Ausgabe 01/2022

Java aktuell

Java aktuell

iJUG Verbund
www.ijug.eu

Java 17
Das neue LTS-Release im Detail

Migration
Wege in die Cloud

Retroperspektive
Welches Format für welches Team?

Praxis. Wissen. Networking. Das Magazin für Entwickler

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

4 191978 304903 01

MIGRATION

Liebe Leser/innen der Java aktuell,

das neue Long Term Support Release Java 17 wurde im September veröffentlicht! Falk Sippach nimmt das Update in seinem Artikel unter die Lupe und geht mit uns alle Neuerungen im Detail durch. Im Java-Tagebuch und der Eclipse Corner am Anfang der Zeitschrift findet ihr zudem wieder relevante Neuigkeiten aus der Community.

Der Hauptfokus dieser Ausgabe der Java aktuell liegt auf dem Thema **Migration**. Dazu haben unsere Autoren aus der Community wieder informative Fachartikel für euch vorbereitet. Los geht's mit Bastian Engelking, der anhand eines Beispielsprojekts aus der Energiebranche den Einsatz und die Vorteile RPA-gestützter Datenmigration erläutert. Im Anschluss zeigen uns Susanne Apel und Christian Fritz anhand eines Praxisbeispiels, wie man eine "Anwendung mit Zustand" mithilfe von JSF und Apache Ignite in die Cloud migriert. Was genau der Döner mit einer Cloudanwendung zu tun hat, erklärt uns Thomas Michael ab Seite 34. Er nimmt uns mit auf die Reise einer manuellen Dönerbestellung für die Kolleginnen und Kollegen hin zu einer in die Cloud migrierten Serverless-Anwendung.

Gerhard Fuchs präsentiert eine Java-Spracherweiterung, mit der man zusammengesetzte vernetzte Systeme programmieren kann, um beispielsweise Spielzeugrobotern Leben einzuhauchen. In seinem Artikel ab Seite 47 stellt Bernd Müller das Projekt Panama näher vor, das es sich zum Ziel gesetzt hat, die Gesprächskultur von Java und C mithilfe von JEPs zu verbessern. Er gibt uns eine Einführung in das Foreign Function & Memory API.

Wie Java und die Evolution zusammenpassen, zeigen uns Andreas Gräßer und Andreas Lau im darauffolgenden Artikel. Sie haben ein Simulationsprogramm in Java erstellt, das ein Modell zur Evolution und Mutation vereinfacht nachbildet. Zum Abschluss dieser Ausgabe gibt es einen nicht-technischen Softskills-Artikel zum Thema Retroperspektive aus Unternehmenssicht von Dr. Dominic Lindner und Martin Weber. Darin stellen die beiden einige Formate näher vor, mit deren Hilfe agile Teams bestimmte Scrum-Sprints oder Projekte aus der Vergangenheit bewerten und etwas daraus lernen.

Wir wünschen euch viel Spaß beim Lesen!

Eure

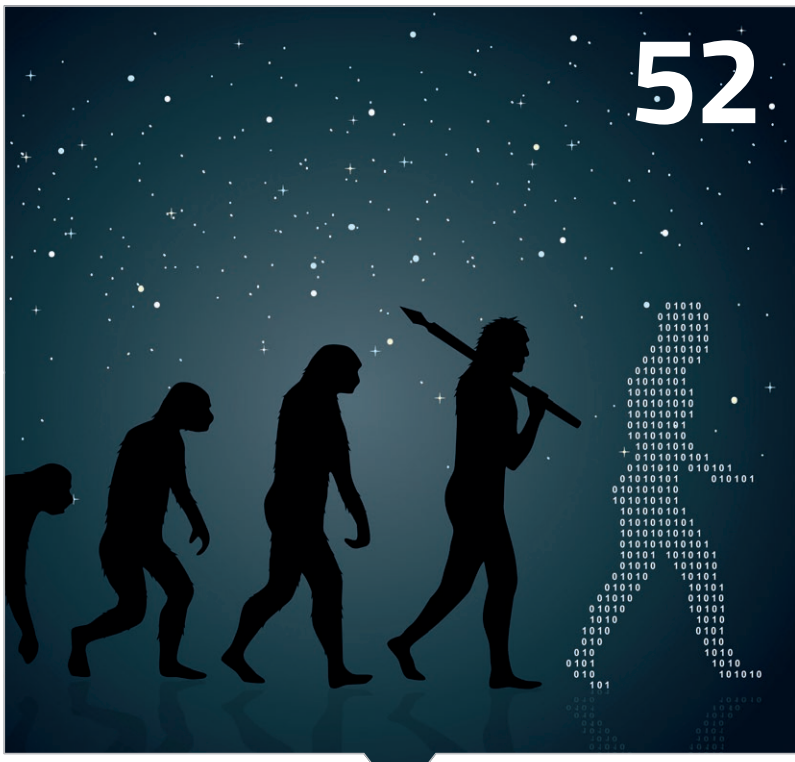
Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

Java 17 ist da!

Migration einer Anwendung mit Zustand in die Cloud

Zusammengesetzte vernetzte Systeme in Java programmieren



Evolutionssimulation in Java

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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

3. August 2021

Adoptium übernimmt (größtenteils)

Der Übergang von AdoptOpenJDK zu Eclipse Adoptium ist im Wesentlichen abgeschlossen, auch wenn die alte Website noch eine Weile online bleiben soll, inklusive der archivierten Builds. Der erste eigene Build, unter dem Namen des dafür zuständigen Sub-Projekts Temurin, ist inzwischen verfügbar. Auch Builds anderer Hersteller sollen bald, wie vorher im AdoptOpenJDK-Projekt, bereitgestellt werden.

Was unter dem Dach von Adoptium weiterhin nicht verfügbar sein wird, sind Builds auf Basis der GraalVM, und auch die OpenJ9 Builds werden nicht „umziehen". Beides ist Teil der Vereinbarungen zwischen der Eclipse Foundation und Oracle – im Gegenzug bekam Adoptium Zugang zu den Java-SE-Test-Compatibility-Kits. OpenJ9 Builds können jedoch direkt bei IBM heruntergeladen werden. Für diejenigen, die Binaries über das AdoptOpenJDK-API herunterladen, stellt Adoptium das gleiche API zur Verfügung. Um den Übergang zu erleichtern, wird die AdoptOpenJDK über das API aber für eine begrenzte Zeit auch Temurin und OpenJ9 Builds zur Verfügung stellen – wenn nach den neuesten Builds für „hotspot" beziehungsweise „openj9" Builds gesucht wird.

6. August 2021

MicroProfile 4.1 erschienen

Mit der neuen Version von MicroProfile werden Anforderungen und Prozesse für „kompatible Implementierungen" konkretisiert: Erfolgreiche Tests gegen die TCKs der Core APIs (Config, Fault Tolerance, Health, JWT Authentication, Metrics, OpenAPI, OpenTracing und Rest Client) sind für die Feststellung der Kompatibilität erforderlich. Optional ist eine kompatible Implementierung der von Java beziehungsweise Jakarta EE übernommenen APIs wie JAX-RS und CDI.

Einzige inhaltliche Änderung in 4.1 sind kleinere Verbesserungen des Health API – das auf Version 3.1 angehoben wurde. Außerhalb des „Umbrella" hat sich alauch etwas getan. Die mit 4.0 eingeführten, ebenfalls nur optional zu unterstützenden „Stand-alone Specifications" (Reactive Messaging, Reactive Streams Operators, Context Propagation und GraphQL) haben alle einen Versionssprung gemacht. Und neu dazugekommen ist LRA 1.0 – eine Spezifikation für Long-Running-Actions (SAGA-Patterns), unter Nutzung von JAX-RS.

1. September 2021

Microsoft macht GCToolkit Open Source

Die Java Engineering Group bei Microsoft hat ihr GCToolkit als Open Source freigegeben, mit dem komplexe Analysen der Java-Garbage-Collector-Logs durchgeführt werden können. Es besteht aus drei Java-Modulen: einem Vert.x-basierten Message-Backend, einer Reihe von Parsern und einem API.

Das Backend stellt über einen (ersten) Message Bus GC-Log-Zeilen als Streams zur Verfügung, die dann von Parsern in GC-Cycle oder Safe-Point-Events konvertiert werden – die wiederum über einen zweiten Bus an eine weitere Gruppe von Parsern verteilt werden. Hier der Blog-Eintrag dazu von Kirk Pepperdine (einem der Gründer des Londoner Java-Performance-Startups jClarity, das Microsoft nicht ohne Grund vor zwei Jahren gekauft hatte) *[1]*.

7. September 2021

Spring 2022

Auf der gerade abgelaufenen SpringOne-Konferenz wurden die nächsten Release-Termine angekündigt: Spring Boot 2.6 kommt im Herbst 2021 und 2.7 im ersten Halbjahr 2022 heraus. Die nächsten Major-Versionen, Spring Boot 3 und Spring 6, sollen dann gegen Ende 2022 folgen; für diese ist dann unter anderem Support für Java 17 und die APIs von Jakarta EE 9 anvisiert – aber auch das Bauen von Spring Apps als Native Images *[2]* soll dann endlich die Beta-Phase verlassen haben.

15. September 2021

Jakarta wird populärer, aber Spring ist weiter vorn

Die Eclipse Foundation hat die Ergebnisse des „2021 Jakarta EE Developer Survey" *[3]* veröffentlicht. Immerhin fast 1.000 Rückmeldungen waren eingegangen. Ein zentrales Resultat: Der hybride Ansatz liegt vor dem Monolithen – sprich, mehr Unternehmen fahren einen Mix aus Microservices und Monolithen. Der reine Microservices-Ansatz liegt jedoch weiterhin klar vorn.

Interessant ist, dass MicroProfile in der Gunst der Entwicklungsteams deutlich hinter Jakarta EE liegt. Vielleicht erklärt sich das durch Spring Boot – das klar vor Jakarta steht und in vielen Microservice-Projekten Verwendung findet. Sowohl Spring als auch Jakarta haben dabei deutlich zugelegt – stärker als die „Konkurrenz". Wobei MicroProfile und Jakarta sich ja eher als Partner sehen.

Ein weiteres interessantes Ergebnis: 48 % der Befragten nutzen bereits Jakarta EE oder wollen zumindest innerhalb der nächsten 24 Monate dorthin migrieren (was nicht heißt, dass sie zum Beispiel nicht mehr Spring nutzen – Mehrfachnennungen waren möglich).

16. September 2021

Java SE 17

Das erste Long Term Release (LTS) seit 2018 (damals Java 11) ist erschienen. Das zentrale neue Sprach-Feature sind „Sealed Classes" (JEP 409), also die Möglichkeit, über die neuen Keywords „sealed"

und „permits“ festzulegen, dass eine Klasse nur von einer definierten Liste anderer Klassen direkt erweitert werden darf. Diese können ihrerseits für ihre eigenen Sub-Klassen Einschränkungen definieren oder aufheben. Dafür wurde mit „not-sealed“ das erste reservierte Keyword mit Bindestrich eingeführt. Ausdrücklich keine Ziele von Proposal 409 sind die Einführung eines „friends“-Konstrukts, wie es aus anderen Sprachen bekannt ist, oder Änderungen an der Behandlung von „final“.

Weitere Neuerungen in Java 17 sind unter anderem „Pattern Matching for switch“ als Preview und die Fortsetzung der Arbeiten an der „strong encapsulation“ von internem JDK-Code; Letzteres mit dem Risiko, dass so manche Applikationen und Frameworks nicht mehr ohne Anpassungen funktionieren, zumal der einfache Ausweg über den Schalter „--illegal-access“ in einem zukünftigen Release wegfallen soll *[4]*.

Mit dem neuen Release hat „Java-Chef-Architekt“ Mark Reinhold auch gleichzeitig eine Änderung am Release-Zyklus angekündigt: Grundsätzlich sei der neue, sechs-monatige Release Train ein Erfolgsmodell und das schnelle Liefern neuer Features komme gut an. Viele Entwicklerinnen und Entwickler seien aber auch frustriert, weil sie nur Features aus LTS-Releases in Produktion bringen dürfen, sodass sie bis zu drei Jahre warten müssen. Aus diesem Grund soll es nun alle zwei Jahre ein LTS-Release geben – also jede vierte Version.

Oracle hat außerdem mitgeteilt, dass seine Versionen des JDK 17 und zukünftiger JDKs unter der neuen „No-Fee Terms and Conditions“ (NFTC) Lizenz kostenlos zur Verfügung stehen werden, und zwar bis ein Jahr nach dem jeweils nächsten LTS-Release. Das gilt auch für (internen!) produktiven Einsatz. Updates gibt es unter dieser Lizenz bis mindestens September 2024. Wer mehr will, muss weiterhin zahlen.

17. September 2021

Angriffe auf Open-Source-Software nehmen drastisch zu

Sonatype hat eine Analyse veröffentlicht, nach der der Einsatz von Open-Source-Software im Vergleich zum Vorjahr wieder deutlich zugelegt hat (gemessen an der Anzahl der Projekte und vor allem der Downloads – 71 % mehr für Java-Projekte), gleichzeitig jedoch auch die Anzahl der Angriffe massiv zugenommen hat; beispielsweise „Malicious Code Injection“ (bewusstes Einfügen von Schadcode in existierende Open-Source-Projekte) und „Dependency Confusion“ – bei dem ein Angreifer interne Komponenten-Namen in Unternehmen „rät“ und Schadcode unter diesen Namen in öffentlichen Repositories ablegt, wo sie bei sorglosem Umgang mit dem Thema Sicherheit dann von firmeninternen Build-Prozessen gefunden und anstelle der eigenen Komponenten verwendet werden.

Die Attacken sind laut Sonatype um 650 % gestiegen, nach schon 430 % Plus im Vorjahr. Keine gute Entwicklung – außer für die Software-Security-Branche.

20. September 2021

Ausblick auf Java 18

Was können wir von Java SE 18 erwarten? Noch sind es fast sechs Monate bis zum Release, aber ein paar Kandidaten stehen schon fest: UTF-8 (endlich) als Default Character Set über alle Standard-APIs hinweg, das Vector-API (wohl immer noch nicht produktiv, sondern als dritte Inkubator-Version), ein @snippet-Tag für Code-Schnipsel in JavaDoc und ein simpler, von der Kommandozeile startbarer Web-Server für statischen Content – zur Unterstützung von schnellem Prototyping, Tests und zu Unterrichtszwecken. Außerdem vermutlich Record und Array Patterns als Preview. Weitere Projekte – beispielsweise die zweite Inkubator-Version des „Foreign Function and Memory“-API – werden sicher folgen *[5]*.

22. September 2021

Kafka 3 ohne Java 8 (fast)

Apache Kafka 3.0.0 ist da. Was hat das mit Java zu tun? Nicht nur das OpenJDK-Projekt, Mark Reinhold persönlich und diverse Hersteller im Ökosystem arbeiten daran, die immer noch verbreitete Nutzung von Java SE 8 zu reduzieren. Auch Projekte wie Kafka gehen immer mehr voran. Mit Version 3.0.0 wird die Unterstützung von Java 8 als „deprecated“ ausgewiesen, spätestens mit Kafka 4.0 soll sie entfallen.

6. Oktober 2021

Quarkus 2.2 und 2.3

Quarkus 2.3 bringt eine Erweiterung der „Dev Services“ um Neo4J, Logging mit Panache, integrierte Testunterstützung für CLI-Applikationen, Unterstützung für MongoDB-Migrationen mit Liquibase sowie Hibernate Interceptors.

Die einen guten Monat vorher freigegebene Version 2.2 hatte sich hauptsächlich auf Bug Fixes und generelle Qualitätsverbesserung fokussiert – und war ironischerweise wegen eines Bugs, der den Dev-Services-Einsatz unter Windows verhinderte, erst als 2.2.1 freigegeben worden. Aber auch in 2.2 sind ein paar interessante Features, unter anderem hat auch Quarkus jetzt eine auf dem Narayana-Projekt basierende Implementierung der MicroProfile „Long Running Actions“-Spezifikation. Dafür, dass LRA erst vor ein paar Monaten angenommen wurde und nur eine optionale („Standalone“) MicroProfile-Spezifikation ist, wird sie schon recht breit unterstützt.

WildFly 25 läuft auf Java SE 17

WildFly läuft auch mit Version 25 weiterhin auf Java SE 8. Es wurde allerdings nicht nur für Kompatibilität mit Jakarta EE 8, sondern unter dem Titel „WildFly Preview“ auch mit den Test-Compatibility-Kits von Jakarta EE 9.1 getestet, zusammen mit Java SE 17. Auch wenn die Unterstützung von EE 9.1 entsprechend vorsichtig angekündigt wird, soll es keine Einschränkungen bei der Nutzung von Java 17 geben –

bis auf diese: Da WildFly an vielen Stellen über „deep reflection" auf Module des JDK zugreift und dies mit Java 17 eingeschränkt wird (vorher gab es nur Warnungen), sind gegebenenfalls Anpassungen an den JPMS-Einstellungen nötig – falls kein „bootable jar" oder die mitgelieferten Startskripte genutzt werden.

9. Oktober 2021

Neue Working-Group-Repräsentanten bei Eclipse

Im Steering Committee der MicroProfile Working Group hat es eine Neubesetzung bei den „Corporate" Mitgliedern gegeben. Dort repräsentiert nun Heiko Rupp den iJUG e.V. als Stellvertreter für Jan Westerkamp.

Jakarta EE kennt gleich drei Committees und vier Mitgliedsstufen („Strategic", „Enterprise", „Participant" und „Committer"). Dort wurden schon Ende letzten Monats die Repräsentanten der beiden unteren Stufen für die nächsten 12 Monate gewählt – beziehungsweise per Akklamation wiedergewählt, Gegenkandidaturen gab es keine. Im Marketing Committee ist dies weiterhin Jelastic („Participants"), vertreten durch Tetiana Fydorenchyk, und Otavio Santana vertritt sich selbst beziehungsweise die individuellen Committers. Im Steering Committee bleiben Martijn Verburg als Vertreter der London Java Community (LJC) sowie Arjan Tijms. Im Specification Committee sind es Mercelo Ancelmo (ebenfalls LJC) und Werner Keil.

19. Oktober 2021

Eclipse Open DI

Oracle hat ein neues Implementierungsprojekt für Jakarta EE eingereicht. Unter dem Namen „Eclipse Open DI" soll eine Implementierung der neuen CDI-Lite-Spezifikation auf Basis von Micronaut entstehen. Das Projekt ist bislang komplett intern bei Oracle vorangetrieben worden. Micronaut ist letztes Jahr von Object Computing mit einer Anschubfinanzierung von zwei Millionen US-Dollar in seine eigene, nicht gewinnorientierte Micronaut Foundation ausgelagert worden. Fast gleichzeitig ist der Mitgründer des Projekts, Graeme Rocher, zu Oracle gewechselt, um dort den Einsatz von Micronaut voranzutreiben (er bleibt aber im Board der Foundation). Und Oracle hat schon einiges für und mit Micronaut umgesetzt, unter anderem in seiner eigenen Cloud – obwohl „Big Red" ja auch noch das eigene Helidon-Projekt als direkte Alternative hat. Mal sehen, was daraus wird.

Referenzen

[1] *https://devblogs.microsoft.com/java/introducing-microsoft-gctoolkit*
[2] *https://github.com/spring-projects-experimental/spring-native*
[3] *https://outreach.jakartaee.org/2021-developer-survey-report*
[4] *https://openjdk.java.net/jeps/403*
[5] *https://openjdk.java.net/projects/jdk/18/*

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

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

Markus' eclipse-Corner

Ihr kennt mich als kritischen, bewusst provozierenden Kolumnisten. Und doch muss ich heute unumwunden zugeben: Ich bin begeistert – völlig uneingeschränkt! Also, nee, nicht von der eigenen Leistung, JAX-RS 3.1 immer noch nicht durch die Tür zu haben. Das ist eher peinlich. Wir – und hier möchte ich insbesondere unsere neuen Kontributoren nennen, die in einem Fall sogar nachweislich den Weg über diese Kolumne in das Projekt gefunden haben – arbeiten derzeit noch daran, das JAX-RS TCK auf moderne Beine zu stellen. Das wird auch noch ein wenig dauern und ist eine andere Geschichte. Nein, begeistert bin ich vielmehr von Eclipse Adoptium.

Nachdem ja lange Zeit die Homepage eine Dauerwarteschleife war, kann man dort seit Kurzem tatsächlich zum ersten Mal einen Installer downloaden, der JDK 17 unter dem Namen „Eclipse Temurin JDK 17" auf die Platte presst. Wow! Auch wenn das Signaturzertifikat offenbar „nur" von der LJC stammt – es ist ein Meilenstein für die gesamte Community! Ich habe Temurin® heute zu meinem Standard-JDK gemacht, lasse damit gerade meine IDE (ja, nach 20+ Jahren immer noch Eclipse) booten und werde ab sofort meine diversen Open-Source-Projekte nur noch mit Temurin builden! Beispielsweise jetzt in diesem Moment, während ich an der aktuellen Kolumne schreibe, werkelt Temurin im Hintergrund daran, einen Snapshot von JDK 18 mit meiner nächsten Contribution für OpenJDK zu kompilieren* (auch das ist eine andere Geschichte). Und dabei bin ich in guter Gesellschaft: Auch IBM, Red Hat, Microsoft, Azul und viele andere Unternehmen sind von Temurin überzeugt. Das ist auch euer Verdienst, denn letztendlich vertritt der iJUG ja euch und eure Interessen in der Adoptium Working Group bei der Eclipse Foundation.

Für mich erfüllt sich mit Temurin ein lang gehegter Traum: Java ist nun nicht mehr gleichbedeutend mit Oracle, sondern mit Community, mit Pluralität, mit Wahlfreiheit. Denn wo es beim Oracle JDK nur kostenpflichtigen Support von Oracle gab, da kann nun bei Temurin jeder Support anbieten. So sind beispielsweise IBM und Azul bereits mit Business-Support-Angeboten für Temurin am Start, parallel zum Support der eigenen Distributionen. Und weil Temurin so verbreitet und der Support leicht verfügbar ist, musste Oracle klein beigeben – zumindest ein bisschen – und gibt nun zwangsweise doch wieder kostenlose JDKs unter das Volk *[1]*. Damit haben wir im iJUG erreicht, was wir seit Oracles Einführung eines Abo-Modells *[2]* 2018 anstrebten: weiterhin kostenloses Java, und eine Distribution in eigener Hand.

Danke an alle Beteiligten, ob im iJUG oder außerhalb, für diesen Meilenstein! Wie gesagt, ich bin uneingeschränkt begeistert. So sehr, dass ich Adoptium-PMC-Member Hendrik Ebbers zu Temurin interviewt habe – was ihr dann in der nächsten Ausgabe zu lesen bekommt.

Ach ja, und noch etwas: Ich würde mich außerordentlich freuen, wenn wir ähnliche Erfolge bald auch mit weiteren wichtigen Bausteinen des Java-Universums erreichen würden: Jakarta EE, MicroProfile oder auch Maven oder JUnit zum Beispiel. Was immer ihr auch nutzt – beteiligt euch daran. Das Java-Universum besteht aus Open Source und lebt von Contributions – euren Contributions!

* Es hat zwar bis zum Ende des Artikels gedauert, aber das Kompilat hat erfolgreich die Test-Suite durchlaufen.

Referenzen

[1] *https://www.doag.org/de/home/news/java-17-oracle-jdk-wieder-kostenfrei/detail/*

[2] *https://www.inside-it.ch/de/post/oracle-lizenziert-java-se-auf-basis-von-abos-20180713*

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.

Unbekannte Kostbarkeiten des SDK Heute: Base64-Kodierungen

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Die Kodierung von Binärdaten in Codepage-unabhängige ASCII-Zeichen wird in der Regel beim Datenaustausch per XML, JSON oder Ähnlichem verwendet. Das Standardformat dafür ist Base64. Seit Java 8 enthält das SDK die Klasse `Base64`, die diese Kodierung zur Verfügung stellt.

Alternative Base64-Kodierungen

Der Autor hat vor vielen Jahren in einem Projekt die letzte vollständig öffentliche Version von iText verwendet, die Version 2.1.7. Diese definierte als Abhängigkeit Bouncy Castle *[1]*, eine Crypto-Bibliothek, die eine Klasse `Base64` zur Kodierung von Binärdaten bereitstellt. Weitere häufig genutzte Implementierungen sind Apache Commons Codec *[2]* oder die nicht öffentlichen SDK-Klassen `sun.misc.BASE64Decoder` und `sun.misc.BASE64Encoder`.

Base64 in Java 8

Mit Java 8 erblickte die Klasse `Base64` im Package `java.util` das Licht der Welt. Laut JavaDoc werden die Base64-Kodierungen der RFCs 4648 *[3]* und 2045 *[4]* durch entsprechende Klassen, nämlich `Base64.Decoder` und `Base64.Encoder`, implementiert. Die RFCs unterscheiden die folgenden Kodierungsarten:

```
public class DeEncodingTest {

    private static final byte[] CAFEBABE = new byte[]{0xC,0xA,0xF,0xE,0xB,0xA,0xB,0xE};
    private static final byte[] ENCODED = Base64.getEncoder().encode(CAFEBABE);

    @Test
    public void encoding() {
  byte[] encodedByJdk = java.util.Base64.getEncoder().encode(CAFEBABE);
  byte[] encodedByBc = org.bouncycastle.util.encoders.Base64.encode(CAFEBABE);
  byte[] encodedByCommons = org.apache.commons.codec.binary.Base64.encodeBase64(CAFEBABE);
  Assertions.assertArrayEquals(encodedByJdk, encodedByBc);
  Assertions.assertArrayEquals(encodedByJdk, encodedByCommons);
    }

    @Test
    public void decoding() {
  byte[] decodedByJdk = java.util.Base64.getDecoder().decode(ENCODED);
  byte[] decodedByBc = org.bouncycastle.util.encoders.Base64.decode(ENCODED);
  byte[] decodedByCommons = org.apache.commons.codec.binary.Base64.decodeBase64(ENCODED);
  Assertions.assertArrayEquals(decodedByJdk, decodedByBc);
  Assertions.assertArrayEquals(decodedByJdk, decodedByCommons);
    }

}
```

Listing 1

- Basic (`getDecoder()`, `getEncoder()`)
- URL und Filename (`getUrlDecoder()`, `getUrlEncoder()`)
- MIME (`getMimeDecoder()`, `getMimeEncoder()`)

Die in den Klammern genannten Klassenmethoden der Klasse `Base64` geben jeweils dedizierte Implementierungen der genannten Kodierungsarten von `Base64.Decoder` und `Base64.Encoder` zurück.

Die Klasse `Base64.Decoder` enthält die Methoden

```
byte[] decode(byte[] src)
int decode(byte[] src, byte[] dst)
byte[] decode(String src)
ByteBuffer decode(ByteBuffer buffer)
InputStream wrap(InputStream is)
```

Die Klasse `Base64.Encoder` enthält die Methoden

```
byte[] encode(byte[] src)
int encode(byte[] src, byte[] dst)
ByteBuffer encode(ByteBuffer buffer)
String encodeToString(byte[] src)
Base64.Encoder withoutPadding()
OutputStream wrap(OutputStream os)
```

Die genannten Methodensignaturen lassen die Verwendungsmöglichkeiten relativ gut erahnen. Dass die grundlegenden Methoden zur Base64-Kodierung des SDK mit denen von Bouncy Castle und Apache Commons Codec übereinstimmen, überprüfen wir mit einfachen Tests, die in der JUnit-Klasse DeEncodingTest in *Listing 1* dargestellt sind. Zur besseren Zuordnung haben wir die verwendeten Klassen mit voll qualifizierten Klassennamen verwendet.

Zusammenfassung

Seit Java 8 erlaubt die Klasse `java.util.Base64` das Kodieren und Dekodieren binärer Daten auf Basis der Base64-Kodierung.

Referenzen

[1] Bouncy Castle. *https://www.bouncycastle.org/*

[2] Apache Commons Code. *https://commons.apache.org/proper/commons-codec/*

[3] The Base16, Base32, and Base64 Data Encodings. *https://www.ietf.org/rfc/rfc4648.txt*

[4] Multipurpose Internet Mail Extensions (MIME). *https://www.ietf.org/rfc/rfc2045.txt*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Java 17 – Das neue LTS-Release

Falk Sippach, embarc Software Consulting GmbH

Der 2018 eingeschlagene Weg – der anfangs umstritten war – wird weiterhin konsequent verfolgt. Die halbjährlichen Updates an der Programmiersprache und der Plattform Java geben uns Entwicklern die einfache Möglichkeit, regelmäßig neue Funktionen auszuprobieren. Potenziell kann man sein Produktivsystem sogar alle sechs Monate aktualisieren und vermeidet damit langwierige Migrationsaufwände, die sonst zwar seltener, aber irgendwann trotzdem anfallen würden.

Konservative Nutzer können jedoch auch weiterhin auf länger unterstützte Long-Term-Support-Versionen setzen, die wie frühere Releases (vor Java 9) über mehrere Jahre mit Updates versorgt werden. Und mit dem OpenJDK 17 ist nun im September 2021 das zweite LTS-Release der neuen Zeitrechnung erschienen.


17

Wenn alle sechs Monate eine neue Version erscheint, kann man natürlich kein Feuerwerk an neuen Features erwarten. Dennoch hat sich insgesamt seit Java 11 doch schon wieder eine Menge getan. Und für das JDK 17 war die Erwartungshaltung auch gar nicht so hoch. Schließlich war klar, dass hauptsächlich die in den letzten zweieinhalb Jahren veröffentlichten Änderungen stabilisiert werden mussten, damit das neue LTS-Release auch in den nächsten Jahren insbesondere für die anstehenden Support-Aufgaben gut dasteht. Die Liste der umgesetzten JEPs (JDK Enhancement Proposals) *[1]* ist allerdings überraschend umfangreich ausgefallen:

- JEP 306: Restore Always-Strict Floating-Point Semantics
- JEP 356: Enhanced Pseudo-Random Number Generators
- JEP 382: New macOS Rendering Pipeline
- JEP 391: macOS/AArch64 Port
- JEP 398: Deprecate the Applet API for Removal
- JEP 403: Strongly Encapsulate JDK Internals
- JEP 406: Pattern Matching for switch (Preview)
- JEP 407: Remove RMI Activation
- JEP 409: Sealed Classes
- JEP 410: Remove the Experimental AOT and JIT Compiler
- JEP 411: Deprecate the Security Manager for Removal
- JEP 412: Foreign Function & Memory API (Incubator)
- JEP 414: Vector API (Second Incubator)
- JEP 415: Context-Specific Deserialization Filters

Nicht alle Themen sind für uns Java-Entwickler direkt relevant. Einige Features sind noch in einem Inkubationsstadium und werden erst in den nächsten Versionen finalisiert. Bei einigen JEPs geht es auch nur darum aufzuräumen, also Teile zu entfernen beziehungsweise für die baldige Entfernung als "Deprecated for Removal" zu markieren. Man merkt auch, dass die drei Jahre zwischen den LTS-Releases nicht immer für größere, lang laufende Änderungen ausreichen. So gibt es auch in Version 17 wieder mal einen ersten Preview für eines der Pattern Matching Features – ein Ende ist da noch nicht absehbar. Weiterhin sind Vector API sowie Foreign Function & Memory API weiterhin im Inkubationsstatus und werden erst in einem der nächsten JDK Releases finalisiert.

Der lange Weg zur Version 17

Tatsächlich sind viele der spannenden Funktionalitäten schon in den letzten Major-Versionen eingeführt und finalisiert worden. Diese Liste ist recht beeindruckend und muss sich inhaltlich nicht hinter einigen der früheren Big-Bang-Releases (8 und älter) verstecken:

- Switch Expressions (ab 12)
- Text Blocks (ab 13)
- Pattern Matching for instanceof (ab 14)
- Helpful NullPointerExceptions (ab 14)
- Records (ab 14)
- Inkubator: Foreign-Memory Access API (ab 14) und Foreign Linker API (ab 16)
- Sealed Classes (ab 15)
- Vector API (ab 16)
- Preview: Pattern Matching for switch (ab 17)
- diverse Änderungen an den Garbage Collectors (ZGC, Shenandoah)
- diverse Neuerungen zum CDS (Class Data Sharing)
- Migration zu Git und GitHub

Einige dieser Neuerungen wollen wir nochmal Revue passieren lassen und uns dann auch mit den neuen JEPs aus dem JDK 17 beschäftigen.

Pattern Matching wirft seine Schatten voraus

Bereits seit einiger Zeit schwebt das Thema Pattern Matching im Raum und erhält nach und nach Einzug in Java. Dazu sind diverse Änderungen in der Sprache selbst notwendig und deshalb erfolgt die Einführung nur schrittweise. Los ging es mit den Switch Expressions bereits im JDK 12. Ab Version 14 folgten „Pattern Matching for instanceof" und Records. Sealed Classes wurden im JDK 15 eingeführt und ganz frisch im JDK 17 ist jetzt „Pattern Matching for switch (Preview)" hinzugekommen. Mit Record und Array-Patterns (JEP 405) werden weitere Mustertypen in den nächsten JDK-Releases folgen. Alle Themen rund um das Pattern Matching sind mit ein oder zwei Previews gestartet. So konnten die Macher von Java frühzeitig Feedback einsammeln und einarbeiten. Mittlerweile sind die meisten Features final und damit in den Sprachumfang von Java SE aufgenommen.

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

Die Spezialform des Pattern Matching beim `instanceof`-Operator spart unnötige Casts auf die zu prüfenden Ziel-Datentypen. Wenn `obj` ein String ist, dann kann direkt mit der neuen Variablen (`s`) weitergearbeitet werden. Das Ziel ist es, Redundanzen zu vermeiden und dadurch auch die Lesbarkeit zu erhöhen *(siehe Listing 1)*. Letztlich passieren drei Schritte: Typ-Check, Cast und Zuweisung.

Der Unterschied zum zusätzlichen Cast mag marginal erscheinen. Für die Puristen unter den Java-Entwicklern spart das allerdings eine kleine, aber dennoch lästige Redundanz ein. Laut Brian Goetz soll die Sprache dadurch prägnanter und die Verwendung sicherer gemacht werden. Manuelle Typumwandlungen werden vermieden und stattdessen implizit durchgeführt.

Sealed Classes

Die Sealed Classes waren zweimal Preview-Feature und wurden jetzt im JDK 17 als JEP 409 finalisiert. Getan hat sich seit Java 16 nichts mehr. Ganz konkret sollen sie im Rahmen des Pattern Matching den Compiler bei der Analyse von Mustern unterstützen, denn sie können die Vollständigkeit (Exhaustiveness) einer Switch-Anweisung sicherstellen. Aber auch für Framework-Entwickler bieten die Sealed Classes einen interessanten Mehrwert.

Die Idee ist, dass versiegelte Klassen und Interfaces entscheiden können, welche Subklassen oder -Interfaces von ihnen abgeleitet werden dürfen. Bisher konnte man als Entwickler die Ableitung von Klassen nur über Zugriffsmodifikatoren (package private, protected, ...) einschränken oder mit „final" komplett untersagen. Sealed Classes bieten nun einen deklarativen Weg, um ganz gezielt nur bestimmten Subklassen die Ableitung zu erlauben *(siehe Listing 2)*.

`Vehicle` darf nur von den vier genannten Klassen überschrieben werden. Damit wird auch dem Aufrufer deutlich gemacht, welche Subklassen erlaubt sind und überhaupt existieren. Mit dem JDK 17 können Sealed Classes im Rahmen des Pattern Matching for Switch eingesetzt werden. Werden dann alle erlaubten Subklassen in den `case`-Zweigen verwendet, kann der Einsatz des `default`-Blocks und damit eine Fehlerquelle (pauschales Catch-All) entfallen. Durch die Information aus der `permit`-Anweisung kann der Compiler sicherstellen, dass mindestens einer der Zweige aufgerufen wird *(siehe Listing 3)*.

Subklassen bergen immer die Gefahr, dass beim Überschreiben der Vertrag der Superklasse und damit das Liskov'sche Substitutionsprinzip verletzt wird. Zum Beispiel ist es unmöglich, die Bedingungen der `equals`-Methode aus der Klasse `Object` zu erfüllen, wenn man Instanzen von einer Super- und einer Subklasse miteinander vergleichen will. Weitere Details dazu kann man in der API-Dokumentation *[2]* unter dem Stichwort Äquivalenzrelationen (konkret Symmetrie) nachlesen. Wenn man als Ersteller einer Klasse aber die volle Kontrolle über alle erlaubten Subklassen hat, muss man später nicht mit bösen Überraschungen rechnen, wenn ein Kollege aus Gründen der Wiederverwendung den Vertrag der Superklasse verletzt.

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

Wenn es nur wenige und eher kleine Subklassen gibt, kann man sie auch direkt in der gleichen Quellcode-Datei wie die Sealed Class deklarieren. Dann spart man sich die `permit`-Anweisung und der Compiler ermittelt die relevanten Sub-Typen selbst *(siehe Listing 4)*.

Records

Die in Java 14 eingeführten und bereits im JDK 16 finalisierten Record-Datentypen dürfen an dieser Stelle natürlich auch nicht fehlen. Immerhin stellen sie eine größere und für viele Entwickler sehr nützliche Funktionalität bereit. Bei den Records handelt es sich um eine eingeschränkte Form der Klassendeklaration, ähnlich zu den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Auch sie sind im Umfeld der Einführung von Pattern Matching entstanden und werden demnächst bei den Record Patterns noch relevanter werden.

Aber auch ohne Pattern Matching gibt es interessante Anwendungsfälle. Durch ihre kompakte Syntax können sie einige Funktionen von Bibliotheken wie Lombok in Zukunft obsolet machen. Die einfache Definition einer unveränderbaren Person mit zwei Feldern kann man in *Listing 5* betrachten. Eine erweiterte Variante mit einem zusätzlichen Konstruktor ist erlaubt. Dadurch lassen sich neben Pflichtfeldern auch optionale abbilden *(siehe Listing 6)*.

```
if (obj instanceof String s) {
    // Let pattern matching do the work!
    ...
}
```

Listing 1

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

Listing 2

```
public BigDecimal calculateExpense(Vehicle vehicle) {
  return switch(vehicle) {
    case Car c -> calculateCarExpense(c);
    case Bike b -> calculateBikeExpense(b);
    case Bus b -> calculateBusExpense(b);
    case Train t -> calculateTrainExpense(t);
  }
}
```

Listing 3

```
abstract sealed class Root { ...
    final class Sub1 extends Root { ... }
    final class Sub2 extends Root { ... }
    final class Sub3 extends Root { ... }
}
```

Listing 4

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

Listing 5

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

Listing 6

Vom Compiler wird eine immutable Klasse erzeugt, die neben den beiden Attributen und den eigenen Methoden natürlich auch noch die Implementierungen für die Accessoren, den Konstruktor sowie equals/hashCode und toString enthält. In *Listing 7* sieht man den Pseudo-Code, den man dafür hätte schreiben müssen.

Die Records werden dann wie normale Java-Klassen verwendet. Der Aufrufer merkt also gar nicht, dass ein spezieller Typ instanziiert wird *(siehe Listing 8)*. Und genau genommen ist es ja auch kein spezieller Datentyp, sondern syntaktischer Zucker des Compilers.

```
public final class Person extends Record {
  private final String name;
  private final Person partner;
  public Person(String name) { this(name, null); }
  public Person(String name, Person partner) {
    this.name = name; this.partner = partner;
  }

  public String getNameInUppercase() {
    return name.toUpperCase();
  }
  public String toString() { /* ... */ }
  public final int hashCode() { /* ... */ }
  public final boolean equals(Object o) { /* ... */ }
  public String name() { return name; }
  public Person partner() { return partner; }
}
```

Listing 7

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

Listing 8

Records sind übrigens keine klassischen Java-Beans, da sie keine echten Getter enthalten. Um auf den Zustand (die Membervariablen) zuzugreifen, verwendet man dafür die gleichnamigen Methoden (`name()` statt `getName()`). Damit verletzen Records genau wie auch Java-Beans bewusst das Information-Hiding-Prinzip.

Letztlich sind sie jedoch einfach nur als Träger für Daten gedacht, um zusammengehörige Informationen typsicher zu gruppieren (ähnlich wie Tupel). Einsetzen kann man sie zum Beispiel als DTOs (Datentransferobjekt) oder für Projektionen bei Datenbankabfragen.

Sie können im Übrigen auch Annotationen oder JavaDocs enthalten. Im Body dürfen zudem statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Nicht erlaubt ist die Definition von weiteren Instanzfeldern außerhalb des Record Header.

Zwischen Sealed Classes und den Record-Typen gibt es eine Integration, wie das Beispiel in *Listing 9* zeigt.

Eine Familie von Records kann von dem gleichen Sealed Interface ableiten. Die Kombination aus Records und versiegelten Datentypen führt uns zu algebraischen Datentypen, die vor allem in funktionalen Sprachen wie Haskell zum Einsatz kommen. Konkret können wir jetzt mit Records Produkttypen (enthalten mehrere Informationen, beispielsweise Vor- und Nachname bei Person) und mit versiegelten Klassen Summentypen (es gibt immer genau einen Typ, ähnlich wie eine Instanz bei Aufzählungstypen) abbilden. Auch hier schließt sich wieder der Kreis zum Pattern Matching. Die Modernisierung von Java entwickelt sich also weiterhin in die vielversprechende Richtung der funktionalen Programmierung. Aber bitte nicht erwarten, dass Java in ein paar Jahren eine rein funktionale Sprache sein wird. Da wird es auch in Zukunft besser geeignete Kandidaten (Haskell, Clojure, ...) geben. Nichtsdestotrotz wird man auch bei der Entwicklung mit Java von einigen dieser Möglichkeiten profitieren.

Pattern Matching for switch (Preview)

Im JDK 17 im JEP 406 neu hinzugekommen ist die Integration des Type Pattern (Prüfung mit instanceof) in die Switch Expression. Diese recht große Änderung ist für ein LTS-Release eher untypisch, da sie in diesem Long-Term-Support-Zyklus nicht mehr abgeschlossen werden kann. Höchstwahrscheinlich wird in den nächsten Releases noch mindestens ein weiterer Preview und dann die Finalisierung erfolgen. Es wird sowieso noch einige Iterationen benötigen, bis das Pattern Matching vollständig in Java umgesetzt ist.

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

Listing 9

```
record Point(int i, int j) {}
enum Color { RED, GREEN, BLUE; }

static void typeTester(Object o) {
    switch (o) {
        case null -> System.out.println("null");
        case String s -> System.out.println("String");
        case Color c -> System.out.println("Color with " + c.values().length + " values");
        case Point p -> System.out.println("Record class: " + p.toString());
        case int[] ia -> System.out.println("Array of ints of length" + ia.length);
        default -> System.out.println("Something else");
    }
}
```

Listing 10

Man muss also im Hinterkopf behalten, dass es sich um ein Preview-Feature handelt, das standardmäßig deaktiviert und sowohl beim Kompilieren als auch Starten mit `--enable-preview` explizit eingeschaltet werden muss.

Listing 10 demonstriert den Funktionsumfang mit einem Beispiel aus der JEP-Dokumentation *[2]*. Der Selektionsausdruck `o` wird auf verschiedene Typen geprüft. Je nachdem, ob es sich um einen String, eine Farbe, einen Punkt oder ein `int`-Array handelt, wird der jeweilige `case`-Zweig aufgerufen. In einem normalen Switch darf im `case` nur gegen primitive Zahlen beziehungsweise deren Wrapper sowie Strings und Enums geprüft werden. Hier wird jedoch jetzt geschaut, ob `o` einem der Datentypen (inklusive Array) entspricht. Ist das der Fall, wird automisch ein Cast auf den Datentyp ausgeführt und in einer Variablen gespeichert. Diese kann dann direkt weiterverarbeitet werden. Sollte `o` eine Null-Referenz sein, dann wird der optionale Zweig `case null` aufgerufen und nicht wie üblich direkt eine NullPointerException geworfen. Ein expliziter Null-Check vor Aufruf des Switch kann somit entfallen.

Die Type Patterns kann man mit Guarded Patterns verfeinern, indem zusätzlich zum Typ noch nach bestimmten Inhalten gefiltert wird. Dazu wird ein boolescher Ausdruck mit einer logischen und-Verknüpfung an das Type Pattern angehängt. Damit erspart man sich if-else-Abfragen im `case`-Zweig. *Listing 11* stellt beide Varianten gegenüber. Mit dem Guarded Pattern wirkt der Switch aufgeräumter und ist verständlicher.

```
static void test(Object o) {
    switch (o) {
        case String s:
            if (s.length() == 1) { ... }
            else { ... }
            break;
        ...
    }
}
static void test(Object o) {
    switch (o) {
        case String s && (s.length() == 1) -> ...
        case String s -> ...
        ...
    }
}
```

Listing 11

Weitere Neuerungen im JDK 17

Wer schon sehr lange in der Java-Welt unterwegs ist, wird sich vielleicht noch an das Java Native Interface (JNI) erinnern. Damit kann man nativen C-Code aus Java heraus aufrufen. Der Ansatz ist aber relativ aufwendig und fragil. Das Foreign Linker API bietet einen statisch typisierten, rein Java-basierten Zugriff auf nativen Code. Zusammen mit dem Foreign-Memory-Access-API kann diese Schnittstelle den bisher fehleranfälligen Prozess der Anbindung einer nativen Bibliothek beträchtlich vereinfachen. Mit letzterer bekommen Java-Anwendungen die Möglichkeit, außerhalb des Heap zusätzlichen Speicher zu allokieren. Beide APIs sind seit dem JDK 14 beziehungsweise 16 mit dabei, im JDK 17 jetzt als JEP 412 zusammengefasst und weiterhin als Inkubator enthalten. Wann sie finalisiert werden, steht noch nicht fest.

Ebenfalls noch im Inkubator ist der JEP 414: Vector API. Da geht es übrigens nicht um die Runderneuerung der alten java.util.Vector-Klasse. Vielmehr will man die Fähigkeiten der SIMD-Rechnerarchitektur mit Vektorprozessoren ins JDK bringen. Single Instruction Multiple Data (SIMD) lässt viele Prozessoren gleichzeitig unterschiedliche Daten verarbeiten. Durch die Parallelisierung auf Hardware-Ebene verringert sich beim SIMD-Prinzip der Aufwand für rechenintensive Schleifen.

Durch das in Java 9 eingeführte Java Platform Modul System (JPMS) können nun JDK-interne Klassen vor dem Zugriff von außen geschützt werden. Bisher gab es zum Übergang jedoch die schwache Kapselung als Default, das heißt, interne APIs konnten weiterhin verwendet werden. Dieses Schlupfloch wurde im JDK 16 standardmäßig geschlossen. Man konnte allerdings noch zum vorherigen Verhalten zurückkehren, wenn der Schalter `--illegal-access` explizit auf `permit` gesetzt wurde. Im OpenJDK 17 wird laut dem JEP 403 dieser JVM-Schalter gar nicht mehr ausgewertet. Damit lässt sich die schwache Kapselung nicht mehr zurückholen. In Zukunft soll der Schalter dann komplett verschwinden (aktuell wird er ignoriert, später wird dann ein Fehler geworfen). Das Ziel dieser Maßnahmen ist die Erhöhung der Sicherheit und der Wartbarkeit des JDK. Man möchte die Entwickler ermutigen, alte, auf Interna basierende Lösungen zukünftig auf den Zugriff über Standard-APIs umzubauen. Somit sollen sowohl Java-Entwickler als auch Endbenutzer viel problemloser auf zukünftige Versionen updaten können. Kritische interne APIs wie sun.misc.Unsafe sind davon übrigens (noch) nicht betroffen und können weiterhin verwendet werden.

Neben den prominenten JEPs gibt es in jeder neuen JDK-Version auch noch kleine Änderungen, zum Beispiel an der Java-Klassenbibliothek.

Mit dem Java Version Almanac *[2]* kann man sehr einfach und kompakt die Differenzen zwischen den Releases, aber auch bei Versionssprüngen etwa von JDK 11 auf 17 einsehen. Im Vergleich zum OpenJDK 16 *[3]* fallen insbesondere Änderungen an den Random-Klassen auf. Der JEP 356 „Enhanced Pseudo-Random Number Generators" hat eine Menge von neuen Pseudozufallszahlengeneratoren und ein API zum einfacheren Austausch der Zufallszahlengenerator-Strategie eingeführt. Es gibt jetzt ein Interface RandomGenerator, von dem nun auch die uralte Klasse java.util.Random ableitet.

Für MacOS-Nutzer gibt es noch zwei neue Features. JEP 382: „Rendering Pipeline für MacOS" stellt eine vollfunktionsfähige Rendering-Pipeline für das Java-2D-API bereit. Sie verwendet das Metal-Framework und soll zukünftig die bestehende Rendering-Pipeline, die auf dem veralteten OpenGL-API basiert, ersetzen. Mit JEP 391: „macOS/AArch64 Port" wird ein auf der AArch64-Prozessarchitektur basierender JDK-Port bereitgestellt. Damit soll Apples neuer Prozessor M1 nativ in Java unterstützt werden.

Das JEP 415: „Context Specific Deserialization Filters" erweitert die Deserialization-Filter, die in Java 9 eingeführt wurden. Es geht um die Validierung eingehender serialisierter Daten, die von nicht-vertrauenswürdigen Quellen kommt. Denn Serialisierung kann zu vielen Security Issues führen. Das Filtern passiert dabei in der JVM.

Abschied nehmen

Java war lange Zeit dafür bekannt, keine Altlasten zu entfernen. Vielmehr blieb aus Abwärtskompatibilitätsgründen alles erhalten. Das hat sich geändert, mittlerweile werden Teile, die zum Löschen markiert sind, in einem der nächsten Releases auch wirklich entfernt.

Im JDK 17 wurde nun ein Klassiker, das Applet-API, zum baldigen Abschuss freigegeben. Denn aus Sicherheitsgründen unterstützt mittlerweile kein Browser-Hersteller mehr diese „Altlast" aus den 90ern. Applets waren ein interessantes Feature in den ersten Jahren und haben auch einen gewissen Anteil daran, dass Java schnell populär wurde. Mittlerweile verwendet man jedoch alternative Lösungsansätze wie klassische Webanwendungen, Single Page Applications oder lädt Desktop-Anwendungen mit dem Webstart-Feature. Wann die Applets endgültig entfernt werden sollen, steht noch nicht fest. Ebenfalls zum Löschen markiert wurde auch der bereits seit Java 1.0 existierende Security-Manager. Er wurde hauptsächlich für die clientseitige Absicherung von Java-Code (beispielsweise in Applets) und weniger auf der Serverseite eingesetzt *[4]*.

Direkt entfernt wurde der RMI(Remote Method Invocation)-Activation-Mechanismus, der bereits im JDK 15 als zu löschen markiert wurde. RMI Activiation ist mittlerweile obsolet geworden, außerdem gibt es Konflikte mit Activation-Mechanismen in Java/Jakarta EE, die aber in keinem Zusammenhang stehen. Die Remote Method Invocation selbst ist davon nicht betroffen, sie kann weiterhin genutzt werden.

Ebenfalls entfernt wurde der Experimental AOT and JIT Compiler. Er wurde wenig genutzt und die Wartungsaufwände waren sehr hoch. Das Java Level JVM Compiler Interface (JVMCI) soll jedoch vorerst bestehen bleiben, sodass Entwickler weiterhin extern gebaute Versionen des JIT-Compilers verwenden können. Interessant ist die Art und Weise, wie ein möglicher Einsatz dieses Compilers ermittelt wurde. In den Oracle JDK 16 Builds wurde er einfach weggelassen und da es keine Beschwerden gab, hat man ihn nun komplett aus dem OpenJDK entfernt.

Ausblick

Interessant ist, dass Java 8 und auch noch ältere Versionen weiterhin sehr verbreitet sind. Es gibt drei einigermaßen repräsentative Umfragen *[5] [6] [7]*, die ihre Nutzer und Kunden jährlich nach den produktiv verwendeten Java-Versionen fragen. Mehrfachnennungen

sind da erlaubt. Java 8 ist immer noch stark vertreten (69 % bei JRebel, 60 % bei Snyk und 72 % bei JetBrains) und eigentlich verwundert das nicht. Es gibt vermutlich viele Anwendungen, die laufen einfach und werden möglicherweise kaum weiterentwickelt. Und solange es von den Herstellern noch Unterstützung für Java 8 gibt, wird es wenig Anreiz für ein Update geben. Neue oder stark in der Weiterentwicklung befindliche Projekte setzen allerdings jetzt immer mehr auf mindestens Java 11, auch wenn da die Spannweite bei den Statistiken deutlich größer ist (36 % bei JRebel, 62 % bei Snyk und 42 % bei JetBrains). Der Trend geht also langsam, aber stetig weg von Version 8 in Richtung 11 oder höher. Spannend dürfte sein, wie sich das Verhältnis zwischen Java 11 und 17 in den nächsten zwei Jahren ändern wird. Da der Umfang der Migrationen zwischen diesen beiden LTS-Versionen überschaubar ist, werden vermutlich viele recht schnell auf die neueste Version wechseln.

Auch mit den JDK-Distributionen hat man mittlerweile immer mehr sehr gute Optionen. Technisch basieren insbesondere die freien, aber auch viele der kommerziellen Varianten sowieso auf dem OpenJDK von Oracle. Insbesondere das frühere AdoptOpenJDK (jetzt Temurin des Projekts Adoptium bei der Eclipse Foundation) *[8]* hat sehr viele unterstützende Firmen mit im Boot und auch eine stetig steigende Nutzerzahl. Interessant ist, dass die Temurin JDK Builds mittlerweile auch gegen das TCK geprüft werden. Das Java SE Technology Compatibility Kit (TCK) wird von Oracle angeboten, um Java Binaries als kompatibel zum Java-Standard zertifizieren zu lassen. Das TCK ist nicht frei verfügbar und muss bei Oracle lizenziert werden. Früher war das beim AdoptOpenJDK nicht der Fall. Dafür gab es dort schon immer eine große Menge von separaten Tests. In der Summe sind die Temurin Java Builds nun also noch besser getestet. Und infrastrukturell wird ebenfalls ein sehr hoher Aufwand betrieben, um die verschiedensten, teilweise exotischen Varianten aus der Kombination von Version, Betriebssystem und Plattform anzubieten.

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

Nach dem LTS-Release ist vor dem nächsten LTS-Release. Jetzt wird bei den kommenden halbjährlichen Major-Releases auf das schon in zwei Jahren erscheinende Java 21 hingearbeitet. Oracle hat nämlich direkt mit der Veröffentlichung von Java 17 den verkürzten Release-Zyklus (statt bisher drei Jahre) angekündigt *[9]*. Themen werden voraussichtlich die Finalisierung des Pattern Matching sein und gegebenenfalls sehen wir auch andere spannende Themenbereiche aus den verschiedenen Inkubator-Projekten (Amber, Loom, Valhalla, Panama, …). Interessant wären zum Beispiel Looms leichtgewichtige Threads (Fibers).

Konkret für das JDK 18 gibt es aktuell (Stand Oktober 2021) fünf eingeplante JEPs. JEP 400 „UTF-8 by Default" hat zum Ziel, UTF-8 als das Default-Character-Set für die Standard-Java-APIs für File IO zu verwenden. APIs sollen sich über alle Implementierungen, Betriebssysteme, Lokalisierungen und Konfigurationen hinweg konsistent verhalten. Im JEP 413 „Code Snippets in Java API Documentation" wird in JavaDoc ein neues Tag @snippet eingeführt, um das Einbinden von Quellcode-Beispielen in die API-Dokumentation zu vereinfachen. Klingt nicht nach einer bahnbrechenden, aber doch nach einer sehr nützlichen Idee. Des Weiteren wird es mit dem JEP 408 eine einfach Web-Server-Implementierung geben und der Reflection Core soll mit Method-Handles neu implementiert werden (JEP 416). Und last, but not least wird in JEP 417 weiter am Vector-API (dritter Inkubator) gearbeitet.

Es werden aber sicher noch einige andere spannende Neuerungen und Syntax-Erweiterungen hinzukommen. Immerhin kann man jetzt nach dem LTS-Release in den nächsten Zwischenversionen wieder einige Previews unter die Leute bringen und so in den nächsten anderthalb Jahren Feedback einsammeln und einarbeiten. Ein heißer Kandidat aus Sicht des Pattern Matching wird übrigens der JEP 405 („Record Patterns & Array Patterns (Preview)") sein *[10]*. Die Zukunftsaussichten für Java und sein Ökosystem sind also weiterhin sehr rosig!

Referenzen

[1] *https://openjdk.java.net/projects/jdk/17/*
[2] *https://javaalmanac.io/*
[3] *https://javaalmanac.io/jdk/17/apidiff/16/*
[4] *https://foojay.io/today/jep-411-what-it-means-for-javas-security-model/*
[5] *https://www.jrebel.com/blog/2021-java-technology-report*
[6] *https://snyk.io/jvm-ecosystem-report-2021/*
[7] *https://www.jetbrains.com/lp/devecosystem-2021/*
[8] *https://adoptium.net/*
[9] *https://blogs.oracle.com/java/post/moving-the-jdk-to-a-two-year-lts-cadence*
[10] *https://openjdk.java.net/jeps/405*

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

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

RPA-gestützte Datenmigration in der Praxis

Bastian Engelking, viadee AG

Datenmigrationen sind meist einmalige Projekte und bedürfen daher individueller Strategien, um Daten zuverlässig und technisch sicher vom Quell- ins Zielsystem zu überführen. Abhängig von der jeweiligen Situation kann der Einsatz von RPA (Robotic Process Automation) und entsprechenden Tools ein Baustein in der Migrationsstrategie sein. In diesem Artikel erklären wir anhand eines beispielhaften Projekts aus der Energiebranche, wann der Einsatz sinnvoll ist und wie Daten mithilfe von RPA migriert werden können. In diesem Zuge werden ein in Java geschriebenes und auf Open-Source-Bibliotheken basierendes RPA-Tool und seine unterschiedlichen Automatisierungstreiber vorgestellt, sodass die Leser/innen ein vollumfassendes Bild über die Technologie erhalten.

Status Quo

RPA ist inzwischen eine etablierte Technologie, um repetitive und strukturierte Arbeitsschritte in Unternehmen zu automatisieren. Der Einsatz von RPA erfolgt häufig dort, wo Schnittstellen fehlen oder es sich nicht mehr lohnt, Systeme weiterzuentwickeln. Dabei ist der Einsatz von RPA zumeist keine langfristige Lösung. Allerdings lassen sich RPA-Lösungen verhältnismäßig schnell umsetzen und flexibel an eine sich ständig verändernde Systemumgebung anpassen. Damit hat RPA seine Daseinsberechtigung in der IT-Landschaft heutiger Unternehmen. Auch in Migrationsprojekten kann RPA eingesetzt werden, wie das im Folgenden vorgestellte Beispielprojekt verdeutlicht.

Migrationsszenario aus der Energiebranche

Als Datenmigration wird ein Vorhaben bezeichnet, das die Übertragung von Daten aus einem System in ein anderes zum Ziel hat. Es gibt meist viele Gründe für ein solches Vorhaben. Häufig führen Fusionen und Unternehmensübernahmen zur Konsolidierung mehrerer Datenbestände.

In dem hier gezeigten Beispielprojekt ist es genau andersherum. Aufgrund gesetzlicher Vorgaben im Zusammenhang mit der Fusion zweier Energieversorger musste ein Geschäftsbereich ausgegliedert werden. Der gesamte Datenbestand für mehrere hunderttausend Kunden musste daher in ein neues System überführt werden. Dies ist eine Größenordnung, für die RPA vermutlich nicht das geeignete Mittel der Wahl ist. Die Migration der Daten von Kunden und Vertragspartnern wurde auf Datenbankebene durchgeführt. Es wurden komplexe ETL-Prozesse (extract, transform, load) entwickelt, um große Datenmengen in kurzer Zeit zu migrieren. Ein solches Projekt bindet viele Ressourcen sowie Know-how und ist mit einigen Risiken verbunden.

Im Beispielprojekt war das Projektmanagement vorrausschauend und hat frühzeitig erkannt, dass es bei der Migration einer bestimmten Entität zu Engpässen kommen würde. Bei den Entitäten handelte es sich um die Datenbestände der Geschäftspartner die als Zahler, Zahlungs- oder Mitteilungsempfänger für andere Kunden agierten. Mengenmäßig handelte es sich hierbei um mehrere tausend Geschäftspartner und damit um eine hinreichend kleine Menge, die mit RPA in einem engen Zeitfenster migriert werden könnte. Das Ziel des Projektes war es, diese Geschäftspartner mit allen Stammdaten aus dem Quellsystem über die Benutzeroberfläche in das Zielsystem zu überführen und wieder den zuvor migrierten Kunden zuzuordnen. Was sich nach einem vielversprechenden Vorhaben anhört, musste jedoch wohl bedacht werden.

Entscheidung für eine RPA-gestützte Migration

Bei der Entscheidung für den Einsatz eines RPA-Tools zur Datenmigration wurden daher die Vor- und Nachteile gegeneinander abgewogen und mit einer klassischen Migration auf Datenbankebene verglichen.

RPA-Tools haben den Vorteil, dass sie einem Low-Code-Ansatz folgen, wodurch in der Regel weniger Programmieraufwand anfällt. Die gängigen Tools bringen out of the box vordefinierte und wiederverwendbare Funktionen mit, sodass Arbeitsabläufe effizient und in kurzer Zeit automatisiert werden können. Der Einsatz solcher Tools ist insbesondere dann lohnenswert, wenn RPA-Entwickler/innen verfügbar sind und gleichzeitig das Personal im Datenbank- und Entwicklungsumfeld knapp ist. Sind die zu automatisierenden Abläufe erst mal konzeptionell entworfen und implementiert, kann mit der richtigen Infrastruktur und durch den Einsatz mehrerer Instanzen des RPA-Tools die Anzahl gleichzeitiger Prozessdurchläufe skaliert werden. Zudem können die Instanzen theoretisch rund um die Uhr ohne Unterbrechung betrieben werden, wenn dies die automatisierten Systeme zulassen und es beispielweise keine Tagesend- oder Nachtverarbeitungen gibt. Hierbei handelt es sich um ein Risiko, das durch die Geschwindigkeit bei einer Migration auf Datenbankebene vermieden werden könnte.

Beim Faktor Datenqualität können RPA-Tools Vorteile gegenüber einer Migration auf Datenbankebene aufweisen. Beispielweise kann bei der Migration auf die bestehenden Regeln und Prüfroutinen des Zielsystems zurückgegriffen werden, um diese zur Qualitätssicherung zu nutzen. Der Großteil der Validierungsregeln eines Systems ist in die Benutzeroberfläche und den Code integriert. Bei der Eingabe unvollständiger oder fehlerhafter Daten über die Benutzeroberfläche verweigert das System die Eingabe. Diese Daten können ausgesteuert und anschließend bereinigt werden. Bei einer ETL-basierten Migration müssten Validierungsregeln zunächst nachgebaut werden oder es würde in Kauf genommen, dass fehlerhafte Daten direkt in die Datenbanken geschrieben werden. Die Nutzung bestehender Workflows und Eingabemasken des Systems hat den weiteren Vorteil, dass auf der bestehenden Geschäftslogik aufgebaut werden kann. Dies führt auch dazu, dass weniger systemspezifisches Know-how benötigt wird und wiederum knappe Ressourcen geschont werden können.

Der Einsatz von RPA kann jedoch auch Nachteile mit sich bringen. Es müssen geeignete organisatorische Sicherheits- und Betriebskonzepte erarbeitet werden, um ein zuverlässiges Abarbeiten der automatisierten Abläufe zu gewährleisten. Häufig gibt es diese Konzepte bereits für die Datenbankentwicklung und entsprechende ETL-Tools, nicht aber für RPA-Tools. Des Weiteren besitzen RPA-Tools in der Regel keine ausgefeilten Datenanalysefunktionen wie ETL-Tools, wodurch Datenbereinigungsaktivitäten und die Nutzung temporärer Datenbanken für die Migration erschwert werden. Gute RPA-Tools sollten jedoch Datenbanktreiber mitbringen, was bei einer Toolauswahl berücksichtigt werden sollte. Das wichtigste Argument, das gegen den Einsatz von RPA-Tools spricht, ist vermutlich die Datenmenge. Bei großen Datenmengen, die über die Benutzeroberfläche migriert werden sollen, kommt RPA an seine Grenzen, da sehr viele Instanzen benötigt werden, die betrieben und überwacht werden müssen.

In dem vorgestellten Beispielprojekt haben die Vorteile von RPA zur Migration der genannten Entität überwogen. Die Verantwortlichen waren sich einig, dass RPA das Ziel einer fachlich richtigen, technisch sicheren und zuverlässigen Datenmigration optimal unterstützt. Somit wurde RPA ein Baustein in der Migrationsstrategie.

Verwendete Automatisierungs-Treiber

Für die Migration wurde das in Java geschriebene und auf Open-Source-Bibliotheken basierende Tool „mateo rpa" verwendet. Das Tool besitzt eine offene und erweiterbare Architektur, die es ermöglicht, verschiedene Automatisierungstreiber zu integrieren *(siehe Abbildung 1)*. Angelehnt an die generische Testautomatisierungs-

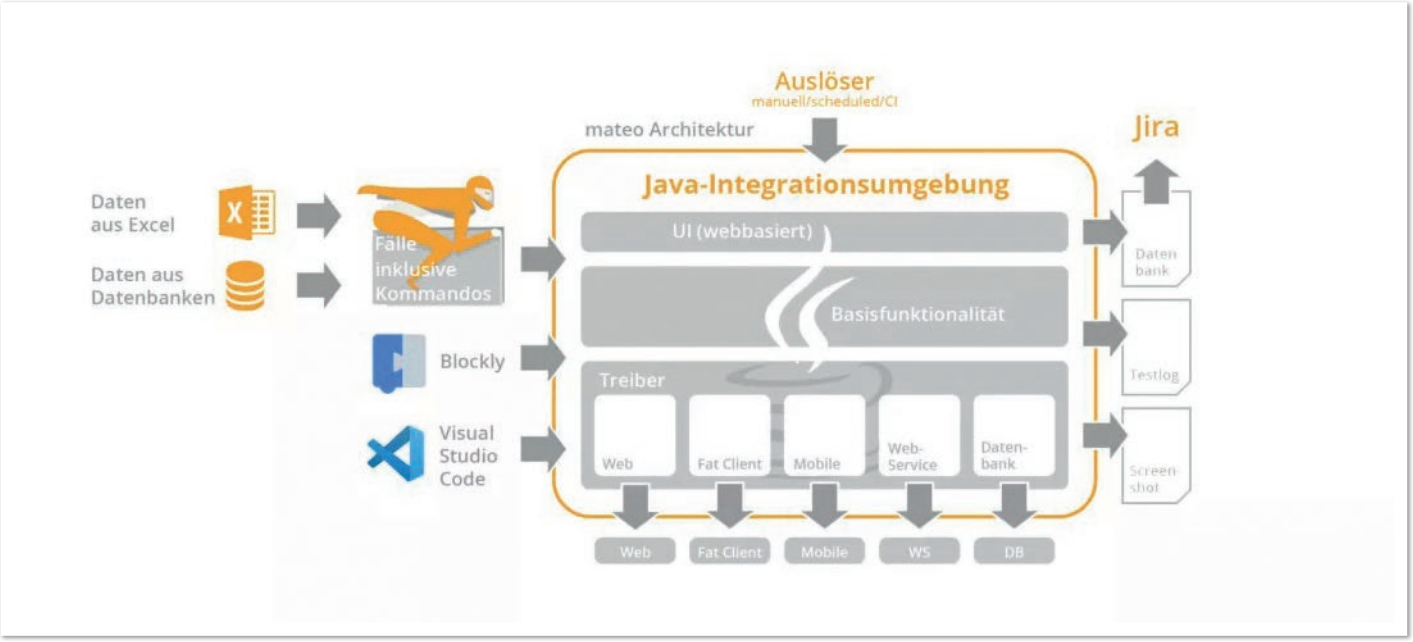


Abbildung 1: Architektur mateo rpa (© viadee Unternehmensberatung AG 2021)

Unter der Basisfunktionalität sind Funktionen zu verstehen, mit denen der Kontrollfluss der automatisierten Arbeitsabläufe abgebildet wird. Es können Bedingungen, Schleifen oder der Vergleich von Werten umgesetzt werden (siehe Listing 1).

```
1  while(MAX_ITERATIONS = "10"):
2     lessThanNum(LEFT = "{index}", RIGHT = "8")
3  do:
4     if():
5        lessThanNum(LEFT = "{index}", RIGHT = "4")
6     then:
7        success(MESSAGE = "Der Index ist kleiner 4.")
8     else:
9        success(MESSAGE = "Der Index ist größer gleich 4.")
```

Listing 1: While-Schleife, Bedingungen und Wertevergleich in mateo rpa

architektur nach ISTQB *[1]* werden Automatisierungstreiber der Testadaptierungsschicht zugeordnet und trennen das zu steuernde System von der im RPA-Skript definierten Logik. Ein Automatisierungstreiber ist daher ein Software-Modul innerhalb des Tools, der es ermöglicht, die zu steuernde Anwendung anzusprechen und mit dieser zu interagieren. Für die Migration wurde neben der Basisfunktionalität des Tools auf Automatisierungstreiber für Web, SAP und Datenbanken zurückgegriffen.

Der verwendete Web-Treiber basiert auf Selenium *[2]*, einem weitverbreiteten und quelloffenen Framework, das sich bei der Testautomatisierung von Webanwendungen bewährt hat, aber auch für RPA-Szenarien verwendet werden kann. Selenium kann mit verschiedenen Programmiersprachen (zum Beispiel Java, Python, C#) genutzt oder, wie im Fall von mateo, als Automatisierungstreiber in ein umfassendes RPA-Tool integriert werden. Diese Einbettung bringt verschiedene Vorteile für die Entwicklungszeit und Qualität der Skripte mit sich. Insbesondere vordefinierte Wrapper-Funktionen, die ansonsten initial entwickelt werden müssten, beschleunigen die Entwicklung. Als Beispiel kann hier die Funktionalität zum Warten auf die Klickbarkeit eines Elements vor dem tatsächlichen Klick auf das Element genannt werden. RPA-Tools besitzen hierfür vordefinierte Funktionen wie *clickWeb*, das einen impliziten Warteschritt vor dem tatsächlichen Klick des Elements einschließt.

Bei dem zu automatisierenden SAP-System handelte es sich um eine Version, die über eine Windows-Oberfläche gesteuert werden musste. Hierzu bringt das RPA-Tool einen auf AutoIt *[3]* basierenden SAP-Treiber mit. AutoIt ist eine leistungsfähige, frei verfügbare Skriptsprache mit BASIC-ähnlicher Syntax. Sie ist in erster Linie für die Automatisierung von Operationen auf der Windows-GUI konzipiert. AutoIt bietet eine COM-Schnittstelle, die es ermöglicht, das SAP-GUI-Scripting-API in AutoIt zu verwenden. Mit diesem API können Benutzerinteraktionen mit SAP imitiert und somit ganze Arbeitsabläufe und Prozesse automatisiert werden. Analog zum Web-Treiber stellt das RPA-Tool mittels AutoIt vordefinierte Funktionen bereit. Mit der Funktion *sapObjSelect* können beispielweise Objekte selektiert werden. Dass AutoIt im Hintergrund zur Interaktion mit SAP genutzt wird, fällt nicht auf.

Um die exportierten Stammdaten sicher zwischenzuspeichern, wurde auf eine temporäre Datenbank gesetzt. Der Datenbank-Treiber des verwendeten Tools basiert auf JDBC (Java Database Connectivity), der Standardschnittstelle für datenbankunabhängige Verbindungen zwischen Java und einer Vielzahl von Datenbankverwaltungssystemen wie MySQL, Oracle und anderen. Damit lassen sich Datenbankverbindungen herstellen und SQL-Anweisungen oder -Abfragen ausführen. Das RPA-Tool hält hierzu Funktionen wie *querySql* bereit, um Datenbankabfragen auszuführen. Über einen

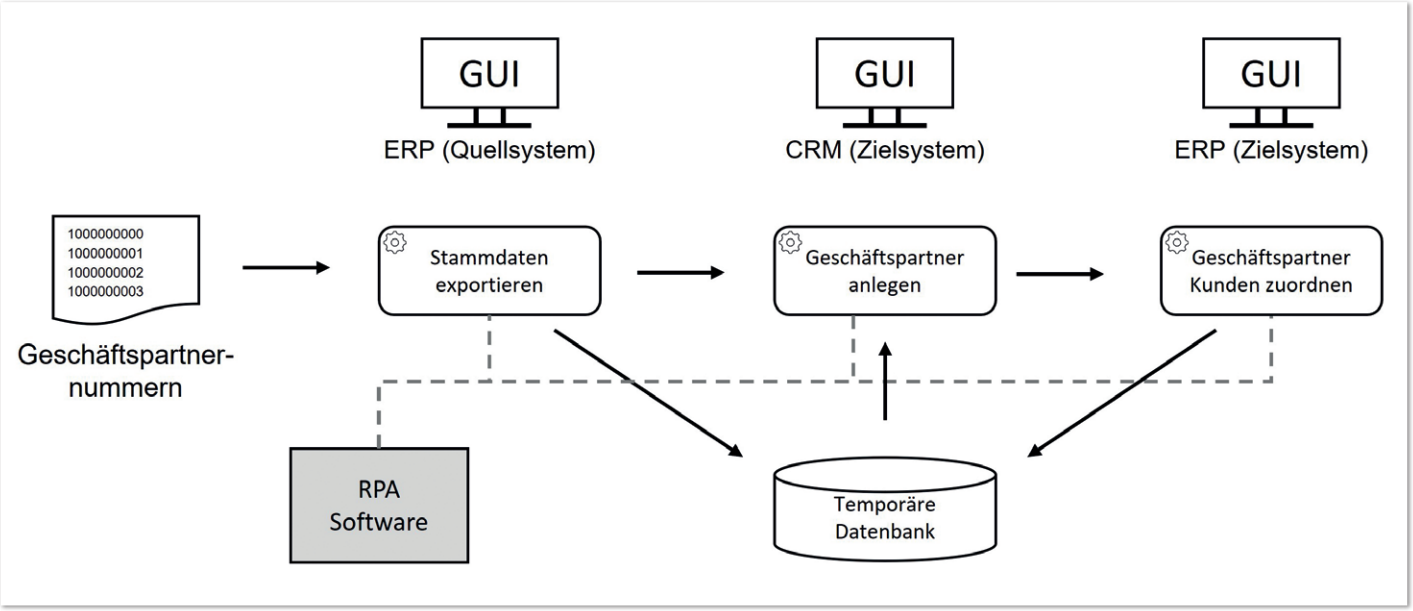


Abbildung 2: RPA-gestützter Migrationsablauf

Parameter wird die konkrete SQL-Abfrage an die Funktion übergeben. Die Rückgabewerte der Abfrage stehen im JSON-Format zur Verfügung. Das Parsen und die Weiterverarbeitung einzelner Werte lassen sich mit der Funktion *queryJson* durchführen.

Es kann festgehalten werden, dass mit einem reifen RPA-Tool verschiedenste Anwendungen gesteuert werden können, indem Automatisierungstreiber geschickt integriert werden. Für die Funktionen der verschiedenen Treiber entstehen im RPA-Tool neue Wrapper-Funktionen und Funktionsnamen. Hierdurch entsteht eine neue anwendungsspezifische Sprache (Domain-specific Language, kurz DSL). Jedes RPA-Tool besitzt seine eigene DSL zur Automatisierung von Prozessen. Zum Schreiben der DSL gibt es jedoch unterschiedliche Ansätze. Einige Tools stellen einen Editor mit einer grafischen Benutzeroberfläche bereit, um Funktionen in Form von Symbolen via Drag-and-Drop in einem Prozessdiagramm zusammenzustellen. Andere Tools sind weniger grafisch und nutzen Texteditoren wie Visual Studio Code.

Für erfahrene RPA-Entwickler/innen, die regelmäßig Prozesse automatisieren, ist letztere Variante in der Regel das Mittel der Wahl, da die Entwicklungsgeschwindigkeit hiermit deutlich erhöht werden kann. Die Nutzung eines RPA-Tools hat den Vorteil, dass nur eine Tool-spezifische Sprache geschrieben werden muss, anstatt dass mehrere kleinteilige Prozesse in unterschiedlichen Sprachen und Bibliotheken umgesetzt werden müssten.

RPA-gestützter Prozess

Der RPA-gestützte Migrationsablauf bestand im Wesentlichen aus drei Prozessschritten: Stammdaten exportieren, Geschäftspartner im neuen System anlegen und Geschäftspartner den Kunden zuordnen.

Für jeden der Prozessschritte gab es ein Skript. Zusätzlich war es notwendig, Skripte für Zwischenschritte zu entwickeln, beispielsweise um Daten zusammenzuführen und Teilmengen dieser auf die unterschiedlichen RPA-Instanzen zu verteilen. Zum Datenexport wurde der SAP-Treiber verwendet. Es wurden verschiedene Stammdaten-Tabellen heruntergeladen, in eine temporäre Datenbank importiert, bereinigt und zu einem neuen Datenmodell zusammengefügt. Das Herz der Datenbank war eine Steuerungstabelle *(siehe Tabelle 1)*, in der die Geschäftspartner-Kunden-Zuordnung vorgehalten und über die der Bearbeitungsstatus eines jeden Datensatzes nachvollzogen werden konnte.

Die Anlage der Geschäftspartner im neuen System erfolgte über das webbasierte Kundenmanagement-System. Das Skript zur Anlage von Kunden war so aufgebaut, dass pro Ausführung genau ein Kunde angelegt wurde. Das RPA-Tool hat in einer großen Schleife das gleiche Skript mehrere Tausend Mal verteilt über sechs Instanzen ausgeführt. Gleiches galt für den letzten Prozessschritt. Die Zuordnung von Geschäftspartnern zu Kunden erfolgte, nachdem alle Geschäftspartner angelegt worden waren. Dies war zugleich der Prozessschritt mit den meisten Ausführungen pro Skript, da ein Geschäftspartner mehreren Kunden zugeordnet war und dabei unterschiedliche Rollen einnehmen konnte. In der Steuerungstabelle waren die verschiedenen Kombinationsmöglichkeiten ersichtlich. Ein Datensatz entsprach der einmaligen Ausführung eines Skriptes. In der Steuerungstabelle wurde die Spalte *Status* als Kontrollmerkmal angelegt. Mithilfe der Spalte wurden die doppelte Verarbeitung einzelner Datensätze vermieden, die Rückverfolgbarkeit fehlerhafter Ausführungen sichergestellt und der Bearbeitungsfortschritt überwacht.

Herausforderung Datenqualität

Das Thema Datenqualität hatte einen großen Stellenwert bei der Migration. Bereits während der RPA-Entwicklung auf den Testsystemen wurde festgestellt, dass einige Datensätze nicht mehr den aktuellen Anforderungen entsprachen und in ihrer derzeitigen Form gar nicht migriert werden konnten. Für diese Fälle wurde vorab definiert, wie im Zuge der Migration mit diesen umzugehen sei.

Beispiel: Bei der Anlage eines neuen Kunden im Kundenmanagement-System gab es ein Dropdown-Menü zur Selektion der korrekten Anredeform. Die Auswahl der zur Verfügung stehenden Anredeformen hatte sich jedoch im Laufe der Zeit verändert. Im

ID	Geschäftspartner	Kunde	Rolle	Status
1	1000000000	3000000000	Zahlungsempfänger	Zugeordnet
2	1000000000	3000000000	Zahler	Zugeordnet
3	1000000000	3000000000	Mitteilungsempfänger	Error
4	1000000000	4000000000	Mitteilungsempfänger	Zugeordnet
5	1000000000	4000000000	Zahlungsempfänger	Bearbeitung
6	2000000000	5000000000	Zahlungsempfänger	Offen

Tabelle 1: Steuerungstabelle

Kundenmanagement-System wurde die Anrede „Lebensgemeinschaft" nicht mehr unterstützt. Im SAP-System wurde diese Anredeform jedoch weiterhin geführt. In diesem Fall wurde definiert, dass aus der Anrede „Lebensgemeinschaft" die Anrede „Herrn/Frau" werden sollte.

Trotz umfänglicher Tests und Datenanalysen im Vorfeld der Migration zeigte sich auch während der Migration, dass die zu migrierenden Daten an einigen Stellen nicht immer der Erwartungshaltung entsprachen.

Beispiel: Die Migration der Kundenstammdaten beinhaltete unter anderem die Übertragung der E-Mail-Adresse des Kunden. Ausgehend von den Anforderungen wurde erwartet, dass jedem Kunden genau eine E-Mail-Adresse zugeordnet ist. Im Kundenmanagement-System gab es ein entsprechendes Textfeld, in das diese eingetragen werden sollte. Nachdem alle zu migrierenden Daten ausgelesen worden waren, zeigte sich, dass in etwa 0,1 Prozent der Fälle zwei E-Mail-Adressen zu einem Kunden hinterlegt worden waren. Welche E-Mail-Adresse sollte nun migriert werden?

Da diese Möglichkeit während der Anforderungsanalyse und der Entwicklung der Skripte nicht bedacht worden war, wurden die entsprechenden Datensätze aussortiert und an die Fachabteilung weitergeleitet. Im Nachhinein stellte sich heraus, dass es in der Vergangenheit Überlegungen zur Hinterlegung einer zweiten E-Mail-Adresse gegeben hatte und dass die in diesem Kontext testweise modifizierten Stammdaten nicht wieder bereinigt worden waren. Diese simplen Beispiele zeigen, dass auch eine RPA-gestützte Migration viel Vorbereitung bedarf und dennoch mit Risiken verbunden ist, auf die flexibel reagiert werden muss.

Fazit

An dem vorgestellten Beispielprojekt zeigt sich, dass sich RPA als Technologie zur Datenmigration in bestimmten Fällen eignet und zu einer erfolgreichen Migration beitragen kann. Es können komplexe, anwendungsübergreifende Prozesse durch die Integration verschiedenster Automatisierungstreiber realisiert werden, wobei nur eine Tool-spezifische Sprache geschrieben werden muss. Gleichzeitig darf der Realisierungsaufwand nicht unterschätzt werden. Die Entscheidung für oder gegen die Aufnahme von RPA in die Migrationsstrategie wird wohl immer vom Einzelfall sowie von den verfügbaren Ressourcen und vorhandenen Tools abhängen.

Letztlich birgt eine RPA-gestützte Datenmigration die gleichen Herausforderungen wie eine klassische Migration. Selbst, wenn auf die Prüfroutinen des Zielsystems zur Qualitätssicherung gebaut wird, müssen die Datenbestände vorab analysiert und Regeln definiert werden, wie mit Abweichungen umzugehen ist. Anforderungsanalysen und Migrationskonzepte sind hierfür die Grundlage.

Quellen

[1] German Testing Board (2019): *Certified Tester Advanced Level Syllabus Testautomatisierungsentwickler.*

[2] *https://selenium.dev*

[3] *https://autoitscript.com*

Bastian Engelking

viadee Unternehmensberatung AG

bastian.engelking@viadee.de

Bastian Engelking ist seit 2019 als IT-Berater für die viadee Unternehmensberatung AG in den Branchen Energiewirtschaft, Versicherungen und Banken unterwegs. Seine Einsatzschwerpunkte liegen in der Business- und Anforderungsanalyse mit Fokus auf Migrationsprojekte. Seine Begeisterung für Automatisierungstechnologien führen ihn immer wieder zur Entwicklung mit RPA- und Testautomatisierungs-lösungen zurück.

Wie bekommt man Anwendungen mit Zustand in die Cloud? Round-Robin-Load-Balancing auch mit JSF

Susanne Apel und Christian Fritz, QAware

Eine alte Anwendung soll in die Cloud gehoben werden – geht das einfach so? Viele ältere, im Cluster betriebene JEE-Anwendungen und Frameworks nehmen ganz natürlich Sticky-Sessions an. In der Cloud möchte man dynamisch skalieren und Round-Robin-Load-Balancing betreiben. Das passt oft nicht zusammen. Wir zeigen am Beispiel von JSF und Apache Ignite, wie wir diesen typischen Fallstrick umschifft haben, welche Art von Herausforderungen in beiden Komponenten sowie in deren Zusammenspiel aufgetreten sind und wie wir diese der Reihe nach alle gelöst haben.


esc 1 2 3 4 5 6 7 8 9 0 - + Backspace
Tab Q W E R T Y U I O P [ ] \
Caps Lock A S D F G H J K L ; ' Enter
Shift Z X C V B N M , . / Shift
Ctrl Alt Alt Ctrl

Das Problem

In unserem letzten Projekt standen wir vor der Aufgabe, Anwendungen in eine Enterprise-Cloud auf Basis von OpenShift auf AWS zu migrieren. Bisher liefen diese als JSF-basierte Portlets auf IBM-Portalservern. Ziel war eine Cloud-Friendly-Migration *[1]*. Diese ist kein Neubau, aber auch keine Lift-&-Shift-Migration. Im Gegensatz zur letzteren Strategie wurde innerhalb der Cloud-Friendly-Migration nicht der komplette alte Ausführungskontext inklusive Portalserver in Container gesteckt und in der Cloud betrieben, stattdessen wurde der Ausführungskontext modernisiert und enthält jetzt zum Beispiel Monitoring, Hyperskalierbarkeit und eben Round-Robin-Fähigkeit. Dieses Vorhaben stellte uns vor verschiedene Herausforderungen. Auf eine der größten davon geht dieser Artikel genauer ein: das Session-Handling.

In der alten Welt werden Sessions gerne durch Sticky-Sessions im Load-Balancer unterstützt. Das bedeutet: Die Session ist nur auf einem Server des Clusters vorhanden und ein Nutzer landet mit jedem Request wieder auf genau diesem Server. Das bedeutet allerdings auch, dass jeder Neustart der Applikation zu einem Verlust der in der Session gespeicherten Daten führt. In der alten Portalserver-Welt war das in Ordnung, da Neustarts und Deployments selten und meist geplant waren. Laufen die Applikationen allerdings in der Cloud, sollten gerade Rolling-Deployments oder Neustarts nicht zu Problemen führen. Ansonsten würde das zu Problemen mit den Resilienz- und Skalierungs-Mechanismen der Cloud führen: Hier sollen kurzfristige Ausfälle schnell kompensiert werden können, es soll nach Bedarf hoch- und runterskaliert werden können. Cloud-native Anwendungen nehmen implizit ein Round-Robin-Load-Balancing an. Sticky-Sessions sind technisch prinzipiell auch in der Cloud möglich, jedoch oft architektonisch nicht sinnvoll. Im konkreten Fall waren die Umsetzungsmöglichkeiten für Sticky-Sessions aufgrund von Sicherheitsvorgaben sowie die konkrete Umsetzung der Landing-Zone nicht nutzbar.

Zur Lösung: Wenn verschiedene Instanzen einer Anwendung auf die gleichen Informationen konsistent zugreifen sollen, dann ist es eine Möglichkeit, diese Information – den Sitzungszustand – zentral zu speichern. Damit auf diesen auch wirklich zugegriffen wird, muss man die Sitzungsverwaltung im ausführenden Container, wie Liberty, GlassFish oder Tomcat, aufbohren, um auf den zentralen Sitzungszustand zugreifen zu können.

Prinzipiell gibt es für dieses Problem verschiedene einsatzfähige Lösungen. So können Apache Tomcat und dessen JEE-Variante TomEE bereits Session-Clustering durchführen. Allerdings setzen sie ein „Trusted Network" voraus *[2]*. Damit sind die einzelnen Kommunikationswege zwischen den beteiligten Knoten nicht verschlüsselt. Das war allerdings eine zentrale Sicherheitsanforderung des Kunden: Alle Kommunikationsverbindungen müssen mit TLS 1.2 geschützt sein.

Um nicht auf einen bestimmten Application-Server angewiesen zu sein, bieten sich In-Memory-Data-Stores wie Hazelcast *[3]*, Redis *[4]* und Apache Ignite *[5]* an. Im konkreten Fall hatte sich bereits das Vorgänger-Programm für Apache Ignite entschieden. Die Gründe dafür: Bei Hazelcast waren zu diesem Zeitpunkt in der kostenlosen Version die Verbindungen nicht verschlüsselt. Redis passte dagegen nicht zum Betriebsmodell das Unternehmens, da man einen zentralen Anwendungsbetrieb hätte aufbauen müssen.

Unsere Lösung: Apache Ignite

Apache Ignite: Grundsätzliche Funktionsweise

Ignite unterstützt explizit das Session-Handling, siehe *[6]* für die kürzere neuere Dokumentation und *[7]* für eine ältere, ausführlichere Dokumentation. Wie zuvor beschrieben, brauchen wir die beiden Bestandteile einer zentralen Datenhaltung und einer Integration in den Ausführungskontext der Anwendung.

Zu letzterem Punkt wird WebsessionFilter von Ignite *[8]* verwendet. Die Kernidee dieses Servlet-Filters ist es, die Session-Instanz des Webservers durch eine Instanz auszutauschen, die die verteilte Ignite-Session verwendet und nach Ende des Request ein Update in den verteilten Ignite-Cache schreibt. Für die Datenhaltung im Ignite-Cache gibt es zwei Möglichkeiten: zum einen eine Embedded-Version, bei der eine Ignite-Instanz zusammen mit der Anwendung hochgefahren wird; alternativ kann auch ein Stand-alone-Ignite-Cluster betrieben werden. Mehr zu unserer Wahl im Abschnitt „Ignite als Stand-alone-Instanz (Grid)".

Technischer Projektkontext der Migration

Um besser beschreiben zu können, auf welche Probleme wir gestoßen sind, gehen wir im Folgenden auf den technischen Projektkontext ein. Bei der Migration wurde nicht nur eine einzelne Anwendung migriert, sondern eine ganze Reihe mehr oder weniger ähnlicher Anwendungen. Diese waren in einem ähnlichen Zeitraum entstanden und nutzten überwiegend dieselben (teils selbstgebauten) Frameworks.

Unser Entwicklungsteam war unter anderem dafür verantwortlich, zentrale Komponenten für die Migration der einzelnen Anwendungen zur Verfügung zu stellen und bei komplizierten Integrationen zu unterstützen. Die technische Ausführungsumgebung war bisher JEE mit Spring-Anteilen im IBM-Portalserver. Durch die Migration wurde auf Spring Boot *[9]* umgestellt, da das der Unternehmensstandard für Neuentwicklungen ist. Die Tatsache, dass wir unsere Lösungen in verschiedenen Anwendungen beobachten konnten, hat bei der Aufdeckung und Behebung der unten beschriebenen Probleme sehr geholfen.

Die tiefe Grube und der Weg hinaus

Im Folgenden beschreiben wir einige auftretende Probleme und deren Lösung. Der Fokus liegt auf der Beschreibung der technischen Ursache der Probleme und ihrer Wirkmechanismen. Die Probleme waren größtenteils unabhängig voneinander, wir beschreiben sie hier der Reihe nach.

WebsessionFilter und Spring Boot

Die zuvor beschriebene grundsätzliche Funktionsweise des WebsessionFilter legt nahe, dass man diesen so konfigurieren muss, dass er den Haupteinstiegspunkt in die JSF-Anwendung – das FacesServlet – umschließt. Damit alles funktioniert, muss man die richtige Reihenfolge der Filter beachten. Insbesondere muss dies mit dem Session-Handling von Spring Boot zusammenpassen.

Will man in Spring Boot in allen Threads Zugriff auf die Session haben (nicht nur in dem, der den Request beantwortet), so kann man einen von drei von Spring Boot vorgesehenen Mechanismen verwenden: RequestContextListener, RequestContextFilter und DispatcherServlet. Diese sind für Spring Boot gleichwertig. Soll gleich-

zeitig aber der WebsessionFilter verwendet werden, so muss der RequestContextListener unbedingt vermieden werden: Da er ein Listener ist, wird er vor allen Filtern aufgerufen. Würde der Listener verwendet, so würde immer auf die originale Tomcat-Session zugegriffen werden, da der Filter sie noch nicht durch eine Ignite-Session ersetzen konnte.

In unseren Anwendungen haben wir uns für den RequestContext-Filter entschieden und diesen nach dem WebsessionFilter um das FacesServlet konfiguriert. Beachte: Um für alle Filter eine korrekte Reihenfolge zu erhalten, sollte auch der Dispatcher-Type betrachtet werden. Zudem sieht man in der Implementierung von org.apache.catalina.core.ApplicationFilterFactory#createFilterChain, dass es in der Reihenfolge der Filter Unterschiede zwischen Registrierung auf einen URL-Pfad und auf einen Servlet-Namen gibt.

Konsistente Session-Daten, Transaktionen – oder: Parallele Requests auf die gleiche Session verhindern

Da die Session per Konzeption der Master für den Anwendungszustand eines Nutzers ist (beziehungsweise aus technischer Sicht: einer Browser-Session), muss diese unbedingt konsistent gehalten werden. Konzeptionell möchte man eigentlich Transaktionssicherheit für Änderungen auf den Session-Daten. Eine echte technische Umsetzung von Transaktionen ist problematisch und nicht nötig, wenn man durch zeitliche Entzerrung sicher ist, dass sich keine Nutzeraktionen überschneiden. Das Problem trat in der alten Welt mit Sticky-Sessions so nicht auf, da hier die Session in einer Anwendungsinstanz gehalten werden konnte und so auch gleichzeitige Manipulationen ohne Fehler bleiben konnten. Finden in der neuen Welt gleichzeitige Manipulationen beispielsweise auf zwei Instanzen statt, wird beim Schreiben in Ignite nur die neuere Manipulation übernommen (Lost Update). Grundsätzlich trifft das auf alle Anwendungen zu, bei denen das Lesen und Schreiben der Session eines Request nicht in der gleichen Transaktion stattfinden. Daher – und aufgrund des Round-Robin-Balancing – müssen parallele Requests des gleichen Nutzers unbedingt vermieden werden.

Tabs, die ebenfalls potenziell parallele Requests ausführen, waren in unseren Anwendungen kein großes Problem, da Tabs von den Nutzern auch zuvor schon nicht verwendet werden konnten: Die Anwendungen stellten im neuen Tab immer den letzten Session-Zustand wieder her. Meistens sogar unabhängig davon, welche XHTML-Seite aufgerufen wurde.

Konkret verblieben bei uns zwei Situationen, in denen parallele Requests aufgetreten sind:

Ressourcen-Requests

JSF fragt unter /javax.faces.resource/jsf.js.xhtml potenziell dynamisch generiert Ressourcen an. Wir hatten Anwendungen, bei denen auch in diesem Fall auf die Session zugegriffen wurde, zum Beispiel durch einen Filter, der auf *.xhtml-Pfade registriert war. Damit liefen wir genau in die Probleme der parallelen Requests. Dieses Session-Handling war fachlich nicht notwendig und somit bestand die Lösung darin, diesen Code für jsf.js.xhtml nicht laufen zu lassen.

Buffering Filter

Im Normalfall liefert eine JSF-Anwendung eine HTML-Seite in Teilen schon aus, bevor die serverseitigen Berechnungen vollständig abgeschlossen sind. Theoretisch kann im bereits gerenderten Anteil also schon eine Nutzer-Interaktion stattfinden, bevor die serverseitige Behandlung abgeschlossen ist. Insbesondere kann damit ein neuer Request erfolgen, bevor das Update auf die erste Session nach Ignite geschrieben wurde. Um das zu vermeiden, haben wir einen Buffering-Filter implementiert, der das HTML erst dann an den Browser sendet, wenn das Ignite-Session-Handling abgeschlossen ist, mit entsprechend erhöhter Time-to-First-Byte.

Objekt-(De-)Serialisierung und Objekt-Vervielfältigung

Vergleicht man die alte und die neue Ausführungsumgebung, so fällt auf: In der alten Sticky-Session-Welt wurden die Sessions nur im Speicher jeweils einer Instanz gehalten. Durch das Verteilen der Sessions in der Cloud muss ein Austauschformat verwendet werden. Wir haben den standardmäßig vorgesehenen Binary Marshaller verwendet. Auch bei anderen Marshallern ließen sich Probleme nachstellen, die ähnlich zu den unten beschriebenen waren. Da die gesamte Session synchronisiert werden muss, sollte diese möglichst klein gehalten werden. Insbesondere sollte beispielsweise kein ganzer ApplicationContext im Feld einer Session-Scoped-Bean gehalten werden.

Wir haben beobachtet, dass es in Ignite zu einer Objektvervielfältigung nach einer Deserialisierung kommen kann. Das passiert dann, wenn es im Objektbaum der Session mehr als einen Pfad zu einer Bean gibt. Als Beispiel siehe *Abbildung 1*. Hier wird Bean A von der Session selbst und von Bean B referenziert. Beim Deserialisieren wird das Feld in Bean B eine andere Kopie von Bean A enthalten als die in der Session aufbewahrte Bean A. Das ist unbedingt zu vermeiden. Der Code der Anwendung geht davon aus, dass es sich um dieselbe Instanz handelt. Wird eine Änderung nach der Deserialisierung an einer Stelle geschrieben, so kommt die Änderung nicht in der Kopie an.

Dass das kein grundsätzliches Problem von Serialisierung ist, zeigt die Dokumentation von ObjectOutputStream: „Multiple references to a single object are encoded using a reference sharing mechanism so that graphs of objects can be restored to the same shape as when the original was written." *[10]*

In der Praxis wird eine problematische, mehrfach referenzierte Bean A Application-Scoped, Session-Scoped oder View-Scoped sein.

Um diese Probleme zu vermeiden, haben wir die folgenden Workarounds in die Anwendungen eingebaut: Im Wesentlichen haben wir auf Lazy-Loading statt Ablage in Feldern von Beans gesetzt. Das Refactoring-Ziel lässt sich so beschreiben: Jede Session-Scoped oder View-Scoped Bean B wird durch Ignite automatisch selbst als Top-Level-Element in der Session gespeichert. B darf kein transitives (nicht-transientes) Feld auf eine Bean A von Scope Session, View oder Application haben. Muss B auf A zugreifen, so haben wir ein Lazy-Loading etwa über den JSF-Context oder den Spring ApplicationContext eingebaut. Konkret haben wir jeden Feld-Zugriff in einen (privaten) Getter gekapselt. Im Getter wird der Wert aus dem gewählten Kontext ermittelt. Die Ermittlung des Kontexts selbst kann in eine Dependency-Injection-fähige Komponente gekapselt werden. So kann man auch Testbarkeit gewährleisten. Leider macht das Dependency-Injection deutlich weniger komfortabel. Für eine saubere Lösung wäre ein fachliches Refactoring

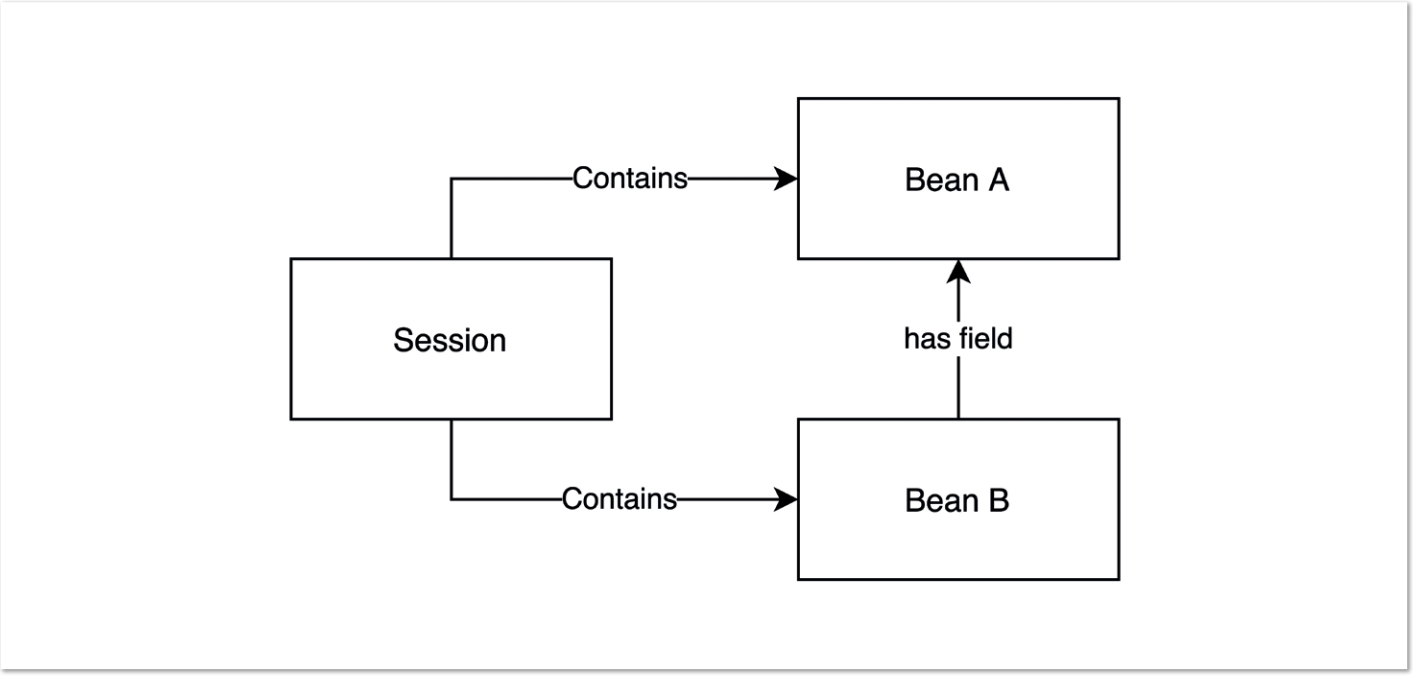


Abbildung 1: Ausgangssituation vor dem Serialisieren der Session und vor dem Refactoring (© Christian Fritz)

und ein anderer Schnitt der Beans vom Scope Session und View nötig gewesen. Bei einer guten Kapselung von Session-Scoped- und View-Scoped-Daten würde man nur selten in dieses Problem laufen.

Für einen effizienten Umbau haben wir Spoon *[11]* eingesetzt. Spoon ist ein mächtiges Framework zur automatisierten Anpassung von Java-Code. Dabei arbeitet Spoon auf dem Abstract Syntax Tree (AST) des Java-Codes und schreibt die Änderungen direkt in den Source-Code. Mit Spoon haben wir nach allen Zugriffen auf Session- und View-Scoped Variablen gesucht und diese durch einen Lookup im FacesContext ersetzt. Dadurch konnten wir uns sicher sein, wirklich alle potenziell gefährdeten Zugriffe zu finden und das Refactoring in allen Anwendungen, wenn nötig auch mehrfach, auszuführen.

Als mögliche Alternative sei Spring Session *[12]* erwähnt. Wir kennen keine Spring-Session-Integration in Ignite, die sich selbst als produktionsreif bezeichnet. Ob das Problem mit Redis auch auftritt, haben wir nicht untersucht.

JSF: Exceptions in der RestoreView-Phase

Gelegentlich sahen wir in mehreren der migrierten Anwendungen schwer einzuordnende Phänomene:

Wir beobachteten ClassCastExceptions und ArrayIndexOutOfBoundsExceptions in der JSF-Phase RestoreView. Die nächsten beiden Abschnitte beschreiben unsere Ansätze, mit diesem Problem umzugehen.

Ignite als Stand-alone-Instanz (Grid)

Zu Beginn hatten wir aufgrund der Erfahrungen im Vorgänger-Projekt und des vermeintlich einfacheren Betriebs Ignite in der Embedded-Version betrieben. Das heißt, die Datenhaltung fand in den Anwendungsprozessen statt, diese synchronisieren sich untereinander.

Als wir die Exceptions in der RestoreView-Phase gesehen hatten, hielten wir eine Race-Condition beim Lesen und Schreiben der View für wahrscheinlich. Daher betrachteten wir den im Aufbau befindlichen Lasttest genauer und dieser stellte sich als Reproducer für dieses Verhalten heraus. Bei Verlangsamung der Anwendung (etwa durch Debugger) traten die Fehler nicht mehr auf. Alles sprach für eine Race-Condition.

Daher testeten wir eine Ignite-Stand-alone-Lösung: Ignite wird als Grid mit zwei Instanzen separat betrieben. Danach traten die Probleme im Reproducer des Lasttests nicht mehr auf. Weiter Vorteile durch das Ignite-Stand-alone-Grid:

- Bei einem Nicht-Rolling-Neustart der Anwendung gehen die Sessions nicht mehr verloren
- Wird im Betrieb über außerordentliche Aktionen nachgedacht, so ist die Datenhaltung im Ignite-Deployment explizit sichtbar und wird weniger wahrscheinlich vergessen
- Bessere Skalierbarkeit der Anwendung: Bei der Verwendung von Ignite sollten nur wenige Instanzen verwendet werden, da sonst der Synchronisationsaufwand steigt (siehe auch Split-Brain-Gefahr im Abschnitt Ignite: Datenkonsistenz). Entkoppelt man Ignite von der Anwendung, kann man die Anwendung unabhängig skalieren

Man beachte, dass bei einem Ignite-Versionshub alle Ignite-Komponenten heruntergefahren und mit der neuen Version wieder hochgefahren werden müssen. Das betrifft hier also alle Knoten des Grid sowie gleichzeitig die Anwendungen, die einen Ignite-Client integriert haben. Da Ignite auch auf allen Komponenten die gleiche Konfiguration fordert, gilt dies auch bei Konfigurationsänderungen.

Selten auftretende persistente Probleme im JSF-State und JSF-Id-Mapper als Ursache für Exceptiones in der RestoreView-Phase

Auch nach dem Einsatz des Ignite-Grid kam es in anderen Konstellationen vereinzelt zu `ClassCastExceptions` und `ArrayIndexOut-`

```
/**
 * Used to provide aliases to Facelets generated unique IDs with tend to be
 * somewhat long.
 */
public class IdMapper {

    private Cache<String,String> idCache = new Cache<>(new IdGen());


    public String getAliasedId(String id) {
        return idCache.get(id);
    }

    private static final class IdGen implements Cache.Factory<String,String> {

        private AtomicInteger counter = new AtomicInteger(0);

        @Override
        public String newInstance(String arg) throws InterruptedException {

            return 't' + Integer.toString(counter.incrementAndGet());

        }

    }
}
```

Listing 1: Gekürzte Implementierung von com.sun.faces.facelets.impl.IdMapper [13]

`OfBoundsExceptions`. Diese waren sehr sporadisch. Zu beobachten war folgendes Fehlerbild: Tauchte der Fehler einmal auf, dann erschien er nutzerübergreifend an der gleichen Stelle beziehungsweise auf der gleichen Seite wieder. Durch einen Anwendungsneustart verschwand dieser Fehler. Das Problem war sehr schwierig festzunageln, weil wir es phasenweise überhaupt nicht gesehen haben und es nicht reproduzieren konnten. Letztendlich haben wir die Lösung im com.sun.faces.facelets.impl.IdMapper gefunden.

Die erzeugten Ids unterscheiden sich prinzipiell in ihrer Struktur danach, ob partial- oder full-state-saving aktiviert ist. Gemeinsam haben beide Modi, dass für die an sich eindeutige Facelet-Id eine html-id generiert wird. Dabei wird zur Vergabe ein Atomic Integer hochgezählt. Man sieht das gut in *Listing 1*.

Um die Auswirkungen dieses Id-Vergabe-Vorgehens zu verstehen, ist es wichtig, zu betrachten, wie lange der `IdMapper` lebt, also was sein Scope ist. Anders als die im `IdMapper` sichtbare Interaktion mit dem FacesContext vermuten lässt, lebt ein IdMapper anwendungsweit (siehe `IdMapper#getMapper(FacesContext ctx)`, `DefaultFacelet`, `DefaultFaceletFactory#idMappers`) und nutzerübergreifend für einen Ressourcen-Alias (also beispielsweise den Pfad auf eine xhtml-Seite). Insbesondere werden die IdMapper aber nicht zwischen zwei Anwendungsinstanzen geteilt. Das bedeutet, dass es prinzipiell denkbar ist, dass Ids in beiden Instanzen unterschiedlich vergeben werden.

Dafür müssen ein paar Dinge zusammenkommen, denn der erste Request auf eine Instanz muss sich vom ersten Request auf die andere Instanz unterscheiden – so stark, dass verschiedene Ids vergeben werden. Das kann beispielsweise passieren, wenn ein Tag-Handler wie `c:if` verwendet wird und dieser genau beim ersten Request auf die eine Instanz zu `true` auswertet und im ersten Request auf die andere Instanz zu `false`. Dann sieht der resultierende JSF-Komponentenbaum in beiden Requests auf den beiden Instanzen unterschiedlich aus und die Komponenten werden unterschiedlich durchnummeriert.

Für dieses Beispiel der abweichenden Ids müssen Tag-Handler und Komponenten ungünstig kombiniert werden. Für den Unterschied siehe *[14]*. Die Probleme waren selten, da die allerersten Requests auf mehreren Instanzen unterschiedlich sein müssen.

Wir haben das Problem gelöst, indem wir den `IdMapper` gepatched haben und in der Methode `getAliasedId` genau den eindeutigen Input als aliased Id zurückgeben.

Ignite: Connection Handling

Wir sind am Ende der Migration in ausgeprägte Performance-Probleme gelaufen. Anfragen an das Grid hatten häufiger Laufzeiten von über drei Minuten. Im Dynatrace beobachten wir, dass die Requests in solchen Lastphasen umso länger dauerten, je älter sie waren. Sie häuften sich auch genau auf einer Instanz. Wir vermuten, dass die Implementierung im GridFutureAdapter letztendlich dafür sorgt, dass Requests pro Connection auf einen Stack gelegt werden. Damit wird der erste Request zuletzt beantwortet. Gleichzeitig ist per Default nur eine Connection pro Node aktiv. Um das Problem zu umgehen, haben wir mehr Connections konfiguriert.

- Number of connections per node: 5. Das ist kein dokumentierter Konfigurationswert, wir haben zur Umsetzung Java-Code geschrieben, um die Instanz der internen Klasse TcpCommunicationConfigInitializer umzukonfigurieren
- Use paired connections: true (siehe Network [15]). So werden aus- und eingehende Nachrichten eines Knotens über verschiedene Verbindungen abgewickelt und wir sind möglichst unabhängig von gleichzeitigen Ereignissen. Wir wollen Bottleneck bei den Verbindungen vermeiden

Ignite: Datenkonsistenz

Neben dem Tuning der Verbindungen zwischen den einzelnen Knoten mussten wir uns auch noch um die Datenkonsistenz kümmern. Dazu gehören: Schutz vor Split-Brain-Szenarien, die Time-to-Live von Einträgen und das Handling von Evictions.

Split-Brain bedeutet, dass mindestens ein Knoten, der Daten speichert, die restlichen Datenknoten nicht mehr erreichen kann. Das führt dazu, dass die beiden Teilgruppen jeweils unterschiedliche Stände als den aktuellen Stand ansehen und diese somit nicht mehr zusammenpassen. Prinzipiell gibt es verschiedene Möglichkeiten, darauf zu reagieren. Apache Ignite bietet in der Open-Source-Variante leider keine komplexeren Mechanismen wie Majority-Voting an. Wir haben uns daher für die denkbar einfachste Lösung entschieden: einen Knoten neu zu starten (siehe *[16]*). Damit wird der andere Knoten automatisch zum Master. Dabei besteht die Gefahr, Sitzungsdaten zu verlieren. Das nahmen wir bewusst in Kauf, da die Auswirkungen in unseren Anwendungsfällen in seltenen Fällen verschmerzbar waren.

Die Time-to-Live ist eine einfache Einstellung. Diese sollte mit der maximalen Laufzeit der Sessions übereinstimmen. Somit werden invalide Sessions zeitnah aus dem Speicher von Ignite entfernt und verbrauchen keinen Speicher.

Die letzte wichtige Einstellung ist das Handling von Daten, sollte der konfigurierte In-Memory-Speicher nicht mehr ausreichen, um neue Daten zu speichern. Dazu bietet Ignite mehrere Mechanismen an, um einzelne Daten-Seiten aus dem Speicher zu entfernen. Eine Seite kann, je nach Größe, Teile einer Session bis zu mehreren Sessions enthalten. Bei unserer Session-Größe gehen wir von einer betroffenen Session aus. Wir haben uns für den Random-2-LRU-Mechanismus entschieden, der von fünf zufälligen Seiten die mit der geringsten Zugriffshäufigkeit entfernt.

Ignite: Konfiguration der Datenaufteilung

Insgesamt muss Ignite so konfiguriert werden, dass es gut zum gewünschten Use Case passt. Wir haben das Ignite-Cluster standardmäßig mit zwei Instanzen betrieben. Manche Einstellungen werden erst bei mehr Instanzen interessant.

Es stellt sich die Frage, wie die Daten auf die Instanzen verteilt sind. Einen guten Überblick bieten *[17]* und *[18]*. Wir beschreiben im Folgenden und auch anhand der Konfigurationsmöglichkeiten die Funktionsweise und unsere Entscheidungen. Man beachte, dass es gerade bei (anderen) Einstellungen, die zu hohem Synchronisationsaufwand führen, bei mehr Instanzen zu längeren Gesamt-Antwortzeiten kommen kann.

- `cacheMode: PARTITIONED`: Die Daten werden auf die verschiedenen Instanzen verteilt, jedoch im Gegensatz zu REPLICATED nicht auf alle
- `Number of Backups: 1` (default: 0, nur im `cacheMode PARTITIONED` relevant, bei `REPLICATED` gilt: #backups= ∞)
- Wir wollen ausfallsicher gegenüber dem Ausfall einer Ignite-Instanz sein.
- `writeSynchronizationMode: PRIMARY_SYNC (default)`
- Zu jedem Datensatz gibt es einen Primärknoten. Die zuvor konfigurierten Backups sind echte Backups. Mit `PRIMARY_SYNC` als Synchronisationsmodus wartet Ignite beim Schreiben, bis der Vorgang auf dem Primärknoten vollendet ist, nicht aber bis das Update in den Backups angekommen sind. Das bringt insgesamt mit sich, dass Backups potenziell minimal veraltet sind
- `Read from backups: false`
- Backups werden durch den Synchronisationsmodus asynchron geupdatet. Daher soll nicht von diesen gelesen werden, um zu vermeiden, dass veraltete Daten gelesen werden
- Wir hatten es mit sehr großen Sessions zu tun. Damit Ignite weniger Verwaltungsaufwand hat, haben wir die Page Size, in deren Blockgröße die Daten in Seiten geschrieben werden, auf das Maximum 16384 (Bytes) gesetzt *(siehe [19])*

Fazit

Der Umzug von so alten Anwendungen auf die Cloud inklusive Round-Robin-Load-Balancing ist möglich, aber nur dann sinnvoll, wenn ein Neubau in absehbarer Zeit nicht umsetzbar ist. Allgemeiner ausgedrückt kann jede verwendete Komponente in der alten Anwendung implizite Annahmen über die Betriebsumgebung enthalten. Das kann auch gut „abgehangene" Komponenten wie JSF betreffen. Es ist eine Herausforderung, diese zu erkennen und abzuwägen, ob sich eine Migration lohnt.

Dafür wird man mit einer Anwendung belohnt, die nun auf einer modernen Cloud-Infrastruktur läuft, deren Vorteile man nutzen und resilient auf Probleme reagieren kann. Daneben ist diese Art einer Migration meist günstiger als ein kompletter Neubau oder der Weiterbetrieb auf der alten Infrastruktur. In unserem Projekt haben sich die Kosten der Migration nach kurzer Zeit durch die gesunkenen Betriebs- und Lizenzkosten amortisiert, und das bei einem deutlichen Stabilitätsgewinn.

Insgesamt war dieses Projekt eine schöne Bestätigung für die Cloud-Friendly-Migrationsstrategie.

Quellen

[1] QAware (2021): *https://info.qaware.de/cloud-friendly-migration*
[2] Apache Tomcat 9: *https://tomcat.apache.org/tomcat-9.0-doc/cluster-howto.html*
[3] Hazelcast: *https://hazelcast.com*
[4] Redis: *https://redis.io*
[5] Apache Ignite: *https://ignite.apache.org*
[6] Apache Ignite: Web Session Clustering (aktuelle Documentation): *https://ignite.apache.org/use-cases/caching/web-session-clustering.html*
[7] Apache Ignite: Web Session Clustering (Legacy Documentation): *https://apacheignite-mix.readme.io/docs/web-session-clustering*
[8] Apache Ignite: WebSessionFilter *https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/cache/websession/WebSessionFilter.html*
[9] Sprint Boot: *https://spring.io/projects/spring-boot*
[10] ObjectOutputStream: *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/ObjectOutputStream.html*
[11] Spoon: *https://spoon.gforge.inria.fr*
[12] Spring Session: *https://spring.io/projects/spring-session*
[13] IdMapper.java: *https://github.com/eclipse-ee4j/mojarra/blob/2.3.15/impl/src/main/java/com/sun/faces/facelets/impl/IdMapper.java*
[14] Roger Keays: *https://rogerkeays.com/jsf-c-foreach-vs-ui-repeat*
[15] Ignite: Network Configuration: *https://ignite.apache.org/docs/latest/clustering/network-configuration*

[16] Ignite: Segmentation Policy: *https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/configuration/IgniteConfiguration.html#setSegmentationPolicy-org.apache.ignite.plugin.segmentation.SegmentationPolicy-*
[17] Ignite: Data Partitioning: *https://ignite.apache.org/docs/latest/data-modeling/data-partitioning*
[18] Ignite: Configuring Caches: *https://ignite.apache.org/docs/latest/configuring-caches/configuring-backups*
[19] Ignite: Werte für die Page Size: *https://ignite.apache.org/releases/latest/javadoc/org/apache/ignite/configuration/DataStorageConfiguration.html#setPageSize-int-*

Susanne Apel
QAware
susanne.apel@qaware.de

Susanne Apel ist Software Architect bei QAware. Sie war technischer Lead im beschriebenen Migrationsprojekt, vertieft sich gerne in kniffelige, ungewöhnliche Probleme und setzt alles zu einer Lösung zusammen. Susanne hat an der TU München Mathematik mit Nebenfach Informatik studiert und in der Geometrie promoviert.

Christian Fritz
QAware
christian.fritz@qaware.de

Christian Fritz ist Senior Software Engineer bei QAware. Er war Entwickler im beschriebenen Migrationsprojekt und beschäftigt sich intensiv mit dem Bau Cloud-nativer Anwendungen und der Sicherheit in der Cloud.

Döner in the Cloud

Thomas Michael, GOD mbH

Vom Bestellzettel über eine Spring-Boot-App hin zur Serverless-Cloud-App. In diesem Artikel werden zwei Migrationen aus dem Büroleben dargestellt. Anfangs wird eine manuelle Dönerbestellung für die Kollegen digitalisiert und danach für eigene Testzwecke zur Verfügung gestellt. Am Ende wird die Anwendung in die Cloud migriert und serverless zur Verfügung gestellt.

Früher, als man noch mit den Kollegen gemeinsam im Büro saß – die Älteren werden sich erinnern –, hatten wir ein Donnerstagsritual. Jeder Donnerstag war Dönerstag und es wurde zum Mittagessen Döner oder Ähnliches bestellt. Anfangs konnte man sich die drei Bestellungen noch merken, fuhr persönlich zum Dönermann seines Vertrauens und brachte den Kollegen die gewünschte Ware mit. Sehr schnell stieg das Nachbarbüro mit ein und so wurde der Bestellzettel für Döner geboren. Bei der Menge konnte man sich den Döner sogar liefern lassen, man musste nur dort anrufen! Dann ging ein Kollege mit einem Zettel bewaffnet durch die Büros, nahm die Bestellungen auf, sammelte das Geld ein und bestellte anschließend das begehrte Gericht. Da immer mehr Büros mitmachten und am Ende sogar die nächste Büroetage, wurde das Aufnehmen der Bestellungen immer langwieriger und anstrengender.

Vom Bestellzettel zur Applikation

Da wir ein Dienstleister für Softwareentwicklung sind, lag es nahe, anstelle eines Bestellzettels eine Anwendung zu schreiben. Aber: in welcher Sprache, welche Technologie, welches Backend und welches Frontend? Der Zufall wollte es, dass sich der Autor in die neue Version von Angular einarbeiten sollte. Zu dem Zeitpunkt wurde Angular 4 veröffentlicht, das sich erheblich von Version 1 unterscheidet. Also lag es nahe, für das Frontend Angular 4 zu verwenden. Ähnlich lief die Entscheidung für das Backend. Motiviert durch den Leitspruch „Make jar not war" arbeitete sich der Autor, der es bisher als J2EE/

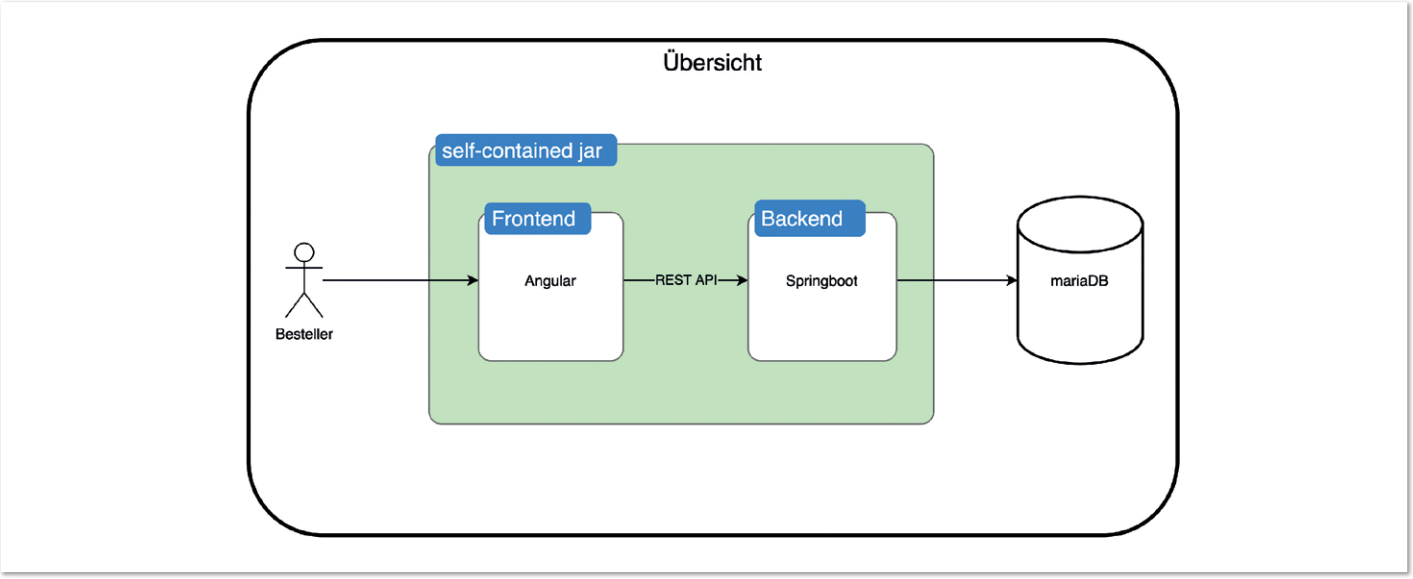


Abbildung 1: Übersicht (© Thomas Michael)

JEE-Entwickler gewohnt war, EAR/WAR-Files für einen Server zu erstellen, in das – damals recht neue – Spring-Boot-Framework ein. Das ergab eine übersichtliche Architektur *(siehe Abbildung 1)*.

Gestartet wurde mit zwei Anwendungsfällen:

1. Abgeben einer Bestellung
2. Übersicht über alle Bestellungen

Dadurch ergab sich im Backend ein sehr übersichtliches REST-API *(siehe Listing 1)*.

```
GET https://localhost:8080/bestellungen/
POST https://localhost:8080/bestellungen/
```

Listing 1: REST-API der Döner-App

In Angular wurden dafür zwei Seiten erstellt, eine zum Anlegen einer Bestellung *(siehe Abbildung 2)* und eine als Übersicht über alle Bestellungen.Eine Bestellung ist ein einfaches JSON-Objekt *(siehe Listing 2)*, das an das REST-API per POST gesandt wird.

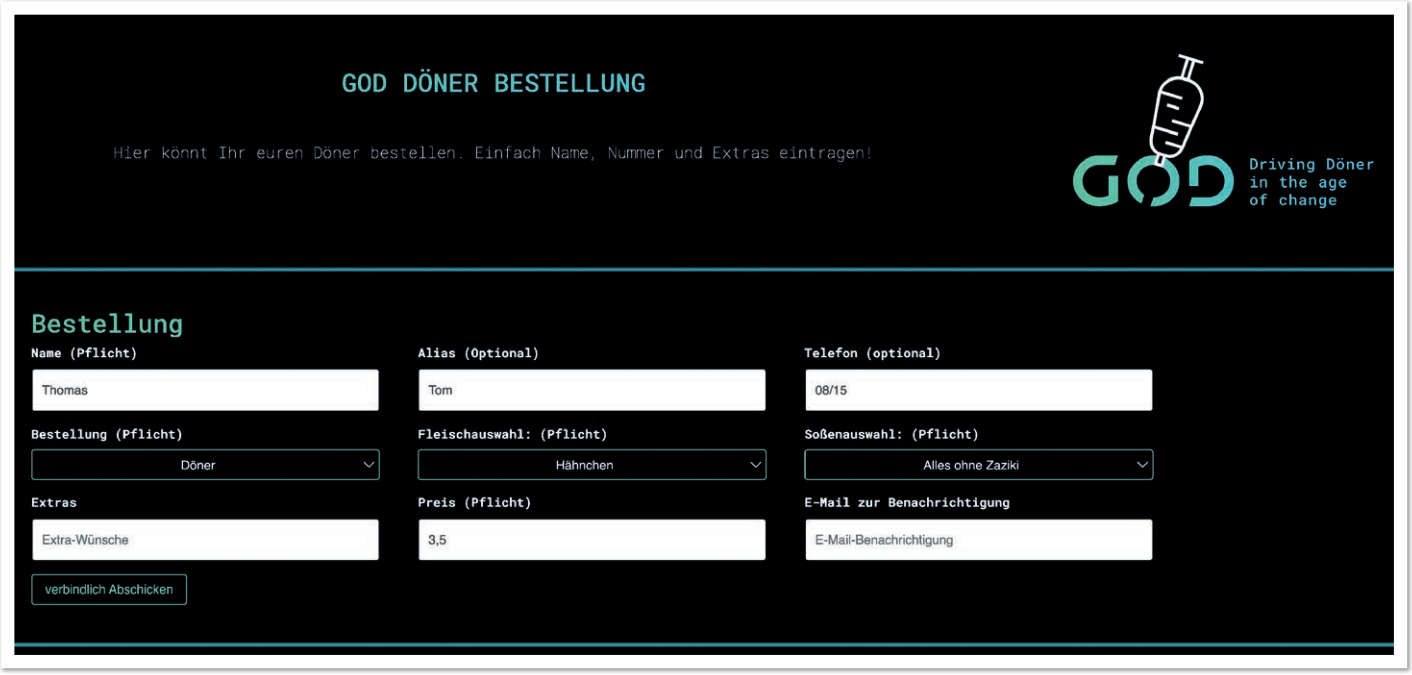


Abbildung 2: Bestellseite der Döner-App (© Thomas Michael)

```
{"bestellung":"Doener","name":"Thomas","alias":"Tom","telefonnummer":"08/15","fleisch":"Haehnchen",
"sauce":"Alles ohne Zaziki","price":3.50}
```

Listing 2: Eine Bestellung im JSON-Format

Server unter dem Schreibtisch

Anfangs reichte es, wenn jeden Donnerstag die Anwendung und das Angular-Frontend einfach über die Entwicklungsumgebung gestartet wurden. Die App sollte zu Beginn nur dem Besteller des Döners helfen. Die Kollegen füllten die Bestellseite aus, Geld wurde eingesammelt und über die Übersichtseite konnte sehr einfach bestellt werden. Für den Anwendungsfall, Döner zu bestellen, genügte dies, aber war das eine würdige Lösung für ein Büro voller Softwareentwickler? Wer hat denn schon erlebt, dass eine „echte" Anwendung im Büro unter dem Schreibtisch läuft, statt auf dem eigenen Server? Bitte nicht die Hand heben, dies ist eine rhetorische Frage!

Mittlerweile gab es die ersten Verbesserungsvorschläge für die App. Ein Server und eine eigene Datenbank mussten her. Alles, bis hin zum Deployment, sollte möglichst automatisiert werden, so wie man sich das immer wünscht, aber nur selten erlebt. Damit hatten wir die ersten DevOps-Anforderungen an unsere Döner-App:

- einen Server,
- eine Datenbank,
- einen Build-Job für unseren Jenkins,
- einen Deploy-Job für Jenkins.

Unglaublich, wie schnell DevOpsler arbeiten können, wenn es um Döner geht! Die Döner-App wurde um Maven erweitert, das war aufgrund der Erfahrung mit Maven am einfachsten. Mithilfe des Maven-Plug-ins von Eirslett *[1]* wurde das Angular-Frontend gleich mitgebaut. Dabei wird das Plug-in so konfiguriert, dass das erstellte HTML und JavaScript direkt in den Server kopiert wird. Für statische Webseiten ist dies der `static`-Ordner auf dem Server. Einfach im `Targetpath` *(siehe Listing 3)* diesen `static`-Ordner setzen.

Zur Datenbankmigration wurde Flyway eingebaut. Durch die einfache Namenskonvention von Flyway *[2]* migriert die Datenbank beim Start des Servers selbstständig, da es die beiden aufgeführten Skripte `V1_0_1_create_table_bestellung.sql` und `V1_0_1_alter_table_bestellung.sql` ausführt. Der Inhalt der Skripte ist in den *Listings 4 und 5* aufgeführt und umfasst die initiale Erstellung und die erste Erweiterung.

Vom Spielprojekt zur Anwendung zum Spielprojekt

Ausgestattet mit Maven und Flyway sowie einem Jenkins, der CI und CD macht, wurde aus dem anfänglichen Spielprojekt schnell eine Anwendung, die überall deployt werden konnte.

Den Spielprojektcharakter behielt es dennoch bei, weil nun auch andere Kollegen anfingen, daran Technologien auszutesten. Docker-Container für die Anwendung und für die Datenbank (nicht nach dem Sinn fragen!) wurden eingebaut. Blue-/Green-Deployments wurden ausgetestet und sogar ein ganzer Docker-Swarm mit mehreren Instanzen ist mal online gegangen. Da die Daten in der Datenbank nur für den Bestelltag wichtig sind, waren Datenverluste bei Misserfolgen ohne Konsequenzen, außer jemand probierte am Dönerstag erfolglos etwas Neues aus.

Döner in the Cloud

Um den Kollegen praktisches Wissen für einige Clouddienste zu vermitteln, wurde ein interner Workshop auf Basis der Döner-App durchgeführt.

```
<build>
    <finalName>doener-ui</finalName>
    <resources>
        <resource>
            <directory>dist </directory>
            <targetPath>static</targetPath>
        </resource>
    </resources>
```

Listing 3: Konfiguration für das Frontend, sodass es gleich im notwendigen Serverordner erstellt wird

```
CREATE TABLE `Bestellung` (
  id  BIGINT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  bestellung VARCHAR(255) NOT NULL,
  bestelldatum DATE NOT NULL
);
```

Listing 4: Inhalt von create_table_bestellung.sql

```
ALTER TABLE  `Bestellung`
        add column alias  VARCHAR(200),
        add column email VARCHAR(200),
        add column extras VARCHAR(200),
        add column fleisch VARCHAR(200),
        add column preis VARCHAR(200),
        add column sauce VARCHAR(200),
        add column bezahlt bit,
        add column telefonnummer VARCHAR(200);
```

Listing 5: Inhalt von V1_0_1_alter_table_bestellung.sql

```
spring:
  profiles: aws
  datasource:
    driverClassName: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://doener.someurl:3306/doener
    username: schoener
    password: ********
```

Listing 6: Spring-Boot-Konfiguration für die Datenbank in AWS

Als Cloudanbieter bot sich damals AWS an. Mit dem Service RDS von AWS kann man sehr leicht eine relationale Datenbank erstellen (Beispiel: *[3]*). Die Verbindungsdaten zur Datenbank müssen dann nur noch in der Anwendung eingetragen werden. Dies ist hier über ein eigenes Profile *(siehe Listing 6)* in der application.yaml gelöst. Datenmigration war aufgrund der Daten nicht notwendig, wäre aber über normalen Import/Export auch möglich gewesen.

Der AWS-Service Elastic Beanstalk kann praktischerweise Jar-Dateien ausführen. Mit seiner Hilfe *[4]* war es sehr einfach möglich, die Döner-App in die Cloud zu bringen. Aber ist das wirklich Cloud, nur weil die Datenbank und ein Jar in der Cloud laufen?

Döner in the Cloud – Part 2: Now Serverless

Wenn schon in die Cloud, dann richtig – was auch immer das bedeutet. Auf jeden Fall serverless, also ohne einen laufenden Server, der ein Jar ausführt und am liebsten ohne eine relationale Datenbank.

Kann man den Applikationsserver durch Serverless-Funktionen ersetzen? AWS bietet dafür die Lambda-Funktionen, die einfach ein Stückchen Code auf einer vorher definierten Hardware und Programmiersprache ausführen. Serverless bedeutet hier nur, dass wir keinen Server aufsetzen, pflegen und warten müssen. Es werden lediglich die Angaben zum gewünschten Arbeitsspeicher und Rechenpower benötigt. Bezahlt wird dann nur die Dauer, die eine Lambda-Funktion zum Ausführen benötigt, mit unterschiedlichen Stundensätzen, je nach Rechenkapazität der CPU. Die Default-Einstellungen reichen in der Regel.

Bei Bedarf wacht die Lambda-Funktion auf, führt ihren Code aus und legt sich wieder schlafen. Dies kann, je nach Programmiersprache, mehr oder weniger lange dauern. Als Faustformel gilt, dass Lambda-Funktionen, geschrieben in Java, sehr viel Zeit zum Aufwachen benötigen und JavaScript oder Python dies deutlich schneller erledigen. Wer produktive Lambda-Funktionen in Java schreiben möchte, sollte lieber über einen eigenen Server nachdenken.

Für die Abbildung eines REST-API gibt es in AWS den Service-API-Gateway. Dieser ist mit wenigen Mausklicks angelegt und mit einer Lambda-Funktion zur weiteren Verarbeitung verbunden. Mithilfe des Service-API-Gateway kann man beliebig komplexe APIs bauen. Diese können dann mittels Swagger/OpenAPI exportiert werden, in den unterschiedlichsten Versionen. Man kann aber vorhandene APIs über Swagger/OpenAPI importieren.

Um die Dönerbestellungen zu persistieren, kann man in diesem einfachen Fall statt einer relationalen Datenbank auch eine NoSQL-Datenbank nutzen. AWS bietet dafür die DynamoDB an, die nahezu problemlos JSON speichern, lesen und updaten kann. Dadurch ergibt sich folgendes Bild *(siehe Abbildung 3)*.

Der Lambda-Zugriff auf die DynamoDB, in Abhängigkeit von den Methoden POST und GET, ist exemplarisch in *Listing 7* dargestellt. Alles in einer Methode, noch mit dem älteren V2-SDK und einem Callback anstelle eines `async/await` für die Promises. Aber dieses Projekt ist klein genug, um problemlos Neues auszuprobieren, wie zum Beispiel das neue JavaScript V3-SDK *[5]*, und um natürlich jeweils ein Lambda für GET und POST zu erstellen.

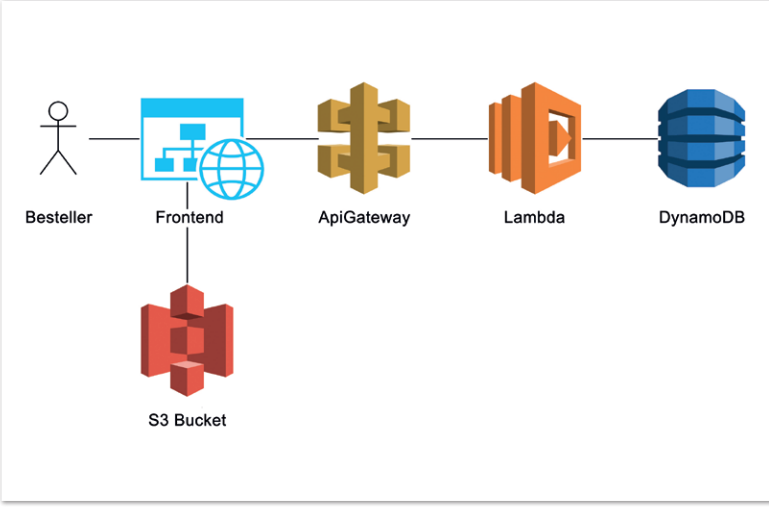


Abbildung 3: AWS-Service-Übersicht (© Thomas Michael)

```
const AWS = require('aws-sdk');
const dynamo = new AWS.DynamoDB();

exports.handler = (event, context, callback) => {
    const done = (err, res) =>
        callback(null, {
            statusCode: err ? '400' : '200',
            body: err ? err.message : JSON.stringify(res),
            headers: {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Headers': 'Content-Type',
                'Access-Control-Allow-Methods': 'OPTIONS,POST,GET'
            },
        });

    switch (event.httpMethod) {
        case 'GET':
            dynamo.scan({ TableName: 'doener' }, (err, res) => {
                const resultAsJson = res.Items.map((item) => AWS.DynamoDB.Converter.unmarshall(item));
                done(err, resultAsJson);
            });
            break;
        case 'POST':
            const itemAsJson = JSON.parse(event.body);
            const itemAsDynamoDbItem = AWS.DynamoDB.Converter.marshall(itemAsJson, true);
            var params = {
                Item: itemAsDynamoDbItem,
                ReturnConsumedCapacity: "TOTAL",
                TableName: 'doener'
            };
            dynamo.putItem(params, done);
            break;
        default:
            done(new Error(`Unsupported method "${event.httpMethod}"`));
    }
};
```

Listing 7: Komplette Lambda-JavaScript-Funktion

```
const resultAsJson = <<Result aus DynamoDB>>.map((item) => AWS.DynamoDB.Converter.unmarshall(item));
…
const itemAsJson = JSON.parse(<<Bestellung als JSON>>);
const itemAsDynamoDbItem = AWS.DynamoDB.Converter.marshall(itemAsJson, true);
```

Listing 8: Code-Snippets der Konvertierung von JSON in ein DynamoDB-Record

Die DynamoDB benötigt die Bestelldaten in einem eigenen JSON-Format: DynamoDB-Record, angereichert mit zusätzlichen Informationen zu den einzelnen Attributen. Mit dem Converter *(siehe Listing 8)* kann unsere Bestellung aber sehr einfach hin und her konvertiert, beziehungsweise „marshalled" und „unmarshalled", werden.

Beim API-Gateway verzichten wir vorerst weiterhin auf eine Autorisierung und Authentifizierung und sichern das API nur mit einem API-X Key ab. Das Frontend, die Angular-Anwendung, benötigt die neuen Pfade zur Cloud und kann dann selbst in der Cloud laufen. Dafür gibt es auch wieder mehrere Möglichkeiten – wir nutzen die kostengünstigste: Aus Angular, genauer dem TypeScript, mittels *Listing 9* einen DIST-Ordner mit HTML und JavaScript zu erstellen und den Inhalt dann einfach in ein S3-Bucket zu kopieren.

Ein S3-Bucket ist ein riesiger Speicherplatz für unterschiedliche Anwendungsfälle. Einer davon ist die Bereitstellung einer Webanwendung. Dies lässt sich mit einem Mausklick und einer Zugriffspolicy *(siehe Listing 10)* sehr leicht und intuitiv erledigen. In dieser Policy muss nur der `<<S3-Bucketname>>` durch den echten S3-Bucketnamen ersetzt werden. Damit haben wir folgende Abbildung der einzelnen Systeme in der Cloud:

System	Von	Nach
Backend-Logik	Spring Boot Jar	AWS Lambda
REST-API	Spring Boot Jar	AWS API Gateway
Datenbank	Relationale MariaDB	NoSQL DynamoDB
Frontend	Angular-Artefakte im JAR inklusive	Angular-Artefakte im S3 Bucket

Man sieht deutlich eine Trennung der einzelnen Services. So sind das REST-API und die Logik streng getrennt abgebildet durch die Services-AWS-Lambda und das AWS-API-Gateway.

Wenn mal wieder Zeit ist, wird die immer noch fehlende Berechtigung eingebaut. Wie man das mit AWS macht und dabei nur das AWS-API-Gateway um einen weiteren Service erweitert, kann man unter *[6]* und *[7]* sehr anschaulich nachlesen.

Fazit

Auch bei scheinbar trivialen Anwendungsfällen lässt sich viel lernen. Die Anwendung wurde mittlerweile zu Lernzwecken in unterschiedlichen Programmiersprachen neugeschrieben. Die Kollegen lernen anschaulich und übersichtlich den Build-Prozess mit CI/CD in Jenkins kennen, können damit rumspielen und ihn verbessern. Und für die Cloud ist nun auch ein kleines Projekt zum Ausprobieren vorhanden.

So wurde aus einem kleinen Projekt mit Bestellzettel ein anschauliches Schulungsprojekt für viele Bereiche der Softwareentwicklung.

```
npm run build
```

Listing 9: Angular kompiliert zu HTML und JavaScript

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::<<S3-Bucketname>>/*"
        }
    ]
}
```

Listing 10: S3-Policy für den Zugriff des S3-Bucket als Webseite

Quellen

[1] *https://github.com/eirslett/frontend-maven-plugin.*
[2] *https://flywaydb.org/documentation/concepts/migrations.html#naming*
[3] *https://youtu.be/vW_G7GwiYn0*
[4] *https://youtu.be/8Ch-stbEW-c*
[5] *https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/index.html*
[6] *https://docs.amazonaws.cn/en_us/cognito/latest/developerguide/what-is-amazon-cognito.html*
[7] *https://docs.aws.amazon.com/de_de/serverless-application-model/latest/developerguide/sam-permissions.html*

Thomas Michael
GOD mbH
Thomas.Michael@god.de

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

Eine Java-Spracherweiterung zum Programmieren von zusammengesetzten vernetzten Systemen

Gerhard Fuchs

Dieser Artikel skizziert eine Java-Spracherweiterung für zusammengesetzte vernetzte Systeme und demonstriert deren Anwendung durch die beispielhafte Java-Programmierung eines aus zwei Spielzeugrobotern zusammengesetzten Schwarms im Detail. Die Spracherweiterung ermöglicht ein deklaratives Programmieren mithilfe von Annotationen und wird von FEPCOS-J prototypisch implementiert. Der Anwender kann Netzwerkprogrammierung automatisieren und bekommt bei der Programmierung systemübergreifender Nebenläufigkeit Unterstützung.

Wie programmiere ich „Zusammensetzen“ in Java?

Abbildung 1 zeigt ein Beispiel eines zusammengesetzten vernetzten Systems: Zwei mobile Spielzeugroboter sind als Teile zu einem neuen Ganzen – dem Schwarm – zusammengesetzt. Die vom Schwarm ausführbaren Aktivitäten sind, verglichen mit den Aktivitäten, die die einzelnen Roboter für sich genommen ausführen können, ein möglicher Mehrwert. Die beiden Roboter können völlig unabhängig voneinander, also systemübergreifend nebenläufig, agieren. Deshalb werden sie bei den Aktivitäten, die sie gemeinsam als Schwarm ausführen, von einem weiteren Computer mithilfe von Netzwerkkommunikation koordiniert.

Das Zusammensetzen von Teilen zu einem neuen Ganzen erinnert an Klemmbausteine. Aber wie programmiere ich „Zusammensetzen“ in Java? Und entsteht dabei aufgrund der Netzwerkkommunikation und der systemübergreifenden Nebenläufigkeit ein Mehraufwand?

Domänenspezifische Java-Spracherweiterung

Im Rahmen des `FEPCOS-Projekts` *[1]* ist eine domänenspezifische Java-Spracherweiterung entstanden *[2]*. Diese basiert auf einem Modell, das das Kompositum-Entwurfsmuster um den Aspekt der Netzwerkkommunikation erweitert. Das Modell erlaubt es, einen Mehrwert zusammengesetzter vernetzter Systeme klar zu benennen. Die Spracherweiterung ermöglicht das deklarative Programmieren von „Zusammensetzen“ mithilfe von Annotationen und wird von `FEPCOS-J` prototypisch implementiert.

Gemäß dem Modell ist ein zusammengesetztes vernetztes System ein Ganzes, das aus Teilen besteht, wobei das Ganze über ein Netzwerk mit den Teilen kommuniziert. Sowohl die Teile als auch das Ganze sind Systeme, die Aktivitäten innerhalb einer endlichen Zeit ausführen können. Die Aktivitäten, die das Ganze ausführen kann, sind ein möglicher Mehrwert. Ein elementares System besitzt im Gegensatz zum zusammengesetzten vernetzten System keine Teile. Es ist sozusagen der Grundbaustein.

Die Realisierung des Modells erfolgt über die Spracherweiterung. Diese ermöglicht es dem Anwender, Systeme zu spezifizieren und diese im Anschluss wie Bausteine zusammenzusetzen: Der Anwender programmiert eine Systemspezifikation unter Verwendung von Annotationen und erzeugt daraus mithilfe des zur Implementierung der Spracherweiterung gehörenden Annotation-Prozessors `fjp` ein Systemexportmodul und ein Systemimportmodul. Das Systemexportmodul enthält die Systemspezifikation und wird mit dem zur Implementierung der Spracherweiterung gehörenden Programm `fjx` exportiert, das heißt innerhalb eines IPv4-Netzwerks zur Verfügung gestellt. Das Systemimportmodul gibt einem Systemnutzer eine von `fjp` generierte Schnittstelle für einen blockierenden beziehungsweise nebenläufigen Zugriff auf das Systemexportmodul über das Netzwerk. Die Adressierung des Systemexportmoduls erfolgt über ein Internet-Socket. `fjp` generiert den für die Netzwerkkommunikation benötigten Quellcode.

Ein Anwender der Spracherweiterung spezifiziert ein System, indem er eine Systemdeklaration und mindestens eine Aktivitätsspezifikation programmiert. Hierbei verwendet er im Wesentlichen die Annotationen `@SYDec`, `@Cap` und `@AYSpec`: `@SYDec` legt fest, dass die annotierte Klasse die Systemdeklaration ist. In dieser Klasse wird mit `@Cap` mindestens eine Fähigkeit, also eine Aktivität, die das System ausführen kann, deklariert. Die Implementierung der Aktivität erfolgt in einer Klasse, die mit `@AYSpec` als Aktivitätsspezifikation annotiert ist. Bei einem zusammengesetzten System werden mit `@Part` die Teile deklariert. Hierbei werden die Socket-Adresse und die generierte Schnittstelle eines Systemimportmoduls angegeben. Details werden mithilfe des nachfolgenden Beispielszenarios erklärt.

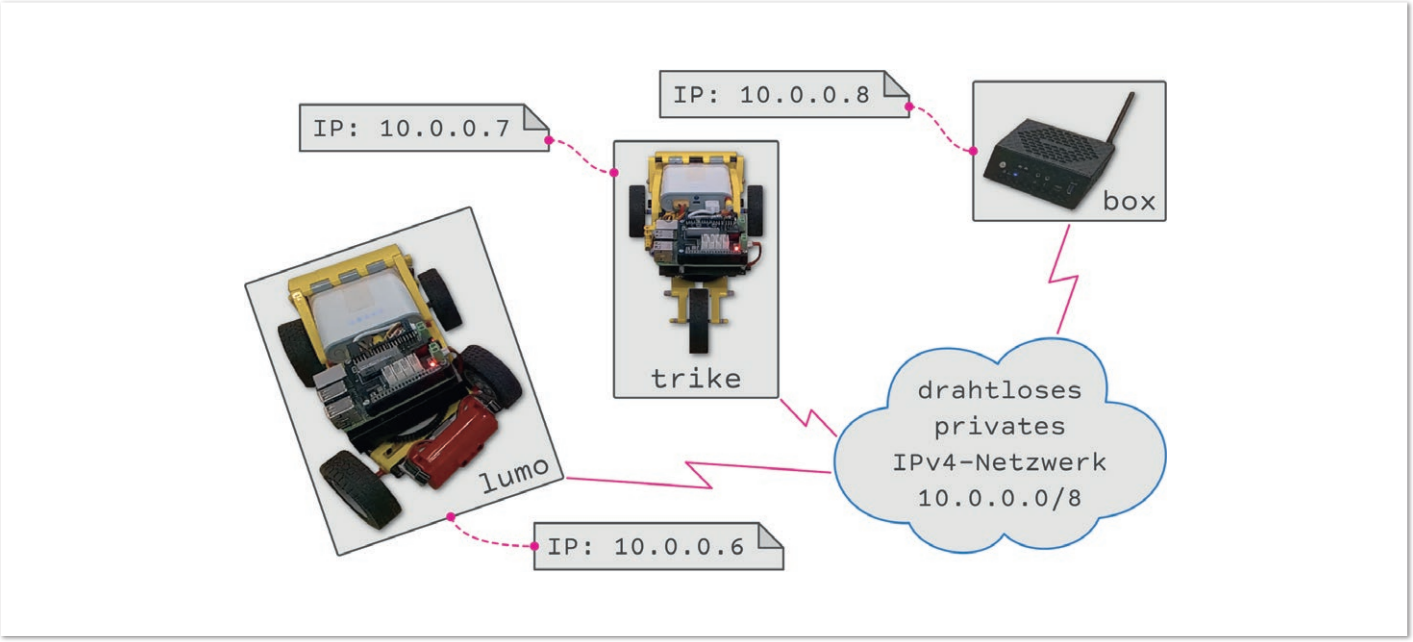


Abbildung 1: Beispiel eines zusammengesetzten vernetzten Systems (© Gerhard Fuchs): Bei der Hardware handelt es sich um drei miteinander vernetzte Computer, genannt `lumo`, `trike` und `box`. `lumo` und `trike` sind `Raspberry Pis` mit 4 x 1.2GHz CPU und 1 GB RAM, die auf Spielzeugroboter montiert sind. box ist ein Computer mit 4x 1.1 bis 2.2 GHz CPU und 8GB RAM. Auf den drei Computern ist die folgende Software installiert: ein `Linux`-Betriebssystem; `OpenJDK 11`; `FEPCOS-J`. Die Kommunikation erfolgt über ein drahtloses privates IPv4-Netzwerk, das die Adresse `10.0.0.0/8` hat. Hierbei hat `lumo` die IP-Adresse `10.0.0.6`, `trike` die IP-Adresse `10.0.0.7` und `box` die IP-Adresse `10.0.0.8`.

Beispielszenario

Die beiden Spielzeugroboter werden zu einem Schwarm zusammengesetzt und sollen wie in *Abbildung 2* dargestellt „tanzen" (sich koordiniert bewegen). Ein Anwender des Modells sieht bei diesem Szenario die elementaren Systeme `Lumo` und `Trike` und ein daraus zusammengesetztes System `Swarm` *(siehe Abbildung 3)*. `Lumo` hat die Fähigkeiten `twinkle()` und `circleLeft()`, wobei `twinkle()` das Blinken mit den LEDs und `circleLeft()` das Linksherumfahren im Kreis ist. `Trike` hat die Fähigkeit `twist()`,das Hin- und Herbewegen des Vorderrads. Schließlich hat `Swarm` die Fähigkeit `dance()`, die das Koordinieren der Fähigkeiten der beiden Roboter ist, sodass diese als Schwarm das gewünschte Verhalten ausführen.

Realisierung des Beispielszenarios

Die Programmierung der Systeme `Lumo`, `Trike` und `Swarm` erfolgt auf den jeweiligen Computern `lumo`, `trike` und `box` unter Verwendung der Annotationen der Spracherweiterung. Begonnen wird mit den beiden elementaren Systemen `Lumo` und `Trike`, die dann im Anschluss als Teile des zusammengesetzten Systems `Swarm` verwendet werden.

Der Anwender der Spracherweiterung programmiert die Systemspezifikation von `Lumo` als Java-Modul `lumo.spec` *(siehe Abbildung 4)*. Er erzeugt daraus mit `fjp` das Systemexportmodul `lumo.exp`, und das Systemimportmodul `lumo.imp`. `lumo.exp` enthält die Systemspezifikation, `lumo.imp` eine Schnittstelle für den Zugriff über das Netzwerk darauf. Das Systemexportmodul bleibt auf `lumo`; das Systemimportmodul kopiert er auf `box`.

Bei `Trike` geht er analog vor. Im Ergebnis befinden sich die Systemexportmodule `lumo.exp` und `trike.exp` auf den entsprechenden Computern. Auf `box` befinden sich die dazugehörigen Systemimportmodule `lumo.imp` und `trike.imp`. Diese verwendet der Anwender für die anschließende Programmierung der Systemspezifikation

Abbildung 2: Beispielszenario (© Gerhard Fuchs, basierend auf einem unter [3] (QR-Code) abrufbaren Video): `lumo` fährt blinkend um `trike` herum, während `trike` gleichzeitig, also systemübergreifend nebenläufig, das Vorderrad hin- und herbewegt. Die Koordination erfolgt auf `box`.

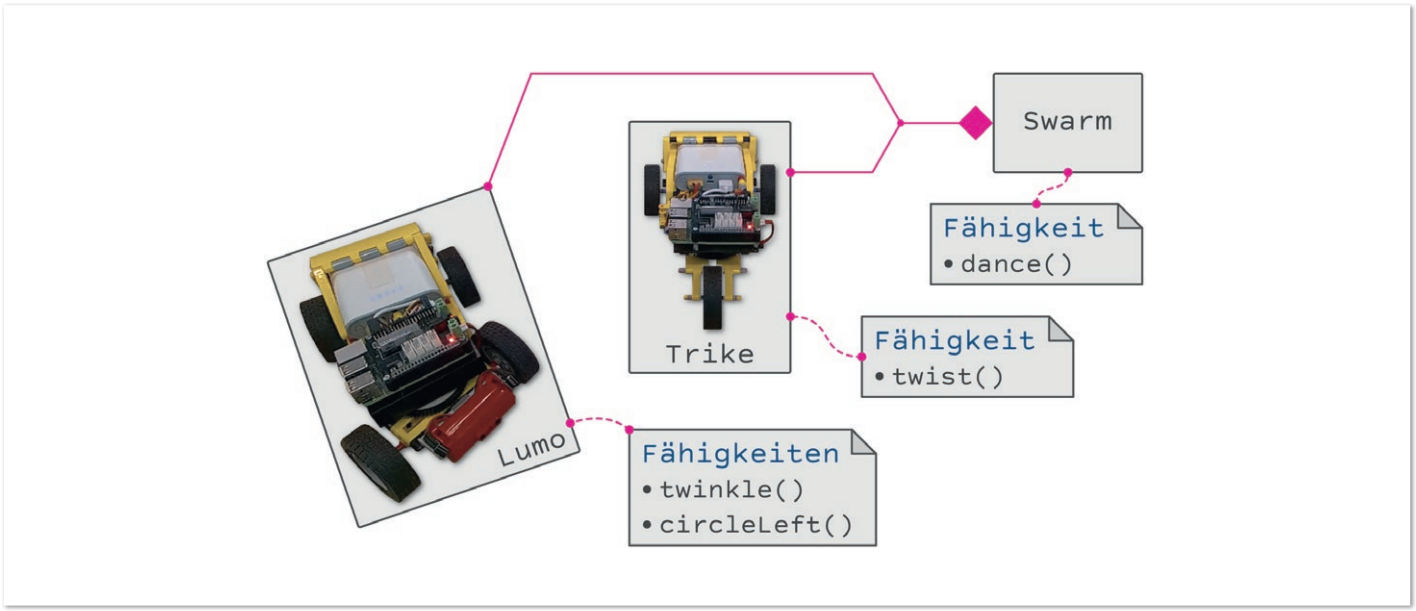


Abbildung 3: Anwendung des Modells (© Gerhard Fuchs).

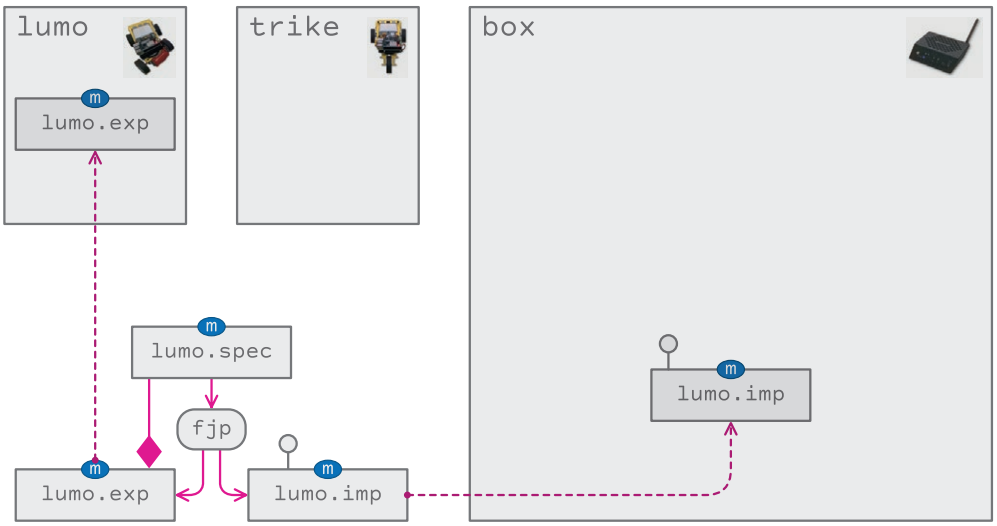


Abbildung 4: Programmierung von Lumo (© Gerhard Fuchs)

von Swarm als Java-Modul swarm.spec *(siehe Abbildung 5)*. Aus swarm.spec erzeugt er mit fjp das Systemexportmodul swarm.exp. Das Systemimportmodul swarm.imp. swarm.exp enthält die Systemspezifikation, swarm.imp eine Schnittstelle für den Zugriff darauf. Diesmal bleiben sowohl das Systemexportmodul als auch das Systemimportmodul auf box.

Auf den drei Computern ist FEPCOS-J installiert *(siehe Abbildung 6)*. Der Anwender exportiert mit fjx die Systemexportmodule auf den entsprechenden Computern. So kann er auf box mit jshell über swarm.imp auf swarm.exp zugreifen. swarm.exp enthält den Algorithmus für die Koordination von Lumo und Trike und greift über das Netzwerk mit lumo.imp auf lumo.exp und mit trike.imp auf trike.exp zu. Bei diesem Beispiel ist beim Exportieren der Systemexportmodule der Port 1111 angegeben worden. Somit erfolgt der Zugriff auf lumo.exp über die Socket-Adresse 10.6:1111, auf trike.exp über 10.7:1111 und auf swarm.exp über 10.8:1111.

Programmierung des elementaren Systems Lumo

Die Realisierung des Systems Lumo erfolgt auf dem Computer lumo im Verzeichnis demo *(siehe Abbildung 7)*. In diesem befinden sich die Unterverzeichnisse mlib und src. In mlib sind die Module der Hardwaretreiber hw.core.jar und hw.lumo.jar. In src befindet sich der Quellcode der Systemspezifikation: module-info.java ist der Moduldeskriptor. Im Unterverzeichnis lumo/spec sind die Systemdeklaration SY.java und die Aktivitätsspezifikationen CircleLeft.java beziehungsweise Twinkle.java.

Der Moduldeskriptor *(siehe Listing 1)* legt fest, dass das Modul der Systemspezifikation lumo.spec heißt. Es benötigt den Hardware-

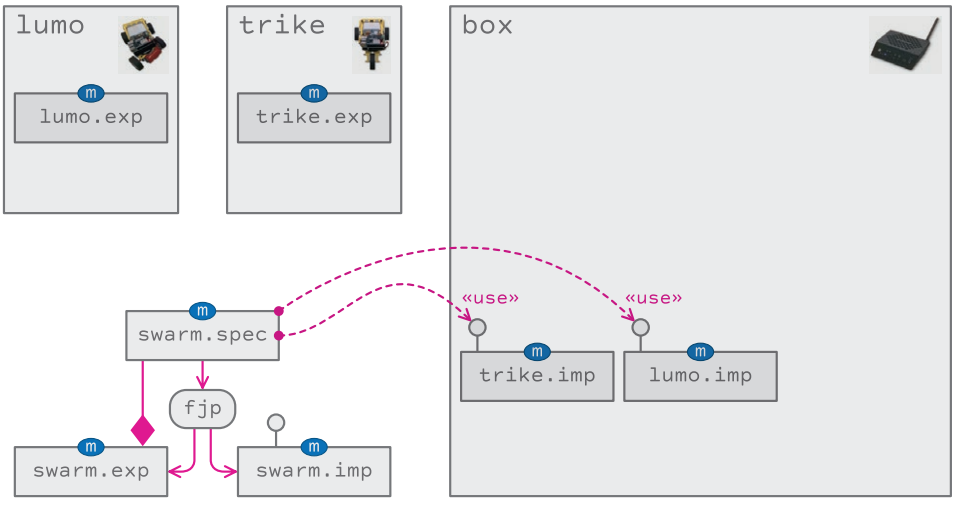


Abbildung 5: Programmierung von Swarm (© Gerhard Fuchs)

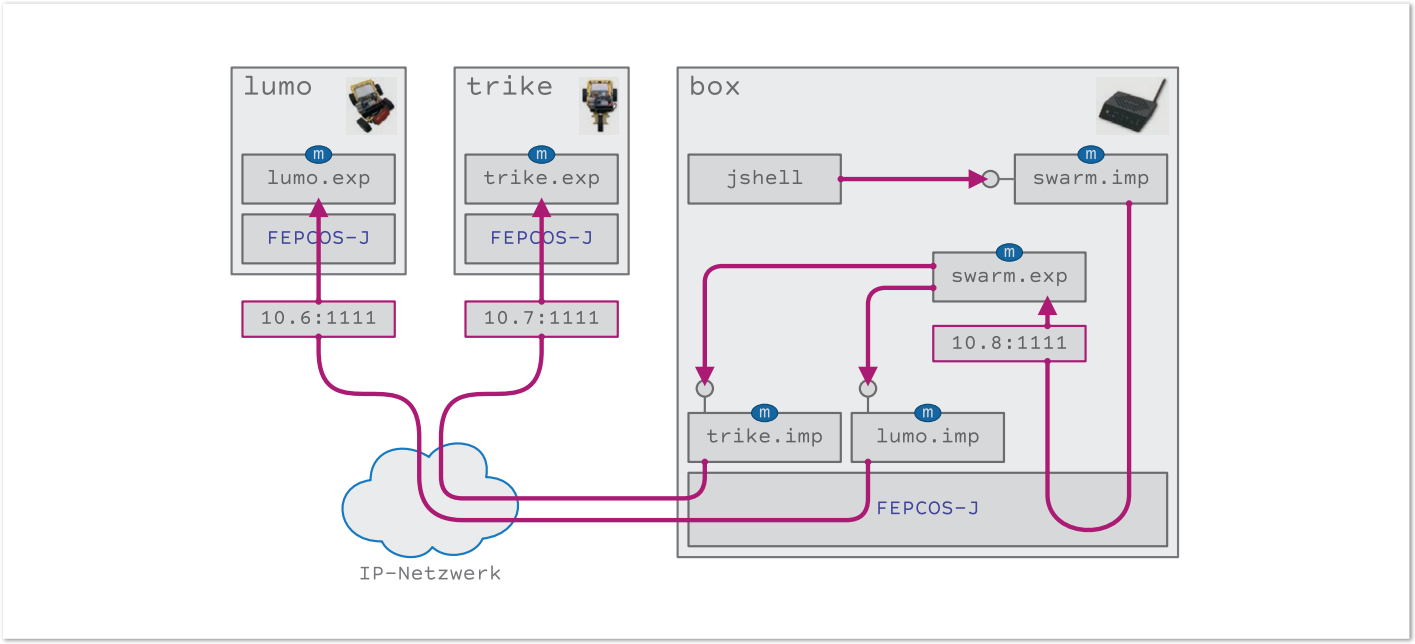


Abbildung 6: Zugriff auf `Swarm` mittels `jshell` und von dort über das Netzwerk auf `Lumo` und `Trike` (© Gerhard Fuchs).

treiber des Moduls `hw.lumo` und benötigt für die Programmierung Annotationen des Moduls `fepcos.annotations`.

Die Klasse `SY` *(siehe Listing 2)* ist im Paket `lumo.spec` vorhanden und importiert die Pakete `fepcos.sy` und `hw.lumo`. `fepcos.sy` enthält die Annotationen `@SYDec`, `@Cap`, `@Start` und `@Stop`. `hw.lumo` enthält den Hardwaretreiber `Lumo`. `@SYDec` legt fest, dass `SY` die Systemdeklaration ist; der übergebene String dient der Dokumentation des zu generierenden Systemimportmoduls.

Die Member-Variable `Lumo hw` ist für die Instanz des Hardwaretreibers vorgesehen. Auf diese wird dann in den Aktivitätsspezifikationen zugegriffen. `@Cap` deklariert die Fähigkeiten `twinkle` und `circleLeft`. Die Implementierung erfolgt mit den Klassen `Twinkle` beziehungsweise `CircleLeft`. `@Start` legt fest, dass die Methode `start()` beim Starten des Systems ausgeführt wird. Es wird mit `hw = Lumo.getInstance();` die Instanz des Hardwaretreibers gesetzt und daran mit `hw.reset()` ein Reset ausgeführt. `@Stop` legt fest, dass die Methode `stop()` beim Stoppen des Systems ausgeführt wird. Es wird an der Instanz des Hardwaretreibers mit `hw.reset()` ein Reset ausgeführt.

```
demo
├─mlib
│  ├─hw.core.jar
│  └─hw.lumo.jar
└─src
   ├─lumo
   │  └─spec
   │     ├─CircleLeft.java
   │     ├─SY.java
   │     └─Twinkle.java
   └─module-info.java
```

Abbildung 7: Verzeichnisstruktur der Systemspezifikation des elementaren Systems `Lumo` (© Gerhard Fuchs)

Die Klassen `CircleLeft` *(siehe Listing 3)* und `Twinkle` *(siehe Listing 4)* sind ebenfalls im Paket `lumo.spec`. Sie importieren `fepcos.ay.*` und so die Annotationen `@AYSpec`, `@In` und `@Behavior`. Die bei den Annotationen übergebenen Strings dienen der Dokumentation des zu generierenden Systemimportmoduls. `@AYSpec` legt fest, dass die Klassen Aktivitätsspezifikationen sind. `@In(…) int d;` deklariert einen Input-Parameter, nämlich eine ungefähre Dauer `d[ms]` – also, wie viele Millisekunden `Lumo` die jeweilige Aktivität ausführen soll. Die mit `@Behavior` annotierte Methode `void go(SY x) throws Exception` spezifiziert das gewünschte Verhalten von `Lumo`.

```
module lumo.spec {
  requires hw.lumo;
  requires static fepcos.annotations;
}
```

Listing 1: `Lumo` – Moduldeskriptor `module-info.java`.

```
package lumo.spec;

import fepcos.sy.*;
import hw.lumo.*;

@SYDec("The lumo system.")
class SY {
  Lumo hw;

  @Cap Twinkle twinkle;
  @Cap CircleLeft circleLeft;

  @Start
  void start() throws LumoException {
     hw = Lumo.getInstance();
     hw.reset();
  }

  @Stop
  void stop() throws LumoException {
     hw.reset();
  }
}
```

Listing 2: `Lumo` – Systemdeklaration `SY.java`.

```
package lumo.spec;

import fepcos.ay.*;

@AYSpec("Circle left for about »d« [ms].")
class CircleLeft {
  @In("Duration in [ms].") int d;

  @Behavior
  void go(SY x) throws Exception {
     x.hw.steerLeft();
     x.hw.driveForward();
     Thread.sleep(d);
     x.hw.stop();
  }
}
```

Listing 3: Lumo – Aktivitätsspezifikation CircleLeft.java.

```
package lumo.spec;

import fepcos.ay.*;

@AYSpec("Twinkle for at least »d« [ms].")
class Twinkle {
  @In("Minimum duration in [ms].") int d;

  @Behavior
  void go(SY x) throws Exception {
     long stop = System.currentTimeMillis() + d;

     while (System.currentTimeMillis() <= stop) {
        x.hw.turnOnLEDs();
        Thread.sleep(500);
        x.hw.turnOffLEDs();
        Thread.sleep(500);
     }
  }
}
```

Listing 4: Lumo – Aktivitätsspezifikation Twinkle.java.

Der Methode wird mit `SY x` die Systemdeklaration übergeben, sodass auf den Hardwaretreiber `x.hw` zugegriffen werden kann.

Bei `CircleLeft` soll `Lumo d[ms]` linksherum im Kreis fahren. `Lumo` lenkt mit `x.hw.steerLeft()` nach links und fährt dann durch den Aufruf von `x.hw.driveForward()` bis auf Weiteres vorwärts, der Treiber sorgt für ein nebenläufiges Ausführen. Nach `d[ms] - Thread.sleep(d)` - wird `Lumo` mit `x.hw.stop()` angehalten.

Bei `Twinkle` soll `Lumo` für mindestens `d[ms]` mit den LEDs blinken. `long stop = System.currentTimeMillis() + d;` bestimmt den Endzeitpunkt der Aktivität. Anschließend wird in einer `while`-Schleife bis zum Erreichen des Endzeitpunkts das Nachfolgende wiederholt: Mit `x.hw.turnOnLEDs();` LEDs einschalten, mit `Thread.sleep(500);` 500 ms warten, mit `x.hw.turnOffLEDs();` LEDs ausschalten, nochmal mit `Thread.sleep(500);` 500 ms warten.

Eventuell geworfene Exceptions werden bei beiden Aktivitätsspezifikationen von der Methode weitergegeben. So kann `fjx` mögliche Fehlschläge dem aufrufenden Systemimportmodul signalisieren.

Programmierung des elementaren Systems Trike

Die Realisierung des Systems `Trike` erfolgt auf dem Computer `trike` im Verzeichnis `demo` *(siehe Abbildung 8)*. In diesem befinden sich die Unterverzeichnisse `mlib` und `src`. In `mlib` sind die Module der

```
demo
├─mlib
│  ├─hw.core.jar
│  └─hw.trike.jar
└─src
   ├─module-info.java
   └─trike
      └─spec
         ├─SY.java
         └─Twist.java
```

Abbildung 8: Verzeichnisstruktur der Systemspezifikation des elementaren Systems Trike (© Gerhard Fuchs)

```
module trike.spec {
  requires hw.trike;
  requires static fepcos.annotations;
}
```

Listing 5: Trike – Moduldeskriptor module-info.java

```
package trike.spec;

import fepcos.sy.*;
import hw.trike.*;

@SYDec("The Trike system which can twist.")
class SY {
  Trike hw;

  @Cap Twist twist;

  @Start
  void start() throws TrikeException {
     hw = Trike.getInstance();
     hw.reset();
  }

  @Stop
  void stop() throws TrikeException {
     hw.reset();
  }
}
```

Listing 6: Trike – Systemdeklaration SY.java.

Hardwaretreiber `hw.core.jar` und `hw.trike.jar`. In `src` befindet sich der Quellcode der Systemspezifikation: `module-info.java` ist der Moduldeskriptor. Im Unterverzeichnis `trike/spec` sind die Systemdeklaration `SY.java` und die Aktivitätsspezifikation `Twist.java`.

Der Moduldeskriptor *(siehe Listing 5)* legt fest, dass das Modul der Systemspezifikation `trike.spec` heißt. Es benötigt den Hardwaretreiber des Moduls `hw.trike` und benötigt für die Programmierung Annotationen des Moduls `fepcos.annotations`.

Die Klasse `SY` *(siehe Listing 6)* ist im Paket `trike.spec` und benötigt die Pakete `fepcos.sy` und `hw.trike`. `fepcos.sy` enthält die Annotationen `@SYDec`, `@Cap`, `@Start` und `@Stop`. `hw.trike` enthält den Hardwaretreiber `Trike`. `@SYDec` legt fest, dass `SY` die Systemdeklaration ist. Der übergebene String dient der Dokumentation des zu generierenden Systemimportmoduls.

Die Member-Variable `Trike hw` ist für die Instanz des Hardwaretreibers vorgesehen. Auf diese wird in der Aktivitätsspezifikation zugegriffen. `@Cap` deklariert die Fähigkeit `twist`. Die Implementierung erfolgt mit der Klasse `Twist`. `@Start` legt fest, dass die Methode `start()` beim Starten des Systems ausgeführt wird. Es wird mit `hw = Trike.getInstance();` die Instanz des Hardwaretreibers gesetzt und daran mit `hw.reset()` ein Reset ausgeführt. `@Stop` legt fest, dass die Methode `stop()` beim Stoppen des Systems ausgeführt wird. Es wird an der Instanz des Hardwaretreibers mit `hw.reset()` ein Reset ausgeführt.

Die Klasse `Twist` *(siehe Listing 7)* ist ebenfalls im Paket `trike.spec`. Sie importiert `fepcos.ay.*` und so die Annotationen `@AYSpec`, `@In` und `@Behavior`. Die bei den Annotationen übergebenen Strings dienen der Dokumentation des zu generierenden Systemimportmoduls. `@AYSpec` legt fest, dass die Klasse `Twist` eine Aktivitätsspezifikation ist. `@In(…) int d;` deklariert einen Input-Parameter, nämlich eine ungefähre Dauer `d[ms]` – also wie viele Millisekunden `Trike` die Aktivität ausführen soll.

Die mit `@Behavior` annotierte Methode `void go(SY x) throws Exception` spezifiziert das gewünschte Verhalten von `Trike`. Der Methode wird mit `SY x` die Systemdeklaration übergeben, sodass auf den Hardwaretreiber `x.hw` zugegriffen werden kann. Eventuell geworfene Exceptions werden von der Methode weitergegeben. So kann `fjx` mögliche Fehlschläge dem aufrufenden Systemimportmodul signalisieren.

Beim Aufruf von `go(…)` soll `Trike d[ms]` lang das Vorderrad hin- und herbewegen. `long stop = System.currentTimeMillis() + d;` bestimmt den Endzeitpunkt der Aktivität. Anschließend wird in einer `while`-Schleife bis zum Erreichen des Endzeitpunkts das Nachfolgende wiederholt: Mit `x.hw.steerLeft();` nach links lenken, mit `Thread.sleep(500);` 500 ms warten, mit `x.hw.steerRight();` nach rechts lenken, mit Thread.sleep(500); 500 ms warten. Abschließend erfordert der Treiber einen Reset, der mit `x.hw.reset();` realisiert wird.

Programmierung des zusammengesetzten Systems Swarm

Nachdem die beiden Teilsysteme `Lumo` und `Trike` programmiert sind, kann das daraus zusammengesetzte System `Swarm` programmiert werden. Dies geschieht auf dem Computer `box` im Verzeichnis `demo` *(siehe Abbildung 9)*. In diesem befinden sich die Unterverzeichnisse `mlib` und `src`. In `mlib` befinden sich die beiden erzeugten Systemimportmodule `lumo.imp.jar` und `trike.imp.jar`. In `src` ist der Quellcode der Systemspezifikation: `module-info.java` ist der Moduldeskriptor. Im Unterverzeichnis `swarm/spec` sind die Systemdeklaration `SY.java` und die Aktivitätsspezifikation `Dance.java`.

Der Moduldeskriptor *(Listing 8)* legt fest, dass das Modul der Systemspezifikation `swarm.spec` heißt. Es benötigt die beiden Systemimportmodule `trike.imp` und `lumo.imp`, die den Zugriff auf `Trike` bzw. `Lumo` über das Netzwerk ermöglichen. Ferner benötigt es für die Programmierung Annotationen des Moduls `fepcos.annotations`.

Die Klasse `SY` *(siehe Listing 9)* ist im Paket `swarm.spec`. Sie importiert `fepcos.sy.*` und so die Annotationen `@SYDec`, `@Part` und `@Cap`. `@SYDec` legt fest, dass `SY` die Systemdeklaration ist. Der übergebene

```
package trike.spec;

import fepcos.ay.*;

@AYSpec("Twist for at least »d« [ms] by " +
  "alternately steering left and right.")
class Twist {
  @In("Minimum duration in [ms].") int d;

  @Behavior
  void go(SY x) throws Exception {
     long stop = System.currentTimeMillis() + d;

     while (System.currentTimeMillis() <= stop) {
        x.hw.steerLeft();
        Thread.sleep(500);
        x.hw.steerRight();
        Thread.sleep(500);
     }
     x.hw.reset();
  }
}
```

Listing 7: `Trike` – Aktivitätsspezifikation `Twist.java`.

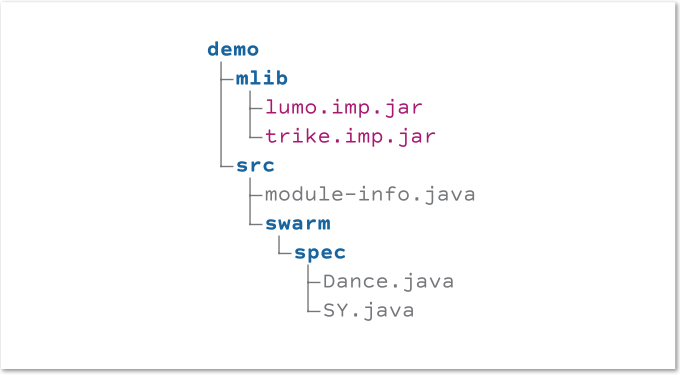


Abbildung 9: Verzeichnisstruktur der Systemspezifikation des zusammengesetzten Systems `Swarm` (© Gerhard Fuchs).

```
module swarm.spec {
  requires trike.imp;
  requires lumo.imp;
  requires fepcos.annotations;
}
```

Listing 8: `Swarm` – Moduldeskriptor `module-info.java`.

```
package swarm.spec;

import fepcos.sy.*;

@SYDec("This swarm, which consists " +
  " of the two robots Lumo and " +
  " Trike, can dance.")
class SY {
  @Part(ip = "10.0.0.6", port = 1111)
  lumo.S lumo;

  @Part(ip = "10.0.0.7", port = 1111)
  trike.S trike;

  @Cap Dance dance;
}
```

Listing 9: `Swarm` – Systemdeklaration `SY.java`.

```
package swarm.spec;
import fepcos.ay.*;
import java.util.concurrent.Future;

@AYSpec("Lumo and Trike dance for " +
  "about »d« [ms].")
class Dance {
  @In("Desired duartion in [ms].") int d;

  @Behavior
  void go(SY x) throws Exception {
     Future<?> r1 = x.lumo.c.twinkle(d);
     Future<?> r2 = x.trike.c.twist(d);
     Future<?> r3 = x.lumo.c.circleLeft(d);
     r1.get(); r2.get(); r3.get();
  }
}
```

Listing 10: Swarm – Aktivitätsspezifikation Dance.java.

String dient der Dokumentation des zu generierenden Systemimportmoduls.

Das „Zusammensetzten" des Schwarms wird mithilfe der Annotationen `@Part` programmiert, indem die Teile deklariert werden. `@Part(ip="10.0.0.6", port=1111) lumo.S lumo;` legt fest, dass `Lumo` ein Teil von `Swarm` und innerhalb des Netzwerks über das Internetsocket `10.0.0.6:1111` adressierbar ist. Der Zugriff in der Aktivitätsspezifikation erfolgt über die Member-Variable `lumo.S lumo`. Hierbei ist `lumo.S` die vom Annotation-Prozessor aus der Systemspezifikation von `Lumo` generierte Schnittstelle.

`@Part(ip="10.0.0.7", port=1111) trike.S trike;` legt dies analog für Trike fest. `@Cap` deklariert die Fähigkeit `dance`. Die Implementierung erfolgt mit der Klasse `Dance`.

Die Klasse `Dance` *(siehe Listing 10)* ist ebenfalls im Paket `swarm.spec`. Sie importiert `fepcos.ay.*` und so die Annotationen `@AYSpec`, `@In` und `@Behavior`. Die bei den Annotationen übergebenen Strings dienen der Dokumentation des zu generierenden Systemimportmoduls. `@AYSpec` legt fest, dass `Dance` eine Aktivitätsspezifikation ist. `@In(…) int d;` deklariert einen Input-Parameter, nämlich eine ungefähre Dauer `d[ms]` - also wie viele Millisekunden `Swarm` die Aktivität ausführen soll.

Die mit `@Behavior` annotierte Methode `void go(SY x) throws Exception` spezifiziert das gewünschte Verhalten von `Swarm`. Der Methode wird mit `SY x` die Systemdeklaration übergeben, sodass auf `Lumo` über `x.lumo` und `Trike` über `x.trike` zugegriffen werden kann. Eventuell geworfene Exceptions werden von der Methode weitergegeben. So kann `fjx` mögliche Fehlschläge dem aufrufenden Systemimportmodul signalisieren.

Beim Aufruf von `go(…)` soll Swarm für `d[ms]` „tanzen". Hierzu wird über die von `fjp` generierten Schnittstellen `x.lumo.c` und `x.trike.c` auf `Lumo` beziehungsweise `Trike` systemübergreifend nebenläufig zugegriffen. `x.lumo.c.twinkle(d);` bewirkt, dass `Lumo` für `d[ms]` mit den LEDs blinkt. `x.trike.c.twist(d);` bewirkt, dass `Trike` das Vorderrad für `d[ms]` hin- und herbewegt. `x.lumo.c.circleLeft(d);` bewirkt, dass `Lumo` für `d[ms]` linksherum im Kreis fährt. Bei den Aufrufen werden die `Future<?>`-Objekte `r1`, `r2` und `r3` zurückgegeben. Mit `r1.get(); r2.get(); r3.get();` wird so lange gewartet, bis alle Aktivitäten ausgeführt sind.

Fazit

Dieser Artikel hat eine Java-Spracherweiterung für zusammengesetzte vernetzte Systeme skizziert und deren Anwendung durch die Realisierung eines Beispielszenarios – inklusive kompletten Quellcodes – demonstriert. Der Anwender der Spracherweiterung musste sich nicht um die benötigte Netzwerkprogrammierung kümmern. Die Implementierung der Spracherweiterung hat den hierfür benötigten Quellcode samt Schnittstellen für systemübergreifende nebenläufige Zugriffe generiert.

Quellen

[1] Fuchs, Gerhard (2021); Das FEPCOS-Projekt, *http://fepcos.info*, Heubach, DE-BW.

[2] Fuchs, Gerhard (2020); Vernetzte Systeme mit FEPCOS-J; In: JAVAPRO OKT/2020; Seite 98-105; Verlag JAVAPRO; Eschborn, DE-HE.

[3] Fuchs, Gerhard (2021); FEPCOS-J, Anwendung im Video vorgestellt, *http://fepcos.info/de/fepcos-j_video.html*, Heubach, DE-BW.

Gerhard Fuchs
Freiberufler
fuchs@fepcos.info

Dipl.-Inf. Univ. Gerhard Fuchs programmiert seit dem Jahr 2000 mit Java und entwickelt Programmierkonzepte und Netzwerkprotokolle. Er hat am Lehrstuhl für Informatik 7 (Rechnernetze und Kommunikationssysteme) der FAU Erlangen-Nürnberg u.a. Netzwerkprogrammierung gelehrt und Programmierkonzepte für Robotersensornetze und die dafür benötigte Aufgabenverteilung sowie Selbstorganisation und Emergenz erforscht. Derzeit arbeitet er unabhängig und ohne Auftraggeber am FEPCOS-Projekt und forscht mit dem Ziel, die Programmierung von zusammengesetzten vernetzen Systemen zu vereinfachen.

Javas neue Gesprächskultur

Bernd Müller, Ostfalia

Java und C können sich über das Java Native Interface (JNI) unterhalten. Das geht ganz gut, Spaß macht es jedoch sicher nicht. Das Panama-Projekt versucht mit einer ganzen Reihe von JEPs, die Gesprächskultur von Java und C auf ein neues Niveau zu heben, und erlaubt es, nativen Code über pures Java ohne zusätzlichen C-Code aufzurufen.

Dieser Artikel ist eine kurze Einführung in das Foreign Function & Memory API mit dem Schwerpunkt auf die Funktionsschnittstelle. Er soll dem Leser einen ersten Eindruck vermitteln, wie das Zusammenspiel von Java und C funktioniert. Das API ist in Java 17 zwar noch im Inkubator-Modus, aber schon durchaus verwendungsfähig.

Die Entstehung von Panama

Leider können wir es nicht mit letzter Sicherheit sagen, doch die Entstehung des Panama-Projekts scheint seinen Ursprung im JEP 191 (Foreign Function Interface, *[1]*) zu haben. Dieser hatte das Ziel, native Methoden des Betriebssystems direkt durch Java-Methoden aufrufen und nativen Speicher direkt manipulieren zu können. Das JEP wurde kurz nach dem Erscheinen wieder zurückgezogen. Die zu lösenden Probleme waren wohl zu umfangreich und zu kompliziert für ein einzelnes JEP, sodass eine breitere Basis etabliert werden musste: das Panama-Projekt *[2]*, dessen Untertitel *„Interconnecting JVM and native code"* lautet.

Neben der Schnittstelle zu nativen Funktionen und nativem Speicher sind auch `jextract`, ein Werkzeug zur Generierung von Java-Interfaces aus C-Header-Dateien, und das Vector-API Teil von Panama. Wir gehen auf die beiden Letzteren an dieser Stelle nicht weiter ein.

Die Panama zuzurechnenden JEPs sind, was die Quantität angeht, mittlerweile recht umfangreich und umfassen die JEPs 370 *[3]*, 383 *[4]*, 389 *[5]*, 393 *[6]* und 412 *[7]*. Das JEP 370 wurde bereits mit Java 14 ausgeliefert, das JEP 412 mit Java 17. Dieses letzte JEP vereint auch das Funktions- und das Speicher-API. Da es das aktuellste Panama-JEP ist, soll hier dessen Zusammenfassung erwähnt werden:

„Introduce an API by which Java programs can interoperate with code and data outside of the Java runtime. By efficiently invoking foreign functions (i.e., code outside the JVM), and by safely accessing foreign memory (i.e., memory not managed by the JVM), the API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI."

Um die genannten negativen Eigenschaften von JNI ins Gedächtnis zurückzurufen, wollen wir ein typisches Hello-World mit JNI realisieren.

Das Java Native Interface

Java 1.0 enthielt bereits ein *Native Method Interface*. Dieses war jedoch an Interna der JVM gebunden und somit nicht plattformunabhängig. So waren etwa die Schnittstellen der jeweiligen JVMs von Sun und Microsoft verschieden. Mit Java 1.1 wurde eine plattformunabhängige Schnittstelle eingeführt, das *Java Native Interface (JNI) [8]*.

Die Verwendung sieht die Deklaration der nativen Methode mit dem `native` Modifier sowie das Laden der nativen Bibliothek mit `System.loadLibrary()` vor. Ein einfaches Hello-World auf dieser Grundlage zeigt *Listing 1*. Das JDK-Programm `javah`, seit Java 8 überführt in `javac` mit der Option `-h`, erzeugt aus dem Java-Code eine C-Include-Datei, die in *Listing 2* dargestellt ist.

Zu guter Letzt muss noch der C-Code erstellt werden, der aufgerufen wird. Dieser ist in *Listing 3* wiedergegeben. Der Code muss nun in eine Bibliothek des zugrunde liegenden Betriebssystems übersetzt werden, also eine Shared Library unter Linux beziehungsweise eine DLL unter Windows. Wir verzichten auf eine Darstellung der Übersetzungsbefehle sowie der Bibliothekserstellung und verweisen auf das GitHub-Projekt *[9]*, das den gesamten Code dieses Artikels enthält. Es bleibt noch anzumerken, dass Shared Libraries unter Linux die Dateinamenserweiterung `.so` und den Präfix `lib` haben, sodass die mit `System.loadLibrary("hello")` geladene Bibliothek den Namen `libhello.so` tragen muss.

Eine oberflächliche Analyse des Quellcodes zeigt, dass beim Aufruf einer C-Funktion durch eine Java-Methode immer eine Indirektion im Sinne einer zusätzlichen C-Funktion benötigt wird, die die Schnittstelle mithilfe der Include-Datei `jni.h` durch verschiedene Typdefinitionen und Datenstrukturen erst ermöglicht. Dies machte die Java-Entwicklergemeinde auf Dauer nicht glücklich, sodass das Projekt Java Native Access (JNA) *[10]* entstand, um diesen in jedem Projekt strukturell identischen Code überflüssig zu machen.

```
package de.pdbm;

public class HelloWorldJni {

  public static void main(String[] args) {
     System.loadLibrary("hello");
     new HelloWorldJni().sayHello();
  }

  private native void sayHello();

}
```

Listing 1

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class de_pdbm_HelloWorldJni */

#ifndef _Included_de_pdbm_HelloWorldJni
#define _Included_de_pdbm_HelloWorldJni
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     de_pdbm_HelloWorldJni
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_de_pdbm_HelloWorldJni_sayHello
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 2

```
#include <jni.h>
#include <stdio.h>
#include "de_pdbm_HelloWorldJni.h"

JNIEXPORT void JNICALL Java_de_pdbm_HelloWorldJni_sayHello(JNIEnv *, jobject) {
  printf("Hello World from C with JNI\n");
}
```

Listing 3

Das Projekt Panama ist ein weiterer Versuch in dieser Richtung, der allerdings im Gegensatz zu JNA in die Java-Sprachdefinition einfließen wird beziehungsweise schon als Inkubator eingeflossen ist.

JEP 412: Foreign Function & Memory API

Das JEP 412, Foreign Function & Memory API *[7]*, kurz FFM, definiert Schnittstellen, um Speicher außerhalb der VM zu allozieren, zu lesen und zu schreiben, den Lebenszyklus dieses Speichers zu beschreiben und letztendlich auch, um Funktionen außerhalb der VM aufzurufen. Damit dies mit verschiedenen Sprachen – nicht nur C und C++ – und verschiedenen Betriebssystemen funktioniert, ergibt sich ein recht hohes Abstraktionsniveau dieser Schnittstellen. Eine vollständige Darstellung verbietet sich daher in dieser kurzen Einführung und wird Thema eines zukünftigen Artikels.

Zunächst ist interessant, wie dem Anspruch einer einfacheren Verwendung als JNI Genüge getan wird. *Listing 4* zeigt die Panama-Version des Hello-World aus *Listing 1*. Der Leser wird recht erstaunt sein zu sehen, dass der Code umfangreicher ist. Dies ist dem Umstand geschuldet, dass die Methode `downcallHandle()` explizit verwendet werden muss, während das JNI-Äquivalent in der Include-Datei und der JVM-Implementierung versteckt ist. Das Ziel des `downcallHandle()`-Aufrufs ist es, ein Method-Handle (eingeführt in Java 1.7) zu bestimmen, das neben dem Bezeichner beziehungsweise der Adresse (`MemoryAddress`) sowohl der Java-seitigen (`MethodType`) als auch der C-seitigen Signatur (`FunctionDescriptor`) entspricht. Der Methodenaufruf erfolgt also ausschließlich mit Java-Bordmitteln und lässt damit ein wenig den Mehraufwand verschmerzen. Recht positiv fällt dagegen die Code-Reduktion auf der C-Seite ins Gewicht. *Listing 5* zeigt die vollständige C-Implementierung. Eine Include-Datei wird nicht benötigt.

Auch hier verweisen wir bezüglich des Bauens sowie des Aufrufs der Anwendung auf das GitHub-Projekt *[9]*. Hier soll lediglich angemerkt werden, dass das FFM-API auch unsichere (unsafe) Methoden verwendet und sich im Augenblick noch im Inkubator-Modus befindet.

```
package de.pdbm;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodType;
import java.util.Optional;

import jdk.incubator.foreign.CLinker;
import jdk.incubator.foreign.SymbolLookup;
import jdk.incubator.foreign.FunctionDescriptor;
import jdk.incubator.foreign.MemoryAddress;

public class HelloWorldPanama {

    public static void main(String[] args) throws Throwable {
  new HelloWorldPanama().sayHello();
    }

    public void sayHello() throws Throwable {
     System.loadLibrary("hello"); // libhello.so
     Optional<MemoryAddress> address = SymbolLookup.loaderLookup().lookup("hello");
     MethodHandle hello = CLinker.getInstance().downcallHandle(
            address.get(),
            MethodType.methodType(void.class),
            FunctionDescriptor.ofVoid());
     hello.invokeExact();
    }

}
```

Listing 4

```
#include <stdio.h>

void hello() {
    printf("Hello World from C with Panama\n");
}
```

Listing 5

Beim Aufruf der JVM sind daher die Optionen `--enable-native-access=ALL-UNNAMED` und `--add-modules jdk.incubator.foreign` zu verwenden.

Gesprächskultur ist immer beidseitig

Mit dem FFM-API ist auch die umgekehrte Richtung möglich: C ruft Java auf. Die Funktion `qsort()` der Standard-C-Bibliothek (`stdlib.h`) sortiert ein Array auf Basis eines Quicksort-Algorithmus. Die Signatur der Funktion stellt sich folgendermaßen dar:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Der Parameter `base` ist ein Zeiger auf das erste Element des Arrays. Da C nicht weiß, wie groß ein Array und seine Elemente sind, müssen diese Informationen angegeben werden. `nmemb` gibt die Anzahl der Elemente, `size` die Größe eines einzelnen Elements an. Der Parameter `compar` ist die Vergleichsfunktion, ganz analog zu der in Java etwa von `Collections.sort()` benötigten Comparator-Klasse.

```
package de.pdbm;

import jdk.incubator.foreign.CLinker;
import jdk.incubator.foreign.FunctionDescriptor;
import jdk.incubator.foreign.MemoryAccess;
import jdk.incubator.foreign.MemoryAddress;
import jdk.incubator.foreign.MemorySegment;
import jdk.incubator.foreign.ResourceScope;
import jdk.incubator.foreign.SegmentAllocator;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.util.Arrays;

import static jdk.incubator.foreign.CLinker.*;

/**
 * Simple example to call C from Java as well as Java from C
 *
 * Based on https://github.com/openjdk/panama-foreign/blob/foreign-jextract/doc/panama_ffi.md
 *
 * @author bernd
 *
 */
public class Quicksort {

    public static void main(String[] args) throws Throwable {
        qsort();
    }

    static class Comparator {
        static int compare(MemoryAddress addr1, MemoryAddress addr2) {
            int v1 = MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr1.toRawLongValue());
            int v2 = MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr2.toRawLongValue());
            return v1 - v2;
        }
    }

    public static void qsort() throws Throwable {
        MethodHandle qsort = CLinker.getInstance().downcallHandle(
                CLinker.systemLookup().lookup("qsort").get(),
                MethodType.methodType(void.class, MemoryAddress.class, long.class, long.class, MemoryAddress.
class),
                FunctionDescriptor.ofVoid(C_POINTER, C_LONG, C_LONG, C_POINTER)
        );

        MethodHandle comparHandle = MethodHandles.lookup()
                .findStatic(Comparator.class, "compare",
                        MethodType.methodType(int.class, MemoryAddress.class, MemoryAddress.class));

        try (ResourceScope scope = ResourceScope.newConfinedScope()) {
            MemoryAddress comparFunc = CLinker.getInstance().upcallStub(
                comparHandle,
                FunctionDescriptor.of(C_INT, C_POINTER, C_POINTER), scope);

            MemorySegment array = SegmentAllocator.ofScope(scope)
                                                  .allocateArray(C_INT, new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7
});
            qsort.invokeExact(array.address(), 10L, 4L, comparFunc);
            int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
            System.out.println(Arrays.toString(sorted));
        }
    }

}
```

Listing 6

Im GitHub-Repository des Panama-Projekts *[11]* findet man ein Beispiel zur Verwendung der `qsort`-Funktion, die als `compar`-Funktion eine Java-Methode verwendet. Java ruft also die C-Funktion `qsort()`, diese wiederum für jede Vergleichsoperation des Sortieralgorithmus eine Java-Methode auf. Das *Listing 6* zeigt eine Überarbeitung dieses Beispiels, dessen Code auch in unserem GitHub-Projekt *[9]* zu finden ist. Die Klasse `Comparator` mit der einzigen Methode `compare()` ist intuitiv gut zu verstehen, wenn wir von den FFM-Typen `MemoryAddress`, `MemoryAccess` und `MemorySegment` absehen.

In der Methode `qsort()` wird zunächst in bekannter Manier ein Method-Handle auf die C-Funktion `qsort()` erstellt. Im Gegensatz zu *Listing 4* wird nicht die Methode `SymbolLookup.loaderLookup()`, sondern `Clinker.systemLookup()` verwendet, die Symbole in der Standard-C-Bibliothek sucht. Das zweite Method-Handle `comparHandle` ist die `compare()`-Methode der Klasse `Comparator`. Im `try`-Block erfolgt letztendlich die Verdrahtung des Ganzen und der eigentliche Aufruf der Sortierfunktion.

Im Gegensatz zu `downcallHandle()` für die Aufrufrichtung *Java ruft C* verwendet man die Methode `upcallStub()` für die Richtung *C ruft Java*. Ebenfalls noch interessant ist die Speicherverwaltung. Da das zu sortierende Array außerhalb der JVM liegt, wird dessen Speicher nach Verwendung nicht „garbage-collected". Ein Speicherbereich mit sogenanntem *confined Scope* muss explizit durch Aufruf der `close()`-Methode wieder freigegeben werden. Da die Klasse `ResourceScope` das Interface `AutoCloseable` implementiert, wird das angelegte Array nach Beendigung des `try`-Blocks automatisch wieder freigegeben.

Wir wollen an dieser Stelle nicht zu sehr in die Details gehen, die durchaus recht komplex sind. Wir hoffen jedoch, dass der Leser mit unserer Einführung einen kleinen Einblick gewinnen konnte, wie sich die Verwendung des Foreign Function & Memory API anfühlt.

Zusammenfassung

Das FFM-API erlaubt es, dass C-Funktionen direkt von Java aufgerufen werden können. Es werden kein weiterer Glue-Code und keine Include-Datei benötigt. Diese Pure-Java-Lösung erkauft man sich mit einem nicht ganz trivialen API, das von den Implementierungsdetails des C-Compilers, des Betriebssystems und letztendlich auch von der konkreten Hardware abstrahiert. Das API befindet sich in Java 17 noch im Inkubator-Modus, scheint sich aber mit dem während des Entstehens dieses Artikels veröffentlichten JEP 419 *[12]* zu stabilisieren.

Referenzen

[1] JEP 191: Foreign Function Interface, *https://openjdk.java.net/jeps/191*

[2] Project Panama: Interconnection JVM and native Code, *https://openjdk.java.net/projects/panama/*

[3] JEP 370: Foreign-Memory Access API (Incubator), *https://youtu.be/0HXID3XZuOQ*

[4] JEP 383: Foreign-Memory Access API (Second Incubator), *https://openjdk.java.net/jeps/383*

[5] JEP 389: Foreign Linker API (Incubator), *https://openjdk.java.net/jeps/389*

[6] JEP 393: Foreign-Memory Access API (Third Incubator), *https://openjdk.java.net/jeps/393*

[7] JEP 412: Foreign Function & Memory API (Incubator), *https://openjdk.java.net/jeps/412*

[8] Java Native Interface Specification Contents, *https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html*

[9] *https://github.com/BerndMuller/Panama*

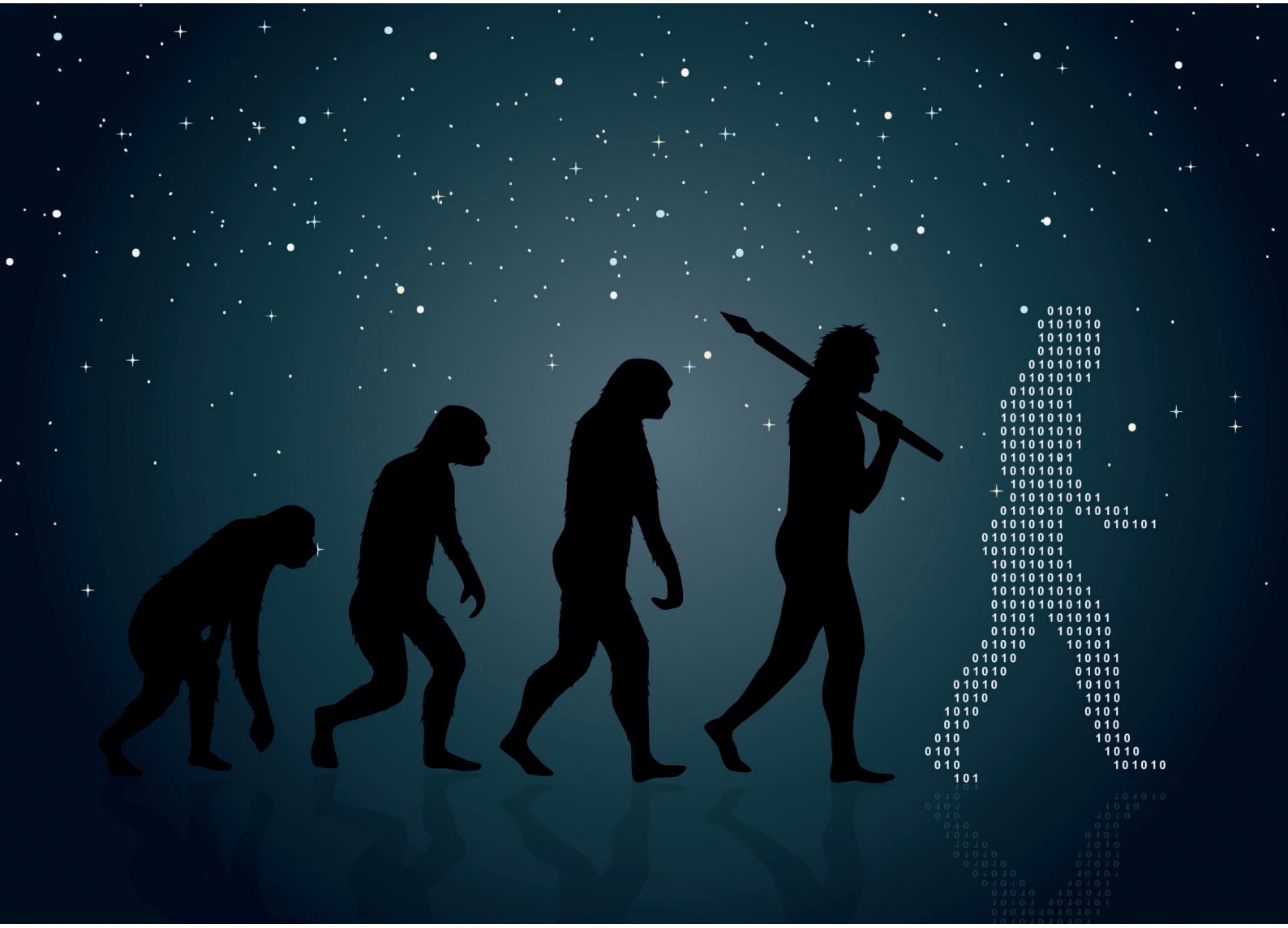
[10] Java Native Access, *https://github.com/java-native-access/jna*

[11] Panama-Foreign Repository, *https://github.com/openjdk/panama-foreign*

[12] JEP 419: Foreign Function & Memory API (Second Incubator), *https://openjdk.java.net/jeps/419*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.



Java und Evolution, wie passt das zusammen? Ein Simulationsprogramm

Andreas Gräßer und Andreas Lau

Seit einiger Zeit ist die Corona-Pandemie in aller Munde und man liest und hört in den Medien immer wieder von dieser, von evolutionären Prozessen, von Mutationen und so weiter. Den Autoren dieses Artikels ist nun wegen dieser „Dauerberieselung" der Gedanke gekommen, den evolutionären Prozess mithilfe eines sehr einfachen Simulationsprogramms ein wenig zu erhellen. Bevor auf dieses Programm eingegangen wird, sollen zunächst in aller Kürze einige Begriffe erläutert werden, die in diesem Zusammenhang eine Rolle spielen.

Evolution, Mutation ...

Unter Evolution versteht man die von Generation zu Generation stattfindende allmähliche Veränderung der vererbbaren Merkmale einer Population von Organismen. Unter Organismen wollen wir Lebewesen und Viren verstehen (Viren weisen zwar Merkmale von Lebewesen auf, gehören allerdings nicht zu den Lebewesen!). Die Erbinformationen sind in den Chromosomen (Bestandteile der Zellen) gespeichert. Ein Chromosom besteht im Wesentlichen aus Desoxyribonukleinsäure (DNA), in der die Gene codiert sind. Als Gen wird meist ein Abschnitt auf der DNA bezeichnet, der Grundinformationen für die Entwicklung von Eigenschaften eines Individuums enthält.

Als Mutation wird nun eine spontan auftretende, dauerhafte Veränderung des Erbgutes beziehungsweise der Gene bezeichnet. Eine Mutation kann Auswirkungen auf die Merkmale eines Organismus haben oder auch nicht. Die Merkmalsveränderungen können negative, positive oder auch gar keine Folgen hinsichtlich der Lebensfähigkeit bewirken.

Bereits 1838 entwarf Darwin seine Theorie der Anpassung von Organismen an ihren Lebensraum – unter anderem durch Mutationen und natürliche Selektion – und erklärte so die Entwicklung der Lebewesen sowie ihre Aufspaltung in verschiedene Arten. Ganz plakativ kann man diesen Sachverhalt kurz und bündig folgendermaßen erklären:

Survival of the fittest!

Diesen Satz hat übrigens nicht Darwin, sondern der britische Sozialphilosoph Herbert Spencer geprägt.

Nach diesem kurzen Ausflug in die Genetik nun wieder zurück zum weiter oben schon erwähnten Simulationsprogramm. Im nächsten Abschnitt wird zunächst erläutert, wie man mit einfachen Modellen Genketten, Mutationen etc. abbilden und auf dieser Grundlage ein Simulationsprogramm entwickeln kann.

Modell zur Nachbildung von Evolution und Mutation, Basis für ein sehr einfaches Simulationsprogramm

Eine „sinnlose" Zeichenkette soll als Modell für eine aktuell vorhandene Genkombination eines Lebewesens oder eines Virus dienen. Ein Lebewesen beziehungsweise Virus mit dieser Genkombination ist nicht optimal an seine Umwelt angepasst. Diese Zeichenfolge soll mit *realGenKom* bezeichnet werden.

Zum Beispiel: *realGenKom* = *awedrtgbdiuzrgt*

Eine eher „sinnvolle" Zeichenkette soll als Modell für eine optimale Genkombination des Organismus eines Lebewesens oder eines Virus dienen. Ein Lebewesen oder Virus mit dieser Genkombination wäre optimal an seine Umwelt angepasst. Diese Zeichenfolge soll mit *idealGenKom* bezeichnet werden.

Zum Beispiel: *idealGenKom* = *dasistevolution*

Wir wollen für die folgenden Ausführungen vereinbaren, dass die Zeichenketten *realGenKom* und *idealGenKom* nur aus Kleinbuchstaben a, ..., z bestehen sollen. Sonderzeichen wie ä, ö, ü und Leerzeichen sind nicht erlaubt. Ferner sollen diese Zeichenketten immer die gleiche Anzahl von Zeichen beinhalten.

Jetzt haben wir bereits Modelle für Genkombinationen entwickelt. Nun kommt der nächste Schritt. Wir müssen Mutationen nachbilden. Das ist ziemlich einfach. Das entsprechende Modell besteht darin, dass man in der Zeichenkette *realGenKom* per Zufall eine Spalte aussucht. Das in dieser Spalte stehende Zeichen wird dann durch ein wiederum per Zufall ausgesuchtes Zeichen ersetzt (dieses Zeichen wird im Allgemeinen von dem ursprünglichen Zeichen abweichen, kann aber auch mit dem ursprünglichen Zeichen übereinstimmen).

Beispiel:

realGenKom = *awedrtgbdiuzrgt* = *realGenKomAlt*

↓

Mutation

↓

realGenKom = *awedrtgbziuzrgt* = *realGenKomMut*

Nun muss noch ermittelt werden, ob das Lebewesen oder Virus durch diese Mutation „fitter" wird, das heißt, ob es in seinem Lebensraum bessere Überlebenschancen hat. Dazu wird eine Größe benötigt, die die Ähnlichkeit zwischen *idealGenKom* und *realGenKomAlt* sowie zwischen *idealGenKom* und *realGenKomMut* abbildet.

Eine solche Größe, sie soll mit *uebereinstimmungsgrad* bezeichnet werden, kann man gewinnen, indem man die Anzahl der gleichen Zeichen in den jeweils gleichen Spalten von *idealGenKom* und *realGenKomAlt* beziehungsweise *realGenKomMut* ermittelt. In unserem Modell soll nun der *uebereinstimmungsgradAlt* zwischen *idealGenKom* und *realGenKomAlt* sowie der *uebereinstimmungsgradMut* zwischen *idealGenKom* und *realGenKomMut* berechnet werden.

Wenn *uebereinstimmungsgradMut* > *uebereinstimmungsgradAlt* gilt, kann man von einer durch die Mutation verursachten Verbesserung der Situation ausgehen. Die weiteren Mutationen basieren dann auf *realGenKom* = *realGenKomMut*. Andernfalls basieren die weiteren Mutationen auf *realGenKom* = *RealGenKomAlt*, man geht also wieder einen Schritt zurück.

Im entsprechenden Simulationsprogramm werden der Mutationsprozess sowie die anschließende Beurteilung, ob sich die Mutation „gelohnt" hat, in einer Schleife so lange wiederholt, bis *realGenKom* = *idealGenKom* gilt. Zum Schluss soll noch die Anzahl der Mutationen *anzahlMut* angegeben werden, die bis zum Erreichen dieses Zieles notwendig waren. Mit dem in *Abbildung 1* dargestellten Struktogramm soll das Simulationsprogramm verdeutlicht werden.

Am Ende dieses Artikels *(siehe Listing 1)* ist ein kleines Java-Programm aufgeführt, das in etwa dem Struktogramm von *Abbildung 1* entspricht. Das Programm ist allerdings etwas komfortabler, dort wird jeweils noch die Anzahl der Mutationen ausgegeben, die erforderlich war, bis eine Mutation zu einer Verbesserung der Übereinstimmung zwischen *idealGenKom* und *realGenKom* geführt hat (es werden also noch Zwischenschritte angezeigt).

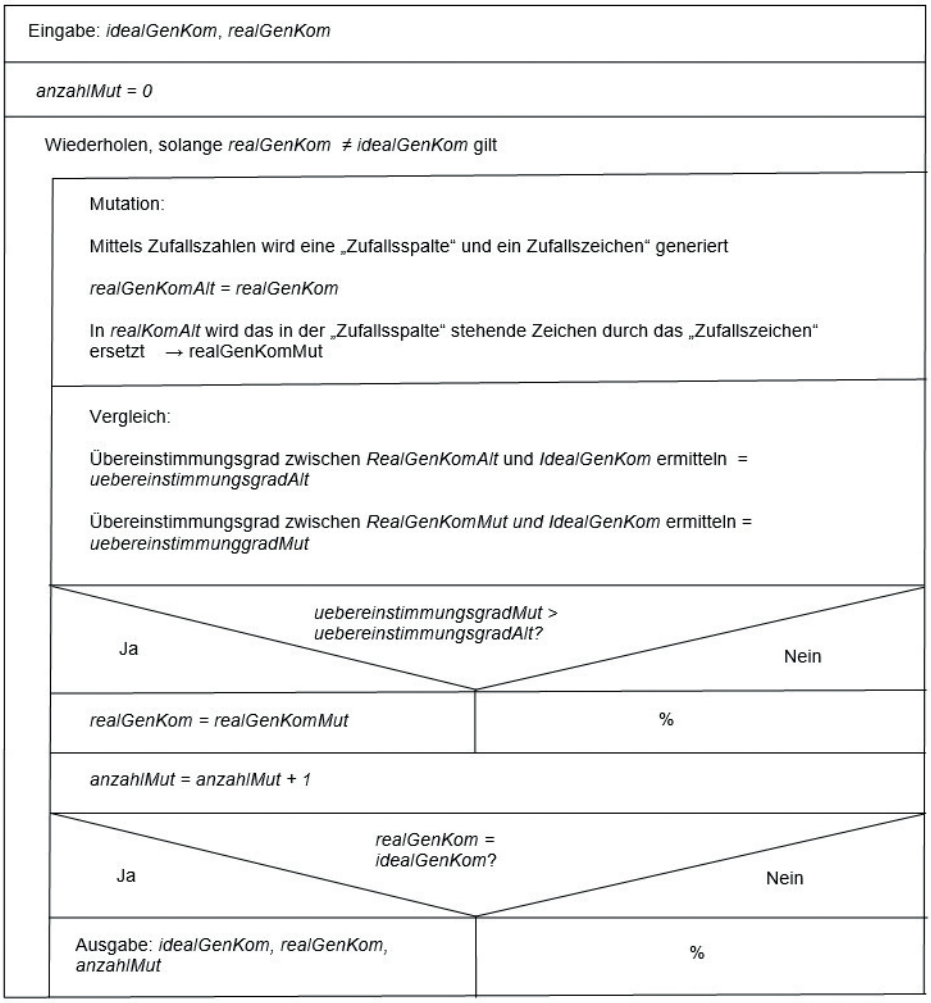


Abbildung 1: Struktogramm des Simulationsprogrammes

idealGenKom	realGenKom	Mutationen, Mittelwert
a (Zeichenzahl = 1)	h (Zeichenzahl = 1)	3, 46, 26, 0, 23, 11, 72, 16, 44, 5 Mittelwert 25
Ah (Zeichenzahl = 2)	bf (Zeichenzahl = 2)	84,11,142,169, 119, 33, 75, 43, 88, 219 Mittelwert 98
hey (Zeichenzahl = 3)	hto (Zeichenzahl = 3)	43, 93, 2, 26, 153, 120, 251, 65, 61, 70 Mittelwert 88
hallo (Zeichenzahl = 5)	ftdio (Zeichenzahl = 5)	62, 185, 356, 441, 88, 60, 203, 415, 285, 671 Mittelwert 277
mutationen (Zeichenzahl = 10)	mermkodrbk (Zeichenzahl = 10)	616, 1090, 768, 1244, 557,1428, 832, 483, 1033, 421 Mittelwert 847
dasistevolution (Zeichenzahl = 15)	vfzkmhwpmntfgzd (Zeichenzahl = 15)	1221, 1013, 1673, 636, 1330, 866, 2799, 703,1300, 1719 Mittelwert 1326
jetztgibtsnochmehrzeichen (Zeichenzahl = 25)	edrtbnjkiufdoplkgtzufrswe (Zeichenzahl = 25)	1074, 2135, 4447, 2702, 2060, 2133, 2384, 1767, 2354, 1282 Mittelwert 2234

Tabelle 1: Versuchsergebnisse, Abhängigkeit der benötigten Mutationen bis zum Erreichen des Idealzustandes (realGenKom = idealGenKom)

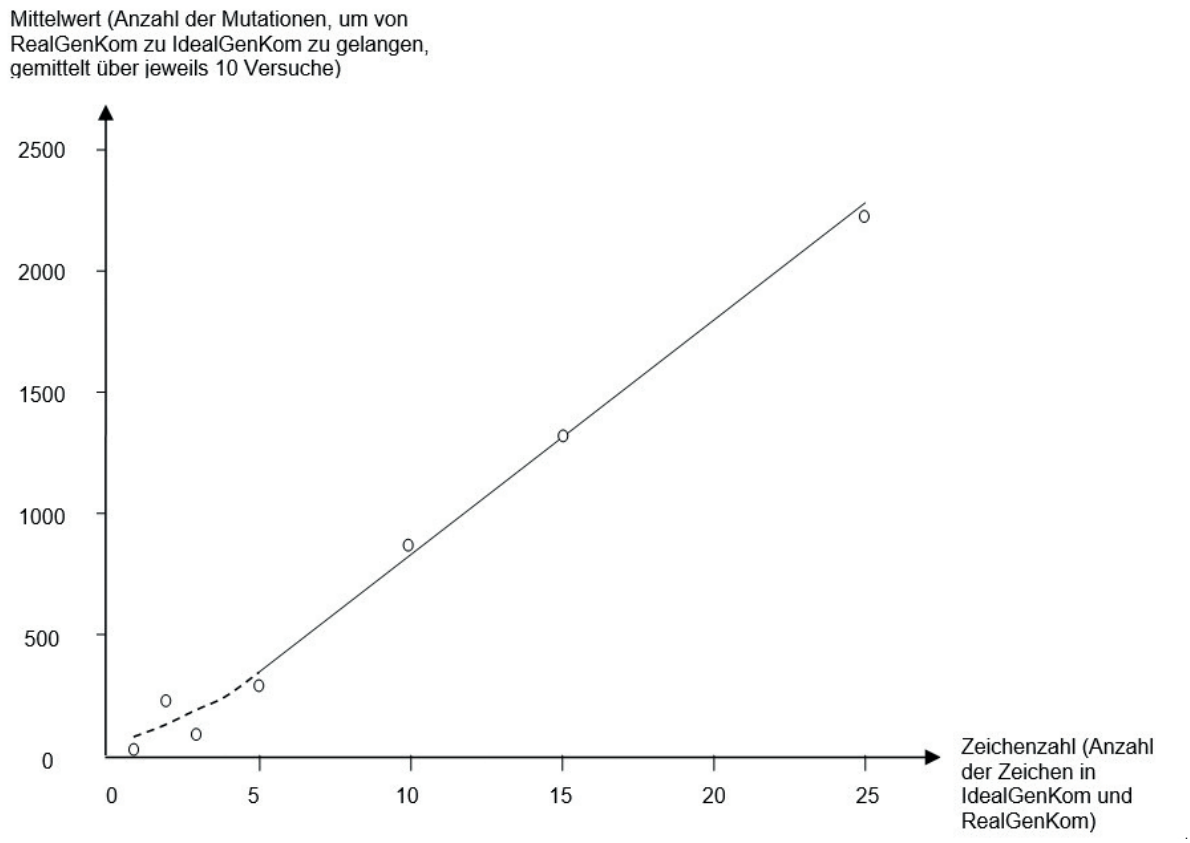


Abbildung 2: Grafische Darstellung der in Tabelle 1 aufgelisteten Messwerte

Das Programm ist sehr einfach gestaltet. Es muss über die Entwicklungsumgebung gestartet werden. Die Dateneingabe und -ausgabe geschehen über die Konsole. Das ist beabsichtigt. Das Programm soll zum Experimentieren anregen, Programmänderungen sollen schnell und einfach möglich sein.

Experimente

Besonders interessant sind Experimente, die Hinweise liefern könnten, wie viele Mutationen nötig sind, um den Idealzustand *realGenKom* = *idealGenKom* zu erreichen. Und natürlich ist es auch von Interesse, zu erfahren, wie die Anzahl der benötigten Mutationen bis zum Idealzustand von der Länge der Zeichenkette *realGenKom* beziehungsweise *idealGenKom* abhängt. Die Autoren haben entsprechende Experimente durchgeführt. In *Tabelle 1* sind einige Versuchsergebnisse dargestellt. Da es sich bei den Versuchen um Zufallsprozesse handelt, wurden immer jeweils zehn Versuche mit Zeichenketten einer definierten Zeichenzahl durchgeführt. Die Ergebnisse wurden gemittelt.

Die grafische Darstellung dieser Versuchsergebnisse ist *Abbildung 2* zu entnehmen. Man kann erkennen, dass ab einer Zeichenzahl von 5 offensichtlich ein linearer Zusammenhang zwischen dem Mittelwert und der Zeichenzahl gegeben ist. Bei einer kleineren Zeichenzahl sind die Mittelwerte nicht mehr aussagekräftig, sie streuen sehr stark, man kann keine (stetige) Funktion zur Verbindung dieser Werte angeben.

Selbstverständlich kann man viele weitere Experimente durchführen, aber das würde den Rahmen dieses Artikels sprengen.

Noch ein paar Bemerkungen zum Abschluss

Die Autoren waren ziemlich überrascht, dass relativ wenige Mutationen bis zum Erreichen des Idealzustandes (realGenKom = idealGenKom) erforderlich waren. Speziell bei längeren Zeichenketten hatte man beträchtliche „Rechenarbeit" bis zum Erreichen des Zieles prognostiziert. Es war auch nicht unbedingt zu erwarten, dass der in *Abbildung 2* dargestellte Zusammenhang zwischen der Anzahl der Mutationen in Abhängigkeit von der Zeichenzahl linear verläuft.

Den Autoren ist selbstverständlich bewusst, dass das hier vorgestellte Simulationsprogramm den Evolutionsprozess nur sehr grob und oberflächlich darstellt. Aber vielleicht regt es ja an, sich mit diesem Thema eingehender zu befassen? Das hier vorgestellte Programm ist sehr überschaubar, ziemlich selbsterklärend und kann leicht verändert oder erweitert werden. Vielleicht ist es deshalb gerade für Programmieranfänger interessant? Und man kann mit diesem Programm wunderbar experimentieren. Viel Spaß!

Quellen

[1] Wikipedia, Erläuterungen zu den Begriffen Evolution, Mutation usw.

[2] Bionik, Natur als Vorbild, Herausgegeben von der Umweltstiftung WWF und PRO FUTURA Vertrieb GmbH, 1993

```
import java.io.PrintStream;
import java.security.SecureRandom;
import java.util.Random;
import java.util.Scanner;
import java.util.StringJoiner;

public class Evolution {
    private static final PrintStream out = System.out;
    private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        usage();

        Gene idealGenKom
                = Gene.fromString(getGeneSequenceAsString());
        Gene realGenKom
                = Gene.fromString(getGeneSequenceAsString(idealGenKom.getSequence()));

        int anzahlMut = 0;
        while (!realGenKom.equals(idealGenKom)) {
            anzahlMut++;

            Gene realGenKomMut = realGenKom.mutateTo();
            int ueGradAlt = realGenKom.getUebereinstimmungsgrad(idealGenKom);
            int ueGradMut = realGenKomMut.getUebereinstimmungsgrad(idealGenKom);

            if (ueGradMut > ueGradAlt) {
                realGenKom = realGenKomMut;
                out.format("%d: %s%n", anzahlMut, realGenKom);
            }

            if (realGenKom.equals(idealGenKom)) {
                out.format("Anzahl der  Mutationen bis zur Idealkombination: %d%n", anzahlMut);
                out.format("realGenKomMut = idealGenKom = %s%n", realGenKom);
                break;
            }
        }
    }

    private static boolean isNotValid(String input) {
        for (char c : input.toCharArray()) {
            if (Gene.ALPHABET.indexOf(c) == -1) return true;
        }
        return false;
    }

    private static String getGeneSequenceAsString() {
        out.print("idealGenKom eingeben: ");
        String idealGenKom = scanner.next();

        while (idealGenKom.isEmpty() || isNotValid(idealGenKom)) {
            out.print("idealGenKom hat falsche zeichenzahl, Eingabe wiederholen: ");
            idealGenKom = scanner.next();
        }
        return idealGenKom;
    }

    private static String getGeneSequenceAsString(String idealGenKom) {
        out.print("realGenKom eingeben: ");
        String realGenKom = scanner.next();

        while (realGenKom.length() != idealGenKom.length()
                || realGenKom.equals(idealGenKom) || isNotValid(realGenKom)) {
            out.println("realGenKom hat falsche zeichenzahl oder ist mit ");
            out.println("idealGenKom identisch, Eingabe wiederholen:");
            realGenKom = scanner.next();
        }
        return realGenKom;
    }

    private static void usage() {
        out.println("Achtung für die folgenden Eingaben: Cursor auf die Konsole");
        out.println("platzieren, Eingaben mit Enter abschließen!");
        out.println();
    }
}

class Gene {
    public static final String ALPHABET = "abcdefghijklmonprstuvwxyz";
    private static final Random RANDOM = new SecureRandom();
    private final String sequence;
```

```
    private Gene(String sequence) {
        this.sequence = sequence;
    }

    public static Gene fromString(String sequence) {
        return new Gene(sequence);
    }

    public int getUebereinstimmungsgrad(Gene ideal) {
        int ueGrad = 0;
        String idealSequence = ideal.getSequence();
        for (int i = 0; i < idealSequence.length(); i++)
            if (idealSequence.charAt(i) == sequence.charAt(i)) ueGrad++;
        return ueGrad;
    }

    public Gene mutateTo() {
        int zufallsSpalte = RANDOM.nextInt(sequence.length());
        char zufallsZeichen = ALPHABET.charAt(RANDOM.nextInt(ALPHABET.length()));
        char[] newSequence = sequence.toCharArray();
        newSequence[zufallsSpalte] = zufallsZeichen;
        return Gene.fromString(new String(newSequence));
    }

    public String getSequence() {
        return sequence;
    }

    @Override
    public String toString() {
        return new StringJoiner(", ", Gene.class.getSimpleName() + "[", "]")
                .add("sequence='" + sequence + "'")
                .toString();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Gene gene = (Gene) o;
        return getSequence().equals(gene.getSequence());
    }

    @Override
    public int hashCode() {
        return getSequence().hashCode();
    }
}
```

Listing 1

Dr. Andreas Gräßer

ahgraesser@web.de

Andreas Gräßer studierte Elektrotechnik an der Technischen Universität Berlin, promovierte und arbeitete mehrere Jahre in der Industrie. Anschließend war er als Dozent an der Hochschule Darmstadt im Fachbereich Elektrotechnik und Informationstechnik tätig. Seit 2008 ist er pensioniert.

Andreas Lau

andi@herrlau.de

Andreas Lau studierte Informatik an der Hochschule für Technik und Wirtschaft in Berlin und ist seit seinem Studienabschluss freiberuflich tätig.

Die perfekte Retrospektive – So finden Sie das passende Format!

Dr. Dominic Lindner und Martin Weber

Eine Retrospektive im Kontext von Unternehmen entspricht, vereinfacht gesagt, einem Blick in die Vergangenheit eines bestimmten Ereignisses, etwa eines Projektes oder eines Scrum-Sprints, und beinhaltet oftmals eine Auseinandersetzung vergangener Ereignisse mit dem Ziel, etwas zu lernen.

Phase	Erklärung
Phase 1: Intro (Onboarding)	Nach einer kurzen Begrüßung stellt die/der Moderator/in eine kurze „Warm-up-Frage". So können alle Teilnehmer/innen in die Thematik eingeführt werden. Zudem lässt sich dadurch die Beteiligung im Verlauf der Retrospektive steigern.
Phase 2: Daten sammeln (Gather data)	In dieser Phase werden für die Retrospektive relevante Fragen gestellt. Diese richten sich nach dem gewählten Format der Retrospektive. Auf die Formate wird im Verlauf dieses Beitrags noch detaillierter eingegangen.
Phase 3: Einsichten gewinnen (Generate Insights)	Nun gilt es, die Antworten sinnvoll zu analysieren: „Warum sind die Dinge, wie sie sind?" Die Auswertung richtet sich ebenfalls nach dem gewählten Format der Retrospektive.
Phase 4: Maßnahmen beschließen (Decide what to do)	Das Ziel ist es nun, aus den Einsichten umsetzbare Maßnahmen zu entwickeln.
Phase 5: Abschluss (Offboarding)	Moderator/in und Teilnehmer/innen werfen einen kurzen Rückblick auf die Retrospektive und geben Feedback zum Ablauf. Das erlaubt dem/der Moderator/in, die nächste Retrospektive zu verbessern.

Tabelle 1: Fünf Phasen einer Retrospektive

In der Regel schauen die Team- oder Projektmitglieder gemeinsam zurück und bewerten, was gut und was schlecht gelaufen ist. Anschließend werden diese Erkenntnisse analysiert, um daraus Maßnahmen für Verbesserungen abzuleiten. Abhängig vom Bedarf finden Retrospektiven in Scrum alle zwei Wochen und in Projekten zumeist einmalig nach Abschluss statt.

Als Vorteile einer solchen Retrospektive lassen sich die Verbesserung der Arbeitsabläufe, der Abbau von Frust und das Teambuilding nennen; zudem bietet sie Raum, um Verbesserungsvorschläge „geschützt" und „moderiert" anzusprechen.

Retrospektiven sind nicht nur in der „agilen Welt" üblich. Für die Studie „Status Quo Agile 2020" *[1]* wurden knapp 500 Projektmanager/innen befragt, mit dem Ergebnis, dass Retrospektiven in über zwei Dritteln der Projekte ein fester Bestandteil sind.

Allgemeiner Ablauf einer Retrospektive

Eine Retrospektive folgt grundsätzlich einem ähnlichen Ablauf. Allgemein lässt sich dieser in fünf Phasen gliedern, die in der folgenden *Tabelle 1* zusammenfassend beschrieben sind.

Wir gehen davon aus, dass Ihnen das Prinzip der Retrospektiven generell bekannt ist und Sie schon an einer Retrospektive teilnehmen konnten. Aus diesem Grund wurde der Ablauf nur kurz beschrieben. Sollten Sie weiterführende Informationen zum allgemeinen Ablauf benötigen, finden Sie in der Quellenangabe am Ende des Artikels eine Referenz auf einen Artikel, in dem diese fünf Phasen detaillierter beschrieben werden *[2]*.

Retrospektive ist nicht gleich Retrospektive

Der Ablauf der Retrospektiven erfolgt in der Regel entsprechend den erwähnten fünf Phasen. Die Unterschiede ergeben sich aus dem Schwerpunkt und den gestellten Fragen. Auch geringfügige Unterschiede können hier wesentliche Änderungen verursachen.

Zudem hat unsere Erfahrung gezeigt, dass nicht jede Retrospektive automatisch zur Folge hat, dass sich die Organisation verbessert. Jede Retrospektive ist genau zu planen, damit beim Ablauf verschiedene Formate passend auf den Kontext angewendet werden.

Hierzu sollen im Folgenden ausgewählte Formate aus unserer Praxiserfahrung vorgestellt werden. Jede Retrospektive wird in einer Art Steckbrief kompakt für Sie zusammengefasst. Eine Vorlage zu jeder Retrospektive können Sie sich kostenlos über die im Anhang angegebene Quelle downloaden.

Format 1: Mad-Sad-Glad *[3]*

Kurzbeschreibung

Es handelt sich um ein Format, in dem Themen auf Basis einer simplen Einschätzung daraufhin gesammelt werden, ob die Teilnehmer/innen sie als gut, eher schlecht oder sehr schlecht einordnen.

Anwendungsfall – Vorteile

Das Format eignet sich besonders, um allgemeine Themen im Team zu finden, ohne bereits eine konkrete Lösung vor Augen zu haben.

Ablauf

Während der Retrospektive sammeln die Teilnehmer im ersten Schritt Themen und notieren diese auf Post-its, die sie einer der drei Spalten zuordnen. Anschließend werden die gesammelten Themen allgemein besprochen. So wird ein gemeinsames Verständnis geschaffen. Die Teilnehmer/innen stimmen anschließend ab, welche Themen sie besprechen möchten.

Diese Themen werden ausführlicher diskutiert mit dem definierten Ziel, das Problem zu durchdringen, Lösungen oder Schritte hin zur Lösung zu überlegen und eine/n Verantwortliche/n zu benennen, die/der sich darum kümmern wird. Bei der nachfolgenden Retrospektive erfolgt dann zu Beginn ein Rückblick auf die definierten Aufgaben mit der Überprüfung, ob diese erledigt wurden.

Unsere Erfahrung

Unsere Erfahrung zeigt, dass sich diese Art der Retrospektive besonders für Situationen eignet, in denen Probleme vorhanden zu sein scheinen, aber das Team nicht genau sagen kann, was zu verbessern wäre. Dieses Format kann schnell zu allgemein werden. Daher ist darauf zu achten, dass Sie direkt Nachfragen stellen, um vom konkreten, persönlich wahrgenommenen Problem zu einer Lösung zu kommen.

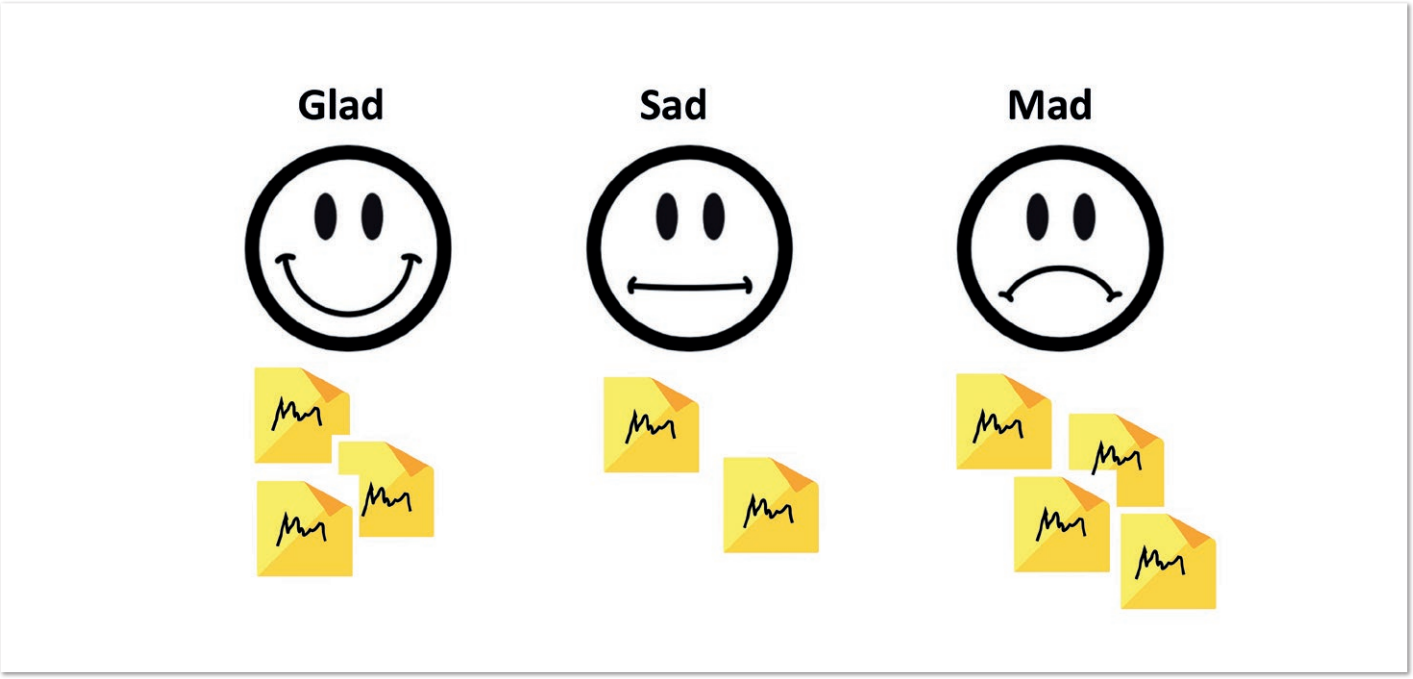


Abbildung 1: Glad-Sad-Mad-Methode

Format 2: Start-Stop-Continue *[4]*

Kurzbeschreibung

Bei diesem Format stehen konkrete Maßnahmen im Fokus. Es unterscheidet sich von Glad-Sad-Mad dadurch, dass explizit nach Maßnahmen gefragt wird, die sich als gut erwiesen haben und die beibehalten werden sollen, und nach solchen, die gestoppt werden sollten.

Anwendungsfall – Vorteile

Das Format ist besonders geeignet, wenn schon konkrete Ideen im Raum stehen und wenn die Probleme dem Team größtenteils bekannt sind und nicht erst analysiert werden müssen.

Ablauf

Der Ablauf der Retrospektive entspricht dem allgemeinen Ablauf und unterscheidet sich lediglich dadurch, dass andere Fragen gestellt werden wie: Was sollen wir starten, was sollen wir nicht mehr tun, was sollen wir beibehalten?

Unsere Erfahrung

Unsere Erfahrung zeigt, dass sich diese Art der Retrospektive besonders für Situationen eignet, in denen die Probleme offensichtlich oder im Team zumindest bekannt sind. Es geht weniger Zeit verloren, um von der Befindlichkeit zum Problem zu gelangen, sodass mehr Zeit für das Besprechen möglicher Maßnahmen bleibt. Es empfiehlt sich, dieses Format nach dem Glad-Sad-Mad-Format anzuwenden.

Abbildung 2: Start-Stop-Continue-Methode

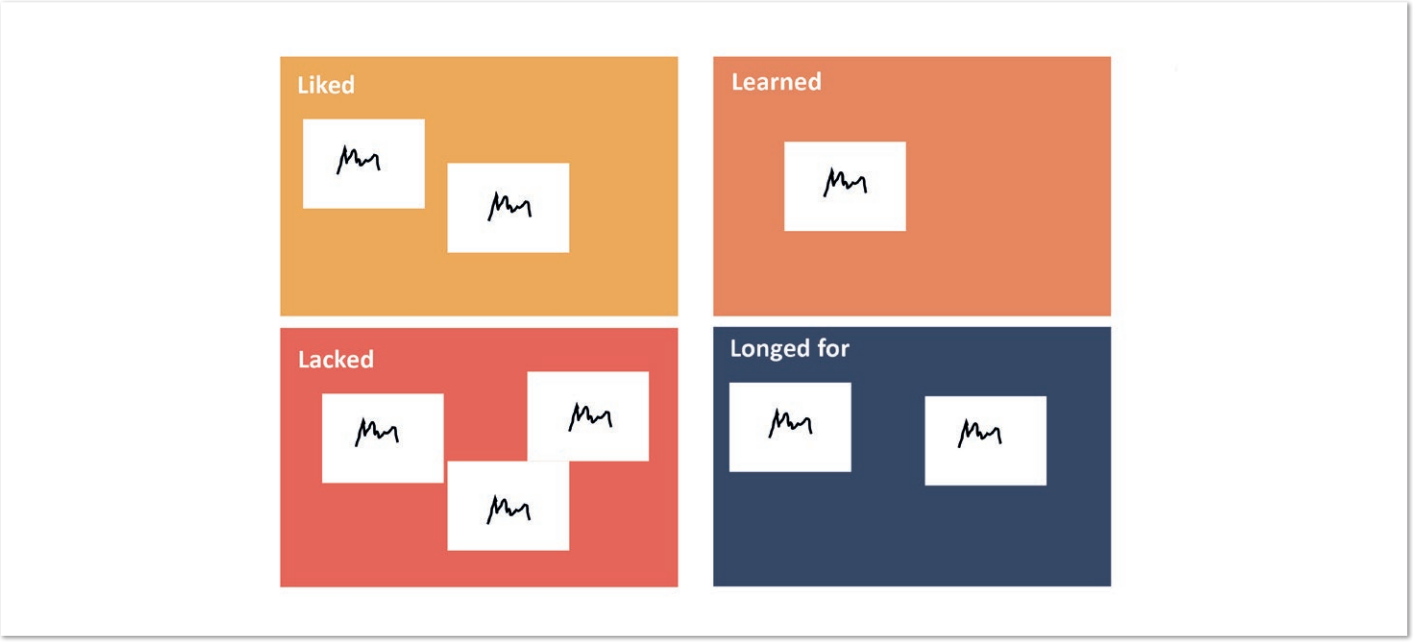


Abbildung 3: Liked-Learned-Lacked-Longed-for-Methode

Format 3: 4 Ls (Liked, Learned, Lacked & Longed For) *[5]*

Kurzbeschreibung

Hierbei handelt es sich um ein Format, in dem sowohl stimmungsbezogene Themen als auch konkrete Erkenntnisse und Probleme behandelt werden. Ausgangspunkte sind die persönlichen Erfahrungen der Teilnehmer/innen.

Anwendungsfall – Vorteile

Diese Retrospektive motiviert die Teilnehmer/innen mit dem Blick aus der persönlichen Perspektive auf die positiven und negativen Erfahrungen, auch Themen anzusprechen, die ihnen auf den ersten Blick nicht bedeutsam erscheinen. Dadurch kann neues Wissen gefördert werden, wodurch ein Ausbrechen aus der Routine ermöglicht wird, etwa wenn Sie seit mehreren Wochen wiederholt die gleichen Punkte in den Retrospektiven besprechen.

Ablauf

Der Ablauf der Retrospektive erfolgt grundsätzlich nach dem allgemeinen Prinzip, wie oben beschrieben. In diesem Fall lauten die Fragen: Was hat mir gefallen? Welche neuen Erkenntnisse konnte ich gewinnen? Wo haben mir Dinge gefehlt, die ich gebraucht hätte? Was sollten wir konkret tun?

Unsere Erfahrung

Unsere Erfahrung zeigt, dass diese Retrospektive vorteilhaft ist, um die Monotonie von klassischen Formaten wie Mad-Glad-Sad oder Start-Stop-Continue aufzulockern.

Besonders die Einbeziehung der positiven Erfahrungen erweist sich als geeignete Ergänzung zum häufig auf die negativen Punkte fokussierten Blick.

Format 4: Energie-Level *[6]*

Kurzbeschreibung

In der Energie-Level-Retrospektive lösen wir uns vom Blick auf das Produkt und die Prozesse und gehen konkreter auf die Befindlichkeiten der einzelnen Teilnehmer/innen ein.

Anwendungsfall – Vorteile

Dieses Format eignet sich, um ein Stimmungsbild im Team zu erhalten, insbesondere dann, wenn der Verdacht besteht, dass einzelne Teammitglieder unzufrieden sind. So können Probleme frühzeitig erkannt werden und es lassen sich Gegenmaßnahmen in die Wege leiten.

Ablauf

Der Ablauf der Retrospektive erfolgt grundsätzlich nach dem allgemeinen Prinzip, wie oben beschrieben. Schwerpunkt der Betrachtung sind die Fragestellungen: Was treibt mich an? Was war gut? Was macht mich fertig? Wie ist meine Motivation für die kommenden Aufgaben?

Unsere Erfahrung

Unsere Erfahrung zeigt, dass sich diese Retrospektive für Situationen eignet, in denen der Verdacht einer deutlichen Unzufriedenheit im Team oder bei einzelnen Teammitgliedern besteht. Manchen Personen fällt es möglicherweise schwer, über Befindlichkeiten oder auch Zufriedenheit zu sprechen, doch ist es hilfreich, in regelmäßigen Abständen die Stimmung und die Motivation im Team zu erfragen.

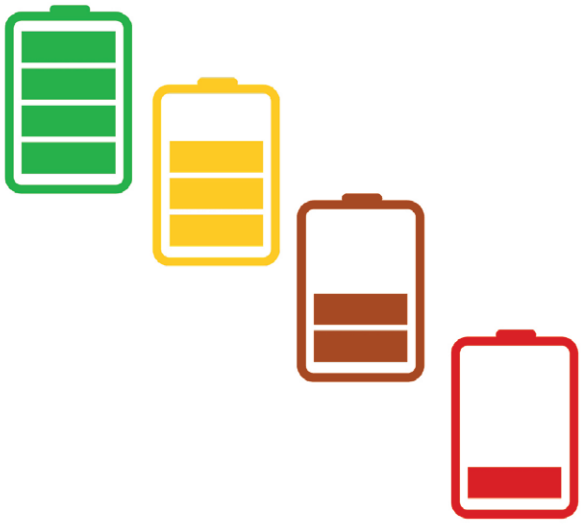
Format 5: Basketball-Retrospektive *[6]*

Kurzbeschreibung

Die Basketball-Retrospektive zielt darauf ab, dass sich das Team selbst reflektiert, um zu ermitteln, wer im Team welche Funktion einnimmt. Dies kann analog zu einem Basketball-Team gesehen werden, wo es verschiedene Spielpositionen und Rollen im Team gibt.

Anwendungsfall – Vorteile

Durch diese Retrospektive soll verdeutlicht werden, wer im Team welche Aufgabe übernimmt, mit dem Ziel, die Zusammenarbeit zu stärken.



Fully Charged
Was treibt mich an?

Medium Power
Was war ok?

Low Battery
Was macht mich ziemlich fertig?

Sprint Energy
Wie ist meine Motivation für die kommenden Aufgaben?

Abbildung 4: Energie-Level-Methode

Die eigene Wahrnehmung wird mit der der anderen Teilnehmer/innen verglichen und es zeigen sich gegebenenfalls Abweichungen oder es werden Aufgaben benannt, für die sich niemand verantwortlich fühlt.

Ablauf

Im ersten Schritt reflektieren die Teilnehmer/innen folgende Fragen: Was habe ich im letzten Sprint geschafft? Wie konnte ich anderen helfen? Wo habe ich Hilfe erhalten und wie sehe ich meine Rolle im Team?

Hier weicht das Format vom allgemeinen Ablauf ab: Basierend auf den Antworten auf die genannten Fragen versuchen die Teilnehmer/innen das eigene Team wie eine Basketball-Mannschaft zu modellieren und aufzuzeigen, wer welche Funktion übernimmt.

Wichtig: Beschränken Sie sich nicht auf Jobbezeichnungen wie Entwickler oder UX-Designer, sondern versuchen Sie auch, komplexere Beziehungen wie Mentor-Mentee oder Ähnliches abzubilden.

Shoot
Was habe ich geschafft?

Assist
Wie kann ich anderen helfen?

Pass
Wie helfen mir andere?

Position
Was ist meine Rolle im Team?

Abbildung 5: Basketball-Methode

Unsere Erfahrung

Unsere Erfahrung zeigt, dass sich diese Retrospektive gut eignet, wenn das Gefühl besteht, dass nicht alle Abläufe im Team reibungslos funktionieren und die Teilnehmer/innen unterschiedliche Vorstellungen davon haben, was sie gut können und was die Teamkolleg/innen denken. Ein Abgleich kann helfen, Problemen frühzeitig entgegenzusteuern und die Abläufe sowie die Motivation im Team zu verbessern.

Format 6: Three little Pigs *[6]*

Kurzbeschreibung

Das Format zielt darauf ab, das Team, das Produkt und die Prozessabläufe von außen zu betrachten und zu bewerten. Es ist angelehnt an das Märchen von den „Drei kleinen Schweinchen".

In diesem Märchen ziehen drei kleine Schweinchen von Zuhause aus. Jedes baut sich ein Haus aus einem anderen Material: Stroh, Holz und Stein. Als der Wolf angreift, zeigt sich, welches Haus den besten Schutz bietet.

Analog hierzu werden in diesem Format Aspekte gesammelt, die stabil, solche, die in Ordnung, aber optimierungsbedürftig sind, und solche, die als stark angreifbar/instabil empfunden werden.

Anwendungsfall – Vorteile

Das Format ist geeignet, um das Produkt sowie die Prozesse der Organisation zu verbessern. Es stellt damit das Gegenstück zur Energie-Level-Retrospektive mit Fokus auf die einzelnen Teammitglieder dar.

Ablauf

Der Ablauf der Retrospektive folgt grundsätzlich dem allgemeinen Prinzip. Der Unterschied liegt wieder in den Fragestellungen: Wo sehen wir ein Risiko (Stroh)? Wo müssen wir mehr geben (Holz)? Wo sind wir solide gebaut (Stein)?

Unsere Erfahrung

Diese Retrospektive eignet sich besonders für eine Gesamtbetrachtung des Produkts und der Prozesse in regelmäßigen Abständen.

Unsere Erfahrung zeigt: Wird diese Retrospektive zu häufig durchgeführt, wiederholen sich die Themen. In geeignetem Abstand wiederholt, bietet sie jedoch ein erfolgreiches Mittel, um den Zustand des Produkts zu bewerten.

Format 7: Das Segelboot *[7]*

Kurzbeschreibung

In diesem Format sieht sich das Team als ein Segelboot, das auf dem Wasser treibt. Das Boot verfügt über verschiedene Anker, wobei jeder für ein konkretes Problem steht. Ein Anker bremst also das Segelboot aus, das durch den Wind angetrieben wird. Dies sind positive Aspekte in der Organisation. Es lässt sich damit ermitteln, welche Faktoren das Team bremsen und welche das Team antreiben.

Anwendungsfall – Vorteile

Der Vorteil dieses Formats besteht darin, dass es auf die äußeren Faktoren/Schnittstellen des Teams konzentriert ist. Es bietet einen neuen Blickwinkel und legt den Fokus primär auf die Teamgrenzen sowie die Schnittstellen mit anderen Teams.

Ablauf

Der Ablauf ist vergleichsweise einfach: Zuerst wird vom Moderator ein Segelboot an die Wand gezeichnet. Im ersten Schritt definieren die Teilnehmer/innen positive und negative Aspekte, die das Team antreiben oder bremsen. Diese Aspekte werden jeweils am Boot visualisiert. In der erweiterten Variante können zusätzlich Hindernisse/Gefahren (Felsen) und Ziele (Land) aufgenommen werden.

Im zweiten Schritt betrachtet das Team die Punkte von beiden Seiten und diskutiert, inwieweit die positiven Faktoren unterstützend wirken, warum die negativen blockieren und ob die negativen durch

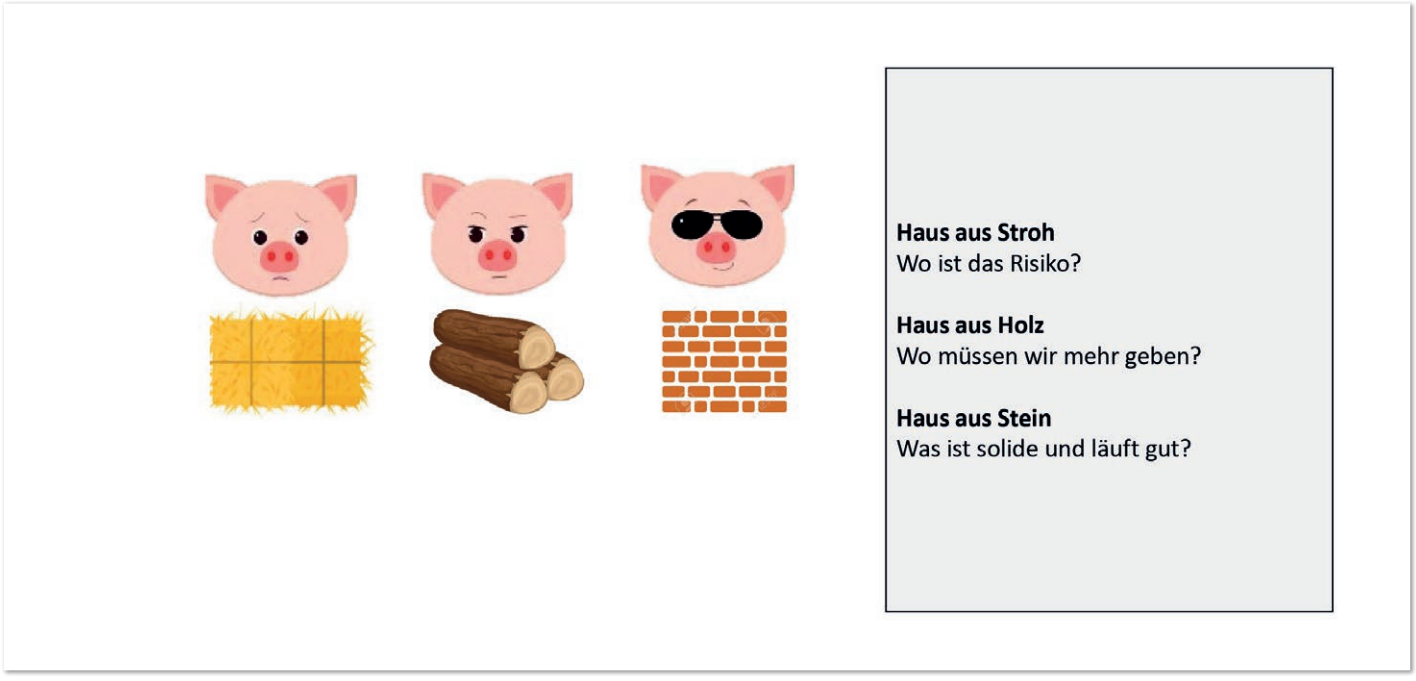


Abbildung 6: Three-little-Pigs-Methode

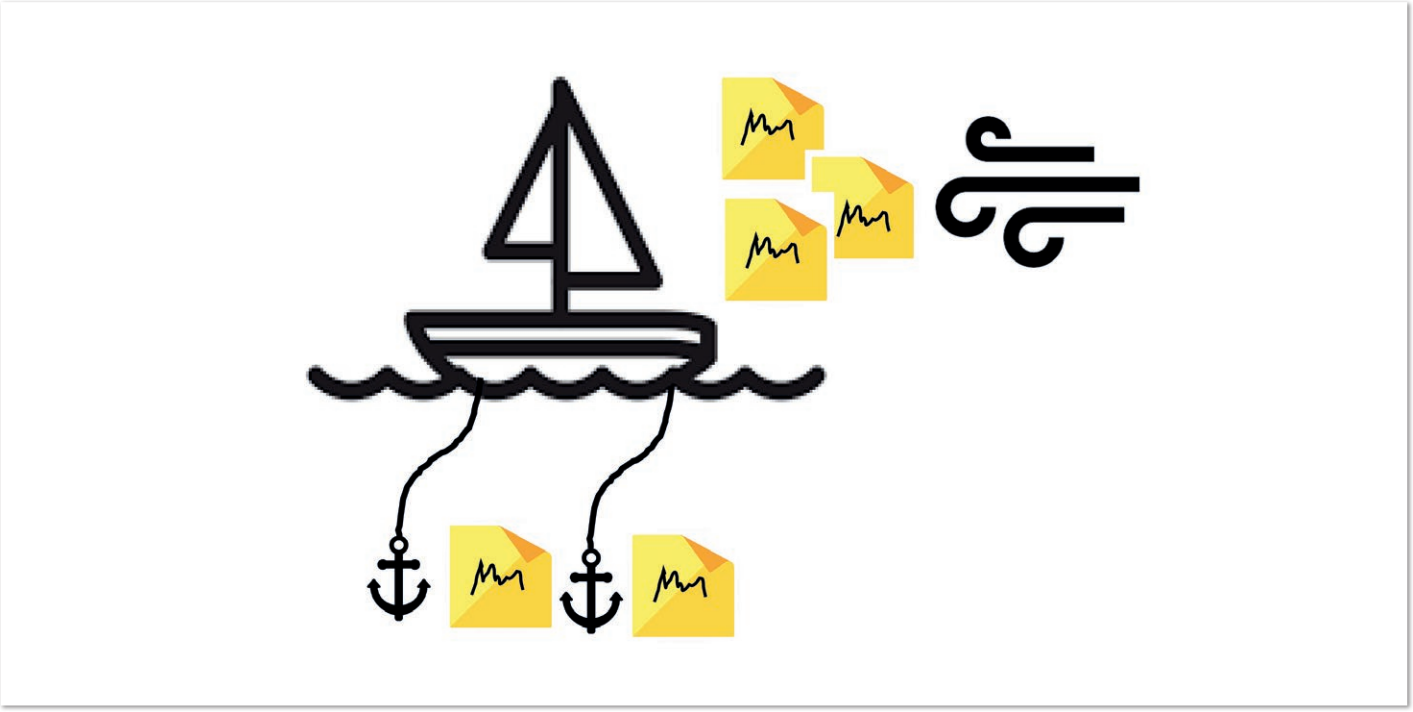


Abbildung 7: Die Segelboot-Retrospektive

gezielte Maßnahmen (Wind) neutralisiert oder sogar ins Gegenteil verkehrt werden können.

Unsere Erfahrung

Unsere Erfahrung zeigt, dass sich diese Retrospektive mit der metaphorischen Darstellung des Teams als Einheit gut eignet, um eine Momentaufnahme der Faktoren zu erhalten, die als hinderlich oder fördernd aufgenommen werden. Gleichzeitig wird durch die bildliche Darstellung eine angenehme, spielerische Atmosphäre geschaffen, durch die sich die Stimmung im Team auflockern lässt.

Weitere Methoden

Alle genannten Formate haben wir bereits in der Praxis erprobt. Sie erhalten hier also unsere jeweiligen Empfehlungen. Sollten Sie weiter experimentieren wollen, möchten wir noch auf die Formate Winning Strike, Mountain-Climber und Each-One-Meets-All hinweisen. Winning Strike ist ein Format, bei dem durch den Rückblick auf die letzten Erfolge des Teams im Sprint Maßnahmen für die Zukunft abgeleitet werden sollen. Die Mountain-Climber-Retrospektive dagegen stellt, ähnlich wie die Segelboot-Retrospektive, den letzten Sprint metaphorisch als Bergsteigertour dar und zeigt helfende, blockierende und unterstützungsbedürftige Faktoren sowie die allgemeine Stimmung auf. Das Each-One-Meets-All-Format dagegen unterscheidet sich deutlich von den anderen: Hier sind die Teammitglieder dazu aufgerufen, einander gegenseitig persönliches Feedback zu geben.

Diese Formate wurden uns bei agilen Stammtischen empfohlen. Wir fanden sie sehr interessant, haben sie allerdings noch nicht selbst in der Praxis ausprobiert. Falls Sie Erfahrung mit diesen oder anderen bisher nicht genannten Konzepten haben, können Sie uns gerne kontaktieren.

Format	Anwendungsfeld
Mad-Sad-Glad	Allgemeine Themen im Team finden, ohne konkret eine Lösung zu wissen.
Start-Stop-Continue	Maßnahmen für konkrete Probleme finden, die dem Team bekannt sind.
4 Ls (Liked, Learned, Lacked, Longed For)	Fördert neues Wissen und bricht aus der Routine aus, wenn seit mehreren Wochen die gleichen Punkte aus den Retrospektiven resultieren.
Energie-Level	Geeignet, um ein Stimmungsbild im Team zu erhalten.
Basketball-Retrospektive	Verdeutlicht, wer im Team welche Aufgabe übernimmt, und stärkt die Zusammenarbeit.
Three little Pigs	Geeignet, um das Produkt sowie die Prozesse der Organisation zu analysieren.
Segelboot	Bestandsaufnahme der äußeren Faktoren/Schnittstellen des Teams.

Tabelle 2: Auswahlhilfe für die Formate zur Retrospektive

Fazit

Retrospektiven sind auch außerhalb der agilen Welt' weit verbreitet und können für Unternehmen zahlreiche Vorteile bringen. Dennoch zeigt unsere Erfahrung, dass nicht jede Retrospektive automatisch einen hohen Mehrwert für das Unternehmen schafft. Es gilt daher, die Retrospektive genau zu planen, das passende Format auszuwählen und durch die Variation der Formate auf ein sinnvolles langfristiges Ziel hinzuarbeiten. Dafür haben wir in diesem Artikel verschiedene Formate vorgestellt. Es liegt nun bei Ihnen, das für Ihre nächste Retrospektive passende auszuwählen. Die folgende *Tabelle 2* kann Ihnen dabei als Entscheidungshilfe dienen.

Quellen

[1] Status Quo Agile 2020: *https://www.process-and-project.net/studien/studienunterseiten/status-quo-scaled-agile-2020/*

[2] Allgemeiner Ablauf von Retrospektiven: *https://agile-unternehmen.de/retrospektive-in-unternehmen/*

[3] Glad, Sad & Mad: *https://nativdigital.com/retro-methoden-mad-sad-glad/*

[4] Start Stop & Continue: *https://t2informatik.de/wissen-kompakt/start-stop-continue-retrospektive*

[5] 4 Ls: *https://www.teamretro.com/retrospectives/4ls-retrospective/*

[6] Energy Levels, Basketball & Three little Pigs: *https://www.parabol.co/resources/agile-sprint-retrospective-ideas*

[7] Speedboat/Sailing Boat: *https://www.agilealliance.org/how-to-improve-the-speedboat-retrospective/*

Vorlagen: *https://agile-unternehmen.de/retrospektive/vorlagen.pptx*

Dr. Dominic Lindner
noris network AG
Dominic.Lindner@agile-unternehmen.de

Dr. Dominic Lindner hat an der FAU Erlangen-Nürnberg am Lehrstuhl für IT-Management zum Thema virtuelle Teamarbeit und Agilität promoviert. Mittlerweile ist er selbst Führungskraft bei noris network in Nürnberg und war vorher viele Jahre als Unternehmensberater tätig. Seine Interessen liegen im Bereich Arbeit 4.0, Digital Leadership und Agilität. Er ist selbst Autor mehrerer Bücher und bloggt aktiv auf agile-unternehmen.de zu den Themen Cloud, Agilität und virtuelle Teams.

Martin Weber
noris network AG
Martin@projektify.de

Martin Weber ist Product Owner bei noris network in Nürnberg. Vorher hat er an der FAU Erlangen-Nürnberg Wirtschaftsinformatik studiert. Er war vorher viele Jahre als Unternehmensberater sowie als Solution-Architekt tätig und konnte umfangreiche Erfahrungen in Unternehmen sammeln. Seine Interessen liegen im Bereich der IT-Architektur und agilen Methoden.

Mitglieder des iJUG

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

iJUG Verbund

www.ijug.eu

Impressum

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

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

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

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

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

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © klyaksun
https://stock.adobe.com
S. 10: Bild © gonin
https://de.123rf.com
S. 12 + 13: Bild © Cienpies Design
https://stock.adobe.com
S. 18: Bild © deniskot
https://de.123rf.com
S. 20 + 21: Bild © Margarita
https://stock.adobe.com
S. 26 + 27: Bild © phonlamaiphoto
https://stock.adobe.com
S. 33: Autorenfotos © Sabine Jakobs
mail@fotografie-jakobs.de
S. 34: Bild © artinspiring
https://stock.adobe.com
S. 39: Bild © kjpargeter
https://de.123rf.com
S. 47: Bild © yupiramos
https://de.123rf.com
S. 52: Bild © 00mate00
https://de.123rf.com
S. 58: Bild © Blue Planet Studio
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

JavaLand

15. – 17. März 2022

im Phantasialand bei Köln

Die Konferenz der Java-Community!

www.javaland.eu

Jetzt Ticket sichern:

Community-Partner: iJUG Verbund

Präsentiert von: DOAG Heise Medien