem Präfix verkettet wird.

```
String title = "My Web Page";
String text = "Hello, world";
String html = STR."""
  <html>
     <head>
        <title>\{ // Comment
           "My company: " + title }
        </title>
     </head>
     <body>
        <p>\{text}</p>
     </body>
  </html>""";
```

Listing 17: Interpolation in mehrzeiligen Strings

Im JDK mitgeliefert werden insgesamt drei *Template-Processor*-Implementierungen, deren Instanzen sich einfach als Variablen verwenden lassen. Im Falle einer einfachen String-Interpolation wird der *Processor STR* verwendet *(siehe Listing 16)*, der anstelle der Platzhalter stumpf die Ausdrücke einfügt. Mit FMT gibt es einen weiteren *Template Processor*, der die aus `String.format()` bekannten Formatierungsregeln auswertet. Während STR automatisch in jeder Java-Klasse importiert wird, muss FMT explizit über den Import

```
User user = new User("dieter@develop.de", new Role("admin"));

String info =
  STR."'\{user.email()}' hat Rolle '\{user.role().name()}'";

// 'dieter@develop.de' hat Rolle 'admin'
System.out.println(info);
```

Listing 16: Einfache String-Interpolation

`import static java.util.FormatProcessor.FMT` aktiviert werden *(siehe Listing 18)*.

```
import static java.util.FormatProcessor.FMT;

double zahl = 2.0;
String formatted = FMT."Zahl: %7.2f\{zahl}";
System.out.println(formatted); // Zahl:    2.00
```

Listing 18: Formatierung der Ersetzung mit dem Format Processor

Um direkt ein String-Template-Objekt zu erzeugen, muss der dritte vom JDK bereitgestellte *Processor RAW* verwendet werden. Dadurch lässt sich der Aufbau des String Templates analysieren und mit der Methode `fragments()` können die statischen Textbausteine sowie mit `values()` die Referenzen auf die einzufügenden Ausdrücke angeschaut werden. Mit `process(Processor)` kann man dann einen beliebigen Prozessor übergeben und die Interpolation anstoßen *(siehe Listing 19)*. Erst dann erfolgt die Auflösung des *String Template*, vorher werden die Einzelteile getrennt im Speicher gehalten (Lazy Evaluation = Bedarfsauswertung).

```
import static java.lang.StringTemplate.RAW;

template = RAW."Today is day
  \{LocalDate.now().getDayOfYear()} of year
  \{LocalDate.now().getYear()}.";

System.out.println(template.fragments()); System.out.
println(template.values());

// Today is day 82 of year 2024.
String result = template.process(STR); System.out.println(result);
```

Listing 19: Raw String Template

Sowohl STR als auch FMT liefern jeweils einen String zurück: String → String. Der große Vorteil im Vergleich zu anderen Programmiersprachen ist, dass ein Prozessor auch einen anderen Datentyp zurückgeben kann. So kann aus einem mehrzeiligen String beispielsweise direkt ein `JSONObject` erzeugt werden. Dazu muss von `StringTemplate.Processor` abgeleitet werden, zum Beispiel über die statische Hilfsmethode `Processor.of()`. Für die Erzeugung des `JSONObject` wird das *String Template* zunächst interpoliert (die Platzhalter werden ersetzt) und das Ergebnis nach JSON konvertiert *(siehe Listing 20)*.

Dieses einfache Beispiel zeigt noch nicht die ganze Mächtigkeit. Anstatt direkt die parameterlose Methode `template.interpolate()`

```
// new Template Processor
var JSON = StringTemplate.Processor.of(
  (StringTemplate template) ->
     new JSONObject(template.interpolate()));

User user = new User("dieter@develop.de", new Role("admin"));

JSONObject json = JSON."""
  {
     "user": "\{ user.email() }",
     "roles": "\{user.role().name()}"
  }""";

// {"roles":"admin","user":"dieter@develop.de"} System.out.println(json);

}
```

Listing 20: Einfacher Custom Template Processor

aufzurufen, kann man zunächst auf den Platzhaltern-Variablen (`template.values()`) Transformationen durchführen, um zum Beispiel potenziell gefährliche Benutzereingaben zu escapen. Anschließend übergibt man der `interpolate`-Methode die Textbausteine (`fragments()`) sowie die angepassten Werte (`newValues`) und kann aus dem Ergebnis wieder das `JSONObject` erzeugen *(siehe Listing 21)*.

```
Processor<JSONObject, JSONException> JSON =
  template -> {
        String quote = "\"";
        List<Object> newValues = new ArrayList<>();
        for (Object value : template.values()) {
           // Ersetzungen durchführen
           // ...
           newValues.add(value);
        }

        var result = StringTemplate.
           interpolate(template.fragments(), newValues);
        return new JSONObject(result);
  };
```

Listing 21: Fortgeschrittener Custom Template Processor

Nach diesem Prinzip können wir beliebige String-Interpolationen durchführen, zum Beispiel mit einem Locale-Objekt Texte internationalisieren, Log-Messages zusammenbauen oder SQL-Statements in ein JDBCPreparedStatement umwandeln. Wann immer es eine textbasierte Vorlage gibt, die transformiert, validiert oder gesäubert werden muss, bieten die *String Templates* in Java eine einfache, ins JDK eingebaute Template Engine ohne Abhängigkeiten zu Drittanbietern. In Zukunft liefern Frameworks und Bibliotheken vermutlich solche Template-Prozessoren mit. Oder wir erstellen für die interne Verwendung leicht eigene Prozessoren und vermeiden dadurch redundanten und potenziell gefährlichen Code. Vorhandene *Template Engines* wie *FreeMarker* oder *Thymeleaf* werden durch *String Templates* nicht obsolet, sie decken einen anderen Anwendungsbereich ab. Während für die im JDK eingebauten *String Templates* zur Compile-Zeit geprüft wird, ob die Ausdrücke in den Platzhaltern aufgelöst werden können, sind *Templates Engines* besser in der Lage, mit dynamischen Inhalten umzugehen.

Kurz vor dem Release vom OpenJDK 22 überraschte Brian Goetz auf der Mailingliste vom Inkubator-Projekt Amber mit der Ankündigung *[6]*, dass er die jetzige Umsetzung der *String Templates* nochmal überdenken wolle. Auf Java 22 hatte das zunächst keine Auswirkungen mehr, da ja bereits im Dezember der Feature Freeze (Rampdown Phase One) stattgefunden hat. Aber für Java 23 bedeutet das, dass die Template Prozessoren gegebenenfalls nochmals in Frage gestellt und durch einfache Methodenaufrufe ersetzt werden können. Diese Vorgehensweise beweist, dass der Prozess mit den Inkubator- beziehungsweise Preview-Phasen gut funktioniert und das gesammelte Feedback ernst genommen wird. Kurz vor Redaktionsschluss kam eine weitere Nachricht auf der Mailingliste, diesmal von Gavin Biermann *[7]*:

Thanks for the extensive feedback following Brian's e-mail. I think it's fair to say that there is still a broad range of opinions on exactly what form this feature should take. [...] So, to be clear: there will be no string template feature, even with --enable-preview, in JDK 23.

So schön die Idee von *String Templates* also ist, wir werden uns weiter gedulden müssen und die Umsetzung wird sich nochmals verändern.

Weitere Neuerungen im Umfeld von *Virtual Threads*

Virtual Threads waren die größte Änderung im OpenJDK 21. Sie erlauben es, die konkurrierende Verarbeitung von parallel ausgeführten Aufgaben auch bei einer sehr großen Anzahl an Threads zu implementieren und dabei sogar gut verständlichen Code zu schreiben. Dieser lässt sich zudem wie sequenzieller Code mit herkömmlichen Mitteln debuggen. Die *Virtual Threads* verhalten sich dabei wie normale Threads, werden aber nicht 1:1 auf Betriebssystem-Threads abgebildet. Stattdessen gibt es einen Pool von Träger-Threads (*Carrier Threads*), denen dann virtuelle Threads vorübergehend zugewiesen werden. Sobald der virtuelle Thread auf eine blockierende Operation stößt, wird er vom Träger-Thread genommen, der dann einen anderen virtuellen Thread (einen neuen oder einen zuvor blockierten) übernehmen kann.

Da *Virtual Threads* so leichtgewichtig sind, können sie jederzeit schnell erzeugt werden. Man muss also keine Threads wiederverwenden, sondern kann immer einfach neue instanziieren. An den *Virtual Threads* selbst hat sich nichts geändert, sie wurden bereits im JDK 21 finalisiert.

Im Umfeld von *Virtual Threads* wurde mit dem OpenJDK 19 die *Structured Concurrency* eingeführt. Bis Java 21 gab es auch immer

wieder Änderungen, diesmal blieb die API im JEP 462 unverändert und man möchte einfach weiteres Feedback sammeln.

Bei der Bearbeitung von mehreren parallelen Teilaufgaben erlaubt *Structured Concurrency* die Implementierung auf eine besonders les- und wartbare Art und Weise. Bisher wurden für die Aufteilung in parallel zu verarbeitende Aufgaben Parallel Streams oder der *ExecutorService* eingesetzt. Letzterer ist sehr mächtig, macht aber auch einfache Umsetzungen ziemlich kompliziert und fehleranfällig. Es ist zum Beispiel schwer zu erkennen, wenn eine der Teilaufgaben einen Fehler produziert, um dann sofort alle anderen sauber abzubrechen. Falls zum Beispiel ein Task sehr lange läuft, erhält man erst spät Feedback, wenn andere Aufgaben in Probleme gelaufen sind. Auch das Debuggen ist nicht einfach, da in den Thread-Dumps die Tasks nicht den jeweiligen Threads aus dem Pool zugeordnet werden können.

Bei der *Structured Concurrency* ersetzen wir den *ExecutorService* durch einen *StructuredTaskScope*, bei dem man verschiedene Strategien auswählen kann. Diese neue API macht nebenläufigen Code besser lesbar und kann zudem leichter mit Fehlersituationen umgehen. In *Listing 22* ist als Strategie *ShutdownOnFailure* verwendet worden.

```
try (var scope =
  new StructuredTaskScope.ShutdownOnFailure()) {

  Future<String> task1 = scope.fork(() -> { … }
  Future<String> task2 = scope.fork(() -> { … }
  Future<String> task3 = scope.fork(() -> { … }

  scope.join();
  scope.throwIfFailed();

  System.out.println(task1.get());
  System.out.println(task2.get());
  System.out.println(task3.get());
}
```

Listing 22: Structured Concurrency

Mit `scope.join()` wird gewartet, bis alle Tasks erfolgreich erledigt sind. Schlägt einer fehl oder wird abgebrochen, werden auch die anderen beiden beendet. Mit `throwIfFailed()` kann der Fehler außerdem weitergegeben werden und das Ausgeben der Ergebnisse wird übersprungen.

Dieser neue Ansatz bringt einige Vorteile. Zum beispiel bilden Task und Sub-Tasks im Code eine abgeschlossene, zusammengehörige Einheit. Die Threads kommen nicht aus einem Thread-Pool mit schwergewichtigen Plattform-Threads, stattdessen wird jede Unteraufgabe in einem neuen virtuellen Thread ausgeführt. Bei Fehlern werden noch laufende Sub-Tasks abgebrochen. Zudem sind bei Fehlersituationen die Informationen besser, weil die Aufrufhierarchie sowohl in der Code-Struktur als auch im Stacktrace der Exception sichtbar ist.

Aufgrund des Preview-Status müssen für die *Structured Concurrency* beim Kompilieren und Ausführen aber weiterhin bestimmte Schalter aktiviert werden *(siehe Listing 23)*.

```
$ javac --enable-preview StructuredConcurrencyMain.
java $ java --enable-preview StructuredConcurrencyMain
```

Listing 23: Kompilieren und Ausführen von Preview-Funktionen

Scoped Values

Im Umfeld der virtuellen Threads wurde im JDK 20 auch eine Alternative zu den ThreadLocal-Variablen vorgestellt. Die *Scoped Values* befinden sich ebenfalls weiterhin im Preview-Status (JEP 464). Es gab keine Änderung zur letzten Version, hier möchte man ebenso Erfahrungen und Feedback sammeln.

Die *Scoped Values* erlauben das Speichern eines temporären Wertes für eine begrenzte Zeit, wobei nur der Thread, der den Wert geschrieben hat, ihn auch wieder lesen kann. Sie werden in der Regel als öffentliche statische (globale) Felder angelegt und sind dann von beliebigen, tiefer im Aufruf-Stack befindlichen Methoden aus erreichbar. Verwenden mehrere Threads dasselbe ScopedValue-Feld, wird dieses je nach dem ausführenden Thread unterschiedliche Werte enthalten. Das Konzept funktioniert von der Idee wie bei ThreadLocal, bringt aber einige Vorteile mit.

Im Beispiel in *Listing 24* werden aus einem Web-Request Informationen zum angemeldeten Benutzer extrahiert. Auf diese Informa-

```
public final static ScopedValue<SomeUser> CURRENT_USER =
  ScopedValue.newInstance();

[..]

SomeController someController =
  new SomeController(
     new SomeService(new SomeRepository()));

someController.someControllerAction(
  HttpRequest.newBuilder()
     .uri(new URL( "http://example.com")
     .toURI()).build());

static class SomeController {

  final SomeService someService;

  SomeController(SomeService someService) {
     this.someService = someService;
  }

  public void someControllerAction(
     HttpRequest request) {

     SomeUser user = authenticate(request);
     ScopedValue.where(CURRENT_USER, user)
        .run(() -> someService.processService());
  }
}

static class SomeService {
  final SomeRepository someRepository;

  SomeService(SomeRepository someRepository) {
     this.someRepository = someRepository;
  }

  void processService() {
     System.out.println(CURRENT_USER
        .orElseThrow(() ->
           new RuntimeException("no valid user")));
     }
  }
}
```

Listing 24: Scoped Values

tionen muss in der weiteren Aufrufkette (im Service, im Repository, …) zugegriffen werden. Eine Variante wäre das Durchschleifen des User-Objektes als zusätzlicher Parameter in die aufzurufenden Methoden. *Scoped Values* verhindern diese redundante und unübersichtliche zusätzliche Parameterauflistung. Konkret wird mit `ScopedValue.where()` das User-Objekt gesetzt und dann mit der `run`-Methode eine Instanz von Runnable übergeben, für deren Aufrufdauer der *Scoped Value* gültig sein soll. Alternativ kann mit der `call()`-Methode ein Callable-Objekt übergeben werden, um auch Rückgabewerte auszuwerten. Der Versuch, den Inhalt außerhalb des Scoped-Value-Kontextes auszuführen, führt zu einer Exception, weil bei diesem Aufruf dann kein User-Objekt hinterlegt ist.

Das Auslesen des *Scoped Value* erfolgt über den Aufruf von `get()`. Bei Abwesenheit eines Wertes kann man mit einem Fallback (`orElse()`) oder dem Werfen einer Exception (`orElseThrow()`) reagieren.

Auch hier müssen beim Kompilieren und Ausführen gewisse Schalter aktiviert werden (analog zu *Listing 23*).

Die Klasse `ScopedValue` ist immutable und bietet dementsprechend keine `set`-Methode an. Dadurch wird der Code besser wartbar, weil es keine Zustandsänderungen am bestehenden Objekt geben kann. Muss für einen bestimmten Code-Abschnitt (Aufruf einer weiteren Methode in der Kette) ein anderer Wert sichtbar sein, kann ein Rebinding des Wertes erfolgen (zum Beispiel auf null setzen wie in *Listing 25*). Sobald der begrenzte Code-Abschnitt beendet ist, wird der ursprüngliche Wert wieder sichtbar.

```
ScopedValue.where(CURRENT_USER, null)
  .run(() -> someRepository.getSomeData());
```

Listing 25: Scoped Value Rebinding

Scoped Values arbeiten auch mit der *Structured Concurrency* zusammen. Die Sichtbarkeit wird, an die über einen StructuredTaskScope erzeugten Kind-Prozesse, vererbt. Somit können alle per `fork()` abgezweigten Kind-Threads ebenfalls auf die im *Scoped Value* befindlichen User-Informationen zugreifen.

Die *Scoped Values* stehen genau wie die ThreadLocals sowohl für Plattform- als auch virtuelle Threads zur Verfügung. Die Vorteile der *Scoped Values* sind vielfältig. Es erfolgt ein automatisches Aufräumen der Inhalte, sobald der Runnable-/Callable-Prozess beendet ist, wodurch Memory Leaks verhindert werden. Die Immutability der *Scoped Values* erhöht zudem die Verständlich- und Lesbarkeit. Bei der *Structured Concurrency* erzeugte Kind-Prozesse haben auch Zugriff auf den einen, unveränderbaren Wert. Die Informationen werden nicht wie bei InheritedThreadLocals kopiert (was zu höherem Speicherverbrauch führen kann).

Was sonst noch geschah?

Ein Punkt fehlt noch, der JEP 423 (*Region Pinning for G1*). Dabei geht es um die Verringerung von Latenzen beim Standard-Garbage-Collector G1 in Bezug auf kritische Regionen bei der Verwendung von JNI (Java Native Interface). Weitere kleinere Neuerungen, für die es keine JEPs gibt, können in den Release-Notes *[8]* nachgelesen werden. Änderungen am JDK (Java-Klassenbibliothek) lassen sich zudem sehr schön über den Java Almanac *[9]* nachvollziehen. In dieser Übersicht finden sich beispielsweise auch alle Neuerungen zu den Stream Gatherers.

Fazit und Ausblick

Nach dem Release ist bekanntlich vor dem Release. Bereits im September 2024 wird das OpenJDK 23 *[10]* erscheinen. Da werden wir einige der hier angesprochenen Preview-Themen erneut wiedersehen. Aktuell eingeplant ist beispielsweise die *Class-File API*. Aber auch neue Themen wie die oben schon kurz angesprochenen *Primitive Types in Patterns, instanceof, and switch* werden mit dabei sein. Wer sich vorab über andere zukünftige Themen informieren möchte, kann sich schon mal im JEP Index unter „Draft and submitted JEPs" *[11]* umschauen.

Java ist und bleibt weiterhin sehr relevant. Im nächsten Jahr wird der 30. Geburtstag gefeiert. Oracle plant zu diesem Jubiläum im März 2025 ein Revival der JavaOne in der San Francisco Bay Area *[12]*. Vielleicht treffen wir uns dort und stoßen auf unser gutes, altes Java an.

Referenzen

[1] *https://openjdk.java.net/projects/jdk/22/*
[2] *https://openjdk.org/jeps/461*
[3] *https://asm.ow2.io/*
[4] *https://openjdk.org/jeps/460*
[5] *https://openjdk.org/jeps/326*
[6] *https://mail.openjdk.org/pipermail/amber-spec-experts/2024-March/004010.html*
[7] *https://mail.openjdk.org/pipermail/amber-spec-experts/2024-April/004106.html*
[8] *https://jdk.java.net/22/release-notes*
[9] *https://javaalmanac.io/jdk/22/apidiff/21/*
[10] *https://openjdk.java.net/projects/jdk/23/*
[11] *https://openjdk.org/jeps/0#Draft-and-submitted-JEPs*
[12] *https://inside.java/2024/03/19/announcing-javaone-2025/*

Falk Sippach
falk@jug-da.de
https://twitter.com/sippsack

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

virtual7

WIR GESTALTEN DIE DIGITALE ZUKUNFT DEUTSCHLANDS.

HIER GESTALTEST DU DEINE
WELT ZUM BESSEREN ALS:

JAVA DEVELOPER (m/w/d)

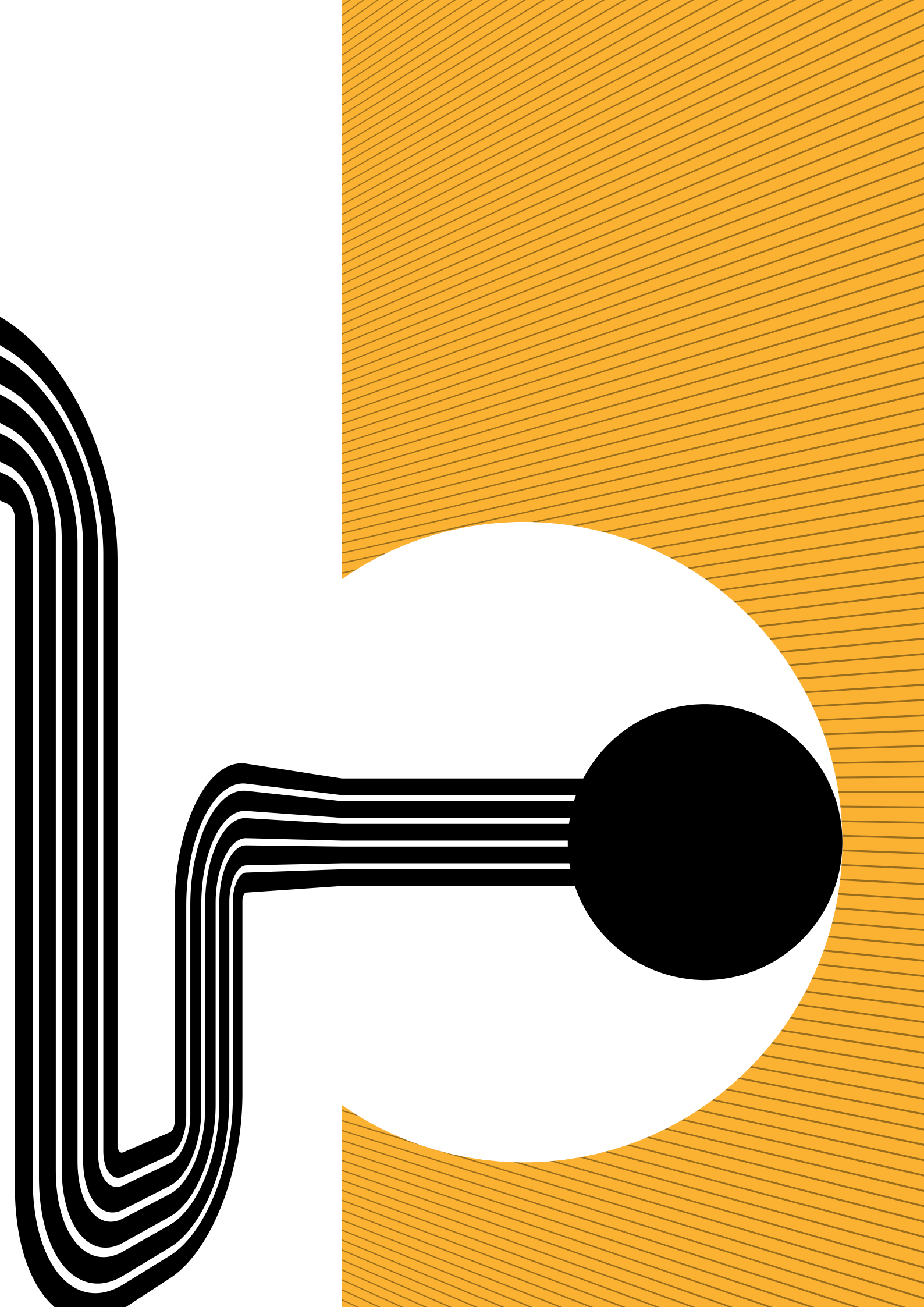
Entdecke spannende Möglichkeiten
in verschiedenen Clustern mit
unterschiedlichen Technologien!

WIR SUCHEN DICH!

Structured Concurrency – Die moderne Art, um Aktionen parallel auszuführen und Ergebnisse zusammenzuführen

Michael Inden, Adcubum AG

Dieser Artikel wirft einen Blick auf JEP 462 Structured Concurrency (Second Preview) (JEP = JDK Enhancement Proposal), einen vielversprechenden Vorschlag, Multithreading einfacher und nachvollziehbarer gestalten zu können. Ziel bei der Structured Concurrency ist es, einen neuen, in der Implementierung einfacheren Mechanismus anzubieten, um Aufgaben in Teilaufgaben aufzuspalten, diese parallel zu verarbeiten und deren Ergebnisse zusammenzuführen oder sogar Teilaufgaben abzubrechen, etwa, wenn Ergebnisse nicht mehr benötigt werden.



Einführendes Beispiel

Als Beispiel einer Businessfunktionalität mit Teilaufgaben, schauen wir uns die Behandlung eines Requests an. Wir beginnen mit einer herkömmlichen Implementierung mithilfe eines `ExecutorService`. Dabei sollen anhand einer User-ID parallel zum einen passende Benutzer mit `findUser()` und zum anderen die zugehörige Bestellung per `fetchOrder()` ermittelt werden. In *Listing 1* sind die Daten der Einfachheit halber als `String` und `Integer` repräsentiert und mithilfe eines Records modelliert.

Wie üblich ist der Happy-Path schnell implementiert, doch wir sollten uns weitere Gedanken machen: Weil die Teilaufgaben parallel ausgeführt werden, können sie unabhängig voneinander erfolgreich sein oder fehlschlagen. Im letzteren Fall tendiert der korrekte Umgang mit Fehlersituationen dazu, recht kompliziert zu werden. Oftmals möchte man nicht, dass das zweite `get()` aufgerufen wird, wenn bereits bei der Abarbeitung der Methode `findUser()` eine Exception aufgetreten ist. Oder genereller: Häufig sollte die umgebende Aktion, hier die Abarbeitung der Methode `handleOldStyle()`, abgebrochen werden, wenn eine ihrer Teilaufgaben fehlschlägt. Obendrein ist `get()` blockierend. Wenn also das erste `get()` nicht terminiert, so kommen wir niemals zum zweiten.

Beides haben wir bereits in diesem einfachen Beispiel erkennen können. Dadurch ist verständlich, dass beim Einsatz von Multithreading ein paar Details zu beachten sind, insbesondere, wenn es zu Exceptions während der Abarbeitung von Teilaufgaben kommt.

Es ergeben sich folgende Fragestellungen: Wie gehen wir damit um, wenn in einer Teilaufgabe ein Fehler auftritt? Wie können wir dann die anderen Teilaufgaben abbrechen? Wie können wir etwa nach Auffinden eines ersten Treffers einer Suche andere Suchen in Form von Teilaufgaben abbrechen, wenn deren Ergebnisse nicht mehr benötigt werden? All das ist beim Einsatz eines `ExecutorService` möglich, erfordert aber komplexen und schwer zu pflegenden Sourcecode. Als Abhilfe lernen wir im Anschluss Structured Concurrency kennen.

Structured Concurrency als Abhilfe

Bei der Structured Concurrency ersetzen wir den `ExecutorService` durch einen `StructuredTaskScope`. Für diesen kann man verschiedene Strategien auswählen, hier beispielsweise `ShutdownOnFailure` zum Abbruch der anderen Teilschritte beim Auftreten eines Fehlers. Statt mit `submit()` spaltet man konkurrierende Teilaufgaben durch einen Aufruf von `fork()` ab und erhält als Ergebnis kein `Future<V>`, sondern einen `StructuredTaskScope.Subtask<T>`. Abgesehen von diesen Details ist die Implementierung der vorherigen Umsetzung recht ähnlich *(siehe Listing 2)*.

Nach dem Start der einzelnen Teilaufgaben sammelt man die Ergebnisse mit einem blockierenden Aufruf von `join()` wieder ein. Dieser wartet, bis alle Teilaufgaben ab- gearbeitet sind oder bei (mindestens) einer ein Fehler in Form einer Exception auftrat. Ist letzteres der Fall, so werden auch die anderen Teilaufgaben beendet. Im Anschluss sollte man alle bei der Abarbeitung der Teilaufgaben aufgetretenen Fehler behandeln. Dazu können Fehler mit `throwIfFailed()` weiterge-

```
static Response handleOldStyle(Long userId)
          throws ExecutionException, InterruptedException
{
     try(var exectuorService = Executors.newCachedThreadPool())
     {
           var userFuture =
                 exectuorService.submit(() -> findUser(userId));
           var orderFuture =
                 exectuorService.submit(() -> fetchOrder(userId));

           var user = userFuture.get();  // Join findUser
           var order = orderFuture.get();  // Join fetchOrder

           return new Response(user, order);
     }
}
```

Listing 1: Klassische Umsetzung einer parallelen Verarbeitung

```
static Response handle(Long userId)
          throws ExecutionException, InterruptedException
{
     try (var scope =
           new StructuredTaskScope.ShutdownOnFailure())
     {
           var userSubtask =
                 scope.fork(() -> findUser(userId));
           var orderSubtask =
                 scope.fork(() -> fetchOrder(userId));

           scope.join();               // Join both forks
           scope.throwIfFailed(); // ... and propagate errors

           // both forks have succeeded, so compose their results
           return new Response(userSubtask.get(),
                                     orderSubtask.get());
     }
}
```

Listing 2: Umsetzung einer parallelen Verarbeitung mit Structured Concurrency

```
static Response handleWithTimeout(Long userId,
                                                    Duration timeOut)
          throws ExecutionException, InterruptedException,
                      TimeoutException
{
      try (var scope =
                    new StructuredTaskScope.ShutdownOnFailure())
      {
            var userSubtask =
                  scope.fork(() -> findUser(userId));
            var orderSubtask   =
                  scope.fork(() -> fetchOrder(userId));

            scope.joinUntil(Instant.now().plus(timeOut));
            scope.throwIfFailed(); // ... and propagate errors

            // both forks have succeeded, so compose their results
            return new Response(userSubtask .get(),
                                          orderSubtask.get());
      }
}
```

Listing 3: Structured Concurrency mit Time-out

reicht werden und die Aufbereitung der Ergebnisse wird übersprungen. Im Erfolgsfall passiert beim Aufruf von `throwIfFailed()` nichts und die Berechnungsergebnisse der Teilaufgaben lassen sich mit `get()` auslesen und dann geeignet kombinieren.

Im Gegensatz zum ursprünglichen Multithreading-Code fällt es mit Structured Concurreny leichter, die Lebensdauer der beteiligten Threads nachzuvollziehen und deren Zusammenführen zu verstehen. Es ist zudem möglich, bei einer Denkweise eher analog zu der rein sequenziellen Abarbeitung zu verbleiben. Hierbei würde die Methode `fetchOrder()` nur dann ausgeführt, wenn zuvor die Methode `findUser()` erfolgreich, also ohne Exception, abgeschlossen wurde. Dagegen würde im Falle eines Fehlers, genauer gesagt einer Exception, die gesamte Berechnung gestoppt.

Zwischenfazit

Die Structured Concurrency zeichnet sich vor allem durch eine gegenüber der sequenziellen Ausführung besseren Performance aus – unter Beibehaltung einer recht gut nachvollziehbaren Abarbeitung. Ein Vorteil, den der `ExecutorService` nicht gleichermaßen bietet, insbesondere dann nicht, wenn es zu Fehlersituationen in der Abarbeitung einzelner Teilaufgaben kommt.

Der `ExecutorService` bietet zwar oftmals eine vergleichbare Performance wie die Structured Concurrency, hat aber gegenüber der Structured Concurrency eben nicht den Vorteil der nachvollziehbaren Abarbeitung.

Variante mit Time-out

Die bisherige Umsetzung enthält noch eine Schwachstelle: Für den Fall, dass eine oder auch mehrere Teilaufgabe(n) entweder sehr lange benötigen, bis sie ein Ergebnis liefern, oder aber gar nicht terminieren, wird die gesamte Verarbeitung entweder stark behindert oder sogar ganz blockiert. Das gilt gleichermaßen sowohl für die sequentielle Abarbeitung, den Einsatz eines `ExecutorService` als auch für die Verwendung von Structured Concurrency beim Aufruf von `join()`. Letzteres bietet mit Time-outs als einzige der drei Varianten eine elegante Möglichkeit, dieses Umstands Herr zu werden. Dazu existiert praktischerweise als Abhilfe die Methode `joinUntil()`, der man einen Time-out, leider etwas umständlich in Form eines Instant, übergibt. Nach Überschreiten dieser Zeit werden alle noch laufenden Berechnungen des StructuredTaskScope beendet und die Fehlersituation mit `throwIfFailed` an den Aufrufer kommuniziert. Der Rest der Implementierung bleibt gleich *(siehe Listing 3)*.

Ausblick

Bislang haben wir die oftmals praktische Strategie `ShutdownOnFailure` kennengelernt, die beim Auftreten eines Fehlers alle anderen Berechnungen stoppt. Mit der Strategie `ShutdownOnSuccess` gibt es eine weitere nützliche vordefinierte Strategie, um mehrere Berechnungen zu beginnen und nachdem eine ein Ergebnis geliefert hat, alle anderen Teilaufgaben zu stoppen. Wozu kann das nützlich sein? Stellen wir uns verschiedene Suchanfragen vor, bei der die Schnellste gewinnen soll. Als Beispiel könnten wir den Verbindungsaufbau zum Mobilnetz in den Varianten 5G, 4G, 3G und Wi-Fi modellieren. Dabei gilt: Sobald eine Verbindung steht – hoffentlich die Schnellste und Stabilste – sollen die Verbindungsversuche zu den anderen Netzen gestoppt werden.

Es ist sogar einfach möglich, eigene Strategien zu implementieren, indem man von der Klasse `StructuredTaskScope<T>` ableitet und ein paar Methoden implementiert, etwa eine Strategie `FirstNSuccessful<T>`, die die Verarbeitung nach n erfolgreichen Teilberechnungen beendet. Das kann beispielsweise für verschiedene Suchanfragen nützlich sein, wo man etwa nur an den ersten 5 oder 10 Treffern interessiert ist.

Fazit

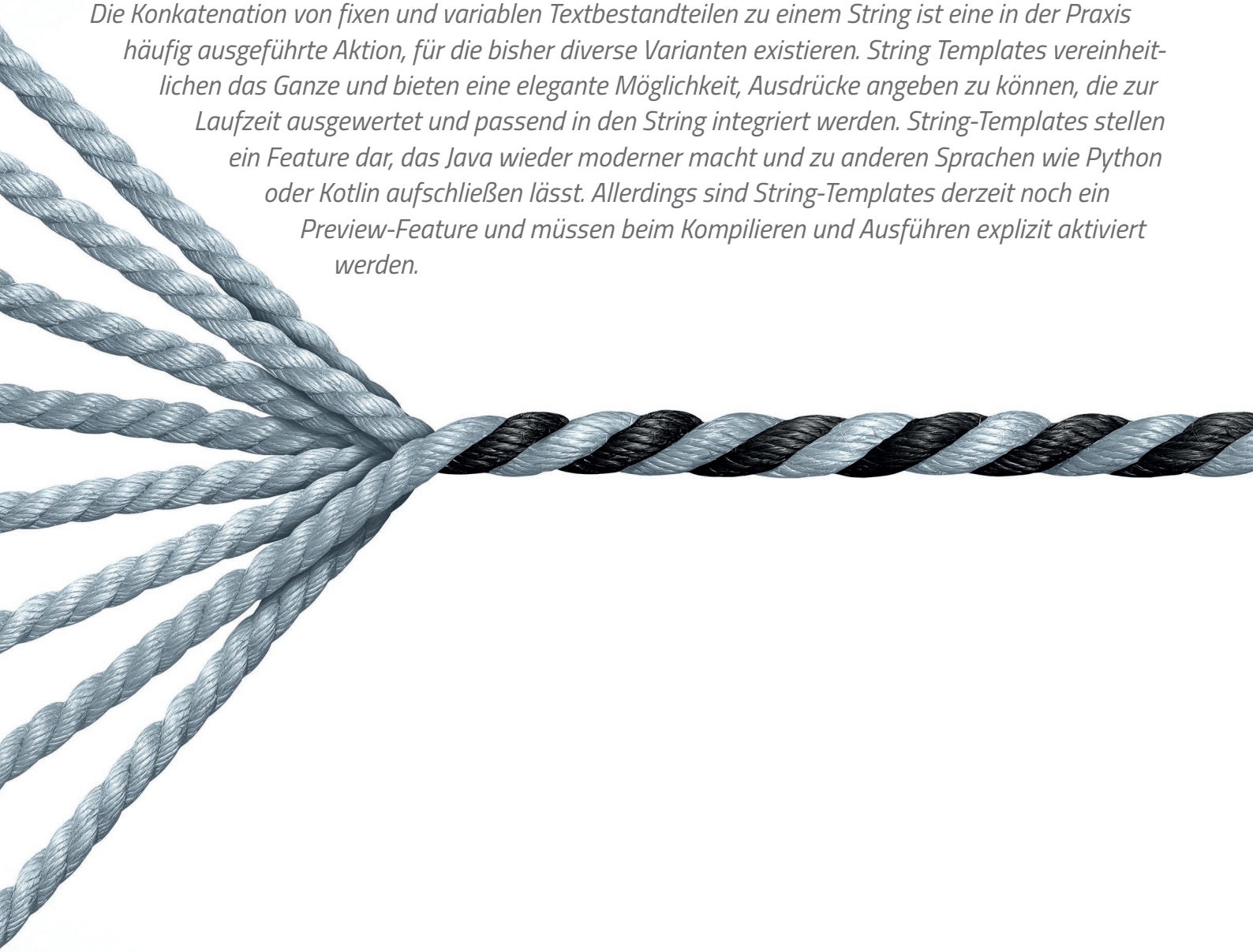
Das Aufteilen von Verarbeitungsschritten auf mehrere Threads, generell Multithreading mit Threads, ließ sich in Java recht einfach implementieren. Eine ziemliche Herausforderung war es jedoch, das Ganze wirklich sauber, verständlich und korrekt umzusetzen. Trotz einiger nachträglicher Ergänzungen im JDK fehlte bis zur Einführung von Structured Concurrency eine eingängige, nachvollziehbare und sogar leicht erweiterbare Möglichkeit, Teilaufgaben parallel auszuführen und deren Ergebnisse strukturiert wieder zusammenzuführen. Dank Structured Concurrency ist dies nun deutlich einfacher möglich.

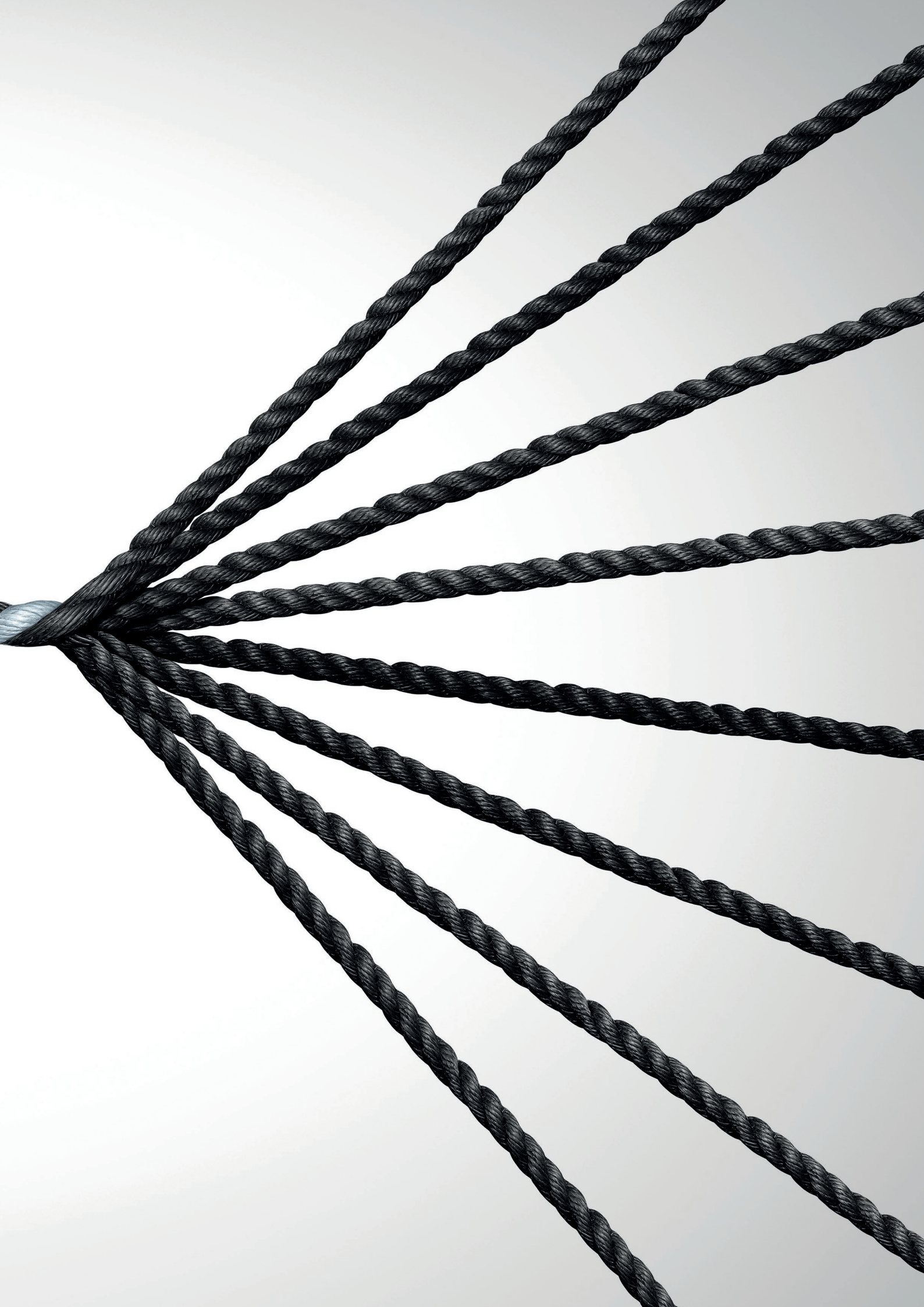
In diesem Sinne: Happy Coding mit dem brandaktuellen Java 22!

String-Templates – Die moderne Art der Stringkonkatenation

Michael Inden, Adcubum AG

Die Konkatenation von fixen und variablen Textbestandteilen zu einem String ist eine in der Praxis häufig ausgeführte Aktion, für die bisher diverse Varianten existieren. String Templates vereinheitlichen das Ganze und bieten eine elegante Möglichkeit, Ausdrücke angeben zu können, die zur Laufzeit ausgewertet und passend in den String integriert werden. String-Templates stellen ein Feature dar, das Java wieder moderner macht und zu anderen Sprachen wie Python oder Kotlin aufschließen lässt. Allerdings sind String-Templates derzeit noch ein Preview-Feature und müssen beim Kompilieren und Ausführen explizit aktiviert werden.





Die Klassiker der Stringkonkatenation

Um formatierte Ausgaben in Java zu erzeugen, kommen oftmals die leicht verständlich und in der Regel recht gut lesbare Stringkonkatenation mit + zum Einsatz. Allerdings leidet die Lesbarkeit mit zunehmender Anzahl an Verknüpfungen. Dramatischer ist der Effekt beim Einsatz `StringBuffers` oder `StringBuilders` und der Methode `append()`, deren Verwendung vorwiegend aus erhofften Performance-Vorteilen motiviert wird. Schon bei den wenigen Aufrufen im nachfolgenden Beispiel sind die Auswirkungen auf die Lesbarkeit offensichtlich *(siehe Listing 1)*.

Es gibt weitere Möglichkeiten, etwa die Methoden `format()` und `formatted()` aus der Klasse `String`. Beide nutzen die gleichen Platzhalter, variieren aber ein wenig in der Handhabung. Darüber hinaus existiert noch die Klasse `MessageFormat` und deren Methode `format()`. Durchaus viele Möglichkeiten für ein und dasselbe Ziel. Wie lösen es andere Programmiersprachen?

Ein Blick über den Tellerrand

Alternativ zur Stringkonkatenation unterstützen Programmiersprachen wie *Python*, *Kotlin*, *Swift* und *C#* die sogenannte Interpolation oder auch formatierte Strings.

Dabei handelt es sich um Strings, in die spezielle Platzhalter integriert sind, die zur Laufzeit durch entsprechende Werte ersetzt werden.

- Python f"Calculation: {x} + {y} = {x + y}"
- Kotlin "Calculation: $x + $y = ${x + y}"
- Swift: "Calculation: \(x) + \(y) = \(x + y)"
- C# $"Calculation: {x} + {y}= {x + y}"

In allen Fällen werden die Platzhalter im String durch die entsprechenden Werte der Variablen, insbesondere auch von Berechnungen, ersetzt. Ähnliches bietet die Methode `String.format()`, die man jedoch explizit aufrufen muss.

String-Templates als Alternative

Mit JEP 430 werden String-Templates ein- und mit JEP 459 weitergeführt. Die grundlegende Syntax startet mit dem Kürzel `STR`, gefolgt von einem Punkt (.) und dann einem String, der einen oder mehrere Platzhalter im Format `\{expression}` enthält. Damit ergibt sich der in *Listing 2* gezeigte schematische Aufbau.

Interessanterweise ist das `STR` per Definition `public static final` und automatisch innerhalb jeder Java-Datei und auch der *JShell* bereits vorhanden. Daher ist kein zusätzlicher Aufwand zum Einbinden nötig.

Betrachten wir als Beispiel den Fall, dass `\{expression}` eine Konstante, hier `Math.PI`, referenziert *(siehe Listing 3)*.

Was passiert im Hintergrund? Das `STR` aktiviert einen sogenannten Template Processor, der dann die Platzhalter ersetzt, im einfachsten Fall die Konstante durch deren Wert.

Praxisrelevanter ist es, Variablennamen oder Berechnungen durch deren Wert beziehungsweise das Ergebnis einer Berechnung zu ersetzen *(siehe Listing 4)*.

Bei der Angabe der Ausdrücke ist man nicht auf einfache mathematische Operationen beschränkt, sondern es lassen sich beliebige Methoden wie `multiplyExact()` oder `repeat()` aufrufen *(siehe Listing 5)*.

Das zweite Beispiel zeigt neben dem Methodenaufruf, dass man innerhalb der Platzhalter normale Anführungszeichen nutzen kann – das mag ein wenig irritierend sein. Auf diese Besonderheit und vor allem die Verschachtelung von String-Templates gehe ich später nochmals ein.

String-Templates in Kombination mit Text Blocks

Praktischerweise kann man String-Templates in Kombination mit Text Blocks nutzen *(siehe Listing 6)*.

Nach der Ersetzung entsteht folgendes HTML *(siehe Listing 7)*.

```
String result = "Calculation: " + x + " plus " + y +
                " equals " + (x + y);
System.out.println(result);

String resultSB = new StringBuilder().append("Calculation: ").
                                      append(x).append(" plus ").
                                      append(y).append(" equals ").
                                      append(x + y).toString();
System.out.println(resultSB);
```

Listing 1: Klassiker der Stringkonkatenation

```
STR."Text 1: \{placeholder1} Text 2: \{placeholder2}"
```

Listing 2: Schematischer Aufbau eines String-Templates

```
jshell> STR."String Template PI: \{Math.PI}"
$1 ==> "String Template PI: 3.141592653589793"
```

Listing 3: Referenzierung einer Konstante in einem String-Template

```
jshell> int x = 47
x ==> 47

jshell> int y = 11
y ==> 11

jshell> STR."Calculation: \{x} + \{y} = \{x + y}"
$15 ==> "Calculation: 47 + 11 = 58"
```

Listing 4: Referenzierung von Variablen und Ausführung von Berechnungen

```
jshell> int x = 7
x ==> 7

jshell> STR."Rechne: \{x} mal 2 = \{Math.multiplyExact(x, 2)}"
$2 ==> "Rechne: 7 mal 2 = 14"

jshell> STR."\{"Moin".repeat(2)}"
$3 ==> "MoinMoin"
```

Listing 5: Methodenaufrufe in String-Templates

```
String title = "My First Web Page";
String text = "My Hobbies:";
var hobbies = List.of("Cycling", "Hiking", "Shopping");

String html = STR."""
<html>
     <head><title>\{title}</title></head>
     <body>
           <p>\{text}</p>
           <ul>
                 <li>\{hobbies.get(0)}</li>
                 <li>\{hobbies.get(1)}</li>
                 <li>\{hobbies.get(2)}</li>
           </ul>
     </body>
</html>""";
```

Listing 6: String-Templates in Kombination mit Text Blocks zur Definition von HTML

```
<html>
     <head><title>My First Web Page</title>
     </head>
     <body>
           <p>My Hobbies:</p>
           <ul>
                 <li>Cycling</li>
                 <li>Hiking</li>
                 <li>Shopping</li>
           </ul>
     </body>
</html>
```

Listing 7: Resultierendes HTML

```
jshell> record Person(String name, int age) {
   ...>  public static final String SPECIAL = "INFO";
   ...> }
|  Erstellt Datensatz Person

jshell> var name = "Peter"
name ==> "Peter"

jshell> var peter = new Person(name, 42)
peter ==> Person[name=Peter, age=42]

jshell> // Variable name

jshell> var useVariable = STR."Greetings \{name}!"
useVariable ==> "Greetings Peter!"

jshell> // Attribut SPECIAL

jshell> var useAttribute = STR."SECRET: \{Person.SPECIAL}"
useAttribute ==> "SECRET: INFO"

jshell> // Methode age()

jshell> var useMethod = STR."You are \{peter.age()} years old!"
useMethod ==> "You are 42 years old!"
```

Listing 8: Verschiedene Varianten des Zugriffs

Speichert man dies als Datei und öffnet diese in einem Browser, so erhält man in etwa folgende Darstellung *(siehe Abbildung 1)*.

Zugriff auf Variablen, Attribute und Methoden

In den Platzhaltern können wir lokale Variablen, statische sowie nicht-statische Attribute und sogar Methoden angeben. Wie zuvor schon gezeigt, sind auch Berechnungen möglich. Die Resultatwerte werden dann zur Laufzeit errechnet. Hier zeige ich explizit nochmals die drei Varianten (Variable, Attribut und Methode) *(siehe Listing 8)*.

Obwohl diverse Aktionen in String-Templates möglich sind, lassen sich beispielsweise Lambdas dort nicht angeben *(siehe Listing 9)*.

Besonderheiten

Interessanterweise ist es möglich, komplexere Aktionen in einem Platzhalter auszuführen. Neben Methodenaufrufen sind auch Aktionen wie die folgenden möglich, etwa ein mehrmaliges Postinkrement *(siehe Listing 10)*.

Auf die Modifikation von Werten innerhalb der Platzhalter sollten Sie zugunsten einer guten Lesbarkeit und Verständlichkeit aber besser verzichten – nicht alles, was möglich ist, ist auch immer sinnvoll – Ihre Kollegen werden Sie sonst in besonderer Erinnerung behalten…

Möglich sind ebenfalls Aktionen mit dem Date-and-Time-API – hier sei nochmals erwähnt, dass sich in einem Muster einfache Anführungszeichen, wie hier für `"HH:mm"`, ohne Escaping nutzen lassen *(siehe Listing 11)*.

Bemerkenswerterweise wird auch die mehrfache Verschachtelung von String-Templates unterstützt, allerdings leiden dann Lesbarkeit und Nachvollziehbarkeit doch deutlich, wie es das Beispiel in *Listing 12* zeigt.

Alternativer String-Prozessor

Neben `STR` existiert `FMT` als weiterer vordefinierter Prozessor, um eine Ausgabe wie mit `String.format()` zu erzielen. Während STR standardmäßig verfügbar ist, muss man FMT geeignet mit `import static java.util.FormatProcessor.FMT` importieren *(siehe Listing 13)*. Als Ausgabe erhält man *Listing 14*.

Fazit

Für die Konkatenation von fixen und variablen Textbestandteilen zu einem String gab es bisher diverse Varianten – alle mit jeweils spezifischen Stärken und Schwächen. In diesem Artikel habe ich gezeigt, dass String-Templates ein sehr praktisches Feature darstellen, um Konkatenationen von fixen und variablen Textbestandteilen zu bewerkstelligen.

Die Arbeitsweise der beiden vordefinierten Template-Prozessoren `STR` und `FMT` ist eingängig. Mitunter möchte man selbst eingreifen und die Art und Weise der Konkatenation beeinflussen können. Da-

```
jshell> STR."String Template PI: \{() -> Math.PI}"
|  Fehler:
|  Hier wird kein Lambda-Ausdruck erwartet
|  STR."String Template PI: \{() -> Math.PI}"
|                            ^-----------^
```

Listing 9: Keine Unterstützung von Lambdas in String-Templates

```
jshell> int index = 0
index ==> 0

jshell> var modified = STR."\{index++}, \{index++}, \{index++}, \{index++}"
modified ==> "0, 1, 2, 3"

jshell> index
index ==> 4
```

Listing 10: (Verwirrende) mehrfache Aktionen

```
var currentTime =
     STR."Current time: \{DateTimeFormatter.ofPattern("HH:mm").
           format(LocalTime.now())}";
```

Listing 11: Anführungszeichen im Muster

```
jshell> String name = "Doctor"
name ==> "Doctor"

jshell> static String confusing()
   ...> {
   ...>     return "STRANGE!";
   ...> }
| Erstellt Methode confusing()

jshell> String result = STR."\{ STR."\{ STR."\{name} " +
                                    STR."\{confusing()}"}"}";
result ==> "Doctor STRANGE!"
```

Listing 12: Verwirrende Verschachtelung von String-Templates

```
int x = 47;
int y = 11;
String calculation1 = FMT."%6d\{x} + %6d\{y} = %6d\{x + y}";
System.out.println("fmt calculation 1: " + calculation1);

float base = 3.0f;
float addon = 0.1415f;
String calculation2 = FMT."%2.4f\{base} + %2.4f\{addon}" +
                      FMT." = %2.4f\{base + addon}";
System.out.println("fmt calculation 2: " + calculation2);

String calculation3 =
         FMT."Math.PI * 1.000 = %4.6f\{Math.PI * 1000}";
System.out.println("fmt calculation 3: " + calcula-
tion3);
```

Listing 13: Formatierte Aufbereitung mit FMT

```
fmt calculation 1:     47 +     11 =     58
fmt calculation 2: 3.0000 + 0.1415 = 3.1415
fmt calculation 3: Math.PI * 1.000 = 3141.592654
```

Listing 14: Ergebnis der obigen Programmzeilen

für ist es möglich, eigene Template-Prozessoren zu erstellen. Den Ausgangspunkt bildet das Interface `StringTemplate.Processor`. Es gibt also noch einiges zu entdecken.

In diesem Sinne: Happy Coding mit dem brandaktuellen Java 22!

Michael Inden
Adcubum AG
michael_inden@hotmail.com

Michael Inden ist Java- und Python-Enthusiast mit über zwanzig Jahren Berufserfahrung. Er hat bei diversen internationalen Firmen in verschiedenen Rollen etwa als Softwareentwickler, -architekt, Teamleiter, CTO und Trainer gearbeitet. Derzeit ist er als Head of Development tätig. Darüber hinaus spricht er auf Konferenzen und schreibt Fachbücher wie „Der Weg zum Java-Profi", „Java – Die Neuerungen in Version 17 LTS, 18 und 19" sowie die Pärchen „Java Challenge"/„Python Challenge" und „Einfach Java"/„Einfach Python". Alle sind im dpunkt.verlag erschienen.

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.

Maven 4, Teil 1: Einfache Versionen

Karl Heinz Marbaise

Im Rahmen des Artikels befassen wir uns mit neuen Features, die Maven 4 (aktuell noch Alpha-Stadium) bietet. Das erste neue Feature, das wir uns genauer anschauen möchten, ist das Versionshandling.


shift
fn
550

Übersicht

Die Versionsnummer, beziehungsweise deren Verwendung, ist in Maven-Projekten durchaus ein Thema. Wie kann die Versionsnummer einfach geändert werden? Vor allem im Zusammenhang mit CI/CD-Setups? Da gibt es unterschiedliche Ansätze, wie zum Beispiel die Nutzung des Maven-Release-Plug-ins*[2]*, die Verwendung des Versions-Maven-Plug-ins*[8]* zusammen mit dem Build-Helper-Maven-Plug-in *[7]* oder auch die Verwendung CI-Friendly*[3]* sowie einige andere Möglichkeiten.

Beispiel-Projekt

Beginnen wir mit einer einfachen POM-Datei, die wie in *Listing 1* aussieht. Der Übersichtlichkeit halber zeigen wir nur die wichtigen Teile der POM-Datei, auf die wir unser Augenmerk legen wollen.

```
<project...>

  <modelVersion>4.0.0</modelVersion>

  <parent>
     <groupId>com.soebes.smpp</groupId>
     <artifactId>smpp</artifactId>
     <version>6.0.5</version>
     <relativePath/>
  </parent>

  <groupId>com.soebes.examples.maven4</groupId>
  <artifactId>basic</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Basic Example</name>
  ..
</project>
```

Listing 1: Einfache POM-Datei

Betrachten wir die POM-Datei etwas genauer: Es wird eine sogenannte Parent-POM (`com.soebe.smpp:smpp:6.0.5`) verwendet, von der wir Konfigurationen und Festlegungen für Plug-ins und so weiter vererbt bekommen. Die Adressierung der POM-Datei geschieht, wie in Maven üblich, über die Koordinaten, auch bekannt als *GAV* (`groupId`, `artifactId`, `version`). Das verhindert, dass wir das in jedem Projekt wiederholen müssen.

Maven 4

Wenn Sie sich die aktuelle Maven-4 -Version (Alpha) heruntergeladen haben *[1]*, prüfen wir zunächst, ob alles funktioniert hat. Das kann einfach, wie in *Listing 2* dargestellt, überprüft werden. Die Ausgabe auf der Konsole sieht dann ähnlich wie in *Listing 3* aus.

Jetzt versuchen wir, das entsprechende Projekt mit Maven 4 zu bauen. Wir führen ein `mvn clean verify` aus und es ergibt sich die (lange) Ausgabe, wie in *Listing 4* dargestellt.

Die Ausgabe `[INFO] Unable to find the root directory...` werden wir in diesem Artikel nicht betrachten, da dies sonst den Rahmen des Artikels sprengen würde. Wir konzentrieren uns hier auf die Versionierung beziehungsweise die Versionsnummern.

```
$> mvn --version
```

Listing 2: Maven-Version feststellen

```
Apache Maven 4.0.0-alpha-13 (0a6a5617fe5ef-
65c44f05903491e170d92cf37fc)
Maven home: /projects/tools/maven
Java version: 22, vendor: Oracle Corporation, runtime:
/projects/.sdkman/candidates/java/22.ea.36-open
Default locale: en_DE, platform encoding: UTF-8
OS name: "mac os x", version: "14.0", arch: "aarch64",
family: "mac"
```

Listing 3: Ausgabe der Maven-Versionsinformationen

SNAPSHOT versus Release-Version

In dem initialen Beispiel *(siehe Listing 1)* wird eine Version `1.0.0-SNAPSHOT` verwendet. Die übliche Vorgehensweise in Maven-Projekten ist es, mit einer sogenannten SNAPSHOT-Version (erkennbar an dem `-SNAPSHOT` Postfix) zu beginnen. Diese ist der Indikator, dass das Projekt sich aktuell noch in der Entwicklung befindet, beziehungsweise noch nicht abgeschlossen ist. Selbstverständlich kommt auch hier irgendwann der Zeitpunkt, an dem der Zustand des Projekts auf final gesetzt werden sollte, beziehungsweise muss (unveränderlich oder *immutable*). Das finalisieren wir durch die entsprechende Kenntlichmachung über die Versionsnummer. Dabei wird dann aus der Version `1.0.0-SNAPSHOT` eine `1.0.0`, die nun Release-Version genannt wird. Im gleichen Zuge werden die entsprechenden Artefakte veröffentlicht, beispielsweise im Central-Repository *[10]* oder auch in internen Repositories innerhalb von Firmen/Organisationen. Da die Entwicklung nicht stillsteht, wird nun die Version auf beispielsweise `1.1.0-SNAPSHOT` geändert und der ganze Zyklus beginnt von vorne. Das kann mithilfe der Plug-ins erreicht werden, die bereits im Abschnitt „Übersicht" genannt worden sind. Das führt aber zu Änderungen in der POM-Datei sowie entsprechenden Einträgen in der Versionskontrolle und ist nicht für alle Anwendungsfälle geeignet. Weiterhin wird auch eine entsprechende Konfiguration beziehungsweise ein Setup benötigt und dies ist unter Umständen im Hinblick auf die Ausführungszeit nicht optimal.

Vereinfachte Versionen

Mit der Verwendung von Maven 4 kann die Version über eine Property in der pom.xml definiert werden *(siehe Listing 5)*.

Führen wir jetzt wieder einen Build via `mvn clean` aus, ergibt sich die Ausgabe wie *Listing 6* dargestellt. Hier wird für den Bau lediglich ein `mvn clean` verwendet, um die Ausgaben übersichtlich zu halten.

Die Ausgabe `Building Basic Example ${revision}` sieht etwas merkwürdig aus. Daraus kann man schlussfolgern, dass das Projekt keine Versionsnummer hat, beziehungsweise die Versionsnummer `${revision}` hat, die keine gültige Versionsnummer darstellt. Wie können wir nun eine Versionsnummer für den Build angeben, beziehungsweise definieren? Zum Glück geht das sehr einfach mithilfe von `mvn clean -Drevision=1.2.0-SNAPSHOT`.

Die Ausgabe in *Listing 7* zeigt jetzt `Building Basic Example 1.2.0-SNAPSHOT`. Das bedeutet, dass die Versionsnummer jetzt korrekt und für die entsprechenden Artefakte verwendet wird.

POM Property

Das ist gar nicht mal so schlecht. Wir können dann per Kommandozeile eine Version definieren. Das geben wir dann in jeder Kommandozeile entsprechend mit an. Moment mal, jedes Mal? Vielleicht

```
[INFO] Unable to find the root directory. Create a .mvn directory in the root directory or add the root="true" attribute
on the root project's model to identify it.
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------------------------< com.soebes.examples.maven4:basic >----------------------------------
[INFO] Building Basic Example 1.0.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------------------------------[ jar ]---------------------------------------------
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ basic ---
[INFO] Deleting /projects/basic/target
[INFO]
[INFO] --- enforcer:3.4.1:enforce (enforce-maven) @ basic ---
[INFO] Rule 0: org.apache.maven.enforcer.rules.RequireSameVersions passed
[INFO] Rule 1: org.apache.maven.enforcer.rules.version.RequireMavenVersion passed
[INFO] Rule 2: org.apache.maven.enforcer.rules.dependency.BannedDependencies passed
[INFO] Rule 3: org.apache.maven.enforcer.rules.RequireNoRepositories passed
[INFO] Rule 4: org.apache.maven.enforcer.rules.RequirePluginVersions passed
[INFO] Rule 5: org.apache.maven.enforcer.rules.property.RequireProperty passed
[INFO] Rule 6: org.apache.maven.enforcer.rules.property.RequireProperty passed
[INFO] Rule 7: org.apache.maven.enforcer.rules.property.RequireProperty passed
[INFO]
[INFO] --- jacoco:0.8.11:prepare-agent (default) @ basic ---
[INFO] argLine set to -javaagent:/projects/.m2/repository/org/jacoco/org.jacoco.agent/0.8.11/org.jacoco.agent-0.8.11-runt-
ime.jar=destfile=/projects/basic/target/jacoco.exec
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ basic ---
[INFO] skip non existing resourceDirectory /projects/basic/src/main/resources
[INFO] skip non existing resourceDirectory /projects/basic/src/main/resources-filtered
[INFO]
[INFO] --- compiler:3.12.1:compile (default-compile) @ basic ---
[INFO] Recompiling the module because of changed source code.
[INFO] Compiling 1 source file with javac [debug release 11] to target/classes
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ basic ---
[INFO] skip non existing resourceDirectory /projects/basic/src/test/resources
[INFO] skip non existing resourceDirectory /projects/basic/src/test/resources-filtered
[INFO]
[INFO] --- compiler:3.12.1:testCompile (default-testCompile) @ basic ---
[INFO] No sources to compile
[INFO]
[INFO] --- surefire:3.2.3:test (default-test) @ basic ---
[INFO] No tests to run.
[INFO]
[INFO] --- jar:3.3.0:jar (default-jar) @ basic ---
[INFO] Building jar: /projects/basic/target/basic-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- site:3.12.1:attach-descriptor (attach-descriptor) @ basic ---
[INFO] Skipping because packaging 'jar' is not pom.
[INFO]
[INFO] --- jacoco:0.8.11:report (default) @ basic ---
[INFO] Skipping JaCoCo execution due to missing execution data file.
[INFO] Copying com.soebes.examples.maven4:basic:pom:1.0.0-SNAPSHOT to project local repository
[INFO] Copying com.soebes.examples.maven4:basic:jar:1.0.0-SNAPSHOT to project local repository
[INFO] Copying com.soebes.examples.maven4:basic:pom:consumer:1.0.0-SNAPSHOT to project local repository
[INFO] ------------------------------------------------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------------------------------------------------
[INFO] Total time: 2.004 s
[INFO] Finished at: 2024-03-16T12:53:58+01:00
[INFO] ------------------------------------------------------------------------------------------------------------------
```

Listing 4: Ausgabe-Build mit Maven 4

```
...
  <groupId>com.soebes.examples.maven4</groupId>
  <artifactId>basic</artifactId>
  <version>${revision}</version>
  <name>Basic Example</name>
...
```

Listing 5: Version-Property in der POM

doch noch nicht so gut. Aber dies lässt sich vereinfachen, in dem wir die Property in der `pom.xml` definieren *(siehe Listing 8)*.

Damit können wir nun ohne die Angabe auf der Kommandozeilenoption `-Drevision=..` einfach bauen (wie ein `mvn clean`), die Ausgabe ist in *Listing 9* dargestellt.

Nun werden einige mit Sicherheit fragen: Haben wir damit wirklich einen Vorteil? Das hatten wir doch schon mit dem ersten Beispiel.

```
..
[INFO]
[INFO] ----------------------------------------< com.soebes.examples.maven4:basic >----------------------------
[INFO] Building Basic Example ${revision}
[INFO]   from pom.xml
[INFO] --------------------------------------------------------[ jar ]-----------------------------------------
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ basic ---
[INFO] Deleting /projects/basic-revision/target
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] Total time:  0.242 s
[INFO] Finished at: 2024-03-16T13:15:11+01:00
[INFO] ------------------------------------------------------------------------------------------------------------
```

Listing 6: Ausgabe mit Revision-Property

```
...
[INFO]
[INFO] ----------------------------------------< com.soebes.examples.maven4:basic >----------------------------
[INFO] Building Basic Example 1.2.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------------------------------[ jar ]-----------------------------------------
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ basic ---
[INFO] Deleting /projects/basic-revision/target
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] Total time: 0.243 s
[INFO] Finished at: 2024-03-16T13:21:41+01:00
[INFO] ------------------------------------------------------------------------------------------------------------
```

Listing 7: Ausgabe mit definierter Revision-Property

```
..
<groupId>com.soebes.examples.maven4</groupId>
<artifactId>basic</artifactId>
<version>${revision}</version>
<name>Basic Example</name>

<properties>
  <revision>1.0.0-SNAPSHOT</revision>
</properties>
```

Listing 8: Version-Property in der POM

```
..
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------------------------< com.soebes.examples.maven4:basic >----------------------------
[INFO] Building Basic Example 1.0.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------------------------------[ jar ]-----------------------------------------
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ basic ---
[INFO] Deleting /projects/basic-revision-pom/target
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] Total time: 0.212 s
[INFO] Finished at: 2024-03-16T13:31:11+01:00
[INFO] ------------------------------------------------------------------------------------------------------------
```

Listing 9: Ausgabe mit Property in der pom.xml

Auf den ersten Blick vielleicht, aber gehen wir einen Schritt weiter. Jetzt möchten wir eine andere Version bauen – zum Beispiel `2.1.0-SNAPSHOT`. Wie können wir das erreichen? Ganz einfach mit Hilfe von `mvn clean -Drevision=2.1.0-SNAPSHOT` *(siehe Listing 10)*.

Betrachten wir das Argument von vorhin noch einmal. Wir haben keine Änderung an der pom.xml-Datei vorgenommen, beziehungsweise vornehmen müssen, sondern haben die Version einfach per Kommandozeile übergeben. Genauer: Wir haben den Wert, der in der pom.xml-Datei ist, überschrieben. Das kann in allen möglichen CI/CD-Tools (*Jenkins [11]*, *Circle CI [12]*, *Woodpecker CI [13]*, *GitHub Actions [14]*, *Gitea Actions[15]* und weiteren) verwendet werden.

Release erzeugen

Das zuvor Beschriebene macht es jetzt sehr einfach, ein Release zu erzeugen. Das erreichen wir mithilfe von `mvn deploy -Drevision=1.2.0`, vorausgesetzt, dass die notwendigen Konfigurationen gemacht wurden (Konfiguration `distributionManagement`, Authentifizierung und so weiter). Die Ausgabe sieht dann ähnlich

```
..
[INFO]
[INFO] ----------------------------------------< com.soebes.examples.maven4:basic >----------------------------
[INFO] Building Basic Example 2.1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------------------------------[ jar ]-----------------------------------------
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ basic ---
[INFO] Deleting /projects/basic-revision-pom/target
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------------------------------------------
[INFO] Total time: 0.241 s
[INFO] Finished at: 2024-03-16T13:21:26+01:00
[INFO] ------------------------------------------------------------------------------------------------------------
```

Listing 10: Andere Version bauen

```
...
[INFO]
[INFO] --- install:3.1.1:install (default-install) @ basic ---
[INFO] Deferring install for com.soebes.examples.maven4:basic:1.2.0 at end
[INFO] Installing /projects/basic-distro/target/basic-1.2.0.jar to /.m2/repository/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0.jar
[INFO] Installing /projects/basic-distro/pom.xml to /.m2/repository/com/soebes/examples/maven4/basic/1.2.0/basic-
1.2.0-build.pom
[INFO] Installing /projects/basic-distro/target/consumer-4964754963249724515.pom to /.m2/repository/com/soebes/exam-
ples/maven4/basic/1.2.0/basic-1.2.0.pom
[INFO]
[INFO] --- deploy:3.1.1:deploy (default-deploy) @ basic ---
Uploading to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0.jar
Uploaded to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0.jar (3.1 kB at 33 kB/s)
Uploading to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0-build.pom
Uploading to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0.pom
Uploaded to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0-build.pom (969 B at 88 kB/s)
Uploaded to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/1.2.0/basic-1.2.0.pom (2.2 kB at 199 kB/s)
Downloading from releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/ba-
sic/maven-metadata.xml
Uploading to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/basic/
maven-metadata.xml
Uploaded to releases: http://localhost:8081/nexus/content/repositories/releases/com/soebes/examples/maven4/basic/
maven-metadata.xml (530 B at 48 kB/s)
[INFO] Copying com.soebes.examples.maven4:basic:pom:1.2.0 to project local repository
[INFO] Copying com.soebes.examples.maven4:basic:jar:1.2.0 to project local repository
[INFO] Copying com.soebes.examples.maven4:basic:pom:consumer:1.2.0 to project local repository
[INFO] ------------------------------------------------------------------------------------------------------------
-------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------------------------------------------
-------------
[INFO] Total time: 1.367 s
[INFO] Finished at: 2024-03-16T13:45:11+01:00
[INFO] ------------------------------------------------------------------------------------------------------------
-------------
```

Listing 11: Release-Version deployen

wie in *Listing 11* aus, worin der Anfang der Ausgabe der Übersichtlichkeit halber entfernt wurde, da dieser schon in den vorigen Beispielen gezeigt wurde.

Multi-Module-Builds

Gehen wir einen Schritt weiter und betrachten ein komplexeres Setup in einem sogenannten Multi-Module-Build. Ein solcher Build besteht aus einer Struktur wie in *Listing 12* dargestellt.

```
.
|-- pom.xml (1)
|-- domain
|   |-- pom.xml
|   `-- src
|-- service
|   |-- pom.xml
|   `-- src
|-- service-client
|   |-- pom.xml
|   `-- src
`-- webgui
    |-- pom.xml
    `-- src
```

Listing 12: Multi-Module-Build-Struktur

Ein Multi-Module-Build zeichnet sich dadurch aus, dass die Kind-Module (wie `domain` oder `service`) immer einen Bezug zum Eltern-Modul (Parent, gekennzeichnet mit (1)) haben. Die Parent pom.xml sieht dann wie in *Listing 13* aus. Dabei fällt auf, dass die Liste der `<modules>..</modules>` den Bezug zu den Kind-Modulen herstellt.

```
<project...>

  <modelVersion>4.0.0</modelVersion>

  <parent>
     <groupId>com.soebes.smpp</groupId>
     <artifactId>smpp</artifactId>
     <version>6.0.5</version>
     <relativePath/>
  </parent>

  <groupId>com.soebes.examples.j2ee</groupId>
  <artifactId>jee-parent</artifactId>
  <version>3.1.4-SNAPSHOT</version>
  <packaging>pom</packaging>
  ...
  <modules>
     ..
     <module>webgui</module>
     <module>domain</module>
     <module>service</module>
     <module>service-client</module>
     ..
  </modules>

</project>
```

Listing 13: Parent-POM, Multi-Module-Build

In einer Kind-Modul `pom.xml` sieht das zum Beispiel für `domain` wie in *Listing 14* aus.

Dabei ist entscheidend, dass die Angabe `parent` sich genau auf das Parent im Multi-Module-Build bezieht (siehe *Listing 12* pom mit

```
<project....>

  <modelVersion>4.0.0</modelVersion>

  <parent>
     <groupId>com.soebes.examples.j2ee</groupId>
     <artifactId>jee-parent</artifactId>
     <version>3.1.4-SNAPSHOT</version>
  </parent>

  <artifactId>domain</artifactId>
  ...
</project>
```

Listing 14: Parent-Child-Beziehung

(1) gekennzeichnet). Ein solcher Multi-Module-Build kann durchaus aus einigen hundert oder gar tausenden Kind-Modulen bestehen und wird häufig auf mehrere Sub-Ebenen aufgeteilt (ähnlich wie ein Verzeichnisbaum). Theoretisch gibt es hier keine Grenzen. Wenn man sich nun vorstellt, die Versionsnummer ändern zu müssen, führt das, wie schon eingangs beschrieben, dazu, dass alle `pom.xml`-Dateien geändert werden müssen. Das kann aber mit der Verwendung eines `${revision} Property`-Ansatzes deutlich vereinfacht werden.

Das führt dazu, dass die Parent-POM eine entsprechende Property bekommt *(siehe Listing 15)* und die Kind-Module ebenfalls. Ein exemplarisches Beispiel an einem Kind-Modul (`domain`) findet sich in *Listing 16*.

Andere Properties

Einige werden sich jetzt fragen: Kann hier etwas anderes genutzt werden als die bisher beschriebene `revision`-Property? Es gibt technisch genau drei Properties, die für diesen Zweck genutzt werden können. Das ist die schon genutzte `revision`-Property, aber auch `sha1` und

```
<project ...>

  <modelVersion>4.0.0</modelVersion>

  <parent>
     <groupId>com.soebes.smpp</groupId>
     <artifactId>smpp</artifactId>
     <version>6.0.5</version>
     <relativePath/>
  </parent>

  <groupId>com.soebes.examples.j2ee</groupId>
  <artifactId>jee-parent</artifactId>
  <version>${revision}</version>
  <packaging>pom</packaging>

  <properties>
     <revision>1.0.0-SNAPSHOT</revision>
  </properties>

  ...
  <modules>
     ..
     <module>webgui</module>
     <module>domain</module>
     <module>service</module>
     <module>service-client</module>
     ..
  </modules>

</project>
```

Listing 15: Parent-POM, Multi-Module-Build

```
<project ....">

  <modelVersion>4.0.0</modelVersion>

  <parent>
     <groupId>com.soebes.examples.j2ee</groupId>
     <artifactId>jee-parent</artifactId>
     <version>${revision}</version>
  </parent>
  ..
  <artifactId>domain</artifactId>
  ...
</project>
```

Listing 16: Parent-Child-Beziehung

`changelist` können wie `revision` -Property genutzt werden. Damit können Kombinationen gebildet werden, wie zum Beispiel `<version>${revision}${sha1}${changelist}</version>`. Für die drei Properties gilt das am Anfang Beschriebene, dass diese entsprechend definiert oder über die Kommandozeile übergeben werden müssen. Ich empfehle, nur eine, nämlich `revision` zu verwenden, aber am Ende ist es Ihnen überlassen, welche Sie nutzen. Wichtig ist, dass man aktuell zum Beispiel nicht `version` anstatt `revision` nutzen kann.

Maven-Konfigurationsdatei

Die Definition der Property wurde in den Beispielen direkt in der `pom.xml`-Datei erledigt. Es besteht aber auch die Möglichkeit, dass man dies mithilfe der `maven.config`-Datei die im Verzeichnis `.mvn` (muss auf der höchsten Ebene des Projektes liegen) lösen kann *(siehe Listing 17)*.

Die `maven.config`-Datei muss im `.mvn`-Verzeichnis abgelegt werden. Die `maven.config`-Datei beinhaltet einfache Kommandozeilen-Optionen wie zum Beispiel `-Drevision=2.0.0-SNAPSHOT` *(siehe Listing 18)*.

```
.
|-- .mvn
|-- pom.xml
|-- domain
|   |-- pom.xml
|   `-- src
|-- service
|   |-- pom.xml
|   `-- src
|-- service-client
|   |-- pom.xml
|   `-- src
`-- webgui
    |-- pom.xml
    `-- src
```

Listing 17: .mvn-Verzeichnis

`maven.config` kann auch noch weitere Optionen wie zum Beispiel `-T 3` oder ähnliche enthalten. Die Unterstützung für die `maven.config`-Datei, gibt es seit *Maven* 3.3.1 *[6]* (seit zirka 9 Jahren). Damit besteht die Möglichkeit, die Property nicht in der `pom.xml`-Datei definieren zu müssen.

Zusammenfassung

Die Verwendung der `revision`-Property (oder auch der anderen beiden) vereinfacht die Versionierung der Artefakte ungemein. In der

```
-Drevision=1.0.0-SNAPSHOT
```

Listing 18: maven.config-Inhalt

Konsequenz ist auch die Release-Erstellung deutlich vereinfacht worden, ohne die `pom.xml`-Dateien ändern zu müssen. Der wichtigste Punkt im Zusammenhang mit *Maven* 4 ist hier, dass die Nutzung out of the box zur Verfügung steht. Mit anderen Worten: Es ist nicht mehr notwendig, ein zusätzliches Plug-in wie das *Flatten-Maven-Plugin [9]*, zu konfigurieren, wie es bei *CI Friendly* in *Maven* 3 *[3]* bisher notwendig gewesen ist. Das heißt auch, dass die Nutzung von beispielsweise *Versions-Maven-Plugin [8]*, um die Versionen in den POM-Dateien zu aktualisieren, nicht mehr benötigt wird. Die Unterstützung seitens des Maven-Release-Plugin *[2]*, dass auch entsprechende Tags (Marken) im Rahmen des Release-Prozesses erstellt werden, fehlt hier selbstverständlich. Das könnte durch einen einfachen ergänzenden Schritt in der CI/CD-Pipeline gelöst werden.

Quellen

[1] *https://maven.apache.org/download.cgi*
[2] *https://maven.apache.org/maven-release/maven-release-plugin*
[3] *https://maven.apache.org/maven-ci-friendly.html*
[4] *https://maven.apache.org/plugins/maven-install-plugin*
[5] *https://maven.apache.org/plugins/maven-deploy-plugin*
[6] *https://maven.apache.org/docs/3.3.1/release-notes.html#jvm-and-command-line-options*
[7] *https://www.mojohaus.org/build-helper-maven-plugin/parse-version-mojo.html*
[8] *https://www.mojohaus.org/versions/versions-maven-plugin/index.html*
[9] *https://www.mojohaus.org/flatten-maven-plugin/*
[10] *https://central.sonatype.com/*
[11] *https://www.jenkins.io/*
[12] *https://circleci.com/*
[13] *https://woodpecker-ci.org/*
[14] *https://docs.github.com/en/actions*
[15] *https://docs.gitea.com/next/usage/actions/overview*

Karl Heinz Marbaise
info@soebes.de

Karl Heinz Marbaise arbeitet als freiberuflicher DevOps mit den unterschiedlichsten Kunden. Er hat zirka 19 Jahre Java-Erfahrung und Expertenkenntnisse im Bereich Apache Maven und weiterhin in der Verwendung von Testing-Frameworks wie zum Beispiel JUnit Jupiter, Mockito, AssertJ und Testcontainers. Er hat auch ausführlichste Erfahrungen im Bereich CI/CD-Systemen wie Jenkins, Drone, Git, Gitea, GitHub und Ansible. Nicht zu vergessen Erfahrungen im Bereich Containern. Zudem ist er Committer beim Apache-Maven-Projekt.

Das dynamische Trio: Java, Kotlin & Scala auf der JVM

Michael Schweiker, sidion GmbH

In diesem Artikel geht es um ein einfaches Beispiel wie Java, Kotlin und Scala zusammen genutzt werden können. Es wird anhand eines kurzen Beispiels gezeigt, wie die Verwendung einer weiteren Sprache die Entwicklerin oder den Entwickler dabei unterstützt, einfachen und verständlichen Code zu schreiben.

In der Welt der Java-Entwicklung gibt es eine Vielzahl von Tools und Frameworks, die im täglichen Gebrauch zum Einsatz kommen. Das reicht von unterschiedlichen Build-Tools wie *Maven* oder *Gradle*, über IDEs wie *IntelliJ*, *NetBeans* oder *Eclipse*, bis hin zu Frameworks wie *Spring* oder *Quarkus*. All diese Varianten sind in den meisten Kreisen geläufig und es wird oft kontrovers und ergebnisoffen diskutiert, was denn die beste Alternative sei.

Was allerdings seltener zur Sprache kommt, ist, dass es neben diesen Optionen noch eine weitere gibt: die Wahl der Sprache selbst. Oft wird selbstverständlich davon ausgegangen, dass das gesamte Projekt in *Java* umgesetzt wird. Allerdings gibt es auch hier eine Fülle an Möglichkeiten. Laut Oracle gibt es mehrere hundert Sprachen, die zu Bytecode für die JVM kompilieren können *[1]*. Und die populärsten können sogar gleichzeitig in einem Projekt verwendet werden. Anders als bei den verschiedenen Tools muss ich mich also nicht für das komplette Vorhaben auf eines festlegen, sondern kann innerhalb des Projektes bei Bedarf neu entscheiden, welche Sprache ich für eine bestimmte Aufgabe nutzen möchte.

Bei der Verwendung mehrerer Sprachen ist es möglich, die Stärken zu kombinieren und Schwächen auszugleichen. Java ermöglicht einen einfachen Einstieg, da die Sprache sehr verbreitet ist. *Kotlin* zeichnet sich beispielsweise durch einen moderneren Stil und den verbesserten Umgang mit null-Werten aus, was schon bei so grundlegenden Dingen wie dem Mapping von Daten für unterschiedliche Schnittstellen einen großen Mehrwert bietet. Entwickelt wird *Kotlin* seit 2011 von JetBrains, dem Unternehmen, das auch *IntelliJ* und weitere IDEs entwickelt und vertreibt. 2016 wurde die erste stabile Version veröffentlicht und seit 2019 ist es die bevorzugte Sprache zum Entwickeln von Android-Apps.

Die Stärken von *Scala* liegen in der Manipulation und dem Filtern von Daten. In diesen Bereichen konnte *Java* zwar durch die Stream-API etwas aufholen, trotzdem sind die funktionalen Möglichkeiten, die *Scala* bietet, immer noch deutlich weiterentwickelt und komfortabler in der Anwendung.

Umsetzung des Mappings mit Kotlin

Das Zusammenspiel können wir an einem einfachen Beispiel verdeutlichen. Es werden Daten von einem Service zur Planung der Abwesenheit von Mitarbeiterinnen und Mitarbeitern abgerufen, aufbereitet und über eine REST-Schnittstelle wieder zur Verfügung gestellt. Als Grundlage für die Anwendung wird *Quarkus* genutzt, da es sehr leichtgewichtig ist und einen schnellen Einstieg bietet. Als Build-Tool und zur Verwaltung der Abhängigkeiten kommt *Maven* zum Einsatz.

Im ersten Schritt wurde das Projekt als reine Java-Anwendung gestartet. Die Datenklassen wurden als *Java Records* umgesetzt. Der REST-Client zur Anbindung des externen Dienstes und die zur Verfügung gestellten REST-Endpoints sind als klassische *Java*-Klassen umgesetzt. Das Mapping zwischen den einzelnen Klassen erfolgt aufgrund der einfacheren Handhabung von null-Werten mit *Kotlin*.

In *Listing 1* ist zum einen das Datenmodell der Schnittstelle und zum anderen das in der Anwendung verwendete Datenmodell zu sehen. Die Schnittstelle verwendet ein mehrfach verschachteltes Modell, was für die Anwendung jedoch nicht nötig ist. Auf jeder Ebene der Daten kann es vorkommen, dass ein Feld nicht gefüllt ist und somit Werte fehlen. Intern sollen jedoch immer alle Felder gefüllt sein. Um das zu gewährleisten, muss in *Java* mit einer Reihe von verketteten if-Abfragen gearbeitet werden, um zu prüfen, auf welcher Ebene ein Wert nicht vorhanden ist.

In *Kotlin* ist das dank dem Aufruf von Parametern und Methoden mit dem Safe-Call-Operator `?.` deutlich einfacher. Wenn in einer Aufrufkette ein Wert null ist, kann mit dem Elvis-Operator direkt ein Standardwert spezifiziert werden. In *Listing 2* ist zu sehen, wie das Mapping mit der Verwendung der erwähnten Funktionen umgesetzt ist. Wenn für eine Person keine oder nur Teile der Informationen zur Verfügung stehen, werden die fehlenden Informationen auf 0 gesetzt.

Bei einer Umsetzung mit *Java* müsste jeder Safe-Call-Operator durch eine eigene Null-Prüfung ersetzt werden. Es ist zwar möglich, das `accumulatedValue?.absences?.` in einem `if` zusammenzufassen, die Zielwerte dann in Variablen zu speichern und diese an den Konstruktor zu übergeben, aber die Lösung wird definitiv länger und schwerer verständlich. Die *Java*-Lösung für die Methode `zuAbwesenheitstage` ist in *Listing 3* zu sehen. Es fällt direkt auf, dass durch das `if` nicht genau erkennbar ist, wann der Wert, auf den die Variablen initialisiert wurden, verwendet und wann der Wert aus dem übergebenen Objekt verwendet wird.

```
public record Absences(Integer holiday, Integer school) {}
public record AccumulatedValue(LocalDate beginDate, LocalDate endDate, Absences absences) {}
public record AccumulatedValueResponse(Map<String, List<AccumulatedValue>> personWithAccumulatedValue) {}

public record Abwesenheitstage(int jahr, int jahresurlaub, int schule) {}
public record Abwesenheiten(Map<String, Abwesenheitstage> personMitAbwesenheitstagen) {}
```

Listing 1: Das Datenmodell der Schnittstelle und das interne Datenmodell

```
fun zuAbwesenheiten(response: AccumulatedValueResponse, jahr: Int) = Abwesenheiten(
     response.personWithAccumulatedValue.map { it.key to zuAbwesenheitstage(it.value[0], jahr) }.toMap()
)
fun zuAbwesenheitstage(accumulatedValue: AccumulatedValue?, jahr: Int) =
     Abwesenheitstage(jahr, accumulatedValue?.absences?.jahresurlaub ?: 0, accumulatedValue?.absences?.schule ?: 0)
```

Listing 2: Mapping von Schnittstellenmodell auf internes Modell

```
public Abwesenheitstage zuAbwesenheitstage(AccumulatedValue accumulatedValue, int jahr) {
  int schule = 0;
  int jahresurlaub = 0;
  if (accumulatedValue != null && accumulatedValue.absences() != null) {
     var absences = accumulatedValue.absences();
     if (absences.jahresurlaub() != null) {
        jahresurlaub = absences.jahresurlaub();
     }
     if (absences.schule() != null) {
        schule = absences.schule();
     }
  }
  return new Abwesenheitstage(jahr, jahresurlaub, schule);
}
```

Listing 3: Mapping von AccumulatedValue zu Abwesenheitstage in Java

```
def summeAllerAbwesenheiten(abwesenheiten: Abwesenheiten) =
  new SummeAbwesenheit(
     abwesenheiten
        .personMitAbwesenheitstagen.asScala
        .map { case (name, abwesenheitstage) => (name, abwesenheitstage.schule + abwesenheitstage.jahresurlaub: Integer) }
        .asJava )
```

Listing 4: Summieren der Abwesenheitstage pro Person

Erweiterung der Daten mit Scala

Nachdem das Mapping der Daten von der Schnittstelle auf das interne Modell erfolgt ist, werden die Daten jetzt aggregiert, um weitere Informationen zu zeigen. Bisher wurde nach verschiedenen Typen der Abwesenheit unterschieden. Manchmal ist die Information, warum jemand nicht anwesend ist, jedoch irrelevant. Es ist auch möglich, dass sich das Interesse rein auf die Gesamtzahl der Abwesenheitstage aller Personen in einem bestimmten Zeitraum beschränkt. An dieser Stelle kommt in unserem Beispiel nun *Scala* zum Einsatz.

Scala vereinfacht das Anwenden funktionaler Prinzipien deutlich. Wie in *Listing 4* zu erkennen ist, muss die Map nicht erst über mehrere Schritte in einen Stream umgewandelt werden, sondern es können direkt Operationen darauf ausgeführt werden. Von *Java* ist man gewohnt, dass Lambdas in Streams meist nur einen Parameter haben. Das führt dazu, dass man sich bei einem Stream über eine Map mit den Einträgen dieser und dann mit dem *Key* und dem *Value* behilft. Durch die Verwendung von Tupeln ist es in *Scala* möglich, den einzelnen Elementen direkt Namen zu geben. Das Bearbeiten aller Einträge einer Map braucht so nur einen Schritt und hat entsprechende Namen. Dadurch wird auch das Zurückkonvertieren in eine Map einfacher. In *Java* wären bis zum ersten Aufruf, in dem Daten bearbeitet werden, mehrere Konvertierungen nötig, um überhaupt per Streamverarbeitung auf die Map zugreifen zu können.

Auffällig sind hier `.asScala` und `.asJava`. Diese Statements werden benötigt, um die Interoperabilität von *Java* und *Scala* herzustellen, wenn hauptsächlich mit Java Datentypen gearbeitet wird.

Risiken

Ein Problem bei der Verwendung mehrerer Sprachen in einem Projekt ist, dass zwar rein technisch mehrere Sprachen verwendet werden, hinsichtlich des Programmierstils aber nur eine. Jede Sprache bringt unterschiedliche Herangehensweisen und Best Practices mit, wie eine ähnlich gelagerte Aufgabenstellung angegangen wird.

Ein Beispiel, an dem das gut verdeutlicht werden kann, ist die Berechnung der Summe der Quadrate von Zahlen in einer Liste. In *Kotlin* gibt es mehrere Wege, diese Aufgabe zu lösen. Drei davon sind in *Listing 5* dargestellt. Die erste Variante ist die, die für eine *Java*-Entwicklerin oder einen -Entwickler am vertrautesten wirkt, da man hier die Methoden-Signatur und anschließend den Methoden-Körper hat. In diesem wird auch schon die *Kotlin*-Funktion zum Summieren der Elemente genutzt, allerdings hier ebenfalls im Java-Stil mit den runden Klammern. Das zweite und dritte Beispiel sind deut-

```
fun sumOfSquares(numbers: List<Int>): Int {
     return numbers.sumOf({number -> number * number })
}

fun sumOfSquares(numbers: List<Int>) = numbers.sumOf { it * it }

fun List<Int>.sumOfSquares() = this.sumOf { it * it }

fun aufrufBeispiel() {
  val liste = listOf(1, 2, 3, 4, 5)
  assertEquals(sumOfSquares(liste), liste.sumOfSquares())
}
```

Listing 5: Drei Wege in Kotlin die Summe der Quadrate einer Liste von Zahlen zu berechnen.

lich mehr im Stil von *Kotlin*. Der größte Unterschied ist, wie die Liste übergeben wird, auf der die Operation ausgeführt werden soll. Bei der zweiten Variante wird die Liste als Parameter übergeben. Beim dritten Beispiel hingegen wird die Klasse Liste für den Typ `Int` so erweitert, dass die Liste nicht explizit übergeben wird, sondern die Funktion auf die Liste aufgerufen wird. Welche der beiden Varianten man bevorzugt, hängt von der Größe des Projekts und der Abstimmung im Team ab.

Um diese Problematik zu vermeiden, ist es wichtig, im Team das richtige Wissen aufzubauen und sich mit den unterschiedlichen Konzepten auseinanderzusetzen. Wenn man diesen Ansatz wählt, muss man sich bewusst sein, dass solche Fälle vorkommen werden und direkt eine Strategie entwickeln und einen Prozess vereinbaren, um diese anzugehen und gemeinsam auszumerzen. Mit einer weiteren Sprache wird dies leider eher noch komplexer.

Fazit

Die Verwendung mehrerer Programmiersprachen ist kein Wundermittel – sie birgt sogar neue Risiken. Trotzdem sollte man offenbleiben, ob es vielleicht eine bessere, leichtere oder verständlichere Art ist, eine Aufgabenstellung anzugehen. Wenn man ausprobiert, mehrere Sprachen zu verwenden, kann das Experiment in unterschiedlichen Kontexten durchaus unterschiedliche Ergebnisse bringen. Eine Möglichkeit wäre, festzustellen, dass es Sinn macht, das ganze Projekt beispielsweise, statt in Java, in *Kotlin* umzusetzen. Ein anderes Ergebnis kann sein, *Kotlin* oder *Scala* aufgrund der Fähigkeit, diese als DSL zu erweitern, hauptsächlich im Bereich des Testens einzusetzen und die Anwendung trotzdem weiter in *Java* zu schreiben. Valide ist es auch, festzustellen, dass der Versuch nicht die Erwartungen erfüllt hat und die Entwicklung wieder zu komplett in *Java* zu gestalten.

Quellen

[1] *https://www.oracle.com/technical-resources/articles/java/architect-languages.html* (Abgerufen am 02.04.2024)

Michael Schweiker
sidion GmbH
michael.schweiker@sidion.de

Michael Schweiker ist Softwareentwickler bei der sidion GmbH in Stuttgart. In den letzten Jahren sammelte er Erfahrungen mit Quarkus, Apache Kafka, Kubernetes sowie React und entwickelte damit Cloud-native Anwendungen. Um eine auf das Problem passende Lösung zu finden, verwendet er agile Methoden in Kombination mit Domain-driven Design und DevOps.

10 JAHRE JAVALAND

JavaLand

www.javaland.eu

JAVALAND 2024 VERPASST?

ALLE ON-DEMAND-ANGEBOTE IM TICKETSHOP

JETZT ON-DEMAND-TICKET BUCHEN UND
VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

Veranstalter: JavaLand

Microfrontends in bestehende Anwendungen integrieren – funktioniert das? Ein Erfahrungsbericht.

Christian Siebmanns, viadee Unternehmensberatung AG

Die Anwendung ist seit mehreren Jahren produktiv – und dann auf einen Hype wie Microfrontends aufspringen, geht das überhaupt? Wir haben das Experiment gewagt, unser bestehendes Frontend umgebaut und Microfrontends eingeführt. Was uns dazu bewogen hat und was wir dabei gelernt haben, erfahrt ihr in diesem Artikel.

Wir hatten folgende Situation: zwei Projektteams, die voneinander unabhängige Webportale in derselben Firma entwickeln, sollen möglichst viele Komponenten wiederverwenden. Die Wiederverwendung wurde beiden Projekten als Ziel vorgegeben.

Aufgrund der Nutzung einer Microservices-Architektur war das serverseitig für einige Komponenten kein Problem: So wurde etwa das Authentifizierungsmodul zwischen den Projekten geteilt. Jedes Anwendungssystem nutzte lediglich seine eigene Instanz. Im Frontend war Wiederverwendung auch in vielen Punkten möglich, beispielsweise wurden der Anmeldebildschirm und das Design-System geteilt.

Eine andere geteilte Komponente im Frontend war die zentrale Administrationsanwendung für die internen Fachanwender:innen. In dieser Anwendung lassen sich Mandant:innen sperren oder einrichten und neue Benutzer:innen hinzufügen. Diese Funktionalitäten hängen fast ausschließlich am Authentifizierungsmodul, das beide Projekte gemeinsam nutzen. Daher wurde während der Entwicklung entschieden, die zentrale Administration auch frontendseitig zu teilen.

Dieser Ansatz funktionierte mehrere Jahre lang problemlos. Dann bekamen wir die Anforderung, im zentralen Administrationsmodul eine Anforderung speziell für eines der Projekte umzusetzen. Die gewünschte Funktion sollte eine Bereinigungsfunktion in einem fachlichen Microservice triggern, der spezifisch für ein Projekt war. Das klingt trivial – war es aber nicht. Die Grundannahme war simpel: Wir integrieren die Funktionalität ins zentrale Administrationsmodul und blenden die dazugehörigen Schaltflächen basierend auf den Berechtigungen der Nutzer:innen ein.

Um statische Typisierung für unsere Projekte sicherzustellen, generiert ein Microservice eine passende Schnittstellenbeschreibung per *OpenAPI*. Diese *openapi.json* entsteht beim Build des Microservices. Sie wird in den Nexus hochgeladen und im Frontendprojekt per Maven heruntergeladen. Im Frontend wird diese Schnittstellenbeschreibung genutzt, um die *TypeScript*-Typen für Requests und Responses der Microservices zu generieren. In der Theorie hervorragend, in der Praxis bedeutet es, dass das Frontend nicht vor dem Backend gebaut werden kann, da sonst die Schnittstellendefinition fehlt.

Um jetzt unsere Anforderung umzusetzen, mussten wir dem zentralen Administrationsprojekt eine Referenz auf den projektspezifischen Microservice hinzufügen – autsch! Die Abhängigkeitssituation ist in *Abbildung 1* verdeutlicht.

In der Regel koordinierten beide Projekte einen gemeinsamen Release-Termin. Das bedeutet, alle Release-Builds wurden nacheinander zum selben Zeitpunkt durchgeführt. Für die ersten Releases funktionierte dieses Vorgehen gut. Probleme gab es erst, als unser Projekt sein Release verschob. Da nun der Entwicklungszeitraum in beiden Projekten zu unterschiedlichen Zeitpunkten endete, wurde festgelegt, dass geteilte Abhängigkeiten, wie die zentrale Administration, zum ersten Release-Termin gebaut werden. Die zentrale Administration

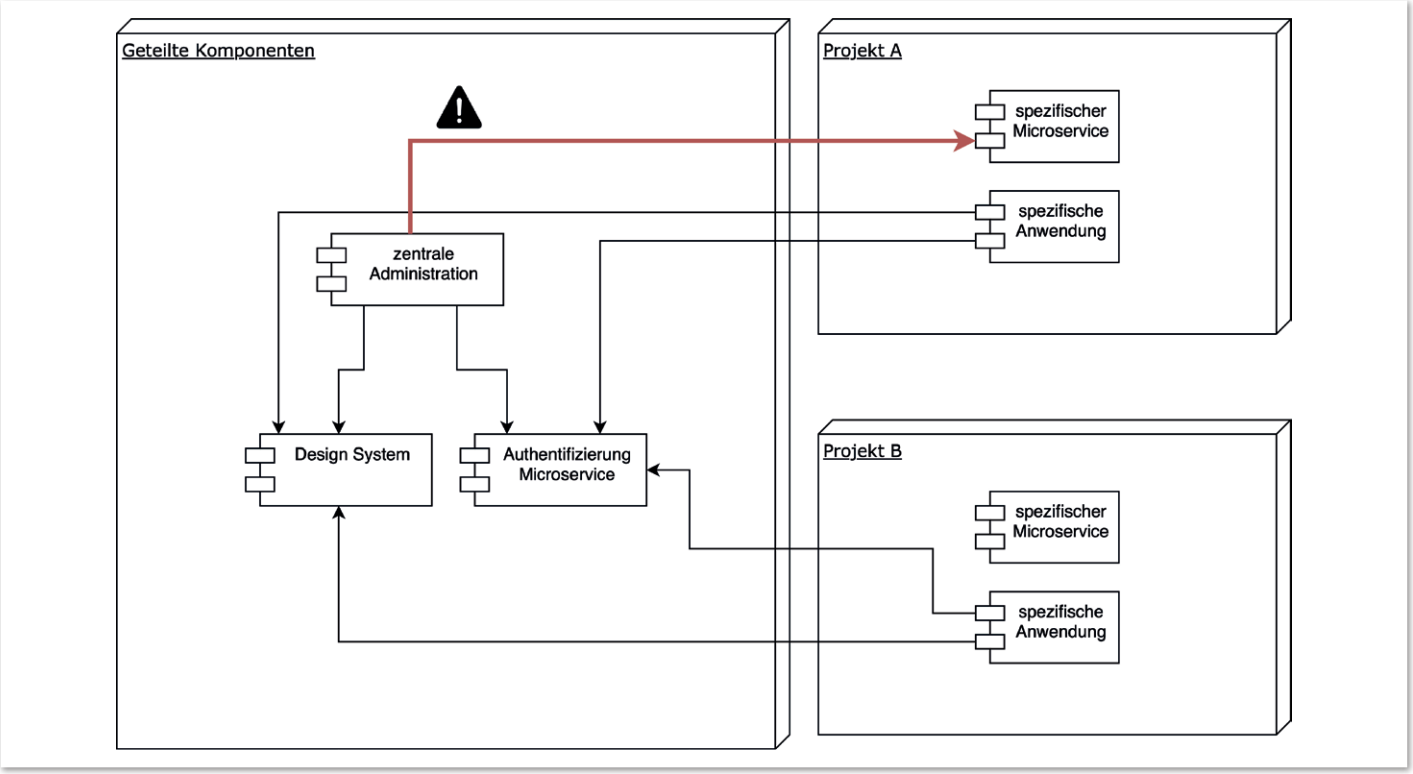


Abbildung 1: Die Abhängigkeitssituation, in die wir uns selbst begeben haben. (© Christian Siebmanns)

verwies per Abhängigkeit allerdings auf unseren projektspezifischen Microservice, der noch weiterentwickelt wurde. Folglich konnten die *TypeScript*-Typen nicht generiert und die zentrale Administration nicht gebaut werden. Dieses Problem ließ sich glücklicherweise temporär schnell lösen: Die betreffende Schnittstelle hatte sich nicht geändert und so konnten wir die Version aus dem vorigen Release nutzen, um unsere *TypeScript*-Typen zu generieren.

Diese Lösung konnte aber nur temporär funktionieren. Nach Abwägung verschiedener Architekturvarianten entschieden wir uns dazu, die projektspezifischen Features als Microfrontends umzusetzen. Der Vorteil lag auf der Hand: Wir haben separate Projekte für unsere Microfrontends, die getrennt gebaut und released werden können.

Um aus unserem Frontend-Code eine produktive Anwendung zu bauen, benutzten wir *webpack* als *Bundler*. Dieser sorgt dafür, dass *TypeScript*-Code in *JavaScript*-Code übersetzt wird und, dass sämtliche Mediendateien (Fonts, CSS-Styles, Bilder und anderes) im richtigen Format vorliegen. Darüber hinaus nimmt der *Bundler* Optimierungen vor, damit wir eine effiziente Anwendung ausliefern können.

Webpack 5 bietet zur Umsetzung von Microfrontends ein neues Feature namens *Module Federation*.

Was ist Module Federation?

Ich habe den Begriff *Module Federation* schon verwendet, doch was ist diese Technologie? Zack Jackson, der Vater von *Module Federation* beschreibt es wie folgt *[1]*:

Module federation allows a JavaScript application to dynamically load code from another application — in the process, sharing dependencies, if an application consuming a federated module does not have a dependency needed by the federated code — Webpack will download the missing dependency from that federated build origin.

In *webpack* können über das *ModuleFederationPlugin* eigene *Federated Modules* definiert werden. Ein *Federated Module* besteht aus folgenden Teilen:

- Der name ist der Name des Federated Module und gleichzeitig sein Scope.
- Der remoteEntry ist der Startpunkt, um ein Federated Module zu initialisieren.
- remotes ist eine Liste der Federated Modules, die ein Federated Module lädt
- Durch exposes werden Elemente für ein ladendes Modul nutzbar.
- shared sind die Abhängigkeiten, die sich Federated Modules untereinander teilen.

Darüber hinaus kann jedes von webpack gebaute Projekt ein Federated Module sein.

Listing 1 zeigt beispielhaft die Definition eines *Federated Modules* unter dem Namen `beispielmicrofrontend`. Es stellt ein Modul `helpers` bereit, also zur Laufzeit `helpers.js`.

Wir bauen Plug-ins für unsere Anwendung

Am Anfang stellte sich die Frage, wie wir die Schnittstelle für unsere Anwendung definieren. Über diese Schnittstelle sollte unsere Anwendung ihre Funktionalität mit Hilfe eines *Federated Module* erweitern. Bisher konnte beliebiger Code für die Funktionalität aufgerufen werden, da der Code Teil derselben Anwendung war. Als Inspiration für diese Schnittstelle diente unsere Entwicklungsumgebung: Sie ist durch Plug-ins flexibel erweiterbar.

Für die Plug-in-Architektur in unserer Anwendung haben wir ein Interface definiert. Die folgende Liste gibt einen kurzen Überblick, über die Dinge, die unser Anwendungsplugin implementieren muss:

- eine Liste aller Routen, die das Plug-in definiert
- einen Scope für Routen, Store-Einträge usw. des Plug-ins
- eine Render-Methode, die für eine aufgerufene Route das zu rendernde UI zurückgibt
- eine Init-Methode, die aufgerufen wird, um das Plug-in zu laden
- diverse Schnittstellen, um Router, Store und ähnliches von der Hostanwendung zu empfangen

Dieses Interface stellen wir über eine gemeinsam genutzte Bibliothek bereit. Mit dieser Schnittstelle erreichen wir, dass eine Anwendung zur Laufzeit über *Federated Modules* beliebig erweitert werden kann.

Auswirkungen auf unsere Codebasis

Nachdem die Plug-in-Schnittstelle implementiert war, mussten die Plug-ins für die projektspezifischen Funktionalitäten umgesetzt werden. Pro Microservice besaßen wir bereits eine passende Frontend-Bibliothek, die wiederverwendbare Funktionen zur Nutzung dieses Microservices enthielt. Wir entschieden, die Plug-ins in diese Bibliotheken zu legen. So weit, so simpel? Nicht ganz!

Da die Bibliotheken jetzt den Code für *Federated Modules* beinhielten, mussten sie auch mit *webpack* gebundled werden. Hierfür

```
const { ModuleFederationPlugin } = require("webpack").container;
const pluginConfig = {
plugins: [
     new ModuleFederationPlugin({
        name: "beispielmicrofrontend",
        exposes: {
           "./helpers": "./src/helpers",
        },

     }),
  ],
}
```

Listing 1: Ein einfaches Federated Module, das eine Datei bereitstellt.

musste eine *webpack*-Konfiguration angelegt und der Build in den bestehenden Build-Prozess integriert werden. Dabei müssen die entstehenden Artefakte der *Federated Modules* auch deployt, versioniert und archiviert werden. Auch wenn es im Nachhinein völlig einleuchtet, die Anpassungen, die für diese Änderungen, insbesondere an unserer CI/CD-Pipeline notwendig waren, sind sehr aufwändig gewesen.

Web Components und *Module Federation* passen gut zusammen

Unsere Anwendungen nutzen *Web Components*. Wir nutzen die Bibliothek Lit, um sie zu entwickeln. Um neue, nachgeladene Komponenten zu registrieren, genügt ein Aufruf von `window.customElements.define`. Es darf kein Elementname mehrfach vergeben werden. Ist das Element bereits registriert, wirft der Browser einen schweren Fehler. Unsere Komponenten sind selbstdefinierend, das heißt, der `define`-Aufruf steht direkt unterhalb der *TypeScript*-Klasse, die das Element beschreibt. Beim Import der Quelldatei wird das Element unter dem vorgegebenen Namen registriert. *Listing 2* zeigt ein einfaches Beispiel einer selbstdefinierenden Komponente.

```
import { LitElement, html } from "lit";
import { customElement } from "lit/decorators.js";

class HelloWorld extends LitElement {
  public render() {
     return html`<p>Hello world!</p>`;
  }
}
window.customElements.define("hello-world", HelloWorld);
```

Listing 2: Eine einfache Web Component mit Lit, die unter dem Tag hello-world verfügbar ist.

Unsere Komponenten werden als npm-Packages ausgeliefert. Sowohl die Anwendungen als auch unsere Plug-ins referenzieren dieses Paket. Ohne *Module Federation* stellt *webpack* sicher, dass solche Abhängigkeiten nur einmal im gebauten Artefakt vorhanden sind. So kann es nicht zur Mehrfachdefinierung von Komponenten kommen. Dieses Verhalten funktioniert mit *Module Federation* nicht. Stattdessen müssen geteilte Abhängigkeiten über die *shared*-Konfigurationseigenschaft angegeben werden.

Abhängigkeiten mit shared teilen

Mit dem Key `shared` werden in der Konfiguration des `ModuleFederationPlugin` Abhängigkeiten angegeben, die sich die *Federated Modules* untereinander teilen. *Listing 3* zeigt die Konfiguration eines *Federated Module* mit Abhängigkeiten.

Es können auch Bedingungen angegeben werden, wann eine Abhängigkeit geteilt werden soll. Hierfür kann die benötigte Versionsnummer spezifiziert werden. Diese kann per *Semantic Versioning* ähnlich zu npm *[3]* beispielsweise als *^ 1.0.0* angeben werden.

Standardmäßig werden allerdings nur die Elemente eines npm-Packages geteilt, die sich im Root des *Packagename* befinden. So wird beispielsweise der Import `import {LitElement} from "lit"` zwischen den *Federated Modules* geteilt. Folgender Import wird jedoch nicht geteilt: `import {ref} from "lit/decorators/ref"`, da sich `ref` nicht direkt im Root des Packages `lit` befindet. Um alle Elemente eines Packages zu teilen, muss in der Konfiguration der *Packagename* mit einem / abgeschlossen werden. Für die Teilung von Abhängigkeiten gibt es noch deutlich mehr Konfigurationsmöglichkeiten. Diese sind in der offiziellen Dokumentation *[2]* beschrieben.

Unerwartete Schwierigkeiten

Wir verwenden *Staging* in unserem Softwareentwicklungsprozess. Das bedeutet, wir haben verschiedene Umgebungen: Entwicklung, Test, Abnahme und Produktion. Auf diesen wird dasselbe Softwareartefakt deployt. Für das `ModuleFederationPlugin` muss die *absolute Url* eines `remoteEntries` spezifiziert werden. Das ist für unser *Staging* nicht praktikabel, da unsere Anwendungen nicht wissen sollen, wo sie deployt werden. Mit dem *external-remotes-plugin* kann die `Url` eines `remoteEntries` um Platzhalter ergänzt werden. Diese Platzhalter werden dann dynamisch zur Laufzeit ersetzt. *Listing 4* zeigt beispielhaft den Plug-ins-Teil einer webpack-Konfiguration. Es wird der Platzhalter `libUrl` definiert. Dieser gibt dynamisch den Host des *Federated Module* an.

Während der Initialisierung der Anwendung setzen wir den Platzhalter `libUrl` mit Hilfe von *window.location.origin*.

Ein weiteres Problem zeigte sich in einem fehlschlagenden CI-Build: Wir nutzen eine Content-Security-Policy, die vorschreibt, dass alles geladene *JavaScript* per SHA-384-Hash identifizierbar ist. Hierfür nutzen wir das Plug-in *webpack-subresource-integrity*. Es generiert die SHA-384-Hashes automatisch und fügt sie in die Ausgabedatei ein. Die Idee von Microfrontends ist jedoch, dass wir eine losere Kopplung erzwingen: Da wir *Federated Modules* zur Laufzeit laden und zur Build-Zeit noch nicht wissen, wie genau sie implementiert sind, funktioniert *Subresource Integrity* nicht für *Federated Modules*. An dieser Stelle blieb uns also nichts anderes übrig, als unsere Content-Security-Policy für die Administrationsmodule zu ändern: Wir verzichten auf die Generierung von SHA-384-Hashes und stattdessen steht die *script-src*-Direktive für diese Anwendung auf `self`. Damit können nur Skripte vom eigenen Server geladen werden.

```
const { ModuleFederationPlugin } = require("webpack").container;
const pluginConfig = {
plugins: [
     new ModuleFederationPlugin({
        name: "admin",
        remotes: {
           lib: "lib@http://localhost/lib/remoteEntry.js",
        },
        shared: {lit: {singleton: true}},
            ]
}
```

Listing 3: Konfiguration eines Federated Module, das die Abhängigkeit lit als Singleton definiert.

```
const { ModuleFederationPlugin } = require("webpack").container;
const ExternalTemplateRemotesPlugin = require("external-remotes-plugin");

// Plugins Teil der Webpack-Konfiguration

plugins: [
     new ModuleFederationPlugin({
         name: "admin",
         remotes: {
            lib: "lib@[libUrl]/lib/remoteEntry.js",
         },
         shared: {"lit/": {singleton: true}},
     }),
new ExternalTemplateRemotesPlugin()
]
```

Listing 4: Plug-ins in der webpack-Konfiguration. Das ExternalTemplateRemotesPlugin erlaubt die Definition von Platzhaltern in der Remote-Url.

Wie wir Remotes dynamisch definieren und nachladen

Der lokale Test funktioniert, der CI-Build läuft. Zum finalen Test prüft der Kollege im anderen Projekt die Änderung. Auch dort funktioniert die Änderung, aber ihm fällt etwas Seltsames in der Entwicklerkonsole auf: Die Anwendung versucht, auf unsere `remoteEntries` zuzugreifen. Offenbar geht *webpack* standardmäßig davon aus, dass `remoteEntries` für `remotes` verfügbar sind und lädt zumindest die `remoteEntries`. In unserem Fall werden sie im anderen Projekt jedoch nie deployt werden. Deshalb sollte die Anwendung nicht standardmäßig versuchen, unsere projektspezifischen *Federated Modules* zu laden. Wir können unsere `remotes` also nicht in der *webpack*-Konfiguration definieren, sondern wir müssen sie dynamisch im Anwendungscode hinterlegen. Dieses Konzept heißt bei *webpack Dynamic Remote Container [4]*. Über das npm-Package `@module-federation/utilities` wird eine Funktion `importRemote` bereitgestellt, mit der `remotes` dynamisch definiert werden können. *Listing 5* zeigt beispielhaft, wie ein Plug-in dynamisch aus einem *Federated Module* nachgeladen werden kann.

Zuerst setzen wir in der Variable `remote` die URL zu unserem Server zusammen, da ein *Federated Module* per vollqualifizierter URL geladen werden muss. Das Objekt, das wir `importRemote` übergeben, hat folgenden Aufbau:

- *url*: Der Dateiname des `remoteEntries`, den wir laden wollen.
- *scope*: Der Name, unter dem das *Federated Module* bereitgestellt wird. Dieser muss identisch sein zum `name`, den das *Federated Module* in seiner Konfiguration definiert.
- *module*: Die Ressource, die aus dem *Federated Module* geladen werden soll. Der Pfad entspricht jenem Pfad, der unter `exposes` in der Konfiguration des konsumierten *Federated Modules* definiert ist.

Das Ergebnis von `importRemote` in *Listing 5* ist die Plug-in-Klasse, die die projektspezifische Administrationsfunktion enthält.

```
import { importRemote } from "@module-federation/utilities";
import { LitElement } from "lit";
import type { Plugin } from "projects/common";



class MyApplication extends LitElement {
  // ...
  protected async setupRemotes() {
if (!this.spezielleAdminFunktionBenoetigt) {
     return;
}
     const remote = `${window.location.protocol}//${window.location.host}`;
     // Klasse laden
     const pluginClass = (await importRemote({
         url: `${remote}/modules/spezielle-admin-funktion`,
         scope: "spezielle_admin_funktion",
         module: "./features/admin-feature",
})) as () => Plugin;
// Instanz des Plugins erzeugen
const plugin = new pluginClass();
//Plugin laden und initialisieren
plugin.load();
     plugin.init();

  }

  // ...
}
```

Listing 5: Dynamisches Laden einer Remote und des dazugehörigen Plug-ins, falls es benötigt wird.

Hiervon müssen wir eine Instanz erzeugen, auf der wir die `load`-Methode aufrufen, um unser Anwendungsplug-in zu laden. In der Realität passiert mit den Plug-ins mehr, als die `Init`-Methode aufzurufen. So nutzen wir beispielsweise deren Render-Methoden, um Seiten aus dem Plug-in zu rendern. Das Beispiel dient nur zur Veranschaulichung. Da unsere `remotes` dynamisch nachgeladen werden, können wir den *remotes*-Eintrag aus der *Federated-Module*-Konfiguration entfernen.

Damit konnte der Kollege wieder testen. Und er gab sein Okay und hat die Änderungen gemerged. Seit diesem Tag setzen wir *Federated Modules* ein. So viel Aufwand die Implementierung im Vorhinein erforderte, in Produktion läuft sie seit langem stabil und ohne Probleme.

Unsere Anwendung ist nicht bereit für Federated Modules

Wie anfangs erwähnt, haben wir versucht mit möglichst geringem Aufwand aus einer bestehenden Anwendung *Federated Modules* zu schneiden. Das unsere Anwendung für diese Operation am offenen Herzen nur bedingt geeignet war, zeigte sich während der Entwicklung an diversen Stellen.

Mein persönlicher Favorit: Mit jedem Neu-Laden einer Unterseite eines Plug-ins meldete unsere Anwendung, die Seite nicht zu kennen. Dies konnten Nutzer:innen etwa durch Drücken der F5-Taste oder der entsprechenden Schaltfläche im Browser provozieren. Der Grund für dieses Verhalten war ein Timing-Problem: Bevor das *Federated Module* komplett geladen war, meldete der Router bereits, dass es sich um eine unbekannte Route handelte und schickte Nutzer:innen auf die 404-Seite. Die zugegeben nicht perfekte Lösung war, den Router erst zu starten, nachdem alle Plug-ins geladen und initialisiert waren. Damit verzögerte sich die Ladezeit der Anwendung beim Erststart ein wenig (danach cached der Browser die *Federated Modules*), für eine nicht häufig genutzte Backoffice-Anwendung war dies jedoch verschmerzbar. Heutzutage würde ich das Problem wahrscheinlich anders lösen: Es ist in Ordnung, wenn die Administrationsanwendung alle ihre Routen kennt. Das ein Plug-in neue Routen in einer Anwendung definiert, ist wohl ein klassisches Beispiel von Over-Engineering.

Ein weiteres Beispiel ist, dass unsere *Federated Modules* aus tausenden kleinen Dateien bestehen, die alle nachgeladen werden müssen. Es zeigt, wie effizient das *Bundling* für Webanwendungen ohne Federated Modules ist. Dass wir so viele kleine Dateien haben, liegt insbesondere daran, dass unser Design-System als *shared* Dependency eingebunden ist. Damit kann ein großer Teil der Komponenten und Funktionen nicht mehr zu einem Bundle zusammengefasst werden (denn *webpack* weiß zur Build-Zeit nicht, wie *Federated Modules* diese nutzen). Ab HTTP/2 sollte die Nutzung von vielen kleinen Dateien jedoch kein Problem mehr darstellen. Gleichzeitig haben viele kleine Dateien eine positive Auswirkung auf das Caching: Eine Änderung führt nur noch zu einer Änderung in wenigen Dateien und nicht zur Änderung des kompletten, großen Bundles.

Ausblick

Unser erstes Projekt mit *Module Federation* nutzt die geschaffene Plug-in-Architektur als klar definierte Schnittstelle. Das funktioniert in diesem Szenario gut, in Zukunft wäre es in einem Monorepository aber wünschenswert, *Federated Modules* beliebig implementieren zu können und bei der Einbindung die passenden *TypeScript*-Typen automatisch zu nutzen. Wir haben hierzu bereits Prototypen umgesetzt, es gibt aber Bestrebungen, dies auch offiziell in *Module Federation* zu ermöglichen *[5]*.

Weiterhin stellt sich die Frage, was passiert, wenn man in einem Grüne-Wiese-Projekt *Federated Modules* nutzt. Meine Zielvorstellung wäre, das Design-System als *Federated Module* bereitzustellen, sodass Updates an den Komponenten vorgenommen werden können, ohne dass die nutzenden Anwendungen neu- gebaut oder deployt werden müssen.

Nicht zuletzt stellt sich die Frage, wie die Zukunft von *webpack* und *Module Federation* aussieht: Der Erfinder von *webpack* hat mit *turbopack* mittlerweile ein neues Projekt, das als Nachfolger positioniert wird. *Turbopack* unterstützt jedoch noch keine *Module Federation*.

Zack Jackson, der Vater von *Module Federation*, scheint mittlerweile mit *rspack* eine Alternative zu *webpack* zu favorisieren *[6]*. So wurde das jüngste Update, *Module Federation* Version 1.5, für *rspack* und *webpack* veröffentlicht. Allerdings ist die aktuelle Version in *rspack* bereits standardmäßig aktiv, bei *webpack* muss sie über ein neues npm-Package erst installiert und genutzt werden. Es bleibt abzuwarten, wie lange diese Koexistenz bestehen bleibt und wie sich *Module Federation* weiterentwickelt.

Quellen

[1] Jackson, Zack. Webpack 5 Module Federation: A game-changer in JavaScript architecture. *https://medium.com/swlh/webpack-5-module-federation-a-game-changer-to-javascript-architecture-bcdd30e02669*.

[2] About semantic versioning. *https://docs.npmjs.com/about-semantic-versioning*.

[3] ModuleFederation-Plugin | webpack. *https://webpack.js.org/plugins/module-federation-plugin/#sharing-hints*.

[4] Module Federation | webpack. *https://webpack.js.org/concepts/module-federation/#dynamic-remote-containers*.

[5] Jackson, Zack. Github. Module Federation Redesign. *https://github.com/module-federation/universe/discussions/1170*.

[6] Module Federation Runtime APIs; Available in Rspack & Webpack. Github. *https://github.com/module-federation/universe/discussions/1936*.

Christian Siebmanns
viadee Unternehmensberatung AG
christian.siebmanns@viadee.de

Christian Siebmanns ist Berater bei der viadee IT-Unternehmensberatung. Er hat seinen Schwerpunkt beim Einsatz in Kund:innenprojekten im Webumfeld und interessiert sich für alles, womit Nutzer:innen interagieren können.

Strukturzementierende Tests und wie man sie vermeidet, Teil 1 – Bauen einer TestDsl

Richard Gross, MaibornWolff GmbH

Tests sollten auf Verhaltensänderungen reagieren, aber unempfindlich gegenüber Strukturveränderungen sein. Tests, die die zweite Bedingung nicht erfüllen, nennen wir strukturzementierend. Sie erschweren oder verhindern Veränderung von Struktur und Architektur. Teams, denen dies widerfährt, testen und verbessern weniger Code. Mit dem richtigen Vorgehen und einer TestDsl können wir die Zementierung allerdings komplett vermeiden und einheitliche Tests schreiben, die nach Änderung von nur einer Testcode-Zeile von Integrations- zu Unittests werden.

Wenn wir Verhalten ändern, sollten wir Tests anpassen müssen. Wenn wir Code umstrukturieren (Methoden, Klassen, Parameter oder Fields hinzufügen, umbenennen oder entfernen), sollten wir die Tests nicht ändern müssen. Beide Bedingungen sind essenziell, weil agile Entwickler ständig zwischen den zwei Modi *Verhalten ändern* und *Struktur ändern* hin- und herspringen. Mit einem fehlschlagenden Test springen wir in den Verhalten-ändern-Modus. Wenn der Test grün ist, haben wir unser Sicherheitsnetz zum Ändern der Struktur und springen zurück *(siehe Abbildung 1)*. Gleiches gilt, wenn wir unsere Tests nach der Implementierung schreiben, dann ist der Übergang zwischen den Modi nur nicht ganz so scharf.

Verhalten ändern ist risikoreich, denn mit einer gewünschten kann auch eine unerwünschte Verhaltensänderung folgen. Die meiste Zeit wollen wir also im *Struktur-ändern*-Modus verbringen und nutzen Techniken wie *preparatory refactoring [1]*, die unsere Verhaltensänderung möglichst einfach und überschaubar macht. Das benötigt aber auch Tests, die Strukturveränderungen erlauben und nicht zementieren. Mit diesen Tests können wir auch in sehr kleinen und überschaubaren Schritten vorgehen. Eine schwierige Änderung wird erst mit 10 bis 20 „r" (refactor*) Commits vorbereitet, um dann einen sehr kleinen „F" (feature) Commit zu machen.

> *When faced with a hard change, first make it easy (warning, this may be hard), then make the easy change.*
> *– Kent Beck [3]*

* Das genutzte Commit-Format ist *Arlo's Commit Notation [2]*, da dieses das Risiko vom Commit modelliert.

Wenn die Struktur allerdings zementiert ist, dann sind nicht nur struktur-, sondern auch featurebedingte Verhaltensänderungen schwieriger. Denn wir bauen Features nicht mehr dort ein, wo sie sinnvoll sind, sondern wo sie einfacher einzubauen sind. Der Code wird in der Folge immer unübersichtlicher. Am Anfang findet man Code nicht mehr, er liegt schließlich an der einfachsten Stelle, nicht an der sinnvollsten. Im späteren Verlauf gibt es immer mehr unerwartete Abhängigkeiten, immer mehr „unknown Unknowns". Wir ändern eine Stelle und eine andere Codestelle ist dadurch verbuggt.

Die strukturzementierenden Tests haben die Wartbarkeit zerstört. Oder wir haben sie gelöscht und uns Regressionen eingefangen.

Tests sind dabei auf zwei Arten strukturzementierend:

1. Durch Redundanz: Tests nutzen an mehreren Stellen immer wieder dieselbe Struktur. Je öfter dieselbe Struktur (Klasse, Methode) verlangt und je exzessiver gemockt wird, desto zementierter ist das Design.
2. Durch Tests auf falscher Ebene: wir testen instabile Elemente und nicht Module.

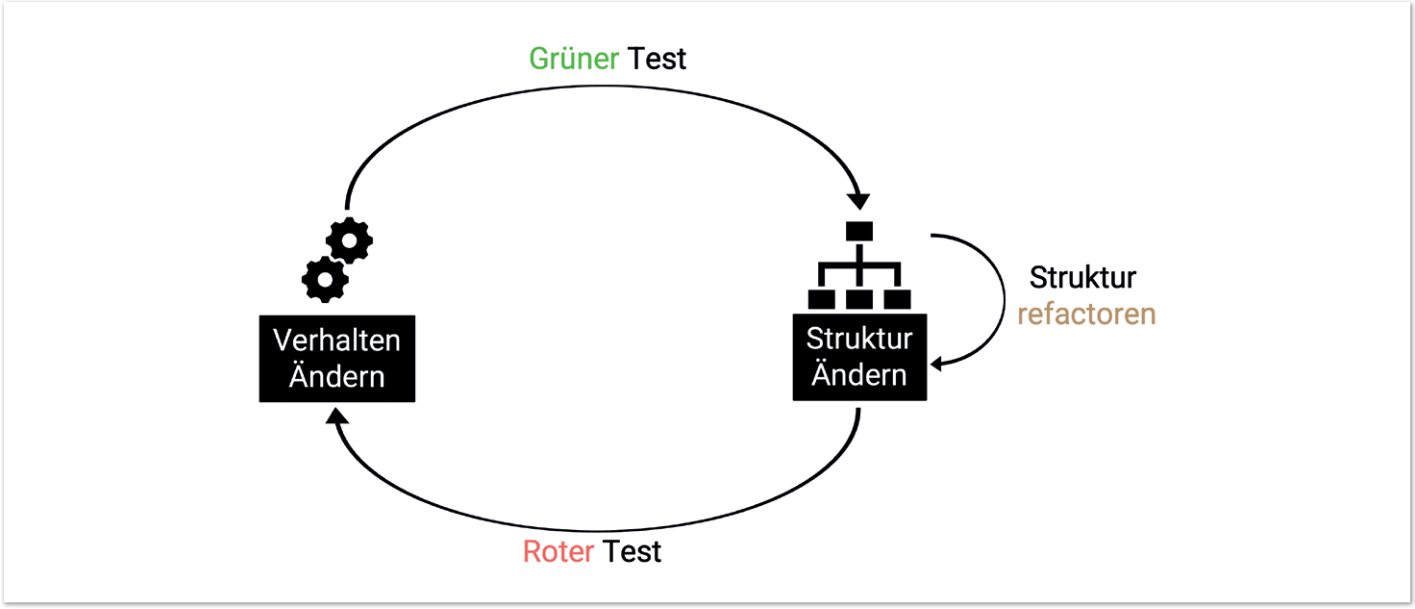


Abbildung 1: Die zwei Entwicklungsmodi frei nach Kent Beck (© Richard Gross)

```
@Test
void should_be_able_to_rent_book(){
     // given
     var book = new Book(bookId("b"), "Refactoring", "Martin Fowler"); // <1>
     var permission = new Permission(permissionId("p"), CAN_RENT_BOOK);
     var role = new Role(roleId("r"), "Renter", permission.id);
     var user = new User(userId("u", "Alex Mack", role.id);

     var books = new InMemoryBooksDouble(); // <2>
     var permissions = new InMemoryPermissionsDouble();
     var roles = new InMemoryRolesDouble();
     var users = new InMemoryUsersDouble();

     books.add(book); // <3>
     permissions.add(permission);
     roles.add(role);
     users.add(user);

     var testee = new RentingService(new ClockDouble(), books, permissions, roles, users, …);
  // <4>
     // when
     var result = testee.rentBook(book, user); // <5>
     // then
     assertThat(result.isRented()).isTrue();
}
```

Listing 1: Initialer Test

In Teil 1 dieser Artikelserie beschäftigen wir uns damit, wie wir mit einer TestDsl die Strukturzementierung durch Redundanz vermeiden können, in Teil 2 erläutern wir die Konzepte der Dsl genauer und Teil 3 geht auf die richtigen Testebenen ein.

Der Initiale Test

Die Zementierung durch Redundanz können wir gut in *Listing 1* sehen.

<1> Die Basis für die meisten unserer Tests sind in diesem Beispiel Bücher. Daher instanziieren wir in den meisten unserer Tests mindestens ein Buch und mit jeder redundanten Instanziierung zementieren wir die Struktur eines Buchs immer mehr, da eine Änderung Anpassungen in hunderten von Tests nach sich ziehen würde.

<2> Die Repositories folgen der Namenskonvention *{Implementierung}{Name der verwalteten Entity mit Plural-s}[Double]*. Am *Double* Postfix erkennt man, dass es sich hier um ein *Test Double [4]* handelt der in *src/testFixtures* (oder *src/test*) liegt. Der Postfix ist nicht notwendig, er macht allerdings das Finden aller erstellten Doubles sehr angenehm.

```
@Test
void should_be_able_to_rent_book(){
  // given
  var book = makeBook();
  var permission = makePermission(CAN_RENT_BOOK);
  var role = makeRole(permission);
  var user = makeUser(role);

  var books = new InMemoryBooksDouble();
  // restlicher Test
  // ...
}

static Book makeBook(){
  return new Book(bookId("b"), "Refactoring", "Martin Fowler");
}
```

Listing 2: Factory-Methoden

<3> Damit unser zu testender Code die Bücher auch aus dem Repository laden kann, müssen diese dort auch abgelegt werden. Wie man das macht, ist für den Test eigentlich ein irrelevantes Implementierungsdetail. Durch Redundanz in mehreren Tests wird dieses allerdings zementiert. Außerdem macht es den Test unnötig lang.

<4> Je öfter wir den `RentingService` instanziieren, desto weniger Interesse haben wir daran, seine Dependencies zu verändern. Die Abhängigkeiten von unserem Service werden zementiert.

<5> Die Methode, deren Verhalten wir eigentlich testen wollen. Mit mehreren Tests die Struktur dieser Methode zu zementieren, ist ein Trade-off, den wir eingehen wollen, weil wir diesmal etwas zurückbekommen: Feedback. Feedback darüber, ob die Methode angenehm und logisch zu benutzen ist.

Neben dem *Test Smell* „strukturzementierend" weist dieser Test noch ein paar weitere Probleme auf:

1. Langer Test: Es gibt viel zu lesen und damit viele, potenziell versteckte Fehler.
2. Irrelevante Details: Muss es ein bestimmtes Buch sein oder ist jedes möglich? Müssen die Felder die angegebenen Werte haben, damit der Test Erfolg hat?

Wir werden den Test daher in mehreren Schritten refaktorisieren und uns an die finale TestDsl *(siehe Listing 13)* herantasten.

1. Refactoring – Factory-Methoden

Wir können die Redundanz brechen und Tests fokussieren, indem Tests nicht mehr direkt Objekte instanziieren, sondern über Factory-Methoden *(siehe Listing 2)* gehen.

Für dieses kleines Beispiel ist das in Ordnung, doch laufen wir schnell in viele Probleme dieses Ansatzes (auch bekannt als *Object Mother Pattern [5]*):

```
@Test
void should_be_able_to_rent_book(){
      // given
      var book = new BookBuilder().build();
      var permission = new PermissionBuilder().withPermission(CAN_RENT_BOOK).build();
      var role = new RoleBuilder().withPermissions(permission).build();
      var user = new UserBuilder().withRole(role).build();

      var books = new InMemoryBooksDouble();
      // restlicher Test
      // ...
}
```

Listing 3: Entity-TestBuilder

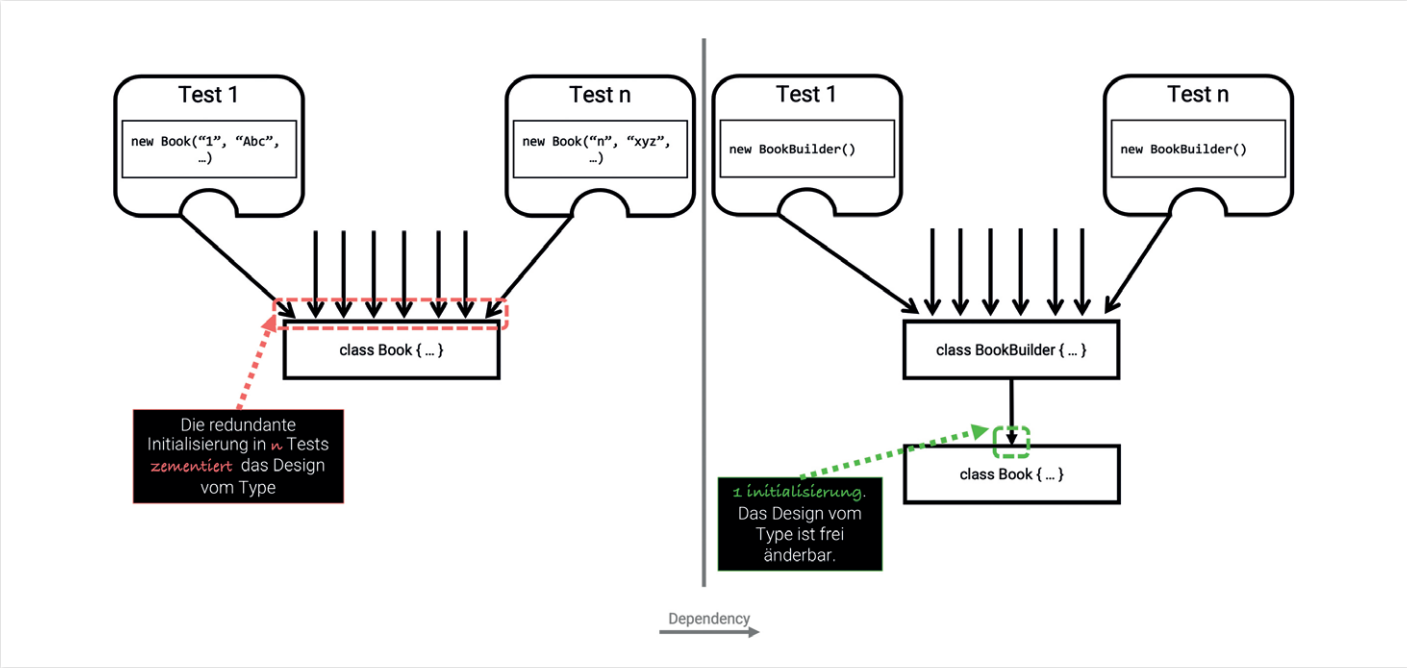


Abbildung 2: Zementieren der Struktur durch Init (© Richard Gross)

1. Entweder jeder neue Anwendungsfall bekommt eine neue Methode (`makeBook()`, `makeExpensiveBook()` etc.)
2. oder die Methode bekommt dutzende optionale Parameter, ohne dass ersichtlich ist, welche Parameter voneinander abhängig sind.

Das heißt nicht, dass man Factory-Methoden nicht nutzen sollte. Gerade, wenn man neue Strukturen einführt, sind Factory-Methoden ideal, denn wir können sie mit wenig Aufwand direkt unter unserem Test erstellen. Wenn wir uns unserer Struktur sicherer sind, sollten wir allerdings den `Builder` aus dem nächsten Abschnitt *(siehe Listing 3)* erst innerhalb der Factory-Methode nutzen und dann mit unseren Refactoring-Tools inlinen.

2. Refactoring – Einfache Builder

Statt der Factory-Methode beziehungsweise dem *Object Mother Pattern* setzen wir lieber auf einen Builder *(siehe Listing 3)*.

Wenn man direkt die `build()`-Methode aufruft, wird die Entity mit Standardwerten belegt. Mit den `withX()`-Methoden können wir die Standardwerte, falls notwendig, auf unseren spezifischen Test anpassen. Wir sind also deutlich flexibler als mit der Verwendung von Factories/*Object Mother Patterns*, da nicht jeder Fall eine eigene Methode braucht.

Mit dem Builder haben wir auch die redundanten Abhängigkeiten auf eine test-spezifische Abstraktion umgelenkt *(siehe Abbildung 2)*. Änderungen an der Struktur der Entity müssen wir jetzt nur noch im Builder, nicht in *n* Tests nachziehen. Wir können die Strukturänderungen im Builder einpflegen, weil wir durch die Tests abgesichert sind, die den Builder bereits nutzen. Werden bestehende Tests rot, haben wir etwas kaputt gemacht.

Neben flexiblem Test-Setup und Vermeiden der Zementierung von Struktur bietet uns so ein `Builder` noch weitere Vorteile:

1. Der Test erwähnt **keine irrelevanten Details** mehr. Der obige Test zeigt uns, dass er irgendein Buch braucht und nicht ein spezifisches.
2. Der Builder stellt **wesentliche Unterschiede** der Tests heraus. Wir sehen durch das Nutzen der `with()`-Methode, dass der User unbedingt das Recht `CAN_RENT_BOOK` braucht.
3. Im Builder haben wir einen eindeutigen Platz, um fachlich sinnvolle Default-Werte zu hinterlegen *(siehe Listing 4)*. Dies ist eine praktische Dokumentation für Entwickler.

<1> Sinnvolle Defaults, die repräsentativ für die Produktion sind, werden hier hinterlegt.

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

      public BookId id = ids.next(BookId.class);
      public String title = "Refactoring"; // <1>
      public String author = "Martin Fowler";
      public Instant createdOn = clock.now();

      public BookBuilder(Clock clock, Ids ids){ // <2>
            super(clock, ids);
      }

      public BookBuilder(){ // <3>
            this(globalTestClock, globalTestIds);
      }

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

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

      // <5>
}
```

Listing 4: Entity-TestBuilder

<2> Wir sehen an dieser Stelle bereits, dass wir die zwei Hauptquellen von nichtdeterministischen Tests (Zeit und Zufallswerte) von außen eingeben können.

```java
@Test
void should_be_able_to_rent_book(){
      // given
      var book = new BookBuilder().build(); // <1>
      var userCombo = new UserComboBuilder.with(it ->
            it.hasPermissions(CAN_RENT_BOOK)
      ).build();
      // Combo includes:
      //   var permissions = userCombo.permissions();
      //   var role = userCombo.role();
      //   var user = userCombo.user();

      var books = new InMemoryBooksDouble();
      // restlicher Test
      // ...
}
```

Listing 5: Entity-TestBuilder

<3> Mit dem TestDsl-Refactoring fällt dieser parameterlose Konstruktor weg.

<4> Die `with()`-Methode beschleunigt das Schreiben des initialen Builders. Man muss sich dann allerdings daran gewöhnen, dass der Builder *public* Fields hat – einen Trade-off, den man für Tests eingehen kann. Etwas flexibler sind die spezifischen `withX()`, da man sie überladen kann.

<5> Als Alternative zur generischen `with()` kann man hier unten Field-spezifische `withX()`-Methoden einführen.

Wir sind allerdings noch nicht fertig, denn die Kombination aus *Permission*, *Role* und *User* kann man noch stärker modellieren und den Test weiter konkretisieren.

3. Refactoring – Bundle Builder

Damit wir mehrere separat gespeicherte Objekte aufeinander abgestimmt bauen können, führen wir das Konzept des *Combo* Builders ein *(siehe Listing 5)*.

```java
public class UserComboBuilder implements TestBuilder<UserCombo> {

      // combination fields
      private List<Permission> permissions = Collections.emptyList();

      public UserCombo build(){
         var role = new RoleBuilder().withPermissions(permissions).build();
         var user = new UserBuilder().withRole(role).build();
      }

      public UserBundleBuilder hasPermissions(PermissionCode... permissionCode) {
         this.permissions = Stream.of(permissionCode)
             .map(code -> new Permission(code))
             .toList();
         return this;
      }
}
```

Listing 6: Entity-TestBuilder

```
private TestState a; // <1>

@Test
void should_be_able_to_rent_book(){
      // given
      var book = a.book(); // <2>
      var userCombo = a.userCombo(it -> it.hasPermission(CAN_RENT_BOOK));

      var books = new InMemoryBooksDouble();
      var permissions = new InMemoryPermissionsDouble();
      var roles = new InMemoryRolesDouble();
      var users = new InMemoryUsersDouble();

      books.add(book);
      permissions.addAll(userCombo.permissions());
      roles.add(userCombo.role());
      users.add(userCombo.user());

      var testee = new RentingService(new ClockDouble(), books, permissions, roles, users, …);
      // WHEN + THEN
      // ...
}
```

Listing 7: TestState der TestDsl

```
private TestState a;
private Floor floor; // contains the floor that the application is build on

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

      var testee = new RentingService(floor); // <3>
      // WHEN + THEN
      // ...
}
```

Listing 8: Floor der TestDsl

```
public class RentingService {
      private final Clock clock;
      private final Books books;
      // etc.

      public RentingService(Floor floor) {
            this.clock = floor.clock();
            this.books = floor.books();
            // etc.
      }
}
```

Listing 9: RentingService zieht sich seine Dependencies

```
private TestState a;
private Floor floor;

@BeforeEach
void init(){
      var dsl = TestDsl.of(unitFloor());
      a = dsl.testState();
      floor = dsl.floor();
}
```

Listing 10: TestDsl im BeforeEach instanziieren

Um die Komplexität des *Combo* Builders gering zu halten, baut er immer nur Standardfälle *(siehe Listing 6)*. Für schwierigere und untypische Situationen, zum Beispiel wenn ein User mehrere Rollen hat, nutzt man wieder die Einzel-Builder von *Permission*, *Role* und *User*. Das ist wichtig, da die ganzen Spezialfälle sehr viel unwartbaren Code erzeugen. Die Faustregel ist, dass ein Builder nie `if` oder `switch` beinhalten sollte.

Das Nutzen der Builder hat den Test bereits gut entschlackt. Bisher haben wir allerdings immer noch das Implementierungsdetail der Repositories. Wir müssen erstellte Entities immer noch in Repositories ablegen und der Test muss wissen, wie das geht.

4. Refactoring – TestDsl

Zunächst führen wir den *TestState* ein *(siehe Listing 7)*.

<1> Der TestState ist eine Klasse, die alle Builder kennt.

<2> Build-Aufgaben werden immer an die bereits geschriebenen Builder delegiert.

Auf den ersten Blick gewinnen wir nur etwas Kompaktheit: `xyzBuilder()` müssen nicht mehr instanziiert werden und wir brauchen auch keine `.build()`-Methode. Hinter den Kulissen haben wir aber noch viel mehr gewonnen. Der `TestState` ist jetzt ein zen-

traler Punkt, der alle erstellten Entities kennt. Wir können den State daher bitten, alle erstellten Entities in den Repositories abzulegen und unseren Test noch weiter verschlanken *(siehe Listing 8)*.

<1> Mit diesem Aufruf speichern wir `book`, `permission`, `role` und `user` in den jeweiligen Repositories. Theoretisch hätte bereits der Aufruf von `a.book();` das Buch im `BookRepository` speichern können. Das `saveTo()` macht das Speichern allerdings expliziter und bietet auch die Flexibilität, Entities zu erstellen, die nicht automatisch in Repositories landen.

<2> Wir gruppieren alle `Ports` im so genannten `Floor`, der Boden, auf dem unsere Anwendung steht, in die Außenwelt. Ein Repository ist so ein `Port`, genauso wie `Clock` oder ein externer `Client`. Durch den Floor können wir in Tests sehr flexibel steuern, wie unser *testee* mit der Außenwelt kommuniziert. Wir können in Tests unserer Anwendung den Boden unter den Füßen wegziehen und einen deutlich testbareren Boden hinstellen. In der *Ports-&-Adapters-Architektur [6]* ist der `Floor` gleichbedeutend mit den *driven*, aber nicht den *driving* Ports. Da man schnell überlesen kann ob etwas *driven* oder *driving* ist, kamen die Begriffe nicht in Frage. `Floor` wurde als Begriff gewählt, weil er kurz ist und man damit eine Analogie für Software- ~~Architekten~~ Gärtner hat, die sich um den *Forest* `Floor`, den *Forest* `Canopy` und den Wald kümmern. Alternative Namen für *driven* (= outcomes) oder *driving* (= triggers) `Port` waren zu dem Zeitpunkt nicht bekannt.

<3> Wir haben den `Floor` zu einem Teil unseres Produktionscodes gemacht. Zum Instanziieren von Service-Klassen braucht man immer nur den `Floor` *(siehe Listing 9)* und muss nicht mehr die konkreten Abhängigkeiten schreiben. Alternativ hätten wir den *Constructor* des Service auch gleich lassen und für Tests eine `configureRentingService(floor)`-Methoden schreiben können, die Dependencies aus dem `Floor` zuweist. Beide Wege vermeiden die Strukturzementierung des `RentingService`. Würden wir zum Instanziieren des Service einen *DI-Container* wie Spring nutzen, hätten wir denselben Vorteil. Viele dieser Container machen allerdings Tests durch ihren Startup-Overhead langsamer und erschweren Testparallelisierung durch *context caching*, weswegen sie für Unittests keine gute Wahl sind. Diese Empfehlung teilt auch das *Spring Framework [7]*.

Damit die Tests voneinander isoliert sind, instanziieren wir `TestState` und `Floor` für jeden Test aufs Neue *(siehe Listing 10)*.

Der Floor selbst ist nur ein Interface, das alle Dependencies kennt *(siehe Listing 11)*. Die Unittest-Implementierung `unitFloor()` liefert beim Aufruf der Methoden dann *InMemoryDoubles* zurück.

Unser Test sieht damit schon sehr kompakt aus *(siehe Listing 12)*.

Mit diesem Refactoring sind wir schon sehr weit gekommen:

1. Wir konnten das Setup für unseren Test in nur 4 Zeilen abbilden.
2. Wir konnten das ganze Setup an derselben Stelle wie unseren Test schreiben. Man sieht auf einen Blick, welche Vorbedingungen der Test benötigt und muss weder scrollen noch eine andere Datei öffnen, um den Kontext zu verstehen.
3. Wir konnten irrelevante Details ausblenden (man braucht irgendein `book` und irgendeinen `user`) und relevante herausstellen (der `user` braucht das Recht `CAN_RENT_BOOK`).
4. Wir haben eine einheitliche Art, das Testsetup für alle Tests zu machen.
5. Wir konnten ein strukturzementierendes Testsetup vermeiden.

Eine Verbesserung können wir allerdings noch machen.

```
public interface Floor {
     public Clock clock();
     public Books books();
     // etc.
}
```

Listing 11: Floor der TestDsl

```
private TestState a;
private Floor floor;

@BeforeEach
void init(){
     var dsl = TestDsl.of(unitFloor());
     a = dsl.testState();
     floor = dsl.floor();
}

@Test
void should_be_able_to_rent_book(){
     // given
     var book = a.book();
     var userCombo = a.userCombo(it -> it.hasPermission(CAN_RENT_BOOK));
     a.saveTo(floor);

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

Listing 12: Vollständiger Test mit TestDsl

5. Refactoring – Extension

Bisher müssen wir in jedem Test redundanten Initialisierungscode für die TestDsl im `@BeforeEach`-Block schreiben. Falls wir mit JUnit 5 unterwegs sind, können wir das Ganze in einer Annotation zusammenfassen *(siehe Listing 13)*.

<1> Unser `@BeforeEach` geht komplett in der Annotation `@Unit` auf.

<2> Durch die Annotation werden die zwei Teile unserer Dsl zu Parametern des Tests.

Die neue Annotation registriert eine *JUnit-5*-Extension *[8]*. Eine solche Extension kann durch Implementierung spezieller Interfaces auf den *Test LifeCycle* reagieren. Uns interessiert hier nur `ParameterResolver`, um die zwei möglichen Parameter `TestState` oder `Floor` unserem Test übergeben zu können *(siehe Listing 14)*.

<1> Mit `@ExtendWith` verbinden wir Annotation mit dem Extension-Code.

<2> Eine normale Java-Annotation. Der Name ist frei wählbar.

<3> Extensions müssen den State immer in einem Store speichern. Dieser ist eindeutig pro Namespace.

<4> Diese Creator-Funktion wird genutzt, wenn noch keine Dsl für den Test erstellt wurde. Pro Test wird die `resolveParameter()`-Methode genau zwei Mal aufgerufen. Einmal für den `TestState` und einmal für den `Floor`. Damit die gleiche Instanz der Dsl zurückgegeben wird, nutzen wir `getOrComputeIfAbsent()`.

<5> Über den *ParameterType* erkennen wir, was zurückgegeben werden soll.

Neben der hier gezeigten Unittest-Extension können wir natürlich noch eine weitere Extension schreiben, die `IntegrationTestExtension`. Diese sieht gleich aus, nutzt aber als Creator-Funktion `(key) -> testDslOf(integrationFloor())`. Der `TestState` bleibt der Gleiche, aber die Implementierung

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

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

Listing 13: Vollständiger Test mit TestDsl-Extension

```
@Target({ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@org.junit.jupiter.api.extension.ExtendWith(UnitTestExtension.class) // <1>
public @interface Unit { } // <2>

class UnitTestExtension implements ParameterResolver {
     @Override
     public Object resolveParameter(
                ParameterContext parameterContext,
                ExtensionContext extensionContext
          ) throws ParameterResolutionException {

          var storeNamespace = Namespace.create(
                getClass(), context.getRequiredTestMethod());
          var store = extensionContext.getStore(store); // <3>

          var dsl = store.getOrComputeIfAbsent(
                "UNIT_TEST_DSL",
                (key) -> testDslOf(unitFloor()), // <4>
                UnitTestDsl.class
          );

          var parameterType = parameterContext.getParameter().getType(); // <5>
          if (parameterType.equals(TestState.class))
                return dsl.testState();
          else if (parameterType.equals(Floor.class))
                return dsl.floor();
          else
                throw new ParameterResolutionException("...");
     }
     // ...
}
```

Listing 14: resolveParameter() der TestDsl-Extension

des `Floor` ist ein `IntegrationFloor`, der keine `InMemory-Doubles` beinhaltet, sondern `Jpa`-Repositories.

Da der `TestState` nur das `Floor`-Interface kennt und nicht die konkrete Implementierung, können wir jetzt jeden beliebigen Test mit dem Ändern **einer einzigen Annotation** von einem `@Integration`- zu einem `@Unit`-Test machen.

Gerade für Legacy-Code ist diese Eigenschaft der TestDsl sehr hilfreich, denn bei diesem Code liegt oft auch viel Domänen-Logik in der Datenbank. Man schreibt daher anfangs viele Integrationstests zur Absicherung von Regressionen. Hat man die Domänen-Logik aus der Datenbank und in den Anwendungscode gezogen, kann man die anfangs geschriebenen Tests mit einem Einzeiler zu Unittests umwandeln. Ohne eine TestDsl müsste man sie auf Unit-Ebene komplett neu schreiben, weswegen viele Teams das nicht machen, auf langsamen Integrationstests sitzen bleiben und trotz steigender Testcoverage nicht viel schneller iterieren können.

Alternativen

Einen ähnlichen Ansatz wie die TestDsl verfolgt auch „Testing without Mocks“ *[9]*. Bei diesem Ansatz muss man allerdings seinen Produktionscode stärker modifizieren, da wir spezielle Test-Doubles, die sogenannten *Nullables* *[10]*, direkt im Produktionscode unterbringen.

Auch die Refactoring-Tools unserer IDE können gewissen Formen der Zementierung von unserem initialen Test *(siehe Listing 1)* abfangen. „Change Signature“ ist dabei das hilfreichste Refactoring gegen Strukturzementierung. Man kann damit sehr gut Konstruktor-Parameter entfernen. Hinzufügen selbiger bietet sich allerdings nur an, wenn der in Tests dann eingefügte Parameter-Default sehr einfach ist und keine Abhängigkeit auf einen anderen *state* im Test hat. Mit dem Refactoring kann man sich außerdem Bugs einfangen, da Default-Werte nicht nur im Testcode, sondern auch im Produktionscode gesetzt werden und man sie dadurch nicht mehr anpassen kann. Die `Builder` der Dsl sind deutlich flexibler und verhindern mehr Zementierung bei Entities. Gleiches gilt für den `TestState`, der ein flexibleres Anpassen der `Ports` ermöglicht. Refactoring-Tools sind daher kein Ersatz, sondern eine Ergänzung für die TestDsl.

Das Refactoring-Framework *Open Rewrite* *[11]* wirkt vielversprechend, scheint aber auf Framework-Migrationen ausgelegt zu sein. Es ist daher eher eine Ergänzung zu der auf Domänen-Logik fokussierten TestDsl.

Zwischenfazit

Mit der TestDsl können wir unser Test-Setup:

1. für alle Tests einheitlich,
2. vollständig (nichts muss ausgelagert werden),
3. kompakt (obwohl nichts ausgelagert wurde),
4. frei von irrelevanten Details,
5. mit herausgestellten relevanten Details,
6. lesbar,
7. wartungsarm,
8. parallelisierbar,
9. schnell,
10. und frei von strukturzementierung

schreiben.

Die TestDsl ist natürlich nicht kostenlos. Sie ist aber auch nicht teuer. Wir müssen einmalig ein Fundament mit Extension, TestState, Floor und BaseInMemoryDouble legen. Die Erfahrung aus mehreren JVM- und Node-Projekten zeigt allerdings, dass der Wartungsaufwand gering ist, sobald man das Fundament gelegt hat.

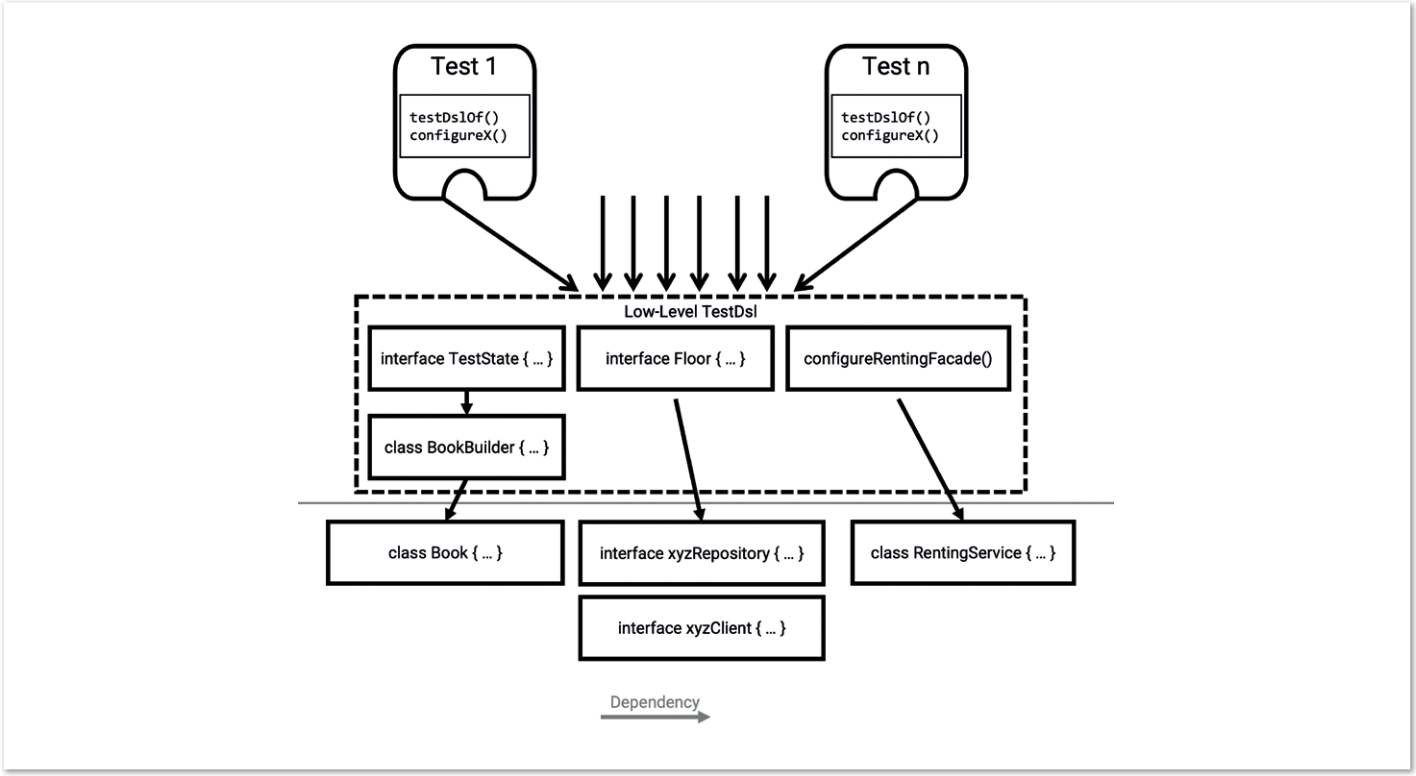


Abbildung 3: Die TestDsl schiebt sich zwischen die Tests und die Struktur des Produktionscodes. (© Richard Gross)

Es kommt selten vor, dass man neue Entities anlegen muss. Man arbeitet viel mehr mit bestehenden Entities sowie Services und strukturiert diese um. Hier zahlt das initiale Investment dann kontinuierlich Dividende. Da das gesamte Setup über die Dsl gemacht wird *(siehe Abbildung 3)*, ist bei Strukturänderungen auch nur noch die Dsl betroffen.

Abbildung 3 zeigt allerdings auch, was der eigentliche Trade-off ist, den wir mit der Dsl eingehen: Verlust von Feedback zu unserem Setup. Ohne Dsl merkt man beim Schreiben von Tests, ob das Setup „nervt". Wenn man zum Testen sehr viel Code schreiben muss, dann fragt man sich, ob das nicht auch einfacher geht. Es entsteht ein ganz natürlicher Druck, der zum Verbessern der Struktur motiviert. Dieses „nervige" Setup kann in der Dsl versteckt werden. Daher ist es umso wichtiger, die Dsl nur für das Setup zu nutzen, sie so schlicht wie möglich zu lassen, so wenig wie möglich Combo-Builder zu definieren und beim Anpassen der Dsl immer wieder zu überlegen, ob die Struktur gerade auf dem richtigen Weg ist.

Wer sich mehr für das Thema interessiert kann den TestDsl-Beispielcode auf GitHub einsehen *[12]* oder sich den Vortrag zur „Beehive Architecture" *[13]* anschauen, der sich ebenfalls um die TestDsl dreht.

Ausblick

In diesem Teil haben wir gesehen, wie man mit der TestDsl Strukturzementierung durch Redundanz lösen kann.

Im nächsten Teil beschäftigen wir uns näher mit den Konzepten, auf der die TestDsl aufbaut. Wir gehen auf das Design der Builder ein, wie man die Dsl mit `@SpringBootTest` kombiniert, was der Unterschied zwischen einer High- und einer Low-Level TestDsl ist, wie man Test-Doubles synchron zum Produktionscode hält und warum der exzessive Einsatz von Mocking-Frameworks auch zu Strukturzementierung führt.

In Teil 3 werden wir erörtern, wie man mit dem richtigen Vorgehen die Strukturzementierung durch Tests auf falscher Ebene verhindert.

Quellen

[1] M. Fowler, „An example of preparatory refactoring". 2015. Verfügbar unter: *https://martinfowler.com/articles/preparatory-refactoring-example.html*

[2] A. Belshee, „Arlo's Commit Notation". 2018. Verfügbar unter: *https://github.com/RefactoringCombos/ArlosCommitNotation*

[3] K. Beck, „Mastering Programming". Verfügbar unter: *https://tidyfirst.substack.com/p/mastering-programming*

[4] G. Meszaros, „Test Double". 2011. Verfügbar unter: *http://xunitpatterns.com/Test%20Double.html*

[5] M. Fowler, „Object Mother". 2006. Verfügbar unter: *https://martinfowler.com/bliki/ObjectMother.html*

[6] A. Cockburn, „Hexagonal architecture". 2005. Verfügbar unter: *https://alistair.cockburn.us/hexagonal-architecture/*

[7] T. Spring, „Unit Testing". 2006. Verfügbar unter: *https://docs.spring.io/spring-framework/docs/2.0.4/reference/testing.html#unit-testing*

[8] T. JUnit5, „JUnit 5 User Guide - Extension Model". Verfügbar unter: *https://junit.org/junit5/docs/current/user-guide/#extensions*

[9] J. Shore, „Testing Without Mocks: A Pattern Language". 2023. Verfügbar unter: *https://www.jamesshore.com/v2/projects/nullables/testing-without-mocks*

[10] J. Shore, „Nullables". 2023. Verfügbar unter: *https://www.jamesshore.com/v2/projects/nullables/testing-without-mocks#nullables*

[11] T. Moderne, „Large-scale automated source code refactoring". 2024. Verfügbar unter: *https://docs.openrewrite.org/*

[12] R. Gross, „TestDsl (Avoid structure-cementing Tests)". 2024. Verfügbar unter: *https://github.com/Richargh/testdsl*

[13] R. Gross, „Beehive Architecture". 2023. Verfügbar unter: *http://richargh.de/talks/#beehive-architecture*

Richard Gross
MaibornWolff GmbH
richard.gross@maibornwolff.de

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

Community-Konferenz organisiert von Java User Groups aus dem Norden

JAVA FORUM NORD

Hannover Congress Centrum - Dienstag, 10. September 2024

Das Java Forum Nord ist die Community Konferenz organisiert von Java User Groups (nicht nur) aus dem Norden. Informiere dich in unzähligen Vorträgen von hochkarätigen Speakern über neuste Technologien und die aktuellen Entwicklungen im Java Umfeld.

Das Java Forum Nord ist eine eintägige, nicht-kommerzielle Konferenz in Norddeutschland mit Themenschwerpunkt "Java für Entwickler und Entscheider". Mit über 30 Sessions wird ein vielfältiges Programm geboten. Der regionale Bezug bietet zudem interessante Networkingmöglichkeiten.

http://javaforumnord.de @JavaForumNord@ijug.social

eck*cellent IT
* software . projekte . prozesse

cloudogu

msg DAVID

SET

CGI

eddyson

Java aktuell

CQRS und DOP mit modernem Java

Simon Martinelli, 72 Services

Die Entwicklung von leistungsfähigen und gleichzeitig wartbaren Softwarelösungen steht im Mittelpunkt der modernen Softwareentwicklung. Das Command-Query-Responsibility-Segregation-Muster (CQRS) bietet hierfür eine effiziente Methode, indem es eine klare Trennung zwischen dem Ausführen von Befehlen und dem Abfragen von Daten vornimmt, was die Systemarchitektur vereinfacht, und die Performance verbessert. Gleichzeitig bringt der Ansatz des Data-Oriented Programming (DOP) eine starke Fokussierung auf die effiziente Handhabung von Daten. Dieser Artikel zeigt auf, wie die Integration von CQRS und DOP zu robusteren, skalierbaren und leichter wartbaren Systemen führt.


XML
CSS
UX
Ai
Ps

Einführung in CQRS

Command Query Responsibility Segregation (CQRS) ist ein Muster, das erstmals von Greg Young *[1]* beschrieben wurde. Es basiert auf dem Prinzip der Trennung von Verantwortlichkeiten und geht auf das *Command-Query-Separation-Prinzip* (CQS) zurück, das ursprünglich von Bertrand Meyer in seinem Buch „Object-Oriented Software Construction" [ISBN: 9780136290490] vorgestellt wurde. Während CQS besagt, dass Methoden entweder Befehle sein sollten, die den Zustand eines Objekts ändern, aber keinen Wert zurückgeben, oder Abfragen, die einen Wert zurückgeben, aber den Zustand nicht ändern, erweitert CQRS dieses Prinzip auf die Architekturebene von Softwareanwendungen.

CQRS trennt die Verantwortung für das Verarbeiten von Befehlen (*Commands*), die den Zustand eines Systems ändern, von der Verantwortung für das Abfragen (*Queries*) von Informationen über diesen Zustand *(siehe Abbildung 1)*. Diese Trennung ermöglicht eine optimierte Gestaltung beider Verantwortungsbereiche, was zu einer besseren Strukturierung führen kann. Durch die getrennte Modellierung von Befehlen und Abfragen können Entwickler auch komplexere Geschäftslogiken klarer und einfacher implementieren.

Die Einführung von CQRS in einem System kann jedoch dessen Komplexität erhöhen, da zwei separate Modelle verwaltet werden müssen. Im Laufe des Artikels werden wir aber sehen, dass dieser vermeintliche Nachteil auch ein großer Vorteil in Bezug auf die Datenabfragen und die Performance sein kann. Ebenfalls kann es zu einer vereinfachten Wartbarkeit führen, da Änderungen an der Abfragefunktionalität unabhängig von der Befehlslogik durchgeführt werden können, und umgekehrt. Darüber hinaus ermöglicht die Trennung eine optimierte Skalierung, da Lese- und Schreiboperationen unterschiedliche Anforderungen an die Systemressourcen stellen und daher unabhängig voneinander skaliert werden können. Wichtig ist hier anzumerken, dass dies bei vielen Geschäftsanwendungen nicht relevant ist, da die Anzahl der Nutzer und das Nutzerverhalten von vornherein bekannt sind.

Data-Oriented Programming

Data-Oriented Programming (DOP) ist ein Paradigma, das einen alternativen Ansatz zur traditionellen objektorientierten Programmierung (OOP) bietet, indem es den Fokus auf die Daten und ihre Strukturen legt, anstatt auf die Objekte und deren Verhalten. Die Schlüsselkonzepte von DOP sind:

1. **Unveränderlichkeit**
   Datenstrukturen sind unveränderlich, was bedeutet, dass sie nach ihrer Erstellung nicht mehr geändert werden können.
2. **Trennung von Identität und Zustand**
   In DOP wird die Identität eines Datenelements von seinem Zustand getrennt. Das bedeutet, dass der Zustand eines Objekts zu einem bestimmten Zeitpunkt einfach eine Momentaufnahme seiner Daten ist, Veränderungen möglich sind und die Historie über die Zeit hinweg leichter nachvollziehbar ist.
3. **Datenmodellierung als zentrales Designelement**
   Im Gegensatz zu OOP, wo das Verhalten und die Methoden von Objekten im Mittelpunkt stehen, konzentriert sich DOP auf die Gestaltung der Datenmodelle. Dies führt zu einer klaren Strukturierung der Daten und erleichtert die Datenmanipulation und -abfrage.

Brian Goetz diskutiert die Implementation von DOP in Java in seinem Artikel *[2]* und beschreibt, wie die Kombination der neuen Java-Features *Records*, *Sealed Classes* und *Pattern Matching* die DOP-Prinzipien unterstütze und zu präziseren, lesbareren und zuverlässigeren Programmen führe. Im Folgenden werden insbesondere die *Commands* aus CQRS gemäß der Ideen von Brian Goetz erklärt/aufgezeigt.

Moderne Java-Features

Java hat in den letzten Versionen mehrere wichtige neue Sprachmerkmale eingeführt, die die Art und Weise, wie Entwickler Code schreiben und strukturieren, wesentlich beeinflussen. Die für diesen Artikel wichtigen Neuerungen sind Records, *Sealed Classes* sowie *Pattern Matching* und werden im Folgenden beschrieben.

Records wurden in Java 16 als Preview-Feature eingeführt und sind seit Java 17 ein fester Bestandteil der Sprache. *Records* sind eine besondere Art von Klassen, die dazu dienen, einfache Datenstrukturen, sogenannte Daten-Träger (*data carriers*), mit minimalem Code zu modellieren. Ein *Record* erstellt automatisch alle Felder als final,

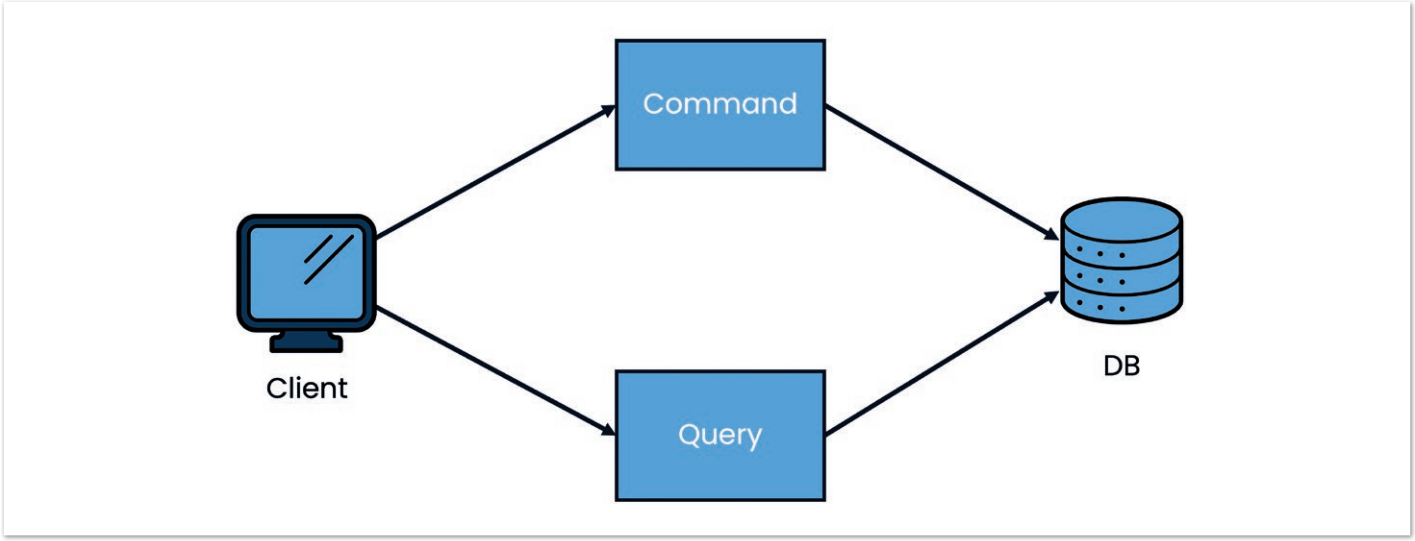


Abbildung 1: CQRS

generiert Getter (jedoch ohne `get`-Präfix) für diese Felder sowie passende Implementierungen von `equals()`, `hashCode()` und `toString()`. Records sind somit ideal für die Modellierung unveränderlicher Datenobjekte und reduzieren die Menge des zu schreibenden Codes erheblich.

Sealed Classes wurden offiziell in Java 17 eingeführt und bieten eine Möglichkeit, die Vererbung einzuschränken. Durch das Versiegeln einer Klasse oder eines Interfaces kann ein Entwickler explizit kontrollieren, welche anderen Klassen oder Interfaces von diesem Typ erben dürfen. Dies wird durch das Schlüsselwort `sealed` zusammen mit `permits` erreicht, die die genauen Typen angeben, die von der versiegelten Klasse erben dürfen. *Sealed Classes* fördern eine präzisere Kontrolle über die Vererbung und ermöglichen es Entwicklern, hierarchische Typsysteme präziser zu definieren und zu sichern, was besonders in der Domänenmodellierung nützlich ist.

Pattern Matching für den `instanceof`-Operator wurde in Java 16 als Preview-Feature eingeführt und seitdem weiterentwickelt. Es ermöglicht eine kompaktere und lesbarere Art, Typabfragen und anschließende Typumwandlungen durchzuführen. Mit *Pattern Matching* kann man in einem `if`- und nun auch einem `switch`-Statement oder einer -Expression nicht nur überprüfen, ob ein Objekt zu einem bestimmten Typ gehört, sondern dieses bei Übereinstimmung direkt in eine lokale Variable des entsprechenden Typs konvertieren. Dies vereinfacht den Code, indem es die Notwendigkeit entfernt, eine explizite Typumwandlung in einem separaten Schritt durchzuführen, und verringert somit die Fehleranfälligkeit bei der Handhabung von Typumwandlungen. In Java 21 hinzugekommen sind schließlich auch noch *Record Patterns*, die es erlauben, direkt auf einzelne Bestandteile eines *Records* zuzugreifen.

CQRS Commands mit modernem Java

Um die Integration von CQRS mit modernem Java und den obgenannten neuen Features zu veranschaulichen, fokussieren wir uns auf die *Commands*. Als Beispiel dient ein Webshop, der Standardfunktionen wie „Bestellung anlegen", „Artikel hinzufügen" und „Menge ändern" anbietet. Bei einem traditionellen Vorgehen würden dafür REST-Schnittstellen erzeugt werden, mit denen die Bestellung angelegt und verändert werden kann. *Listing 1* zeigt eine Implementation, wie ich sie oft in der Praxis antreffe. Grundsätzlich wird immer die ganze Bestellung übertragen, egal, welche Daten sich geändert haben. Auch ist das `PurchaseOrderDTO` eine 1:1-Kopie der `PuchaseOrder`-Entity und lässt sich somit mithilfe eines Mappers wie *ModelMapper* oder *MapStruct* einfachen mappen.

Dieses Design hat gleich mehrere Probleme. In der `put()`-Methode ist nicht ersichtlich, welche Daten durch die Schnittstelle geändert

```
@ResponseStatus(HttpStatus.CREATED)
@PostMapping
void post(@RequestBody PurchaseOrderDTO purchaseOrderDTO) {

  var purchaseOrder = modelMapper.map(purchaseOrderDTO,
                                      PurchaseOrder.class);

  customerRepository.findById(purchaseOrder.getCustomer().getId())
     .ifPresent(purchaseOrder::setCustomer);

  purchaseOrderRepository.save(purchaseOrder);
}

@PutMapping("{id}")
void put(@PathVariable Long id,
         @RequestBody PurchaseOrderDTO purchaseOrderDTO) {

  if (id.equals(purchaseOrderDTO.getId())) {
     throw new IllegalArgumentException();
  }

  var purchaseOrder = modelMapper.map(purchaseOrderDTO,
                                      PurchaseOrder.class);

  purchaseOrderRepository.save(purchaseOrder);
}
```

Listing 1: Traditionelle REST-API

```
sealed interface OrderCommand {

  record CreateOrder(long customerId) implements OrderCommand {}

  record AddOrderItem(long orderId, long productId, int quantity)
                                                  implements OrderCommand {}

  record UpdateQuantity(long orderItemId, int quantity)
                                        implements OrderCommand {}
}
```

Listing 2: Commands als Java Records

```
switch (orderCommand) {

  case OrderCommand.CreateOrder(long customerId) -> {
     var purchaseOrder = orderService.createOrder(customerId);
     return created(...).buildAndExpand(...).toUri()).build();
  }

  case OrderCommand.AddOrderItem(long orderId, long productId,
                                 int quantity) -> {
     var orderItemRecord = orderService.addItem(orderId, productId,
                                                quantity);
     return created(...).buildAndExpand(...).toUri()).build();
  }

  case OrderCommand.UpdateQuantity(long orderItemId,
                                 int quantity) -> {
     orderService.updateQuantity(orderItemId, quantity);
     return ok().build();
  }
}
```

Listing 3: Switch-Expression

werden. Zudem werden zu viele Daten übertragen, da immer die ganze Bestellung vom Client auf den Server geschickt wird, was in den meisten Fällen völlig unnötig ist. Auf Client-Seite ist unklar, welche Daten im Schnittstellenobjekt verändert werden dürfen.

CQRS hilft uns, diese Probleme zu lösen, indem wir anstelle von Objekten Befehle vom Client auf den Server schicken. Die Befehle lassen sich von den Anforderungen zu Beginn des Abschnitts ableiten: „Bestellung anlegen", „Artikel hinzufügen" und „Menge ändern". Das Fokussieren auf Befehle wirkt sich positiv auf das Verständnis der Applikation aus, da es dem Code Semantik hinzufügt.

Da Befehle unveränderlich sind, bietet es sich an, diese als *Java-Records* zu modellieren *(siehe Listing 2)*. Zur Gruppierung und nochmaligen Verbesserung der Verständlichkeit verwendet das Beispiel ein *sealed Interface*, das von allen *Commands* implementiert wird. Dank des *sealed Interface* können wir zur Implementierung der Behandlung der *Commands* auf die *Exhaustiveness* der `switch`-Expression zurückgreifen. *Exhaustiveness* (Vollständigkeit) bedeutet, dass der Compiler prüft, ob sämtliche Werte behandelt wurden. Das ist erstens ein großer Vorteil im Vergleich zu einem `if-/else-if-/else`-Konstrukt und zweitens wird man während der Weiterentwicklung beim Hinzufügen eines neuen *Command* darauf hingewiesen, wenn dieser nicht verarbeitet wird.

In *Listing 3* sieht man neben der Verwendung der `switch`-Expression auch noch einen Anwendungsfall für Pattern *Matching* mit *Record Patterns*. Die Befehle `CreateOrder`, `AddOrderItem` und `UpdateQuantity` werden dekonstruiert und die einzelnen Felder werden direkt an den `OrderService` weitergereicht. Das hat in diesem Beispiel den Vorteil, dass der `OrderService` keine Kenntnisse über die Befehle hat und somit unabhängig bleibt. Der gesamte Quellcode der Beispiele ist auf GitHub *[3]* zu finden.

Fazit

Java hat sich stetig weiterentwickelt, um mit den Veränderungen in der Technologiewelt Schritt zu halten. In den letzten Versionen hat Java mehrere moderne Sprachfeatures eingeführt wie *Records*, *Pattern Matching* und *Sealed Classes*. Diese Erweiterungen verbessern nicht nur die Lesbarkeit und Schreibbarkeit des Codes, sondern ermöglichen auch funktionalere Programmieransätze und verbesserte Datenmodellierung, die Java-Entwicklern helfen, effizientere und ausdrucksstärkere Programme zu schreiben.

Der Artikel hat aufgezeigt, dass der Einsatz von CQRS mit der Trennung der Verantwortlichkeiten die Verständlichkeit und Wartbarkeit erleichtert und insbesondere auf der Befehlsseite bei der Umsetzung sehr stark von den neuen Java-Sprach-Features profitiert.

Quellen

[1] Greg Young (2010): CQRS Documents by Greg Young
https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

[2] Brian Goetz (2022): Data Oriented Programming in Java, InfoQ
https://www.infoq.com/articles/data-oriented-programming-java/

[3] *https://github.com/simasch/cqrs-meets-modern-java*

Simon Martinelli
72 Services
simon@martinelli.ch

Simon Martinelli ist Java-Champion, Vaadin-Champion und Oracle ACE Associate. Er teilt sein Wissen regelmäßig in Artikeln, spricht auf internationalen Konferenzen und schreibt seinen Blog: https://martinelli.ch. Sein aktuelles Interesse gilt der Effizienzsteigerung der Full-Stack-Entwicklung mit Java.

Er ist Inhaber der 72 Services GmbH und arbeitet seit drei Jahrzehnten als Softwarearchitekt, Entwickler, Berater und Trainer, insbesondere im Java-Enterprise-Umfeld. Nebenbei ist er seit 2007 Dozent an der Berner Fachhochschule BFH und der Fachhochschule Nordwestschweiz FHNW.

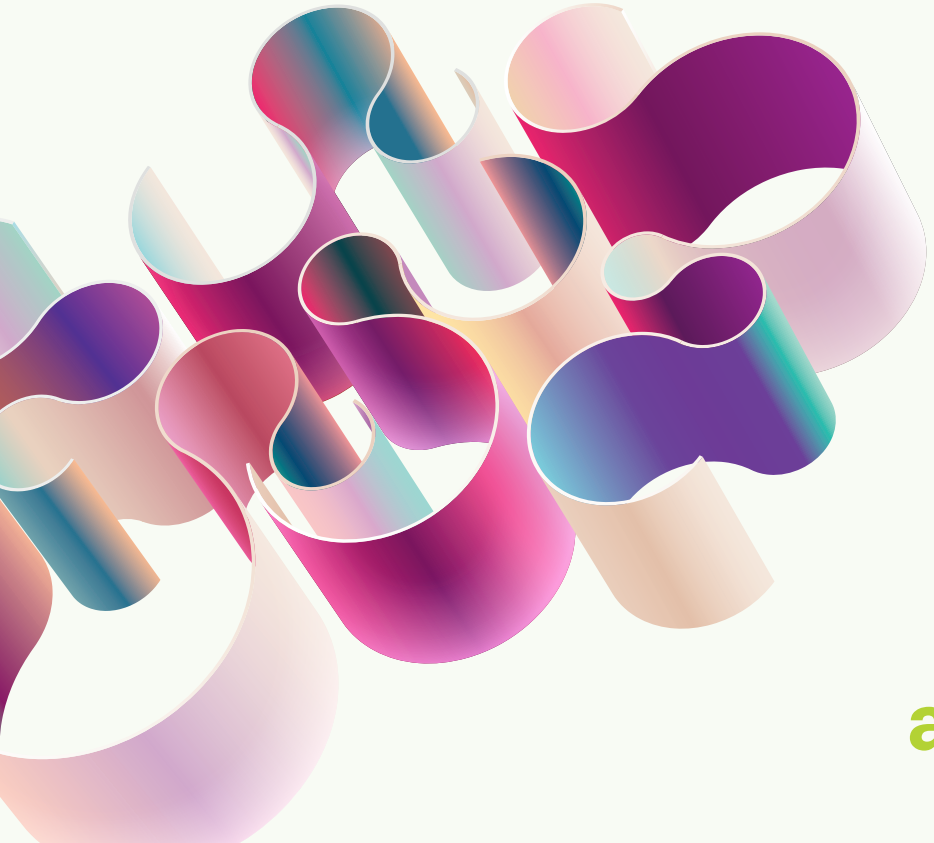
APEX connect

on demand

by DOAG

APEX CONNECT 24 VERPASST?

Jetzt On-demand-Ticket buchen und Vortragsaufzeichnungen anschauen!



apex.doag.org

Nachhaltige Enterprise-Architekturen

Prof. Dr.-Ing. Stefan Wagenpfeil, PFH Göttingen

Große Unternehmen, sogenannte Enterprises, müssen sich seit jeher besonderen Herausforderungen stellen, wenn es um die Entwicklung von unternehmensweiter oder geschäftskritischer Software geht. Investitionssicherheit, die Integration der Entwicklung und Unternehmens- und Entscheidungsprozesse, die Berücksichtigung von regulatorischen sowie politischen Rahmenbedingungen und natürlich der große Bereich „Altsysteme" – all das sind Themen, die insbesondere in Enterprises gravierende Auswirkungen haben können. In diesem Artikel werfen wir einen Blick auf die wichtigsten Themen und mögliche Lösungsansätze.

Mit dem Begriff „Enterprise" bezeichnet man gemeinhin Unternehmen, die eine gewisse Größe oder Relevanz erreicht haben. Firmen, wie beispielsweise Apple, Amazon, Meta, aber auch SAP, Siemens oder die Hersteller der Automobilbranche, gehören schon allein aufgrund ihrer Umsätze, Mitarbeiter- und Kundenzahlen zu dieser Kategorie. Doch auch kleine Unternehmen mit wenig Mitarbeitern können in diese Kategorie fallen, wenn Sie wichtige Komponenten oder Lösungen für große Unternehmen zur Verfügung stellen oder über eine Vielzahl von Kunden verfügen. Viele der Enterprises sind Unternehmen, die schon lange am Markt sind und bereits mehrere Innovationszyklen hinter sich haben, das bedeutet, die IT-Systemlandschaft ist gewachsen, es gibt eine Reihe von Altsystemen und etablierten IT-Prozessen. Benutzer, Rollen, Rechte, Infrastrukturen, Standards, juristische Rahmenbedingungen, Wartung und Betrieb müssen berücksichtigt werden. Entscheidungen werden typischerweise langfristig ausgelegt und wollen gut überlegt sein. Projekte in diesem Umfeld sind teuer, aufwändig und zeichnen sich durch große Optimierungspotenziale dank der Einführung neuer Lösungen aus. Der rasante technologische Fortschritt und die Abhängigkeit von Release-Zyklen der genutzten Standard-Software zwingen Unternehmen jedoch immer häufiger zu kurzfristigen Änderungen, die nunmehr in die langfristigen Strategien integriert werden müssen.

Um nachhaltige Enterprise-Architekturen zu etablieren, müssen daher eine Reihe von Aspekten übereinander gebracht werden, die bereits jeweils für sich gesehen Herausforderungen für Unternehmen jeglicher Größe darstellen. Im Enterprise-Kontext entsteht daraus schnell ein schier unlösbares Problem. Damit wir uns diesen Herausforderungen stellen können und durch gezielte Maßnahmen das „unlösbare" in ein „beherrschbares" Problem überführen, werden nun im Folgenden zunächst die wichtigsten Teilbereiche beleuchtet und am Ende einige konkrete Handlungsempfehlungen beschrieben.

Invididualentwicklung, Standardprodukte und Plattformen

In der Branche hat sich mittlerweile der Trend zum Einsatz von Standardprodukten etabliert. Man hat grundsätzlich erkannt, dass es besser ist, das „Rad" einzukaufen, als es selbst zu erfinden. Allerdings zeichnen sich Enterprises durch ein hohes Maß an Individualität aus, sodass Standardprodukte typischerweise durch Individualentwicklung adaptiert oder ergänzt werden. Ob die Entwicklung dann innerhalb des Standardproduktes oder als eigenständige Software stattfindet, hängt von den Möglichkeiten des Produkts ab. Immer mehr Hersteller bieten daher Plattformen an, auf deren Basis dann solche Anpassungen einfacher vorgenommen werden können, ohne den Kontext zu verlassen. In den seltensten Fällen wird heute noch Software „from scratch" entwickelt. Da sich hinter derartigen Plattformen jedoch auch ein gut funktionierendes Geschäftsmodell verbirgt, sind Enterprises permanent auf Expertise, Ausbildung und operative Unterstützung bei der Entwicklung/Verwendung dieser Plattformen angewiesen.

Prozessbetrachtung, Daten und Transaktionen

Im Enterprise-Umfeld spielt die Betrachtung der systemübergreifenden Prozesse eine entscheidende Rolle. Kaum ein System kann hier losgelöst von anderen Systemen funktionieren, erst im Verbund entstehen die geplanten Anwendungen. Auch wenn beispielsweise große Systeme im Bereich ERP, CRM, MES oder PDM eingesetzt werden, so müssen bei Enterprises permanent Daten aus einem System an das nächste System weitergegeben werden. Es entstehen Schnittstellen, komplexe Kommunikationsflüsse und Systemabhängigkeiten, ohne die ein Enterprise nicht funktionieren kann.

Die Definition der „führenden Systeme" spielt hierbei eine entscheidende Rolle: zu welchem Zeitpunkt besitzt welches System die „Quelle der Wahrheit" in Bezug auf einen Datensatz? Beispielsweise könnte ein übergreifender Prozess so organisiert werden, dass für die Akquise das CRM-System führend in Bezug auf Kontaktdaten ist. Sobald ein Kontakt jedoch zum Kunden wird, könnte das ERP-System die Führung übernehmen. Änderungen am Kunden (zum Beispiel die Rechnungsadresse) würden dadurch nicht mehr im CRM-System, sondern im ERP-System vorgenommen und via Schnittstelle zurück ins CRM transportiert werden. Das führende System würde abhängig vom Prozessschritt gewechselt. Wenn mehrere Systeme dieselben Business-Objekte verwalten (beispielsweise, wenn ein Kontakt im CRM einem Kunden im ERP-System entspricht), müssen die Beziehungen zwischen den Systemen verwaltet werden.

Begonnen bei einem einfachen Mapping der IDs der jeweiligen Systeme, bis hin zur Einführung von Masterdata-Management-Systemen, führt dies zu einer zusätzlichen Ebene, die über den Prozess gelegt wird. Klare Datenflüsse und eindeutige Beziehungen zwischen den systemübergreifenden Daten stellen schließlich auch die Basis für transaktionelle Prozesse dar.

Transaktionen sind seit jeher das beste Mittel, um Daten und Prozesse konsistent zu halten. Im Enterprise-Umfeld haben wir es häufig mit verteilten Transaktionen zu tun, die über mehrere Systeme hinweg konsistente, atomare Prozesse abbilden. Das Aufsetzen verteilter Transaktionen (zum Beispiel durch die Einführung von Jakarta-EE-Systemen unter Nutzung der JTA) erfordert hierbei gute Kenntnis von Prozessen und Programmiersprache. Eine der guten Eigenschaften der JEE ist es, die Komplexität durch einfache Annotationen zu kapseln. Ein Entwickler kann so durch die Annotation `@Transactional(TxType.REQUIRED)` definieren, dass eine

bestimmte Methode im Enterprise-Umfeld bei ihrem Aufruf den Applicationserver dazu bringt, eine solche verteilte Transaktion zu starten. Alles, was nun innerhalb der Methode an externen Aufrufen passiert, wird automatisch in diese Transaktion gesteckt – also jeder Datenbankzugriff, jeder RMI-Call und so weiter. Verteilte Transaktionen zeichnen sich dadurch aus, dass sie erst dann terminieren, wenn alle(!) beteiligten Systeme die Anfragen erfolgreich verarbeiten konnten. So lange wartet dann auch die aufgerufene Methode.

Im Fehlerfall werden alle beteiligten Systeme automatisch auf den Zustand vor der Transaktion gesetzt, der Entwickler bekommt dies durch eine Fehlermeldung signalisiert. So weit, so gut. Problematisch wird es allerdings, wenn Entwickler versehentlich oder aus Unwissenheit per Copy & Paste solch eine Annotation an die falsche Stelle platzieren. Das kann sehr schnell dazu führen, dass Enterprise-Anwendungen sehr langsam werden. Gerade die vielfältigen Möglichkeiten für die Anbindung von Systemen *(siehe Abbildung 1)* stellen für die Konzeption solcher Systeme eine große Herausforderung dar.

Release-Zyklen

Die eingesetzten Software-Produkte unterliegen ihrerseits Weiterentwicklungs-Zyklen, die in Form von Releases veröffentlicht werden. Durch den Einsatz von Cloud-Services und dem nach wie vor großen Trend, Standard-Software zu verwenden, werden solche Releases häufig von den Herstellern vorgegeben. Auch Java und die JEE veröffentlichen mittlerweile mindestens jährlich ein neues Release. Dasselbe gilt für Microsoft, SAP, Oracle, IBM und nahezu alle namhaften Hersteller von Standard-Software.

Isoliert betrachtet, ist das sowohl für Anwender als auch Unternehmen von Vorteil. Man erhält regelmäßig neue Features, ist auf dem aktuellen Stand, Bugfixes und Security-Anforderungen werden regelmäßig ausgerollt. Im Enterprise-Umfeld ist dies jedoch ein Problem. Jeder Release-Wechsel kann sich zwangsläufig auf die Systemlandschaft und die systemübergreifenden Prozesse auswirken. Potenzielle Änderungen an Prozessen, Daten, Schnittstellen müssen erkannt, getestet und gegebenenfalls durch Anpassung der beteiligten Systeme integriert werden. Ein Release-Wechsel im CRM „zwingt" also auch das ERP-Team zu einem erneuten Test und gegebenenfalls zu Anpassungen der Schnittstellen. Diese Aufwände fallen permanent an und müssen in der Budgetplanung entsprechend berücksichtigt werden. Insbesondere problematisch ist hierbei, dass für bestimmte Tests immer Kollegen aus den Fachbereichen involviert werden müssen, die ihre Zeit somit in Regressionstests investieren, ohne neue Funktionalität zu bekommen, was in Enterprises regelmäßig zu Unverständnis und Unmut führt.

Hinzu kommt, dass es pro Jahr nicht nur einen einzigen Release-Wechsel gibt. In einer typischen Enterprise-Landschaft mit zirka 100 Kernsystemen, die alle ihren eigenen Release-Zyklus haben, mit parallellaufenden Eigenentwicklungen oder Altsystemen, ist allein die Koordination solcher Release-Wechsel zwischen den Herstellern und dem Enterprise ein Vollzeit-Job.

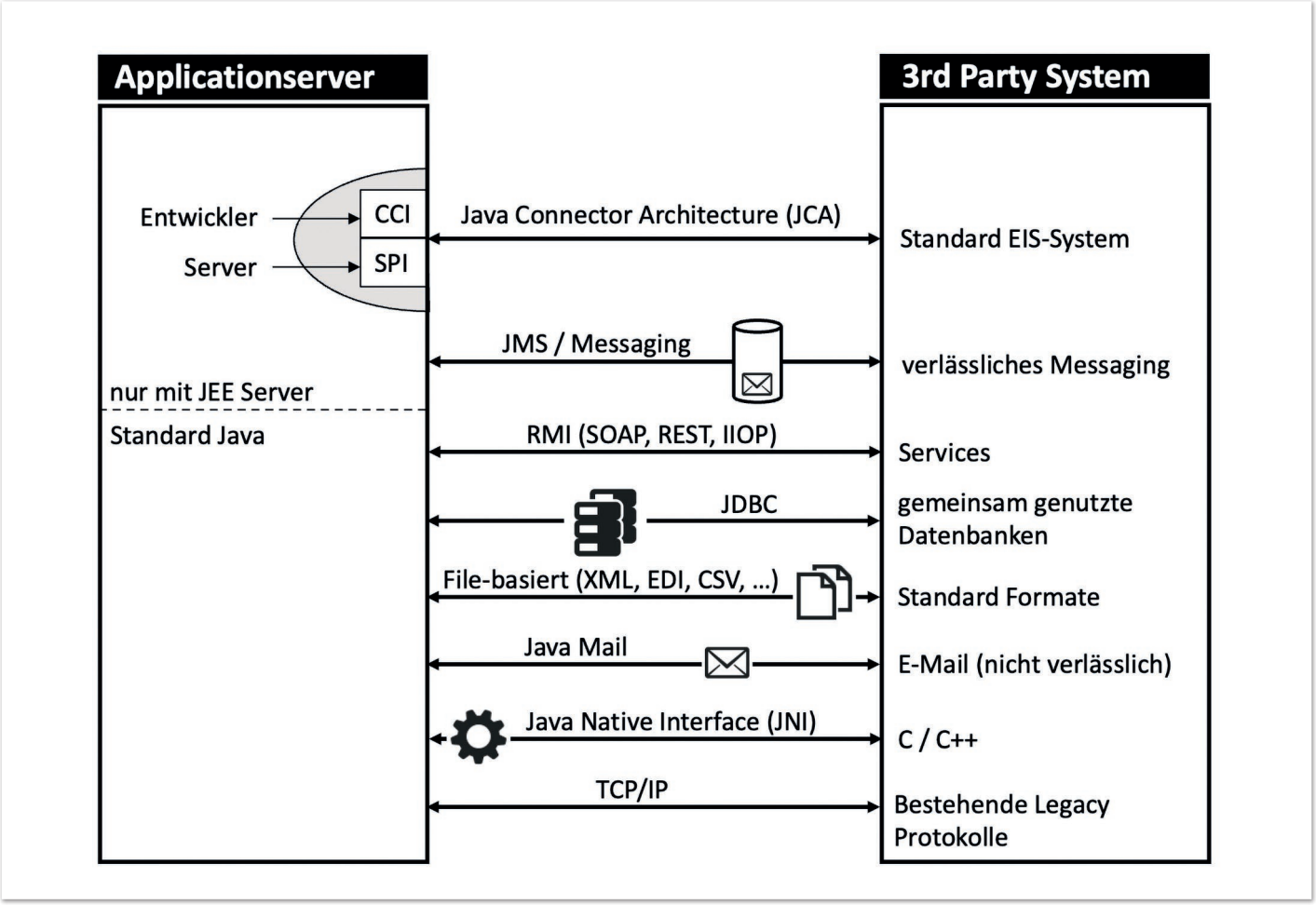


Abbildung 1 - Gängige Schnittstellenvarianten mit Java und der JEE

Technologischer Fortschritt und Standardisierung

Die Geschäftsmodelle von Enterprises wecken seit jeher Begehrlichkeiten. Enterprises haben bewiesen, dass bestimmte Dienstleistungen oder Produkte funktionieren, Märkte erschlossen und Kunden gewonnen werden können. Häufig werden Enterprises dabei ein wenig träge.

Startups nutzen diese Trägheit aus, indem sie disruptive Geschäftsmodelle auf Basis modernster Technologien entwickeln, diese in die etablierten Märkte bringen und schneller und schlagkräftiger agieren können als die bestehenden Enterprises. Ein hervorragendes Beispiel hierfür ist die Elektromobilität, bei der ein ehemals kleines Startup aus Kalifornien, Tesla, es geschafft hat, eine ganze Branche aus etablierten Automobilherstellern zunächst ins Wanken zu bringen, und dann zu überholen. Nunmehr ist Tesla selbst ein großes Enterprise geworden, das versucht, der drohenden Trägheit zu entgehen.

Unternehmen müssen sich immer wieder neu erfinden und den technologischen Fortschritt schnell in ihre Prozesse und Produkte integrieren. Häufig folgt irgendwann später eine Welle der Standardisierung, die dann ebenfalls integriert werden muss und in deren Folge bestehende Prozesse oder Formate nochmals adaptiert werden müssen.

Wartung und Betrieb, Staging

Enterprises beschaffen mittlerweile immer mehr Dienstleistungen und Produkte „as-a-Service". Die bereits genannten Abstimmungen bezüglich der Release-Wechsel müssen dennoch intern koordiniert werden. Dasselbe gilt für die vorgenommenen Anpassungen und Individualentwicklungen. Hierfür muss – insbesondere im Enterprise-Umfeld – eine geeignete Infrastruktur geschaffen werden. Zahlreiche Entwicklungs- Test-, Quality-Systeme (sogenannte *Stages*) werden bereitgestellt, Software wird ebenfalls passend zu den Releases oder in Wartungszyklen entwickelt.

Die erst kürzlich entstandene DevOps-Sparte adressiert genau dieses Problem, dass Entwicklungen zur richtigen Zeit auf die richtige Systemlandschaft mit den richtigen Abhängigkeiten und Konfigurationen kommen. Enterprise-Entwicklung ist schon allein dadurch immer deutlich aufwändiger. Hinzu kommt, dass Enterprise-Anwendungen typischerweise lange in Betrieb sind und dadurch immer wieder gewartet und überarbeitet werden müssen. Sogenannte Refactoring-Phasen werden eingeplant, um Software immer wieder aufzuräumen und auf dem aktuellen Stand zu halten.

Qualifikation

Aus Sicht eines Softwareentwicklers geht es im Enterprise-Umfeld längst nicht mehr darum, Java- oder Jakarta-EE-Anwendungen handwerklich sauber umzusetzen. Vielmehr müssen Enterprise-Software-Entwickler sich in Plattformen, Architekturen, Prozessen, Organisationsabläufen und einer durchaus herausfordernden Tool-Landschaft sicher bewegen können. Während in klassischen Entwicklungsprojekten der Anteil des „Codings" bei gut 70 % liegen kann, ist man im Enterprise-Umfeld häufig bei weniger als 30 % angekommen. Die restlichen Aufgaben eines Software-Entwicklers beziehen sich auf Kommunikation, Dokumentation, Test-Unterstützung, Dev-Ops, Testen von Plattform-Releases auf Kompatibilität und vieles mehr.

Hinzu kommt die Verzahnung mit anderen Teams, da Enterprise-Anwendungen selten allein losgelöst entwickelt werden können. Es gibt Projekt-Manager, die all das koordinieren müssen, Change-Manager, die die Kunden und Mitarbeiter abholen und unterstützen, und viele weitere Arten von Managern, die dafür sorgen müssen, dass Enterprise-Projekte sauber und termingerecht geliefert werden können.

Ab einer gewissen Unternehmensgröße müssen solche Projekte dann zusätzlich durch Qualitäts-Audits von Wirtschaftsprüfern begleitet werden. Daher sind Enterprise-Projekte fast immer teuer und erfordern viele besondere Fähigkeiten aller Beteiligten. Auch Entwickler müssen sich mit den angrenzenden Rahmenbedingungen auseinandersetzen und brauchen eine solide Ausbildung in den Bereichen Prozesswissen, Organisation, Security und Recht. Insbesondere in Enterprises müssen beispielsweise die Lizenzbedingungen von verwendeten Bibliotheken geprüft und validiert werden. Häufig scheiden Bibliotheken, die Entwickler beispielsweise auf *Stack Overflow* finden, aus Compliance-Gründen schon aus. Noch häufiger allerdings wird das nicht überprüft und kann im Nachhinein zu massiven Problemen nicht-technischer Art führen.

Vorgehensmodelle

Als letztes Beispiel für Rahmenbedingungen im Enterprise-Umfeld sei noch das generelle Projektvorgehen angeführt. In Enterprises gibt es typischerweise Entscheidungsstrukturen, Budgets, Roadmaps, Programm-Management und Unternehmens-/IT-Strategien. Diese eher langfristig ausgelegten Planungen stehen häufig im Widerspruch zu typischen Vorgehensmodellen aus der Software-Entwicklung, wie beispielsweise der agilen Entwicklung. Denn dem Management wäre es am liebsten, eine klare Planung mit Zielen und Budgets für die nächsten 12 Monate durchzuführen, während Entwickler und Fachbereiche am liebsten in 3-Wochen-Sprints ohne einen garantierten 12-Monats-Scope und Jahres-Budget arbeiten würden.

Wie also schafft man es nun, all diese und noch weitere Themen übereinander zu bringen und trotzdem flexibel und effizient zu bleiben?

Ein Schlüssel hierzu sind Architektur-Leitplanken. Im Enterprise-Umfeld lässt sich beispielsweise agile Entwicklung nur dann umsetzen, wenn es einen klar vorgegebenen Rahmen für die Freiheitsgrade der Entwicklung gibt. Dies erlaubt im Regelfall auch eine genauere Schätzung der Aufwände und ein besseres Scoping. Auch wenn es häufig aufwändiger ist, Standards zu implementieren, so zeichnen sich diese im Normalfall durch Langlebigkeit aus. Wenn es einen Standard gibt, sollte dieser auch eingeführt werden, anstatt auf „einfachere", aktuelle Libraries zu setzen, die im Zweifel in einem oder zwei Jahren nicht mehr existieren. Die Wahl der Software-Komponenten (und die damit zusammenhängenden Lizenzbedingungen) kann entscheidend für die Nachhaltigkeit von Lösungen sein. Eine Strategie sowohl für die Kommunikation zwischen den Entwicklungsteams als auch bezüglich der Wiederverwendbarkeit von Komponenten führt zum zielgerichteten Einsatz von Entwicklungsaufwänden. Es macht beispielsweise wenig Sinn, Aufwand in die Wiederverwendbarkeit zu stecken, wenn man anschließend niemandem von den entwickelten Komponenten erzählt und diese nicht zentral nutzbar zur Verfügung stellt.

In Bezug auf die Vorgehensmodelle sollten Enterprises versuchen, kleinen Teams möglichst viel Freiheitsgrad einzuräumen und nicht – wie häufig aus der Management-Perspektive gewünscht – alle Teilprojekte einheitlich aufsetzen. Diese Flexibilität zusammen mit guten Leitplanken hilft, die Themen zusammenzuhalten und dennoch gleichzeitig offen für Innovation zu bleiben.

Gerade in der aktuellen Zeit, in der Technologien wie KI, augmented oder virtual Reality in nahezu jedes IT-System integriert werden, ist es für Enterprises eine enorme Herausforderung, an der technologischen Spitze zu bleiben und gleichzeitig die Stabilität des Systemverbunds aufrecht zu erhalten. Wem das gelingt, der hat einen unschätzbaren Wettbewerbsvorteil.

Stefan Wagenpfeil
PFH Göttingen
s.wagenpfeil@pfh.de

Stefan Wagenpfeil ist Professor für Software-Engineering und IT-Management and der PFH Göttingen. Seit 1998 arbeitet er als Java- und JEE-Trainer und freiberuflicher Software- und Enterprise-Architekt für große Industrie-Unternehmen. Insbesondere die Themen Nachhaltigkeit von Softwareentwicklungen und die langfristige Sicherung von Investitionen stellen einen Schwerpunkt seiner Arbeit dar. Als Autor hat er beispielsweise 2023 beim Springer Verlag das Buch „Moderne Software-Entwicklung mit Java und JEE" veröffentlicht.

Über die wichtigsten Fähigkeiten in der Softwareentwicklung: Intellectual Playfulness, Resilienz, Kommunikation und Verantwortungsbewusstsein

Eldar Sultanow, Capgemini

Unsere Welt verändert sich ständig und rasant. Deshalb sind bestimmte Fähigkeiten unverzichtbar geworden. Dieser Artikel hebt vier Schlüsselkompetenzen hervor, die in der modernen Arbeitswelt von zentraler Bedeutung sind: Intellectual Playfulness, Resilienz, Kommunikation und Verantwortungsbewusstsein. Geprägt von hohen Ansprüchen an Sinnhaftigkeit und Nachhaltigkeit in ihrer Arbeit, streben junge Menschen nach Tätigkeiten, die im Einklang mit ihren Werten stehen. Die Kritik, die Generation Z sei lediglich auf Weltrettung aus und zeige wenig Loyalität gegenüber Arbeitgebern, greift zu kurz und übersieht das Potenzial und Engagement dieser jungen Talente. Junge Nachwuchskräfte brillieren

mit tiefem Fachwissen und verfolgen ein Arbeitsverständnis, das über das Individuelle hinausgeht und das Gemeinsam in den Fokus rückt.

In den Vorstellungsgesprächen merke ich es immer wieder: Bewerber haben einen hohen Anspruch an ihre Arbeit und damit an ihren Arbeitgeber. Inhalte müssen mit den eigenen Werten im Einklang stehen; und diese beziehen sich zunehmend auf Nachhaltigkeit und das Engagement für eine bessere Welt.

Der Gastkommentar „Generation Z floppt in der Arbeitswelt" von Susanne Nickel im Handelsblatt bewertete im Sommer 2023 die Arbeitseinstellung der Jungen mit den Worten „Unter der Weltrettung machen sie es nicht. Denn die Wohlstandskinder der Generation Z fühlen sich zu Höherem geboren […] Loyalität gegenüber dem Arbeitgeber? Langfristige Perspektive? Dankbarkeit, dass das Unternehmen in mich investiert? Gilt alles nicht mehr für die Generation Z, wie auch die Generationenauswertung des Job-Netzwerks Xing ergeben hat." *[1]*

Am Trend zu Sinn und Nachhaltigkeit ist etwas dran, allerdings in allen Altersgruppen – denn schließlich haben wir unsere Welt zu retten. Allerdings lässt sich keine Generation über einen Kamm scheren. Und es gibt sehr wohl sehr viele fleißige Nachwuchskräfte, die mit fachlich tiefem Wissen brillieren und deren Arbeitsverständnis nicht beim „Ich", sondern beim „Wir" ansetzt.

Die Welt wird unvorhersehbarer

Das Stichwort heißt VUCA – ein Akronym für *volatility* (Volatilität), *uncertainty* (Ungewissheit), *complexity* (Komplexität) und *ambiguity* (Ambiguität). Das heißt, die Welt ist im Wandel, der Wandel ist die Ordnung.

Daher ist Resilienz eine unerlässliche Fähigkeit, sprich die Fähigkeit, mit Veränderungen umzugehen und sich schnell von Rückschlägen zu erholen. Damit einhergehend werden Anpassungsfähigkeit und Flexibilität zu Schlüsselkompetenzen, sowohl auf individueller als auch auf organisatorischer Ebene. Ein resilienter Mensch lernt aus Fehlern und passt sich an die ständig wechselnden Bedingungen an.

Was bedeutet das für Organisationen? Diese müssen ein Umfeld schaffen, das nicht nur das Wachstum und die Entwicklung dieser Fähigkeiten fördert, sondern auch eine Kultur der Unterstützung und des Mitgefühls pflegt. Mitarbeitende müssen offen über Unsicherheiten und Ängste sprechen können, ohne Angst vor negativen Konsequenzen zu haben. Genau das fördert eine Atmosphäre, in der Innovation und Kreativität gedeihen. Genau das ermutigt, Risiken einzugehen und neue Ideen auszuprobieren.

Führungskräfte brauchen eine starke emotionale Intelligenz, um die Emotionen ihrer Teams zu verstehen und darauf einzugehen, ein Gefühl der Sicherheit und Stabilität zu vermitteln, auch wenn sich die äußeren Umstände schnell ändern. Schließlich sind die Zeiten der Unsicherheit ein zur Normalität gewordener Dauerzustand.

Ständiger Wandel verlangt uns Menschen lebenslanges Lernen ab. Nicht nur Nachwuchskräfte, sondern jeder Mensch jeden Alters muss bereit sein, sich fortlaufend weiterzubilden und sich neuen Herausforderungen zu stellen. Das müssen Organisationen unterstützen und Lernmöglichkeiten sowie Ressourcen bereitstellen, mit denen jede und jeder die eigenen Fähigkeiten kontinuierlich erweitern und sich an ständig wandelnde Anforderungen anpassen kann.

Die Welt wird technischer und komplexer

Dadurch, dass die Welt immer technischer, digitaler und komplexer wird, verschieben sich die in der Zukunft benötigten Skills weiter in Richtung technischer Tiefe. Dies verlangt von den Nachwuchskräften nicht nur ein tiefes Verständnis für die Materie, sondern auch die Fähigkeit, interdisziplinär zu denken und zu handeln. Demnach spielt neben der Tiefe auch die Breite eine wichtige Rolle.

Es gibt eine umstrittene Theorie, und zwar die des sogenannten „Flynn-Effekt", die besagt, der IQ der Menschen steige ständig an. Ich stelle die provokante Frage: Wird infolgedessen KI im Kindergarten gelehrt und die Quanteninformatik zum Pflichtprogramm der ersten Grundschulklasse? Sicherlich (noch) nicht.

Dafür gewinnt technische Verspieltheit immer mehr an Bedeutung. Sie umfasst die Neugier und den Ehrgeiz, sich mit neuen Technologien und Systemen auseinanderzusetzen, ihre Funktionsweise zu verstehen und innovative Anwendungen zu finden. Diese spielerische Herangehensweise an Technologie ermutigt zum Experimentieren und Lernen durch Versuch und Irrtum. Und auf diese Weise entwickeln wir kreative Lösungen.

Diesen dem Menschen von Natur aus mitgegebenen Drang nach dem Probieren, die Neugier und der ungeduldige Forschergeist, dies nennt Prof. Yasmin Weiß von der TH Nürnberg „Intellectual Playfulness" *[2]*. Es ist eine Fähigkeit, die jeder Mensch als Kind besessen hat. Es ist eine Fähigkeit, die wir im Erwachsenenalter oftmals abbauen oder gar verlernen. Es ist die Freude, sich auf Unbekanntes zu stürzen, sich in unbekannte Abenteuer zu stürzen, sich den lieben langen Tag Fragen zu stellen und nicht aufzugeben, bis sie endlich beantwortet sind. Es ist der Drang, die Welt erobern zu wollen. Und das Wichtigste: Diese intellektuelle Verspieltheit kennt keine Altersgrenze und ist erlernbar *[2]*. In einer sich ständig und immer schneller ändernden Welt kommen wir Menschen nicht drumherum, ein Leben lang zu lernen.

Eric Hoffer bringt es folgendermaßen auf den Punkt: „In a world of change, the learners shall inherit the earth, while the learned shall find themselves perfectly suited for a world that no longer exists."

Die Welt wird sozialer und kommunikativer

Technologie, Digitalisierung und auch die enorme Entwicklung von KI machen bestimmte menschliche Arbeit überflüssig. Und zwar die, die monoton und repetitiv ist. Mit KI setzen wir heute fort, was die Industrialisierung vor gut 200 Jahren begonnen hat. Es geht aktuell um eine Verschiebung von unkreativer nach kreativer, von isolierter nach kommunikativer Arbeit. Das erzeugt Veränderungsdruck und setzt Möglichkeiten frei, die bislang unter den Routinen verborgen waren.

Warum ist Kommunikation eine so wichtige Fähigkeit im Software-Engineering? Softwareentwicklung ist keine isolierte Tätigkeit, sondern ein kollaborativer Prozess, der Menschen mit unterschiedlichen Fachkenntnissen und Hintergründen zusammenbringt. Kom-

munikation dient nicht nur der Weitergabe von Informationen, sondern dem Aufbau von Verständnis und dazu, gemeinsam Lösungen zu finden. Wir verdanken es einer effektiven Kommunikation, dass Entwicklerteams Anforderungen klar definieren, Missverständnisse minimieren, Ziele abstimmen und letztendlich qualitativ hochwertige Softwareprodukte liefern.
Entwickler und Architekten müssen in der Lage sein, komplexe technische Konzepte allgemeinverständlich zu erklären, zum Beispiel genau dann, wenn sie mit Stakeholdern wie Projektmanagern oder Kunden sprechen.

Darüber hinaus spielt Kommunikation eine wichtige Rolle beim Wissensaustausch und beim Lernen innerhalb des Teams. Eine Kultur, die offene Diskussionen, Fragen und Feedback fördert, fördert damit auch das Wachstum eines jeden Teammitglieds und die kontinuierliche Verbesserung des Teams als Ganzes.

Die soziale Natur der Softwareentwicklung erstreckt sich auch auf die Online-Community. Entwickler weltweit teilen ihr Wissen und ihre Erfahrungen über Plattformen wie GitHub, Stack Overflow und über verschiedene soziale Medien. So profitieren wir alle von dieser globalen Wissensbasis und tragen aktiv zur Community bei.

Letztendlich kann die Qualität der Kommunikation innerhalb eines Softwareentwicklungsteams den Unterschied zwischen Erfolg und Scheitern eines Projekts ausmachen. Gute Kommunikationsfähigkeiten führen zu besserer Zusammenarbeit, höherer Produktivität, effektiverem Problemmanagement und einer stärkeren Teamkultur.

Systeme in unserer Welt werden kritischer

Denken wir an autonomes Fahren, an medizinische Diagnosesysteme oder Energiemanagement. Ja, – die Systeme in unserer Welt werden komplexer, anspruchsvoller und wichtiger. Diese kritischen Systeme beeinflussen das tägliche Leben von Millionen von Menschen und bringen sowohl enormen Nutzen als auch signifikante Risiken mit sich.

Verantwortungsbewusstsein bedeutet, die potenziellen Auswirkungen der Technologie auf die Gesellschaft, die Umwelt und die individuelle Sicherheit zu verstehen und zu berücksichtigen. Es geht darum, sorgfältige Risikoanalysen durchzuführen, robuste Sicherheitsprotokolle zu implementieren und stets ethische Überlegungen in den Entwicklungs- und Entscheidungsprozess einzubeziehen.

Die Entwicklung kritischer Systeme verlangt allen Beteiligten ein tiefes Verständnis der Technologien ab, eine fortlaufende Bewertung der Sicherheits- und Ethikstandards sowie die Bereitschaft, transparent und rechenschaftspflichtig zu agieren. Wir brauchen eine enge Zusammenarbeit zwischen Entwicklern, Regulierungsbehörden, Industrieexperten und der Öffentlichkeit, um kritische Systeme verantwortungsvoll zu gestalten und einzusetzen. Dieser interdisziplinäre Ansatz stellt sicher, dass Systeme nicht nur technisch fortgeschritten, sondern auch sozial verträglich und ethisch vertretbar sind.

Verantwortungsbewusstsein ist nicht nur eine individuelle, sondern auch eine kollektive Fähigkeit. Alle Beteiligten – von den Ingenieuren und Entwicklern bis hin zu den Endnutzern und politischen Entscheidungsträgern – müssen sich der Bedeutung und der Auswirkungen dieser Systeme bewusst sein und eine sichere, ethische und nachhaltige Zukunft sicherstellen.

Fazit

Intellectual Playfulness, Resilienz, Kommunikation und Verantwortungsbewusstsein sind Fähigkeiten, mit denen (nicht nur) Nachwuchskräfte die Herausforderungen in einer immer unvorhersehbareren und technisch komplexer werdenden Welt meistern. Wenn Organisationen ein Umfeld schaffen, das die Ausprägung dieser Fähigkeiten und den Drang nach lebenslangem Lernen sowie den Aufbau von Netzwerken fördert, dann können Individuen nicht nur klarkommen, sondern auch in dieser sich ständig verändernden Umgebung wachsen.

Die Diskrepanz zwischen der Wahrnehmung der Generation Z in Teilen des öffentlichen Diskurses und in der Realität zeigt, wie sehr wir den Dialog zwischen den Generationen fördern und voneinander lernen müssen. Die Arbeitswelt ist im Wandel; dieser Wandel bietet Chancen wie auch Herausforderungen. Wir müssen uns von Stereotypen verabschieden und die Vielfalt der Talente und Perspektiven anerkennen, die jede Generation in die Arbeitswelt einbringt.

Quellen

[1] Handelsblatt (2023). Gastkommentar: Generation Z floppt in der Arbeitswelt (handelsblatt.com)

[2] Y. Weiß (2023). "Intellectual Playfulness": Die unterschätze Superkompetenz und wie wir sie stärken (linkedin.com)

Eldar Sultanow
Capgemini
eldar.sultanow@capgemini.com

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

DIE DOAG ANWENDERKONFERENZ

ANWENDERKONFERENZ.DOAG.ORG

Save the Date

19. BIS 22. NOVEMBER

IN NÜRNBERG

2024

DOAG

Konferenz + Ausstellung

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.

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

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

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

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

Bildnachweis:
Titel: Bild © shctz
https://stock.adobe.com
S. 10: Sloth McSloth
https://stock.adobe.com
S. 12 + 13: Bild © Zense
https://stock.adobe.com
S. 26 + 27: Bild © Sylverarts
https://stock.adobe.com
S. 30 + 31: Bild © freshidea
https://stock.adobe.com
S. 36 + 37: Bild © Designed by freepil
https://freepik.com
S. 44 + 45: Bild © Nuthawut
https://stock.adobe.com
S. 50 + 51: Bild © Designed by macrovector
https://freepik.com
S. 58 + 59: Bild © Designed by freepil
https://freepik.com
S. 70 + 71: Bild © Designed by macrovector
https://freepik.com
S. 76 + 77: Bild © ma
https://stock.adobe.com
S. 82 + 83: Bild © Flash concept
https://stock.adobe.com

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

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

Inserentenverzeichnis

European NetSuite User Days

November
2024
18 - 19

at NCC Ost
Nuremberg

netsuite.doag.org
#NetSuiteUserDays