Anzeige

B1 SYSTEMS

Sie schreiben den Betrieb von Linux-Umgebungen aus?

Kontaktieren Sie uns – wir übernehmen!

info@b1-systems.de ROCKOLDING · BERLIN · KÖLN · DRESDEN Mehr dazu auf S. 9!

Mächtige Analysefunktionen
Data Science mit R

c't

2020

SPECIAL

Moderne Programmiersprachen

Per Anhalter durchs Code-Universum

Neue Programmieransätze
Quanten-computing

Haskell, Idris, F#, Elixir, Elm
Funktionale Programmierung

Von TypeScript bis WebAssembly
Moderne Webentwicklung

Kotlin und Clojure vs. Java
Alternativen für die JVM

Go, Rust, D
C weitergedacht

4 196296 314900 01

14,90 €
Österreich 16,40 €
Schweiz 27.90 CHF
Luxemburg 17,10 €

www.ix.de

Hochverfügbarer S2D Micro-Cluster

Inklusive Azure Stack HCI Badges & Windows Server Solution Brief

Der extrem leistungsstarke S2D Micro-Cluster bietet Ihnen ein perfektes Rundumpaket. Zusammen mit der Kombination aus neuesten Datacenter Lizenzen von Microsofts Windows Server 2019 sorgt er für eine effiziente Storage-Verwaltung mit größtmöglicher Flexibilität. Zusätzlich haben wir extra für Sie eine spezielle Windows Admin Center Extension entwickelt: Damit haben Sie über eine grafische Weboberfläche das hochverfügbare 2-Node-System jederzeit im Blick . Neben der exzellenten Storage-Performance überzeugt der S2D Micro-Cluster außerdem mit seinem besonders kompakten Format – ideal für den Einsatz in Büroumgebungen.

Windows Server 2019

Certified

Microsoft Azure Stack HCI



Preisvorteil:

Basic Version statt

~~17.150,00 EUR~~

12.990,00

EUR

Jetzt erhältlich!

Im Thomas-Krenn-Onlineshop

Kontaktieren Sie uns:

+49 (0) 8551.9150-300

thomas-krenn.com/basic

THOMAS KRENN®

Probieren Sie doch mal was Neues!

Laut einer aktuellen Studie von HackerRank suchen Unternehmen vor allem nach Fullstack-Entwicklern. An der Frage, was genau hinter dieser Bezeichnung steckt, scheiden sich die Geister. Sicher lässt sich aber sagen, dass Fullstack-Entwickler zumindest ein grundlegendes Verständnis aller Schichten eines Technikstacks benötigen – und damit mit unterschiedlichen Programmiertechniken und -sprachen umgehen können müssen.

In der Praxis trifft man dann freilich häufig auf Berge von Softwarealtlasten. Ein Freund und Ex-Entwickler hat einmal gesagt: „Kaum eine Softwareentwickler*in findet sich in IT-Projekten auf der sprichwörtlichen grünen Wiese wieder. Allzu oft türmen sich Legacy-Berge auf, die verwaltet oder abgetragen werden wollen."

Doch die technische Erblast ist nur das eine. Das andere ist der Ruf nach Innovation. HackerRank berichtet, dass fast zwei Drittel der Befragten im letzten Jahr ein völlig neues Framework lernen mussten. Entwickler*innen müssen also den Spagat zwischen alt und neu schaffen. Oft kommen sie dabei auch nicht daran vorbei, eine neue Programmiersprache zu lernen. Persönliche Weiterbildung und permanentes Dazulernen ist unumgänglich in einem Arbeitsbereich, der sich so schnell ändert wie die Softwareentwicklung.

In den letzten Jahren ist die Auswahl an Programmiersprachen beachtlich gewachsen. Viele moderne Sprachen bringen neue Möglichkeiten, erleichtern beispielsweise das Programmieren nebenläufiger Anwendungen, bauen dabei jedoch auf altbekannten Konzepten auf. Entsprechend hält sich der Lernaufwand in Grenzen. Es lohnt sich also, gezielt nach Weiterentwicklungen oder Ablegern klassischer Sprachen zu schauen. Besonders nahe liegen dabei Verwandte der Sprachen, die in den eigenen Projekten bereits Verwendung finden. Aber auch völlig neue Welten haben ihren Reiz, wie alle Entwickler*innen bestätigen werden, die den Einstieg in die funktionale Programmierung gewagt haben.

Wer sich auf die Suche nach einer neuen Sprache macht, ist jedoch gut beraten, eine Reihe von Fragen im Gepäck zu haben. Zuerst einmal geht es darum, zu überprüfen, ob sich alt und neu vertragen. An welchen Stellen könnte es haken? Natürlich sind die Features der potenziellen Kandidatin wichtig, aber ebenso sehr stellt sich die Frage nach einer brauchbaren Dokumentation, nach Bibliotheken und Schnittstellen. Nicht zuletzt ist auch ein Blick in die Community aufschlussund hilfreich. Wer treibt die Sprache voran, wie viele Leute arbeiten mit und wie rege ist die Beteiligung?

Den Blick über den Tellerrand zu wagen ist nicht nur faszinierend, sondern hilft auch, Probleme neu zu denken. Nicht jede neue Sprache muss gleich im nächsten Projekt zum Einsatz kommen. Schon die Beschäftigung mit neuen Ansätzen reicht aus, den eigenen Horizont zu erweitern.

CARINA SCHIPPER





C weitergedacht

C ist die Wurzel zahlreicher moderner Programmiersprachen. Dazu gehören nicht nur die Weiterentwicklungen C++ und C#, sondern auch Newcomer wie Rust, D und Go. Sie orientieren sich an der C-Syntax, bieten aber viele Verbesserungen und zeitgemäße Features.

ab Seite 12

Programmiersprachen heute

Am Anfang war … die Maschine **6**
Sprachwelten **10**

C weitergedacht

Modernes Programmieren mit C++20 **12**
Rust: nicht nur für den Browser **16**
Go für skalierbare und verteilte Systeme **20**
D – die C-Alternative **26**
Typsicherheit mit Swift **30**
C# 8.0: Änderungen bei Schnittstellen und Compiler **34**
Webanwendungen mit Blazor und C# **38**

Alternativen für die JVM

Java – die jüngsten Entwicklungen **46**
Einstieg in Kotlin **50**
Clojure: funktional programmieren auf der JVM **56**

Moderne Webentwicklung

Eine kurze Geschichte von ECMAScript **60**
TypeScript: JavaScript mit Typsystem **66**
ClojureScript: funktional im Browser **72**
WebAssembly für mehr Performance **78**
PHP: Programmiersprache fürs Web **84**
Das JavaScript-Framework Svelte **88**

Funktionale Programmierung

Elm-Apps als Web Components **98**
Elixir: pragmatisch und leicht zu erlernen **104**
Funktionale Programmierung mit Haskell und Idris **110**
Python: erste Wahl für Data Science und Machine Learning **118**
Perl-Nachfolger: Raku verstehen und anwenden **126**
F# in der Enterprise-Entwicklung **132**

Wissenschaftliches Rechnen

R: Statistikumgebung für Datenanalyst*innen **137**
Sentimentanalyse mit R **140**
Quantencomputer programmieren – ein Einstieg **146**
Julia für die Datenanalyse **152**

Rubriken

Editorial: Probieren Sie doch mal was Neues! **3**
Impressum, Inserentenverzeichnis **121**



Alternativen für die JVM

Seit über 20 Jahren belegt Java durchgängig Platz eins oder zwei im TIOBE-Ranking der populärsten Programmiersprachen. Allerdings lässt sich das Java-Ökosystem inzwischen auch mit anderen Sprachen nutzen: Kotlin glänzt mit kompakterer Syntax, Clojure bringt funktionale Programmierung auf die JVM.

ab Seite 46



Moderne Webentwicklung

Der Browser wird zunehmend zur Plattform der Wahl für die Anwendungsentwicklung. Neben JavaScript-Alternativen wie TypeScript und ClojureScript macht derzeit WebAssembly als Format zum Ausführen von Binärcode im Browser von sich reden.

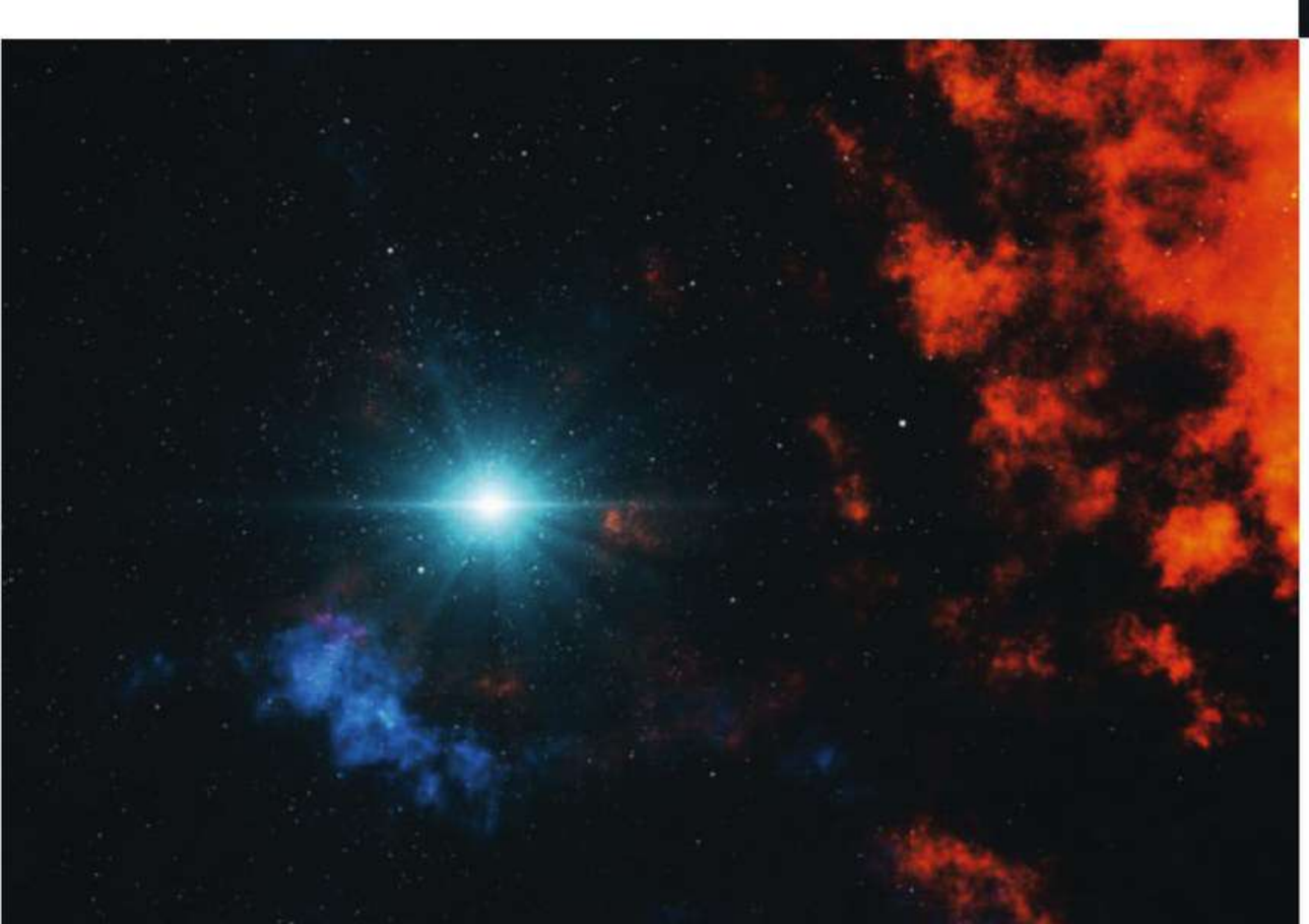
ab Seite 60

Funktionale Programmierung

Funktionale Sprachen galten lange als theoretisch faszinierend, aber in der Praxis sperrig und umständlich. Doch in der neuen Welt der skalierbaren Anwendungen für verteilte Systeme liefert das funktionale Programmierparadigma einfache Lösungen für komplexe Probleme.

ab Seite 98





Wissenschaftliches Rechnen

Im Bereich des wissenschaftlichen Rechnens versuchen spezialisierte Sprachen wie R und Julia der Datenflut Herr zu werden. Quantencomputer sind mittlerweile mehr als technische Spielereien und verschieben die Grenzen des Möglichen.

ab Seite 137



Welche Programmiersprache ist die beste im Land?

Am Anfang war … die Maschine

Michael Stal

Sie sind nur ein Werkzeug für die Softwareentwicklung, wenn auch das wichtigste. Um sie entstehen Glaubenskriege und unterschiedliche Weltanschauungen: Gehen wir auf eine Reise durchs fabelhafte Universum der Programmiersprachen.

Ein historischer Blick auf die Entwicklung von Programmiersprachen verhilft zu einer besseren Perspektive auf die heutige Sprachlandschaft. Die ersten Programmiersprachen waren gar keine, sondern zwangen Programmierer*innen dazu, nah an der Maschine zu codieren. Stichwort Maschinensprache. Das erwies sich als fehlerträchtig und mühsam, erforderte viel Spezialisierung und machte eine Portierung auf andere Hardware so gut wie unmöglich. Das Lambdakalkül (Church, Kleene) sowie Zuses Plankalkül markieren die ersten Versuche, höhere Abstraktionsebenen einzuführen.

Die weitere Evolution von Programmiersprachen war letztendlich von dem Wunsch getrieben, immer höhere Abstraktionsebenen einzuführen, um Entwickler*innen näher an die Problemdomäne und weiter weg von den Niederungen der Maschine zu bringen. Seit den 1960er-Jahren kamen zunehmend Konzepte hinzu, um die Entwicklung großer Softwaresysteme zu fördern. Durch immer leistungsfähigere Computersysteme ließen sich auch stärkere Programmierumgebungen realisieren, um das Programmieren von einer Kunst weniger zu einem Handwerk für viele zu transformieren. Zudem hat die Durchdringung aller Lebensbereiche und Domänen mit Software zu einer Inflation allgemeiner und spezialisierter Programmiersprachen geführt. Es fällt zunehmend schwerer, einen Überblick zu behalten und für eine Problemstellung die geeignete Sprache auszuwählen.

Die Qual der Wahl

Mittlerweile lesen sich Aufreihungen von Programmiersprachen schon fast wie ein Adressbuch. Beispielsweise gibt es in Wikipedia eine große Vergleichsliste bekannter Sprachen, deren Unterscheidung anhand der unterstützten Paradigmen erfolgt. Solche Verzeichnisse erlauben einen tiefergehenden Einblick in die Vielzahl existierender Programmiersprachen, ermöglichen aber keine gezielte Auswahl. Ihre Beliebtheit ist die naheliegendste Art, verschiedene Programmiersprachen zu vergleichen und einzuordnen. Vergleichsindizes wie TIOBE oder RedMonk stellen eine Reihenfolge der beliebtesten Programmiersprachen dar. Als Basis dient dabei die Zahl der Erwähnungen auf Webseiten wie Stack Overflow. Im Mai 2020 lautete zum Beispiel die Reihenfolge im TIOBE-Index wie in der Abbildung gezeigt.

Die üblichen Verdächtigen wie Java, C, Python rangieren dort auf den Spitzenplätzen. Ganz abgesehen davon, dass die diversen Indizes verschiedene Reihenfolgen produzieren, sind sie zwar

interessant, aber wenig hilfreich bei der Entscheidung, welche Programmiersprache sich für ein Problem am besten eignet. Für manche Aufgabenstellung kann eine Nischensprache die richtige Wahl sein, obwohl sie auf dem TIOBE-Index erst in den unteren Regionen auftaucht.

Sprachen klassifizieren

Zum Vergleich und zur Betrachtung verschiedener Programmiersprachen existiert eine ganze Reihe von Kriterien. Eine rein technische Liste von systemnahen Aspekten wie Sprachmerkmalen oder Syntax genügt allerdings nicht. Stattdessen ist essenziell, welches Ziel ein Projekt verfolgt, um die dafür geeigneten Sprachen zu eruieren. Folgender Katalog versucht, ein paar Bezugspunkte für die Auswahl zu setzen, ohne Anspruch auf Vollständigkeit zu erheben.

Enthalten sind sowohl technische als auch nicht technische Aspekte. Welche Art von System möchten Entwickler*innen erstellen? Zum Beispiel eine reine Geschäftslösung, ein eingebettetes System, einen Echtzeitprozess zur Automatisierung, eine browserbasierte Anwendung, eine Edge-Anwendung, eine Software-as-a-Service-Anwendung?

Für jede Problemdomäne existieren entsprechende Programmiersprachen. Im Falle einer Echtzeitanwendung erweist sich zum Beispiel JavaScript als völlig ungeeignet, ganz im Gegensatz zu webbasierten Frontends oder Backends. Traditionelle Sprachen wie C oder C++ glänzen hingegen für systemnahe Domänen à la Embedded und Echtzeit. Go gilt als die Sprache der Wahl für verteilte Netzwerkanwendungen, während Elixir als Sprache mit Ruby-Wurzeln hervorragend für IoT-Edge-Knoten oder verteilte eingebettete Systeme prädestiniert ist.

Moderne Programmiersprachen wie Swift oder Rust implementieren in der Regel mehrere Paradigmen, wobei objektorientiertes Programmieren mittlerweile ebenso zu den Standards gehört wie funktionale Programmierkonzepte. Generisches oder generatives Programmieren taucht häufig auf, während Ansätze für sicheres Programmieren wie Ownership in Rust selten anzutreffen sind.

Ein weiteres Gebiet ist die Unterstützung ereignisorientierten Programmierens mit Sprachen wie F#, Groovy oder Kotlin. Konzepte zur Modularisierung wiederum spielen in vielen Sprachen eine wichtige Rolle. Meta Programming, also die Fähigkeit einer Software, dynamisch Näheres über sich selbst in Erfahrung zu bringen oder sogar eigene Eigenschaften zu verändern, findet sich heute in Sprachen wie zum Beispiel Clojure oder Prolog.

Prolog und Erlang gehören zur Familie der Programmiersprachen, die deklarativ statt imperativ arbeiten. Die meisten Sprachen verfolgen also einen Multiparadigmenansatz. Typisches Beispiel ist Scala, das beispielsweise Generizität, funktionale Programmierung und objektorientierte Programmierung in sich vereint und sogar interne DSLs ermöglicht.

Year	Winner
2019	C
2018	Python
2017	C
2016	Go
2015	Java
2014	JavaScript
2013	Transact-SQL
2012	Objective-C
2011	Objective-C
2010	Python
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++

TIOBE-Index Mai 2020

Unterstützung durch Dritte

Das Laufzeitsystem einer Sprachimplementierung ist die eine Seite der Medaille. Die andere ist die Verfügbarkeit von Zusatzbibliotheken. Moderne Sprachen wie Rust versuchen die Sprache möglichst klein zu halten und wichtige Funktionen stattdessen in Standardbibliotheken vorzuhalten. Da eine Programmiersprache und ihre Bibliotheken in der Praxis nur die Spitze des Eisbergs darstellen, gehört auch eine Betrachtung der Unterstützung durch Dritte zu den wichtigen Aspekten. Beispielsweise die Verfügbarkeit weiterer Werkzeuge wie IDEs, Bibliotheken, Frameworks, Datenbanken, Cloud-Integration und Tools. Oder auch die Verbreitung entsprechender Communitys und die Verfügbarkeit von Fachliteratur oder anderen Lernquellen.

Um moderne Multicoreprozessoren optimal auszureizen, sollte die Laufzeitumgebung einer Programmiersprache entsprechende Concurrency-Konzepte bereitstellen. Dazu gehören Parallelisierung durch Multithreading und Synchronisationsmechanismen für den Zugriff auf gleiche Ressourcen. Im Idealfall verlässt sich eine Sprache auf ein eigenes einheitliches konzeptionelles Modell, statt direkt auf Betriebssystemebene durchzugreifen.

Beispielsweise glänzt Rust durch ein integriertes Modell für Concurrency. Sein Ownership-Modell bei Daten vermeidet mögliche Race Conditions schon zur Übersetzungszeit gezielt. Hier zeigt sich übrigens, dass eine geschickte Wahl von Sprachmerkmalen positive Konsequenzen auf Parallelität besitzt. In Sprachen auf Basis der BEAM VM wie Erlang und Elixir laufen leichtgewichtige Threads geschützt voneinander auf separaten CPU-Kernen. In diesen Fällen bietet das Laufzeitsystem Unterstützung für Concurrency.

Brückenbau

Viele Anwendungen sind über mehrere Systeme verteilt, zum Beispiel Geschäftsanwendungen oder IoT-Systeme. Daher erweisen sich Mechanismen für die prozessübergreifende Kommunikation als wichtig. Die Art der Kommunikation, etwa synchron, asynchron oder nachrichtenbasiert, spielt dabei eine ebenso große Rolle wie die darunterliegenden Kommunikationsprotokolle und Datenübertragungsformate, etwa RESTful, JSON oder

iX-TRACT

- Die ersten Programmiersprachen waren Maschinensprachen, die eine hohe Spezialisierung der Programmierer*innen erforderten. Erst mit der Zeit entwickelten sich höhere Abstraktionsebenen, die den Fokus weg von der Maschine hin zu den Problemdomänen lenkten.
- Da sich jede Sprache aufgrund ihres Charakters nur für bestimmte Einsatzzwecke eignet, ist es ratsam, nicht an einer festzuhalten, sondern mehrere zu erlernen.
- Bei der Wahl der Sprache gilt es, das Projektziel im Blick zu haben. Es ist zu fragen, welche Art von System zu entwerfen ist: ein eingebettetes, eine Edge- oder SaaS-Anwendung, ein Echtzeitprozess zur Automatisierung oder eine reine Geschäftslösung.

TCP/IP. Die Implementierung von Verteilungs- und Kommunikationsaspekten erfolgt meistens in Bibliotheken.

Da nicht alle Projekte auf der grünen Wiese beginnen, sondern sich in eine bereits bestehende Umgebung einfügen müssen, ist entscheidend, wie weit die gewünschte Programmiersprache dies überhaupt zulässt. Gibt es zum Beispiel Anbindungen an C- oder C++-Code, ist der Aufruf existierender C- oder C++-APIs möglich. Zum Zugriff auf Microservices bedarf es REST-Unterstützung. Was auf den ersten Blick oft einfach erscheint, offenbart auf den zweiten Blick manchmal eine große Komplexität. Wer schon einmal versucht hat, die C-Anbindung von Python zu durchdringen, kann ein Lied davon singen.

Für die einen gelten statische Typsysteme als erste Wahl, weil sie schon zur Übersetzungszeit Typfehler erkennen und als Komfort Typinferenz mitbringen, während andere auf dynamische Typsysteme schwören, frei nach dem Erlang-Mantra: „Let it crash." Statische Sprachen sind meist übersetzte Sprachen, für die ein Compiler Maschinencode erzeugt, dessen Ausführung auf einer realen oder virtuellen Maschine erfolgt. Die meisten Skriptsprachen sind hingegen dynamisch, das heißt, ihre Ausführung erfolgt in einem Interpreter – Stichwort: REPL (Run-Eval-Print Loop).

Lernkurve

Wer bereits einschlägige Erfahrungen mit Java oder C# besitzt, dürfte mit dem Erlernen anderer Sprachen aus der C-Familie wie Swift nur wenig Mühe haben. Erfahrenen Entwickler*innen mit JDK- oder .NET-Vorkenntnissen fällt es leichter, sich mit anderen Sprachen aus dem JVM- oder CLR-Universum anzufreunden. Die Lernkurve ist somit ein wichtiges Kriterium bei der Sprachenwahl. Das gilt speziell in solchen Fällen, in denen eine komplette Organisation oder ein Projekt sich für eine „neue" Programmiersprache entscheiden muss.

Zur Komplexität einer Programmiersprache trägt auch fehlende Transparenz bei. In C++ gibt es eine ganze Menge von Seiteneffekten, die Entwickler*innen beachten müssen, um in keine Falle zu laufen, etwa in Hinblick auf das Erzeugen und Löschen temporärer Variablen durch das Laufzeitsystem. Sprachen wie C übertragen die Bürde der Speicherverwaltung weitgehend auf Programmierer*innen. Dieses Vorgehen sorgt für die notwendige Freiheit bei systemnaher oder effizienzorientierter Software, verpflichtet Entwickler*innen aber andererseits zu größerer Disziplin und Vorsicht. Garbage Collection bei Java und C# reduziert die Freiheiten zugunsten der Programmiersicherheit, macht aber Speicherfreigaben intransparent.

In der Compilersprache Go findet sich zwar ein Garbage Collector, aber im Gegensatz zu Java und C# organisiert Go den Heap nicht in Generationen und führt des Weiteren eine Analyse durch, ob Objekte auf dem Heap oder auf dem Stack landen sollen. Sprachen mit Automatic Reference Counting wie Swift verfolgen einen Mittelweg, bei dem der Compiler Retain- und Release-Instruktionen in den Code einfügt. Zyklische Abhängigkeiten lassen sich allerdings nicht automatisieren, sondern verlangen die explizite Mithilfe der Entwickler*innen.

In RAII-Ansätzen (Resource Acquisition is Initialization) wie bei C++ oder Rust nutzt das Laufzeitsystem ein Lebenszyklusmodell für Objekte. Sie beginnen ihr Leben im Konstruktor und beenden es im Destruktor. Das Ownership-Modell in Rust setzt auf RAII auf. Es erfordert, dass jeder Datenwert eine Variable als Besitzer hat. Die Ownership kann zum Beispiel bei Funktionsaufrufen auf den Empfänger übergehen. Gerät ein Objekt außerhalb des Bereiches seines Besitzers, gibt es das Laufzeitsystem mithilfe eines Referenzzählermechanismus frei.

Erfolgsgeschichten

Sobald eine bestimmte Programmiersprache in den Fokus und damit in die engere Auswahl gerät, lohnt sich der Blick darauf, welche Erfahrungen andere Projekte mit der Sprache der Wahl gemacht haben. In der Ära von Open-Source-Software und dem Internet finden sich zum Glück recht schnell entsprechende Informationsquellen und Codebasen. Im Optimalfall lassen sich dank Google, Bing oder DuckDuckGo sogar vergleichbare Projekte aufspüren, die zum eigenen Profil passen.

Für einzelne Programmierer*innen wahrscheinlich wenig relevant, aber für kommerzielle Projekte von maßgeblicher Bedeutung dürfte die Frage sein, wer eine Programmiersprache kontrolliert. Einige Programmiersprachen unterliegen der Kontrolle von Forschungseinrichtungen, Open-Source-Konsortien oder kommerziellen Unternehmen, etwa Rust, Scala, Swift, Java oder Kotlin. Andere haben einen Standardisierungsprozess durchlaufen wie C++ oder Ruby.

Beide Ansätze bringen Vor- und Nachteile mit sich. So hat Andrew Tanenbaum bereits vor Jahren treffend zum Thema Standardisierung formuliert: „Das Schöne an Standards ist, dass es so viele davon gibt, aus denen sich auswählen lässt." Standardisierungsprozesse erlauben Nutzer*innen bessere Einflussnahme, können aber auch sehr zäh verlaufen.

Das Projektziel sich immer vor Augen halten

Das Erlernen einer neuen Programmiersprache erfordert eine große Investition von Organisationen und Entwickler*innen. Daher fällt es oft schwer, ausgetretene Pfade zu verlassen, um neues Terrain zu erforschen. Heutige Entwicklungsprojekte adressieren zumeist verschiedene Lösungsdomänen, weshalb das starre Festhalten an einer einzigen Sprache schon längst nicht mehr funktioniert.

Hemmungsloses Anwenden einer „Jeder Topf bekommt seinen eigenen Deckel"-Strategie erweist sich aber ebenfalls als kontraproduktiv. Wer möchte schon eine polyglotte Codebasis verwalten, die unübersichtlich viele Sprachen umfasst. Wenn aber schon eine neue Programmiersprache her muss, dann sollte sich die Auswahl nicht nur am „Fun"-Faktor, sondern hauptsächlich an den eigenen Bedürfnissen orientieren. Immerhin gibt es mittlerweile eine Vielzahl von Programmiersprachen mit unterschiedlichsten Konzepten und Neuerungen. Da fällt die Auswahl zunehmend schwer.

Auf der positiven Seite bleibt festzuhalten, dass man heute aus einer Vielzahl von Sprachen auswählen kann und dass andererseits alle Sprachen auf Konzepte von Vorgängersprachen aufbauen, weshalb sich die Lernkurve für professionelle Entwickler*innen mit jeder neuen Sprache abflacht. Nicht zuletzt macht es Spaß, den eigenen Horizont durch eine neue Programmiersprache zu erweitern. Jedes neue Werkzeug erhöht die eigenen Möglichkeiten. Bekanntlich ist Spaß die beste Motivation. (nb@ix.de)



Prof. Dr. Michael Stal

beschäftigt sich bei der Corporate Technology der Siemens AG mit Software- und Systemarchitekturen, Digitalisierung und KI. An der University of Groningen in den Niederlanden hält er Vorlesungen und betreut Doktoranden.



B1 Managed Service & Support

individuell – umfassend – kundenorientiert

Neue oder bestehende Systemlandschaften stellen hohe Anforderungen an Ihr IT-Personal. Mit einem individuellen Support- und Betriebsvertrag von B1 Systems ergänzen Sie Ihr Team um die Erfahrung und das Wissen unserer über 120 festangestellten Linux- und Open-Source-Experten.

Unsere Kernthemen:

Linux Server & Desktop · Private Cloud (OpenStack & Ceph) · Containerization (Docker) · Orchestration (Kubernetes) · Monitoring (Icinga, Nagios & ELK) · Patch Management · Automatisierung (Ansible, Salt, Puppet & Chef)

Wir sind für Sie da – mit qualifizierten Reaktionszeiten ab 10 Minuten und Supportzeiten von 8x5 bis 24x7!

B1 SYSTEMS

B1 Systems GmbH - Ihr Linux-Partner

Linux/Open Source Consulting, Training, Managed Service & Support

ROCKOLDING · KÖLN · BERLIN · DRESDEN

www.b1-systems.de · info@b1-systems.de



Sprachwelten

Jenseits der eigenen Galaxie

Carina Schipper

Das Universum der Softwareentwicklung erstreckt sich über schier unendliche Weiten. Wer sich darin bewegt, braucht vor allem eins: Entdecker*innengeist.

Angesichts des Sprachspektrums, um das sich vielfältige Ökosysteme entwickelt haben, fällt es oft schwer, vor lauter Bäumen den sprichwörtlichen Wald zu sehen. Gefühlt jagt eine Innovation die nächste. Projekte verlangen von Softwareentwickler*innen ein hohes Maß an Spezialisierung und viel Detailwissen. Zeitgleich zwingt der stetige Wandel innerhalb der IT sie, sich mit neuen Herausforderungen auseinanderzusetzen und das gesamte Umfeld um das eigene Projekt im Blick zu behalten. Als eines von zahlreichen Beispielen für den Wandel und dessen weitreichende Konsequenzen lässt sich die Cloud mit Entwicklungen wie Platform as a Service und Serverless Computing anführen.

Von I-Shape zu T-Shape

Wissen in die Tiefe in einem oder nur wenigen Gebieten reicht nicht mehr aus, um im Arbeitsalltag als Programmierer*in zu bestehen. Immer neue Anforderungen beispielsweise durch den Einsatz von KI, die Verarbeitung immer größerer Datenmengen oder neue Programmierparadigmen wie Serverless führen das Bild der hoch spezialisierten I-Shape-Entwickler*in ad absurdum. Breitenwissen ist gefragt, auf drei Ebenen, der technischen, fachlichen und sozialen. Daraus ergibt sich vereinfacht das Bild eines T-Modells (siehe Abbildung). Bekannt ist dieses Konzept vor allem aus dem Bereich der agilen Softwareentwicklung – Stichwort crossfunktionale Teams.

Übertragen auf den Aspekt der Programmiersprachen bedeutet das, die Sphäre der eigenen Sprache zu verlassen und in neue Gefilde einzutauchen. Dabei geht es nicht unbedingt darum, sich in eine polyglotte Entwickler*in zu verwandeln, die alle möglichen Programmiersprachen fließend spricht. Schon sich mit den grundlegenden Merkmalen anderer Sprachen oder beispielsweise der Frage, wie sie mit Problem der Nebenläufigkeit umgehen, zu beschäftigen, kann hilfreich für das eigne Projekt sein. Daneben lohnt sich auch ein Blick auf Themen wie Programmierparadigmen und Softwarearchitekturen.

Spitzenreiter, Dauerbrenner und Newcomer

Ein Blick auf den TIOBE-Index liefert eine monatliche Momentaufnahme der Beliebtheit verschiedener Programmiersprachen. Die oberen fünf Plätze der Hitliste sind seit Jahren mit den üblichen Verdächtigen besetzt. Hier rangiert die alteingesessene Prominenz der Szene. Beispielsweise gewinnt im Mai 2020 C gegen Java den scheinbar ewig währenden Kampf um

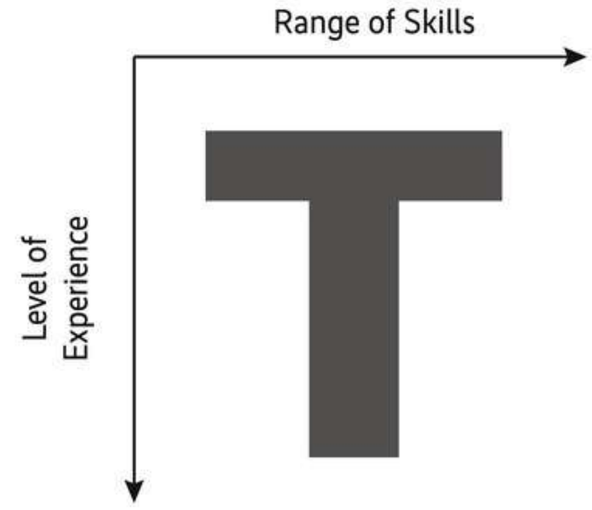
die absolute Spitzenposition. Vergleicht man die Liste der Dauerbrenner über die letzten 30 Jahre hinweg, tauchen die beiden dort ebenfalls auf dem Siegertreppchen auf. Aber auch Python, C++, C# oder JavaScript feiern schon lange immer wieder Erfolge. Zu den Neueinsteigern im TIOBE-Index zählen Sprachen wie Go, Kotlin oder TypeScript, wobei Go der einzige relativ junge Kandidat ist, der es in die Top 20 schafft. Die anderen Newcomer versammeln sich eher im Mittelfeld des Rankings.

Der TIOBE-Index und andere Quellen zur Verbreitung von Programmiersprachen liefern zwar eine Rangliste, sagen aber nichts darüber aus, warum sich einzelne Vertreter mehr oder weniger großer Beliebtheit erfreuen. Wer wissen will, was es damit auf sich hat, braucht mehr als ein Kratzen an der Oberfläche. Die Zusammenstellung der Artikel in diesem Sonderheft erhebt sicherlich keinen Anspruch auf Vollständigkeit. Trotzdem können die Texte als erster Einstieg dabei helfen, den Sprung in neue (Sprach-)Welten zu wagen.

Und wie geht es dann weiter?

Zweifellos, das Berufsbild der Software*entwicklerin geht Hand in Hand mit der Evolution der Technik. Vor allem in der jüngsten Vergangenheit scheint die Digitalisierung vehement um sich zu greifen und in nahezu alle Bereiche des Lebens vorzustoßen. Programmieren ist schon lange nicht mehr nur die Sache dezidierter Expert*innen. Da gibt es zum Beispiel die Schar der Blogger*innen, die sich mit HTML, CSS und JavaScript eigene Webseiten bauen, oder Fachkräfte in Unternehmen, die eigene Datenauswertungen programmieren.

Softwareentwickler*innen brauchen neben Detail- auch Breitenwissen.



Auch in Jobs, die vermeintlich fernab der IT liegen, sind daher IT-Skills immer wichtiger. Karrierebibel etwa bezeichnet Fähigkeiten wie einen Onlineshop zu modifizieren oder ein Add-on für eine Anwendung zu schreiben als unschlagbare Zusatzqualifikationen (siehe ix.de/z8ey).

Zusätzlich werfen neue Techniken wie Low- und No-Code-Plattformen eine Reihe von Fragen auf. Wo geht die Reise in der IT hin und welche Rolle spielen Softwareentwickler*innen darin? Zwei Dinge sind sicher: Die Zeit steht nicht still und es lohnt sich, mit ihr zu gehen und den eigenen Horizont zu erweitern.

(nb@ix.de)

Quellen

Alle Links unter ix.de/z8ey

heise Events
Conferences, Seminars, Workshops

Online-Konferenz

Neue Cyberangriffe –

Wie können Unternehmen sich schützen –
Kritische Infrastrukturen & Industrie 4.0 im Fokus

27. August 2020, 9 – 16 Uhr

Neben der Notwendigkeit von Cyberabwehrmaßnahmen werden auf dieser Online-Konferenz auch verschiedene Konzepte und Vorgehensweisen zur Sicherung der unternehmensinternen IT- und Prozessnetze dargelegt.

Dabei sollen folgende Fragen behandelt werden: Welche Tools benötigt ein Unternehmen, um sich gegen aktuelle Cyberangriffe abzusichern? Wie werden potentielle Angriffe möglichst schnell erkannt? Und welche Vorgehensweisen sind am besten zur Bedrohungsanalyse geeignet?

In Kooperation mit HSB Hochschule Bremen City University of Applied Sciences

Preis: 159,00 Euro

©LuckyStep

www.heise-events.de/konferenzen/it-sicherheitstag-august



Modernes Programmieren mit C++20

Paradigmenwechsel

Nicolai Josuttis

Mit C++20 steht eine große Release vor der Tür. Die neuen Features Modules, Coroutines, Concepts und Ranges ändern die Art, mit C++ zu programmieren.

C++20 ist seit Februar 2020 inhaltlich finalisiert und wartet auf die formale Verabschiedung als Standard. Nach zwei kleineren Releases (derzeit erscheint alle drei Jahre eine neue C++-Version) handelt es sich diesmal wieder um eine große, die ähnlich wie C++11 das Programmieren mit C++ nachhaltig verändern wird. Die großen Neuerungen sind Concepts, Ranges, Modules und Coroutines.

Die erste Neuerung betrifft generische Programmierung. Man kann nun auch außerhalb von Lambdas `auto` als Parameter verwenden:

```
void printColl(const auto& coll) {
    for (const auto& elem : coll) {
        std::cout << elem << '\n';
    }
}
```

Diese Funktion ist im Prinzip ein Funktionstemplate für einen Parameter `coll` von beliebigem Typ. Konzepte erlauben es nun, Anforderungen und Einschränkungen an generische Datentypen zu formulieren. Dabei handelt es sich um eine mit einem Namen versehene Spezifikation der von einem Datentyp unterstützten Eigenschaften und Operationen. Damit kann man dokumentieren und sicherstellen, dass ein Template wie oben nur bestimmte Typen unterstützt (Listing 1; alle Listings siehe ix.de/zwbr).

Die in Listing 1 definierte generische Funktion `print()` lässt sich nur für Datentypen einsetzen, für deren Objekte man `begin()` und `end()` aufrufen kann. Das schützt nicht nur vor Fehlverhalten, sondern verbessert auch die Fehlermeldungen beim Verwenden von generischem Code. Die C++20-Standardbibliothek definiert hierzu über fünfzig Standardkonzepte wie `predicate` oder `movable`.

Der zweite große Meilenstein sind Ranges. Mit der Aufnahme der Standard Template Library (STL) in den ersten C++-Standard etablierte sich das grundsätzliche Prinzip, dass man zum Bearbeiten aller Elemente einer Menge einen halboffenen Bereich in Form von zwei Parametern, Anfang und Ende, übergibt:

```
std::sort(coll.begin(), coll.end());
```

Die neue Ranges-Bibliothek ersetzt dieses Prinzip nun durch die Option, ganze Mengen als einen Parameter zu übergeben:

iX-TRACT

- Die inhaltlichen Arbeiten von C++20 sind abgeschlossen, der Standard liegt zur endgültigen Genehmigung und Veröffentlichung dem ISO-Komitee vor (DIS, Stand: April 2020).
- Das Update der Sprache enthält zahlreiche Neuerungen. Wesentliche sind Ranges, Module, Konzepte (Concepts) und Koroutinen (Coroutines).
- Weitere Features sind der Drei-Wege-Vergleichsoperator <=>, Synchronisierungsmechanismen wie Semaphore, Latches und Barriers sowie Kalender- und Zeitzonenerweiterungen für die chrono-Bibliothek.

Listing 1: Verwenden von Concepts und auto in Funktionen

```
template <typename T>
concept IsContainer = requires(const T& t) {
    { t.begin() };
    { t.end() };
};

void print(const IsContainer auto& coll) {
    for (const auto& elem : coll) {
      std::cout << elem << ' ';
    }
}
```

Listing 2: Iteration mit einer Ranges-View

```
// Spezifikation einer View:
rng::filter_view over36View{coll | rng::views::filter(isOver36)};

// Verwendung der View (hier wird iteriert):
for (int i : over36View) {
  std::cout << "over Limits: " << i << '\n';
}
```

```
std::ranges::sort(coll);
```

Eine Unix-Pipe-ähnliche Syntax erlaubt es dabei, Mengen in Filterketten zu bearbeiten:

```
auto isOver36 = [] (int val) { return val > 36; };

for (int i : coll | rng::views::filter(isOver36)) {
    std::cout << "over 36: " << i << '\n';
}
```

Dazu lassen sich auch Views und Adapter definieren (Listing 2). Zu beachten ist, dass erst die `for`-Schleife tatsächlich über die Elemente iteriert. Das geschieht allerdings nicht nebenläufig, etwa durch mehrere Threads oder Prozesse.

Mit Modulen arbeiten

Als dritter großer Meilenstein gilt die Aufnahme von Modulen. Diese gehen das Problem der physischen Strukturierung von umfangreichem Quellcode mit Millionen oder Milliarden von Codezeilen an. Die große Verbreitung von Templates führt dazu, dass immer wieder der gleiche Code kompiliert. Auch ist es schwierig, Komponenten zu definieren, die aus mehreren Dateien bestehen und nach außen eine klare Schnittstelle haben. Module bieten nur die Möglichkeit, Komponenten zu definieren. Neben einer sauberen Schnittstellendefinition lässt sich ein Modul, auch wenn es in mehreren Dateien implementiert wird, eigenständig kompilieren.

Hier ein einfaches Beispiel: Das Modul `My.Account` definiert seine Schnittstelle in genau einer Datei (Listing 3). Darin exportiert die Schnittstelle die Klasse `Customer` und die Funktion `print()`. Aber auch private Aspekte sind definierbar. Denn bei der Verwendung des Moduls müssen die Objektgrößen und manche Definitionen bekannt sein, um das Kompilieren von Code und Optimierungen zu ermöglichen.

Um Code zu kompilieren, der das Modul nutzt, ist nur ein Import dieser Schnittstelle erforderlich:

```
import My.Account;

Customer c{"test");
print(c);
```

Listing 4 zeigt, wie das Modul dann in mehreren Dateien implementiert wird.

Module lösen damit das Konzept von Headerdateien ab, die vor allem den kompletten Code von Templates enthalten. Es entstehen aber auch neue Dateiformate für vorkompilierte Module und Repositories, um den Code später zu Programmen zusammenzubinden. Eine sanfte Migration von Headerdateien als Schnittstellenbeschreibung wird unterstützt.

Listing 3: Definition einer Modulschnittstelle

```
export module My.Account;

export class Customer {
      std::string name;
    public:
      Customer(std::string name);
      std::string getName() const;
};

export void print(const Customer&);

void helper(); // private in module
```

Listing 4: Modul in mehreren Dateien implementieren

```
// 1. Datei mit Modulimplementierungen
module My.Account;

#include <iostream>

void print(const Customer& c) {
    std::cout << c << '\n';
}

// 2. Datei mit Modulimplementierungen
module My.Account;

void helper() {
    ...
}
```

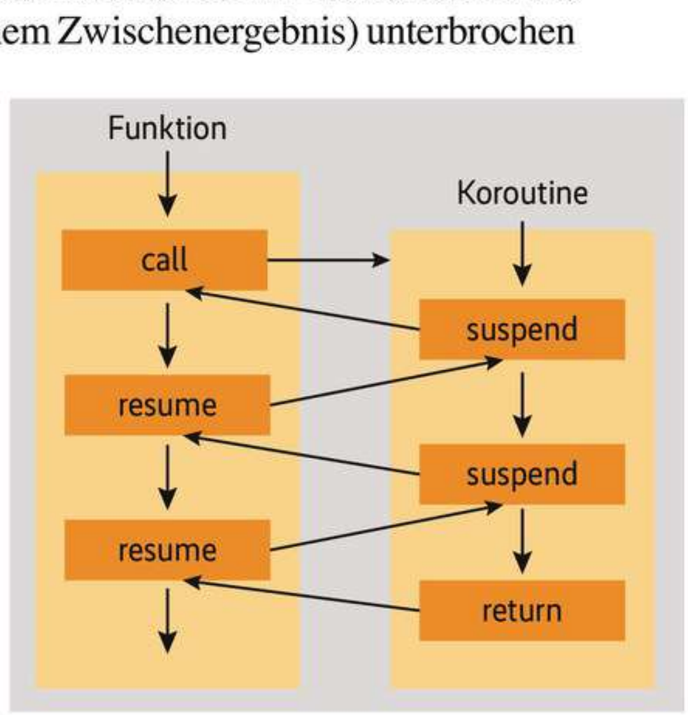
Die praktische Verwendung von Modulen bedarf noch der Klärung technischer Details und Konventionen. Offene Punkte sind deren Dateiendungen, wo und in welchem Format sich vorkompilierte Module speichern lassen, ob und wie diese zwischen verschiedenen Compilern austauschbar sind und wie die C++-Standardbibliothek von Headerdateien auf Module umzustellen ist. Eine Arbeitsgruppe befasst sich im Rahmen der weiteren Standardisierung von C++ damit.

Auch bei einigen anderen C++20-Features wurden erste Teile von neuen Sprachmitteln eingeführt, mit der Idee, diese dann in nachfolgenden C++-Versionen noch zu ergänzen und abzurunden. C++20 stellt in dieser Hinsicht nur den Beginn eines Zyklus von zwei bis drei Releases dar.

Ein Beispiel ist das Verlagern von Berechnungen von der Laufzeit auf die Kompilierzeit. Im Kontext von `constexpr` können jetzt zwar dynamischer Speicher, `try/catch` (nicht `throw`), virtuelle Funktionen, zwei neue Schlüsselwörter (`consteval` und `constinit`) und damit auch Strings und Vektoren zur Kompilierzeit verwendet werden, allerdings lassen sich noch keine Mengen zur Kompilierzeit aufbauen und dann zur Laufzeit verwenden. Alle Features sind erforderlich, um vermutlich mit C++23 Reflection einzuführen.

Asynchrone Programmierung

C++20 ermöglicht auch ein erstes Arbeiten mit Koroutinen. Das sind Funktionen, die (nach einem Zwischenergebnis) unterbrochen und später wieder aufgenommen werden können (siehe Abbildung). Die Aufgabe einer Koroutine könnte zum Beispiel sein, nach und nach das nächste Element einer Menge zu liefern (Listing 5).



Koroutinen können unterbrochen und fortgesetzt werden.

Listing 5: Eine Koroutine hält die Ausgabe an und setzt sie später wieder fort

```
IntGen nextElemGen(const auto& coll)
{
    for (int elem : coll) {
      co_yield elem;  // Schleife und Funktion mit elem als 
                                                  Zwischenergebnis anhalten
    }
}
```

Mit `co_yield` wird die Funktion zunächst angehalten und der aktuelle Wert von `elem` zurückgeliefert. Eine weitere Funktion kann diese nun nach und nach nutzen, um zusätzliche Werte abzurufen, was die Koroutine dann fortsetzt:

```
IntGen gen = nextElemGen(coll);
for (int val : gen) {  // iteriert über die jeweils fortgesetzte
                       //Coroutine
    doSomeStuff(val);
}
```

Das Bindeglied zwischen der Koroutine und ihrem Aufruf ist eine Generator-Klasse, die den Zustand von Koroutinen hält. Sie zu implementieren, ist nicht trivial. Man muss dies noch selbst erledigen (Listing 6). Insofern benötigen auch Koroutinen noch Ergänzungen, die die nächsten Releases sicher enthalten.

Ein weiteres Beispiel einer schrittweisen Verbesserung ist der neue Datentyp für UTF-8-Zeichenliterale `char8_t`. Das Typsystem stellt damit endlich die semantische Bedeutung eines UTF-8-Zeichens sicher. Darauf aufbauend wird es in den nächsten C++-Versionen signifikante Verbesserungen im Umgang mit UTF-8 geben.

Raumschiffoperator

Neue Regeln und ein neuer Operator machen die Definition von Vergleichsoperatoren einfacher. Zunächst ist in Klassen mit `operator==` nun auch automatisch `operator!=` definiert. Für andere Vergleiche liegt der neue Drei-Wege-Vergleichsoperator `<=>` vor, auch Raumschiffoperator genannt: `x <=> y` liefert einen Wert kleiner 0, wenn x kleiner ist, 0, wenn x und y gleich sind, und größer 0, wenn y größer ist. Seine Definition in Klassen definiert implizit die anderen Vergleichsoperatoren (wenn keine Probleme vorliegen). So kann man in Klassentemplates grundsätzlich alle Vergleichsoperatoren wie in Listing 7 definieren. Als Default erfolgt der Vergleich über die ganze Hierarchie Member-weise.

Während in C++17 String-Views Einzug hielten (Read-only-Sichten mit einem String-Interface auf existierende externe Zeichenfolgen), enthält C++20 nun Sichten auf existierende Sequenzen beliebiger Objekte mit Schreibzugriff. Damit können Entwickler*innen Code implementieren, der sequenzielle Mengen oder Teilmengen bearbeitet, ohne sie zu besitzen. Die folgende Funktion verdoppelt alle Elemente einer Sequenz von Integerwerten (siehe auch Listing 8):

```
void doubleContent(std::span<int> coll) {
    for(auto& val : coll) {
      val *= 2;
    }
}
```

Spans sind keine Templates, sondern eine Art Wrapper, mit dem sich auf existierenden Mengen operieren lässt. Das ist so, als hätte man Zeiger oder Iteratoren auf die Elemente übergeben, um die Elemente zu bearbeiten; nur dass weiter eine containerartige Schnittstelle verwendet wird. Deshalb müssen Programmierer*innen auch unbedingt darauf achten, dass die von Span bearbeiteten Mengen lange genug existieren.

Spans können by value übergeben werden, und seit C++17 sind dank Class Template Argument Deduction (CTAD) bei der Initialisierung keine Elementtypen mehr anzugeben. Auch Teilsequenzen sind bequem erzeug- und verwendbar:

```
// verdoppele die ersten beiden Werte in v:
doubleContent(std::span{v}.first(2));
```

Die chrono-Bibliothek wurde so erweitert, dass sie nun mühelos für alle möglichen Kalenderfunktionen (inklusive Behandlung von Zeitzonen) verwendet werden kann. Dabei galt wie immer das Designprinzip, möglichst viele Fehler zur Kompilierzeit zu finden. Ein Tag lässt sich sowohl mit 2019y/11/14 als auch mit 14/11/2019y definieren, das y ist aber wichtig, um zu wissen und zu prüfen, ob Tag, Monat und Jahr korrekt benutzt werden. Auch so etwas wie Thursday[last]/November/2019 für den letzten Donnerstag im November ist möglich. Zeitzonenbehandlung ist einfach (Namespaces sind weggelassen):

```
zoned_time tp1{current_zone(), now()};  // aktuelle lokale Zeit
zoned_time tp2 {"Europe/Paris", tp};    // aktuelle Zeit in Paris
```

Zusätzlich gibt es neue Clock-Datentypen, um Zeitpunkte mit Zeitangaben aus anderen Kontexten zu vergleichen oder diese umzurechnen: `file_clock` (für Filesystem-Einträge), `utc_clock` (zur Behandlung von Schaltsekunden) sowie `gps_clock` und `tai_clock` für Zeitpunkte aus den Standards GPS (Navigation) und TAI (internationale Atomzeit).

Formatierte Ein- und Ausgabe

C++ führte von Anfang an ein neues Konzept für Ein- und Ausgaben ein: I/O Streams. Auch nach 20 Jahren verwenden Entwickler*innen für formatierte Ausgaben noch Funktionen wie `printf()`, weil sie leichter zu benutzen sind und mitunter auch weniger Ressourcen brauchen. C++20 enthält deshalb eine neue, standardisierte Bibliothek zur einfachen formatierten Ausgabe. Sie ist schnell, typsicher und erweiterbar.

`std::format()` erzeugt einen formatierten String, in dem sich mit geschweiften Klammern die nachfolgenden Argumente ansprechen lassen:

```
std::string str{"hello"};
// - String und int nach Reihenfolge der Argumente
std::format("'{}' size: {}\n", str, str.size());
```

Listing 6: Generator-Beispiel für eine Koroutine

```
class IntGen {
    public:
      struct promise_type {
        int current_value;
        ...
        auto yield_value(int value) {
          current_value = value;
          return std::suspend_always{};
        }
      };
    private;
      using handle = std::coroutine_handle<promise_type>;
      handle coro;
      IntGen(handle h) : coro{h} {
      }
    public:
      bool nextValue() {
        return  coro ? (coro.resume(), !coro.done()) : false;
      }
      int getValue() {
        return coro.promise().current_value;
      }
      ...
      using iterator = ...   // ++ setzt die Coroutine fort
      iterator begin();
      iterator end();
};
```

Listing 7: Vergleichsoperatoren in Klassentemplates

```
template<typename T1, typename T2>
class P {
    private:
      T1 x1;
      T2 x2;
    public:
    ...
      friend auto operator<=>(const P&, const P&) = default;
};
```

Listing 8: Jede Sequenz von Integern verdoppeln

```
std::vector<int> v{42, 43, 44, 45};
doubleContent(v);

std::array<int,4> a{-14, 55, 24, 67};
doubleContent(a);

int arr[]{0, 8, 15};
doubleContent(arr);
```

Die Ausgabe der Argumente erfolgt formatiert anhand ihrer Datentypen. Die Reihenfolge der Argumente ist dabei unerheblich:

```
// - Reihenfolge wie spezifiziert
std::format("'{1}' size: {0}\n", str.size(), str);
```

Es sind verschiedene Formatierungen möglich, wie rechtsbündig mit führenden Unterstrichen:

```
std::format("'{:_>20}' has {0} chars\n", str, str.size());
```

Mit `format_to()` kann hierbei auch in existierenden Speicherplatz für Zeichen geschrieben werden. Wie bereits beschrieben, ist der Mechanismus erweiterbar. Die chrono-Bibliothek verwendet diese Erweiterungsmöglichkeit, um Zeitpunkte und Zeitdauern formatiert ausgeben zu können:

```
std::format("Datum: {:%d.%m.%Y %T%z}\n", zonedTimePoint);
```

Der Standard enthält drei weitere Klassen für die Synchronisierung von Threads: Latches, Barriers und Semaphore. Ein Latch ist eine Art Countdown, der initialisiert mit einem Startwert bestimmte Ereignisse herunterzählt und bei 0 „fertig“ meldet. So kann man 10 Threads beauftragen, etwas zu tun, und den Zähler mit 5 initialisieren. Wenn dann jeder Thread sein Ende dem Latch als Ereignis mitteilt, meldet dieser nach 5 fertigen Threads, dass er die Ausführung abgeschlossen hat (Listing 9).

Beliebig viele Threads können einen Latch zum Herunterzählen auf 0 verwenden. Jeder Thread darf beliebig oft und viel herunterzählen. Ist er abgeschlossen und steht somit auf 0, kann er seinen Zustand nicht mehr verändern; er eignet sich also nur für einen einmaligen Countdown.

Die hier verwendete Threadklasse `std::jthread` ist neu und behebt einige Designschwächen von `std::thread`. Es handelt sich um einen echten RAII-Typ, der aufräumt, wenn man das Threadobjekt zerstört (ohne `join()` aufgerufen zu haben). Der Destruktor teilt dem laufenden Thread sogar vorher mit, dass dieser sich beenden soll. Das kann ein kooperativ implementierter Thread nun bequem auswerten, um sich sauber zu beenden.

Barriers sind so organisiert, dass sie mehrfach Ereignisse verschiedener Threads synchronisieren können. Auch hier handelt es sich um einen Zähler, den jeder Thread aber nur einmal herunterzählen kann. Steht der Barrier auf 0, wird eine vordefinierte Aktion aufgerufen und der Barrier lässt sich erneut verwenden. Das ist hilfreich, wenn das Programm nach einem bestimmten Ereignis immer wieder darauf warten muss, dass eine bestimmte Anzahl paralleler Reaktionen erfolgt ist. Ein Semaphor ist ein verallgemeinerter Mutex. Damit kann der Zugriff auf eine Ressource synchronisiert werden. Im Gegensatz zu einem Mutex müssen der Thread, der eine Ressource anfordert, und der Thread, der die

Listing 9: Latch mit zehn Threads, von denen fünf fertig werden müssen

```
#include <latch>

// Menge von Threads:
const int numThreads{10};
std::vector<std::jthread> threads;

// 5 muessen fertig werden:
std::latch ready{5};

// alle Threads starten, die dann fertig melden:
for (int i=0; i<numThreads; ++i) {
    threads.push_back(std::jthread{[&ready]{
      doSomeStuff();
      ready.count_down();
      }});
}
// warten, dass 5 fertig sind:
ready.wait();

...  // Reaktion auf mindestens 5 fertige Threads
```

Listing 10: Limitierung auf je drei parallel laufende Threads

```
std::counting_semaphore sem{3};
for (int i=0; i<numThreads; ++i) {
    threads.push_back(std::jthread{[&sem]{
                                      sem.acquire();
                                      doSomeStuff();
                                      sem.release();
                                    }});
}
```

Anforderung wieder freigibt, nicht identisch sein. Außerdem kann man dabei zulassen, dass eine bestimmte maximale Anzahl paralleler Zugriffe möglich ist. So lässt der Code in Listing 10 immer nur maximal drei Threads auf einmal `doSomeStuff()` aufrufen.

Weitere neue Merkmale von C++20 sind:

- Range-based for-Schleifen, die jetzt mit einer zusätzlichen Initialisierung beginnen;
- Gleitkommawerte und Objekte von Klassen, die nun Templateparameter sein können;
- Erweiterungen bei atomaren Datentypen;
- bessere Unterstützung zum Debuggen (`std::source_location`).

Fazit

Der C++-Standard ist zwar nur um etwa 15 Prozent gewachsen, aber die Neuerungen werden die Art, in Zukunft C++ zu programmieren, wesentlich beeinflussen. Zu den genannten großen Themen Konzepte, Ranges, Module und Koroutinen wird es Verbesserungen in den nächsten C++-Standards C++23 und C++26 geben. Daher sollten sie mit etwas Vorsicht genutzt werden, auch wenn sie in einigen Compilern schon zur Verfügung stehen.

Bis zur vollständigen Implementierung von C++20 in Compilern werden sicher noch zwei bis drei Jahre vergehen. Alle Beispiele in diesem Artikel sind deshalb ohne Gewähr, weil eine standardkonforme Verwendung bisher nur marginal und experimentell unterstützt wird. Aber kleinere Erweiterungen werden zeitnah zur Verfügung stehen. Wer in C++ programmiert, sollte wie immer wissen, was er tut. Er wird dann aber weiter mit guter Performance bei hoher Flexibilität belohnt. (nb@ix.de)

Quellen

Alle Listings zum Download: ix.de/zwbr



Nicolai Josuttis

ist seit 25 Jahren deutscher Vertreter bei der Standardisierung von C++ und Buchautor etlicher Standardwerke zur Programmierung mit C++.



Rust: nicht nur für den Browser

Lichtblick

Jens Breitbart und Stefan Lankes

Die Sprache Rust eignet sich für den Einsatz im Browser. Ihre Effizienz und Eigenständigkeit macht sie aber auch für Cloud-Services und den Einsatz im Embedded-Bereich interessant.

Rust entsprang 2010 einem privaten Projekt des Mozilla-Mitarbeiters Graydon Hoare. Fünf Jahre später erschien die erste stabile Version der noch jungen Programmiersprache, deren Weiterentwicklung ursprünglich Mozilla Research vorantrieb. Mittlerweile hat sich jedoch eine rege Community herausgebildet, die die Weiterentwicklung übernimmt. Hier gibt es eine Reihe fester Teams und anwendungsgetriebene Working Groups; eine genaue Übersicht findet sich auf der Rust-Homepage.

Die Entwicklung der Sprache orientiert sich an drei Zielen. Zum Ersten ist das die Performance. Rust arbeitet daher ohne Laufzeitumgebung und Garbage Collector, was Anwendungen mit derselben Laufzeitgeschwindigkeit und Speichereffizienz wie vergleichbare in C entwickelte Programme ermöglicht. Das zweite Ziel ist Verlässlichkeit. Bereits zur Kompilierzeit kann der Rust-Compiler verschiedene Klassen von Bugs entdecken, insbesondere im Bereich illegaler Speicherzugriffe und Race Conditions. Wenn Rust Code ohne Fehlermeldung kompiliert, können Entwickler*innen im Allgemeinen davon ausgehen, dass tatsächlich keine entsprechenden Fehler vorhanden sind.

Als drittes Ziel hat sich die Community das Thema Produktivität vorgenommen. Für sie haben Tools und Dokumentation einen hohen Stellenwert: Der Compiler soll hilfreiche Fehlermeldungen liefern und das eingebaute Build-Tool und die Paketverwaltung Cargo sollen die typischen Probleme von Build-Umgebungen für Rust-Projekte lösen. Auch die meisten bekannten Editoren beherrschen inzwischen den Umgang mit Rust.

iX-TRACT

- Die von Mozilla geförderte Sprache Rust erschien 2015 in einer ersten stabilen Version.
- Die Syntax lehnt sich der von C an. Aufeinanderfolgende Anweisungen lassen sich mit Semikolon abtrennen und Blöcke über geschweifte Klammen markieren.
- Die Ziele der Sprachentwicklung sind Performance, Verlässlichkeit und Produktivität.

Rust ist nicht nur für den Webbrowser von Mozilla

Die genannten Ziele bieten Vorteile für eine Reihe verschiedener Einsatzszenarien. Bekannt ist sicherlich, dass Rust im Firefox-Browser von Mozilla Verwendung findet, insbesondere ist die Browser-Engine Servo in Rust geschrieben. Aber auch System-

komponenten bekannter Anwendungen setzen darauf. So benutzt Dropbox Rust für Teile ihres Storage Backends und auch Amazons leichtgewichtige Firecracker-VM ist komplett in Rust geschrieben. Überhaupt bieten sich Cloud-Services für eine Umsetzung in Rust an, da sich die Effizienz der Sprache, abhängig von der Anzahl der Nutzer, für Betreiber direkt finanziell bezahlt machen kann. Cloudflare nutzt das für ihre HTML Rewriting Engine und auch NPM implementiert verschiedene Services in Rust. Dass die Rust-Standardbibliothek explizit Teile ausweist, die ohne Betriebssystem funktionieren, ist schließlich auch im Kontext von eingebetteten Systemen interessant, zumal die Cargo-Paketverwaltung die Handhabe entsprechender Projekte unterstützt.

Die Rust-Syntax: fast wie in C/C++

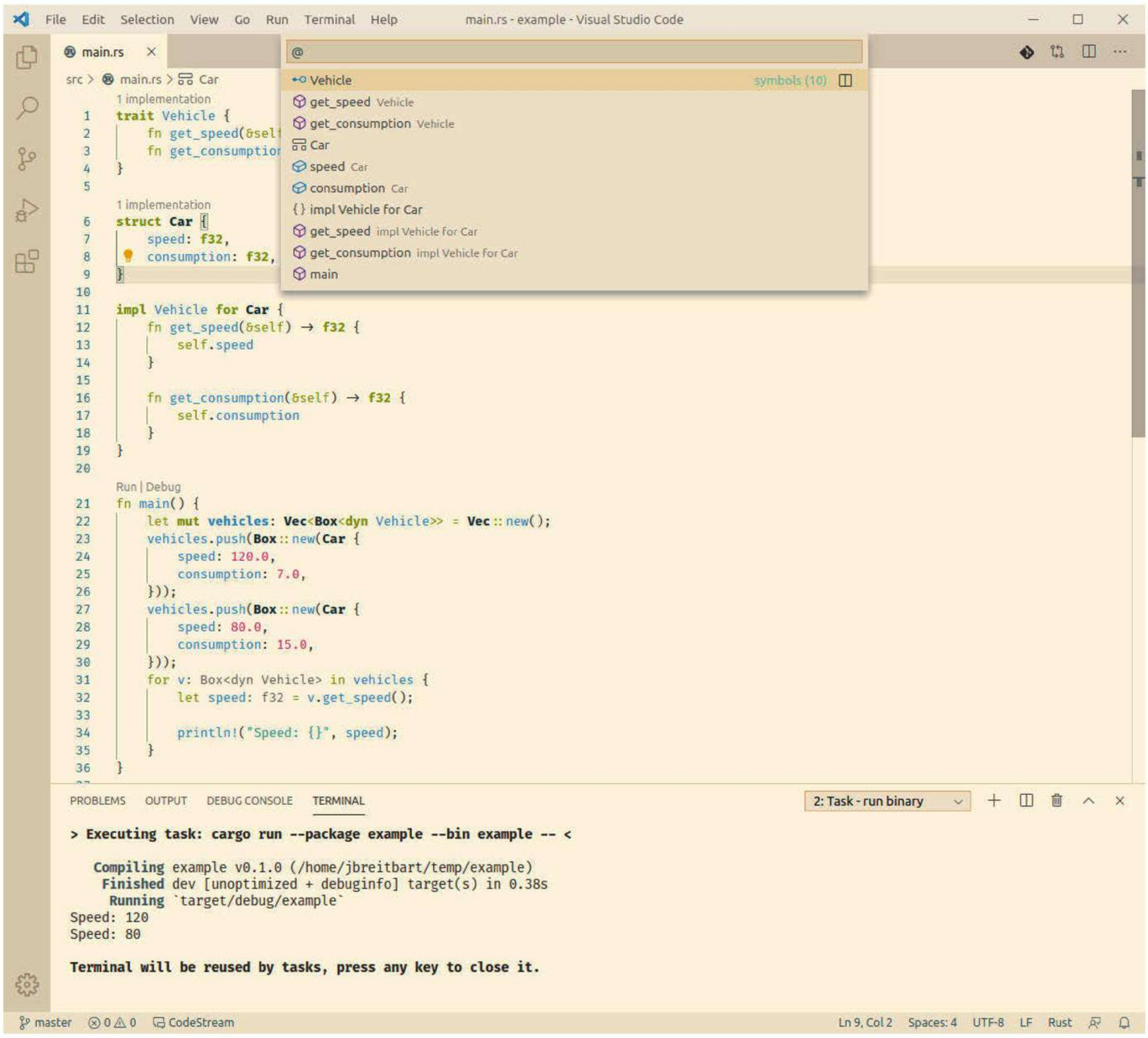
Die Syntax von Rust lehnt sich an C/C++ an. Entwickler*innen trennen aufeinanderfolgende Anweisungen mit Semikolon ab und markieren Blöcke über geschweifte Klammen. Ein einfaches Hello-World-Programm sieht entsprechend wie folgt aus:

```
fn main() {
    println!("Hello, world!");
}
```

Im Detail gibt es dabei durchaus Abweichungen von C, so sind Variablen wie folgt zu definieren:

```
let x: Typ = Wert;
let mut v: Typ = Initialwert;
v = NeuerWert;
x = v;
```

Das Snippet zeigt nicht nur, dass die Syntax für die Variablendefinition deutlich von der von C abweicht, sondern stellt auch einen entscheidenden semantischen Unterschied heraus: Variablen sind im Normalfall unveränderlich, außer Entwickler*innen markieren sie über das Schlüsselwort `mut` als `mutable`. Im gezeigten



Mit Visual Studio Code und dem Plug-in rust-analyzer ist es möglich, die Symbole, also Funktionen, Strukturen und Traits des Projektes, anzeigen zu lassen und an die entsprechende Stelle im Code zu springen. Weiterhin lässt sich die Anwendung auch direkt mit oder ohne Debugger ausführen.

Listing 1: Beispiel für das Verwenden von Traits

```
trait Vehicle {
    fn get_speed(&self) -> f32;
    fn get_consumption(&self) -> f32;
}

struct Car {
    speed: f32,
    consumption: f32,
}

impl Vehicle for Car {
    fn get_speed(&self) -> f32 {
      self.speed
    }

    fn get_consumption(&self) -> f32 {
      self.consumption
    }
}
```

Listing 2: Einen Vektor mit Trait Objects erzeugen

```
let mut vehicles: Vec<Box<dyn Vehicle>> = Vec::new();

vehicles.push(Box::new(Car::new()));
vehicles.push(Box::new(Truck::new()));

for v in vehicles {
    let speed = v.get_speed()

    // do something with speed
}
```

Beispiel erkennt der Compiler den Versuch, der Variablen x einen neuen Wert zuzuweisen, und wirft eine entsprechende Fehlermeldung.

Rust bietet an C angelehnte Strukturen an, allerdings mächtiger und in vielen Bereichen den Strukturen und Klassen aus C++ ähnlich. Konkret können Structs in Rust Member-Funktionen enthalten und auch sogenannte Traits implementieren. Letztere beschreiben eine Sammlung von Methoden, im Unterschied zu einer abstrakten Oberklasse erlauben sie jedoch eine horizontale Wiederverwendung von Methodensammlungen und vermeiden so Mehrfachvererbungen. Anders als C++ und Java stellt Rust also keine Vererbung zur Verfügung.

Listing 1 zeigt die Definition des Traits `Vehicle`, das die maximale Geschwindigkeit und den Treibstoffverbrauch eines Fahrzeugs beschreibt (alle Listings siehe ix.de/z1rn). Die Struktur `Car` implementiert die Eigenschaften von `Vehicle`, das heißt, sie implementiert die entsprechenden Methoden. Analog können die Fahrzeuge `Truck` und `Caravan` angelegt werden, die jeweils unterschiedliche Implementierungen besitzen. Im Unterschied zu C übergeben Entwickler*innen den Methoden die Selbstreferenz `self`, worüber diese auf die Elemente zugreifen können. Das `&` beschreibt wie in C eine Übergabe per Referenz. Auffällig ist, dass der Rückgabewert nicht unbedingt mit dem Schlüsselwort `return` zurückgeliefert wird, sondern Entwickler*innen im Code dem Compiler durch Weglassen des Semikolons signalisieren, dass hier das Ergebnis der Zeile zurückzugeben ist.

Es ist naheliegend, alle Fahrzeuge in einer gemeinsamen Datenstruktur zu verwalten. Dafür kommt ein Vektor (`Vec`) infrage, der ein dynamisches Feld darstellt und mit der Anzahl der Elemente wächst. Um dies zu implementieren, benötigt der Compiler zur Kompilierzeit die Größe der einzelnen Elemente. Für ein Trait wie `Vehicle` kann der Compiler das jedoch nicht bestimmen. Entwickler*innen haben aber die Möglichkeit, Referenzen zu einem `Vehicle` zu verwalten, denn deren Größe ist bekannt.

Das Beispiel in Listing 2 verpackt das Fahrzeug in eine Box. Der Datentyp `Box` stellt einen sogenannten Owning Smart Pointer dar, der das Objekt auf den Heap legt und automatisch löscht, sobald es nicht mehr in Verwendung ist.

Entsorgungssystem

Wie bereits erwähnt, bietet Rust ein sicheres Speichermodell an und verzichtet hierbei auf Garbage Collection, die zur Laufzeit versucht, nicht länger benötigte Speicherbereiche zu identifizieren und freizugeben. In Rust erfolgt dies bereits zur Kompilierzeit. Das Konzept der Ownership, das einen der größten Unterschiede zu C/C++ ausmacht, ermöglicht das.

Das Beispiel in Listing 3 zeigt die Möglichkeit, gültigen C++-Code zu schreiben, der zwar erfolgreich kompiliert wird, aber ungültige Speicherzugriffe enthält. Der Vektor z wird nach dem Verlassen des Codeblocks zwischen den geschweiften Klammern gelöscht, obwohl noch die Referenz x darauf existiert.

Ein entsprechender Fehler lässt sich auch in Rust programmieren (Listing 4). Im Gegensatz zu einem C/C++-Compiler übersetzt ein Rust-Compiler diesen Code nicht, obwohl er syntaktisch korrekt ist. Er meldet es als Fehler, dass z nicht lange

Listing 3: Beispiel einer ungültigen Referenz in C++

```
std::vector<std::string>* x = nullptr;

{
    std::vector<std::string> z;

    z.push_back("Hello World!");
    x = &z;
}

std::cout << (*x)[0] << std::endl;
```

Listing 4: Versuch zum Erzeugen einer ungültigen Referenz in Rust

```
let x;

{
    let z = vec!("Hello World!");

    x = &z;
}

println!("{}", x[0]);
```

Listing 5: Objekte ausleihen

```
let mut x = vec!("Hello World!");

{
    let z = &mut x;
    // Do something with z...
}

println!("{}", x[0]);
```

Listing 6: Generische Typen verwenden

```
struct Vector<T> {
    x: T,
    y: T,
    z: T,
}

fn do_something<T>(v: Vector<T>) {
    // do something with the vector v
}
```

Listing 7: Fehlerbehandlung in Rust

```
enum MathError {
    DivisionByZero,
    NonPositiveLogarithm,
    NegativeSquareRoot,
}

fn div(x: f64, y: f64) -> Result<f64, MathError> {
    if y == 0.0 {
      Err(MathError::DivisionByZero)
    } else {
      Ok(x / y)
    }
}
```

genug lebt. Um Speichersicherheit zu gewährleisten, führte Rust die Konzepte Ownership (Besitz) und Borrowing (Ausleihen) ein. Dabei gibt es immer nur einen Besitzer. Er hat das Recht, Objekte freizugeben und alle Ressourcen, etwa durch das Objekt benutzten Speicher, zurückzugeben. Durch das Erstellen von Referenzen lässt sich das Objekt verleihen (Listing 5).

Das Erstellen der Referenz auf `x` verleiht das Objekt. Die geschweiften Klammern sind in diesem Beispiel essenziell. Ohne sie verweigert der Compiler in der letzten Zeile das Ausleihen des Vektors an `println`, da nicht `x`, sondern `z` der Besitzer des Vektors ist. Es gilt die Grundregel, dass nur der aktuelle Besitzer etwas verleihen darf.

Das Ownership-Borrowing-Konzept ist gewöhnungsbedürftig, hat aber deutliche Vorteile: Die Speicherverwaltung ist sicher und nebenläufiger Code bei nur einem Besitzer einfach zu realisieren.

Rust ermöglicht die Verwendung generischer Typen. Datenstrukturen und Funktionen lassen sich so formulieren, dass beim Schreiben der genaue Typ nicht bekannt ist. Erst zur Kompilierzeit kann man ihn bestimmen und einsetzen.

Das Beispiel in Listing 6 definiert einen Vektor, bei dem noch nicht bekannt ist, ob eine einfach genaue (`f32`) oder eine doppelt genaue Fließkommazahl (`f64`) zur Beschreibung der Koordinaten verwendet wird. Daher kommt der generische Typ `T` zum Einsatz.

Um sinnvoll eine Berechnung in `do_something` zu realisieren, müssen Entwickler*innen bekannt geben, welche Methoden auf `T` anwendbar sind. Dazu dienen Traits, die die auf `T` ausgeführten Operationen beschreiben. `T` lässt sich nur für Typen substituieren, die die entsprechenden Traits implementieren. Der Trait `std::ops::Add` beschreibt, dass sich die Werte des Typs, der den Trait implementiert, addieren lassen.

Fehlerbehandlung

Vielen Programmiersprachen ist gemein, dass sie das Ergebnis nicht auf mögliche Fehler überprüfen. Typischerweise werden in Rust Ergebnisse in ein Result-Objekt (`Result<T, E>`) gepackt. Hier kann der Typ `T` ein beliebiger Datentyp für das Ergebnis sein, während `E` den Fehler repräsentiert.

Das Beispiel aus Listing 7 liefert durch die mathematische Division das Ergebnis als doppelt genaue Fließkommazahl zurück, während im Fehlerfall die Fehlerart durch `MathError` beschrieben ist. Die Rückgabe eines validen Ergebnisses erfolgt über `Ok`, ein Fehler wird in `Err` verpackt.

Der Aufruf der Funktion `div` erzwingt eine Auswertung des Rückgabewertes, um zu überprüfen, ob das Ergebnis korrekt war oder ein Fehler vorliegt. Es gibt sonst keine Möglichkeit, an das benötigte Ergebnis zu kommen. Im Aufruf lässt sich dies mit dem Schlüsselwort `match` realisieren, das den Rückgabewert

Listing 8: Auswerten des Rückgabewertes

```
match div(42.0, 2.0) {
    Err(why) => panic!("math error"),
    Ok(result) => println!("result {}", result);
}

let result = div(24.0, 0.0).unwrap();
println!("result {}", result);
```

Listing 9: Optionale Rückgabewerte verwenden

```
fn find(s1: String, s2: String) -> Option<usize> {
    ...
}

match find("Heise", "eise") {
    Some(i) => println!("Found substring at position {}", i),
    None => println!("Unable to found substring"),
}
```

auspackt und in den entsprechenden Zweig springt (Listing 8). Den Vorgang kann man mit `unwrap` verkürzen, das das Programm abbricht, wenn der Rückgabewert ein Fehler ist, und es andernfalls auspackt.

Es gibt aber auch Situationen, in denen Funktionen abhängig von ihren Eingaben kein Ergebnis zurückliefern können. Bei einer Suche innerhalb eines Strings stellt das Nichtauffinden eines Teilstrings nicht zwingend ein Fehler dar, sondern kann ein gewolltes Verhalten sein. In einem solchen Fall ist die Verwendung einer `Option<T>` zu bevorzugen, der Index der Fundstelle zurückzuliefern und diese in `Some` zu packen. Ist der Substring nicht auffindbar, gibt die Funktion `None` zurück. Das Auswerten des Ergebnisses lässt sich äquivalent mit der `match`-Anweisung realisieren (Listing 9).

Beliebte Sprache

Die immer noch relativ neue Programmiersprache Rust erfreut sich großer Beliebtheit und hat zuletzt viermal in Folge die Umfrage nach der beliebtesten Sprache auf Stack Overflow gewonnen. Den Einstieg erleichtert der Onlineeditor Rust Playground, für die ersten Schritte reicht der integrierte Compiler aus und selbst die lokale Installation ist dank rustup einfach gehalten. Wer sich ein Bild verschaffen will, welcher Editor welche Features unterstützt oder welche Plug-ins zu empfehlen sind, geht auf die Webseite „Are we (I)DE yet?“ (ix.de/z1rn). (nb@ix.de)

Quellen

Listings zum Download, zu Editoren und Plug-ins: ix.de/z1rn



Dr. Jens Breitbart

arbeitet als Softwarearchitekt bei Bosch Automated Driving. Er arbeitet zudem am Softwareprojekt HermitCore mit.



Dr. Stefan Lankes

arbeitet als akademischer Direktor am Institute of Automation of Complex Power Systems der RWTH Aachen University. Er forscht im Bereich Hochleistungsrechner und ist Initiator des Open-Source-Projekts HermitCore, eines Unikernel-Betriebssystems.



Go für die Cloud

Gut verteilt

Frank Müller

Die Sprache Go beherrscht Nebenläufigkeit durch Goroutines. Sie eignet sich damit für den Einsatz in skalierbaren und verteilten Systemen.

Ende 2007 dauerte den drei Google-Mitarbeitern Ken Thompson, Rob Pike und Robert Griesemer das Kompilieren ihrer Software mal wieder zu lang. Darin und in weiteren Punkten erkannten sie ein Dilemma der damals existierenden Sprachen. So reifte bei ihnen als erfahrenen Sprachentwicklern eine Idee: Go. Alle waren zu diesem Zeitpunkt mit einigen Projekten beschäftigt. Ken Thompson, Träger des Turing Award, und Rob Pike arbeiteten zuvor an Multics, Unix, B als Vorgänger von C, Plan 9, dem Acme-Editor, UTF-8 und weiteren Projekten. Robert Griesemer befasste sich mit dem statisch typisierten Smalltalk-Ableger Strongtalk, der Java-VM HotSpot und der JavaScript-Engine V8. Seinerzeit waren sie jedoch frustriert, dass keine Mainstreamsprache gleichzeitig ein effizientes Kompilieren und Ausführen sowie eine einfache Programmierung bot. Und so investierten sie etwas Konzeptarbeit, starteten Mitte 2008 mit der Implementierung und kündigten am 10. November 2009 eine Pre-Release an. Dieses Datum gilt heute als das offizielle Geburtsdatum der Sprache, auch wenn es bis zur Release 1.0 noch bis zum 28. März 2012 dauern sollte. Seitdem haben die drei, weitere Google-Mitarbeiter und Mitglieder der weltweiten Community einiges geschaffen.

Der Sprachstil ähnelt der Sprache C und ist statisch typisiert. Neben `bool` und `string` kennt Go eine Reihe numerischer Typen. Es sind Integer mit und ohne Vorzeichen in der Breite von 8, 16, 32 und 64 Bit. Sie werden als `int` und `uint` plus ihrer Breite angegeben, also `uint8` und `int64`. Daneben liegen die Typen `int` und `uint` ohne eine angegebene Breite vor. Diese hängt von der Rechnerarchitektur ab und beträgt hier 32 oder 64 Bit. Den Abschluss der Ganzzahlen bilden die beiden Alias `byte` und `rune`. Letzterer nimmt UTF-8-Zeichen auf.

Weitere Typen sind der Pointer-Typ `uintptr`, die Gleitkommazahlen `float32` und `float64` sowie `complex64` und `complex128` für komplexe Zahlen. Damit verfügt die Sprache über eine gute Basis für systemnahe Anwendungen.

Neben direkt nutzbaren gibt es auch zusammengesetzte Typen wie Arrays und die mit ihnen verwandten Slices. Sie basieren auf den oben genannten und weiteren, selbst definierten Typen. Während Arrays sich mit einer festen Größe definieren lassen, erlauben Slices eine Längenveränderung zur Laufzeit. Auch Maps verwalten große Mengen von Daten, greifen allerdings über die Schlüssel eines definierten Typs darauf zu.

Arbeiten mit Strukturen

Wie andere Sprachen kennt Go Strukturen. Funktionen sind hier ebenfalls Typen und einfach und direkt zu definieren. Auch ist es möglich, Typen mit Funktionssignaturen zu deklarieren. Diese finden Verwendung als Argumente, Rückgabewerte oder Variablen und lassen sich als Closures definieren. Hier ähnelt Go einer funktionalen Programmiersprache (Listing 1; alle Listings siehe ix.de/zj85).

iX-TRACT

- Die statisch typisierte Sprache Go erschien 2009 als Pre-Release. Dieses Jahr gilt als Geburtsdatum, auch wenn die erste stabile Release erst 2012 vorlag.
- Go orientiert sich syntaktisch an C. Statt Vererbung nutzt Go Komposition.
- Als Systemsprache eingesetzt, hat sich Go vor allem im Cloud-Bereich etabliert.

Die Objektorientierung in Go verzichtet auf Vererbung, bietet aber im Falle von Strukturen eine Komposition. Methoden lassen sich als Interfaces festlegen. So kann eine Funktion oder Methode definieren, was sie von einem Argument als Fähigkeit erwartet. Oder sie gibt ein Interface zurück und lässt den Aufrufer über die Implementierung im Dunkeln. Dabei muss ein Typ nicht deklarieren, welche Interfaces er implementiert. Beim Kompilieren prüft der Compiler, ob ein übergebener oder ein zurückgegebener Typ ein erwartetes Interface erfüllt. Dies bezeichnet man als Duck Typing. Es fühlt sich leicht an, löst aber bei einer ungültigen Zuweisung beim Kompilieren und nicht erst zur Laufzeit einen Fehler aus – einfache Programmierung trifft auf die Sicherheit der statischen Typisierung. Listing 2 zeigt, wie ein Interface `Duck` zum Quaken auch durch einen Funktionstyp zu implementieren ist. Das Package net/http enthält den Typ `HandlerFunc`, der einfache Web-Handler in nur einer Funktion definiert. Er implementiert dabei das Interface `Handler`, das der im Package enthaltene Server erwartet.

Die Kontrollkonstrukte und Schleifen sind wenig spektakulär. Ein Merkmal von Go ist, dass `else` oder ein tiefes Nesting nur selten zum Einsatz kommen. Die Funktion prüft auf negative Zustände und das Verlassen der Funktion im Fehlerfall. Der positive Fall setzt hingegen den Ablauf auf der gleichen Ebene fort (Listing 3).

Listing 3 demonstriert weitere Eigenschaften der Sprache. Der Operator := deklariert die Variablen der linken Seite mit den Typen der rechten Seite, was den Code leichtgewichtig wie in dynamisch typisierten Sprachen macht (siehe auch Listing 1).

Weiterhin sind Mehrfachzuweisungen möglich – ein wichtiger Punkt, da die Sprache keine Exceptions kennt. Ein Umstand, der sich beheben lässt, indem ein Fehler erzeugt oder ein zuvor erhaltener Fehler mit hilfreichen Informationen angereichert und als einziges oder letztes Argument zurückgegeben wird. Ohne Fehler ist die Rückgabe `nil`, da der Typ `error` ein Interface ist. Das Package errors enthält hilfreiche Funktionen für den Umgang mit Fehlern. Für mehr Komfort sind zwei Handle-Blöcke innerhalb von Funktionen in der Diskussion. Sie arbeiten ähnlich wie ein `catch` im Exception Handling und erlauben es, mehrere Fehler eines Blocks gemeinsam zu behandeln.

Ein weiterer Baustein der Sprache ist das Kommando `defer`. Es legt den ihm folgenden Funktionsaufruf auf einem Stapel ab. Eine Funktion kann mehrere solcher Statements enthalten. Der Stapel wird dann beim Verlassen der umgebenden Funktion rückwärts ausgeführt. Hier zeigt sich, wie leicht der lineare Ablauf ist. Im Codebeispiel in Listing 3 schließt ein durch einen Fehler ausgelöstes `return` automatisch eine bereits geöffnete Datei.

Listing 1: Mit Funktionen spielen

```
// Deklaration.
type Converter func(string) string

// Fabrik.
func MakePrefixer(prefix string) Converter {
    return func(in string) string {
      return prefix + ":" + in
    }
}

type Repeater int

// Methode mit Signatur des Converter.
func (r Repeater) Repeat(in string) string {
    return strings.Repeat(in, int(r))
}

// Anwendung des Converters als Argument.
func ConvertAll(in []string, c Converter) []string {
    out := make([]string, len(in))
    for i, s := range in {
      out[i] = c(s)
    }
    return out
}

...

var r Repeater = 5

in := []string{"fun", "with", "functions"}

// Verschiedene Anwendungen des Converters.
foos := ConvertAll(in, MakePrefixer("foo"))
repeated := ConvertAll(in, r.Repeat)
upperCased := ConvertAll(in, strings.ToUpper)
```

Umgang mit Nebenläufigkeit

Ein weiteres mächtiges Feature der Sprache ist die Nebenläufigkeit auf Basis des Konzepts der Communicating Sequential Processes, kurz CSP, von Tony Hoare. Ein Programm besteht aus einer Reihe leichtgewichtiger Prozesse, den Goroutines. Dabei handelt es sich um einfache Funktionen und Methoden, die mit dem Schlüsselwort `go` in den Hintergrund geschickt werden. Goroutines haben einen viel geringeren Overhead als Threads, und die Runtime betreibt sie in einem Threadpool. Kontextwechsel zwischen ihnen kann das System sehr schnell vollziehen, die Kommunikation erfolgt über typisierte Channels. So sind Prozesse mit mehreren Tausend Goroutines gleichzeitig möglich. Die erste von ihnen ist immer das Hauptprogramm `main()`.

Goroutines übernehmen verschiedene Aufgaben: So können Entwickler*innen Funktionen mit den notwendigen Daten zur Verarbeitung in den Hintergrund schicken; sie beenden sich abschließend. Der in net/http enthaltene Server arbeitet auf diese Weise für jeden Request und ist dennoch sehr schnell. Andere Goroutines nehmen die über einen Channel zu verarbeitenden Daten entgegen und liefern über einen zweiten die Ergebnisse zurück. Die Rückgabechannels lassen sich für individualisierte Aufträge aus unterschiedlichen Goroutines auch mit den Eingabedaten mitgeben. Im Anschluss wartet nur die versendende Goroutine auf die Antwort. Dieser Vorgang ähnelt der Objektorientierung. In einer Endlosschleife betreibt die Goroutine das Statement `select` für das Auslesen mehrerer Channels. Diese reagieren auf die erhaltenen Daten, verarbeiten sie und verändern bei Bedarf auch den verwalteten Zustand. So werden alle lesenden und schreibenden Zugriffe serialisiert.

Goroutines lassen sich zusammen mit Convenience-Funktionen oder Methoden problemlos in nebenläufigen Anwendungen

Listing 2: 'Hello, World!' umständlich

```
package main

import "fmt"

type Duck interface {
    Quack() string
}

type NamedDuck func() string

func (nd NamedDuck) Quack() string {
    return fmt.Sprintf("Quack, I'm %s.", nd())
}

func greet(duck Duck) {
    fmt.Println(duck.Quack())
    fmt.Println("Hello, and I'm the computer.")
}

func main() {
    var donald NamedDuck = func() string {
      return "Donald"
    }
    greet(donald)
}
```

betreiben, auch wenn es Risiken bei Race Conditions und Deadlocks gibt. Als Beispiel für eine Goroutine dient eine kleine State Machine, die Datenpakete verarbeitet. Ein Funktionstyp bildet den jeweiligen Zustand ab, erhält das Datum als Argument und gibt den Folgezustand und einen eventuellen Fehler zurück.

Die State Machine soll `context.Context` für die Kontrolle der Goroutine und einen Channel für die Verarbeitung der Daten enthalten sowie den jeweils aktuellen `State` kennen (Listing 4). Für die Arbeitsweise der Goroutine kommt ein dem Actor verwandtes Muster zum Einsatz: das Versenden eines Closure. Es erleichtert den Umgang mit einem eventuell zurückgegebenen Fehler.

Um Daten komfortabel umsetzen zu können, erhält der Typ eine exportierte Methode. Sie verpackt die Kommunikation mit der Goroutine. Hierfür erzeugt sie die Aktionsfunktion zum Aufruf des aktuellen Zustands und Setzen des Folgezustands und gibt im Fehlerfall den entsprechenden Fehler zurück. Die in der Konstruktorfunktion genannte Methode `sm.off` ist wie die Goroutine eine der privaten Methoden der State Machine, die den jeweils aktuellen Zustand beschreiben. Sie ist die erste, die eintreffende Daten verarbeitet.

Die Zustandsmethoden können sich nun um die Datenverarbeitung kümmern. Eventuell benötigte weitere Zustandsdaten lassen sich entweder als weitere Felder der State Machine definieren, wie Zähler oder Register, oder die Zustände werden nicht als Methoden, sondern als Closures definiert. Sie enthalten damit auch immer Informationen aus der Definitionsumgebung. Hier treffen Nebenläufigkeit, Objektorientierung und funktionale Programmierung aufeinander.

Mitbringsel

Der in Listing 4 genutzte `context.Context` ist ein einfacher, aber mächtiger Typ der Standardbibliothek. Er hilft, Goroutines-Hierarchien kontrolliert über unterschiedliche Bedingungen zu beenden, oder fügt dem Call Path Informationen bei. Wie Sprache und Runtime definieren die mitgelieferten Packages den Einsatzort. Der Inhalt der Bibliothek ist klar gegliedert, die Packages sind überwiegend technisch orientiert. Themen wie relationale Datenbanken, verteilte Transaktionen oder grafische Oberflächen sind selten. Dafür sind Hilfen für die Standardtypen vorhanden, Ein- und Ausgabe sind eingebunden sowie Komprimierung, Encodings und Templates geboten. Hinzu kommt der Zugriff auf

Listing 3: Flacher Programmverlauf

```
filename := "/path/to/my/file"
file, err := os.Open(filname)
if err != nil {
    return fmt.Errorf("cannot open %q: %v", filename, err)
}
defer file.Close()

// Weitermachen ...
```

das Betriebssystem, auf das Netzwerk als Client und als Server sowie auf die Kryptologie. Auch ein Package für Unit-Testing ist im Standard enthalten.

Das Package net/http zeigt, wie sehr die Typen ineinandergreifen und dass für den Aufbau einer Web-API keine großen externen Packages notwendig sind. Startpunkt für eine Komponente, die einen Request entgegennimmt und ihn bearbeitet, ist das Interface `Handler`. Der Typ `Server` erwartet es. Das Interface definiert die Methode `ServeHTTP(ResponseWriter, *Request)`, die `Request` entgegennimmt und eine Antwort über `ResponseWriter` zurückschreibt. Dabei ist nicht ein eigener Typ mit der obigen Methode zu implementieren. Liegt nur eine Funktion ohne einen Zustand vor, genügt die Definition einer Funktion vom Typ `HandlerFunc`. Dieser Funktionstyp verfügt über die gleiche Signatur und über die Methode `ServerHTTP()`, in der der implementierte Funktionstyp selbst aufgerufen wird. So lässt sich die Funktion als Handler verwenden:

```
func (hf HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    hf(w, r)
}
```

Ein einfacher Server ist mit wenigen Zeilen programmiert (Listing 5). Das Package net/http/pprof enthält eine Webschnittstelle für Profilingdaten, das Package expvar einen Webzugriff auf freigegebene Debug-Informationen im JSON-Format.

Selten kommt eine Webanwendung mit nur einer Funktion aus und ebenso wenig möchte man für ihren gesamten Umfang alles von nur einem Handler aus selbst implementieren: URL und HTTP-Methode analysieren, in Methoden oder Instanzen anderer Typen verzweigen und Sicherheit gewährleisten. Mag dies bei kleinen APIs noch mit ein paar `switch`-Statements möglich sein, wird es bei größeren Systemen zunehmend unwartbar. Hier führt der Ansatz mit dem einfachen Interface zum Erfolg.

Listing 4: Nebenläufige State Machine

```
type Data struct { ... }
type State func(Data) (State, error)
type StateMachine struct {
    ctx  context.Context
    state State
    actions chan func()
}

func New(ctx context.Context) *StateMachine {
  sm := &StateMachine{
    ctx: ctx,
    actions: make(chan func()),
  }
  sm.state = sm.off
  go sm.loop()
  return sm
}

func (sm *StateMachine) loop() {
    for {
        select {
        case <-sm.ctx.Done():
          return
        case action := <-sm.actions:
          action()
        }
    }
}

func (sm *StateMachine) Process(d Data) error {
    var err error
    action := func() {
      var state State
      state, err = sm.state(d)
      if err == nil {
        sm.state = state
      }
    }
    select {
    case sm.actions <- action:
    case <-time.After(timeout):
      err = errors.New("timeout")
    }
    return err
}
```

Listing 5: Einfache Handler-Funktion

```
var h http.HandlerFunc = func(
    w http.ResponseWriter,
    r *http.Request,
) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("Hallo, liebe Leser!"))
}

http.ListenAndServe(":8080", h)
```

Listing 6: Multiplexer nutzen

```
mux := http.NewServeMux()

mux.Handle("/network/", NewNetworkHandler())
mux.Handle("/network/addresses/", NewAddressesHandler())
mux.Handle("/network/interfaces/", NewInterfacesHandler())
mux.HandleFunc("/", func(
    w http.ResponseWriter,
    r *http.Request,
) {
    w.WriteHeader(http.StatusNotFound)
})

http.ListenAndServe(":8080", mux)
```

Listing 7: Method Wrapper für die Verteilung nach HTTP-Methode

```
type GetHandler interface {
    ServeGet(w http.ResponseWriter, r *http.Request)
}

type PostHandler interface {
    ServePost(w http.ResponseWriter, r *http.Request)
}
...
type MethodWrapper struct {
    handler http.Handler
}

func (mw MethodWrapper) ServeHTTP(w http.ResponseWriter, ⤸
                                                    r *http.Request) {
    switch r.Method {
    case http.MethodGet:
      if h, ok := mw.handler.(GetHandler); ok {
        h.ServeGet(w, r)
        return
      }
    case http.MethodPost:
      if h, ok := mw.handler.(PostHandler); ok {
        h.ServePost(w, r)
        return
      }
    case ...:
      ...
    }
    mw.handler.ServeHTTP(w, r)
}
```

Ein erster Typ ist der Multiplexer `ServeMux` – ein `struct`, der das Interface `Handler` implementiert (Listing 6). Es lässt sich damit durch den Server nutzen. Die Instanz eines Multiplexers registriert verschiedene Handler für unterschiedliche Pfade. Der Typ kennt zwei Varianten: absolute Pfade für individuelle Ressourcen und Pfade für alle Ressourcen innerhalb dieser Pfade. Die verwandten Pfade dürfen sich überlappen: Es kann ein Handler für /network/ und einer für /network/addresses/ registriert werden. Dabei hat der längere Pfad höhere Priorität, wenn er mit dem des Requests übereinstimmt. Und `StripPrefix(prefix string, h Handler) Handler` zeigt, wie individuelle Wrapper zu erzeugen sind. Ein erhaltener Pfad verkürzt sich um das Präfix.

Mit Handlern arbeiten

Etwas Vergleichbares für die HTTP-Methoden ist zwar nicht in der Bibliothek enthalten, lässt sich aber schnell für eigene Zwecke implementieren. Zum einen mit registrierten individuellen Handlern (einfacher als `ServeMux`), zum anderen mit nur einem Handler, der unterschiedliche Interfaces pro Methode implementiert. Dazu sind zunächst für alle von einer RESTful API implementierten HTTP-Methoden individuelle Interfaces zu definieren. Mit einem Wrapper kann ein Handler dann die unterschiedlichen HTTP-Methoden getrennt verarbeiten. Dabei kommt die Technik der Type Assertion zum Einsatz. Sie prüft, welchen konkreten Typ ein Interface implementiert. Ein Test des Interface `Handler` findet heraus, ob es auch den passenden Handler zur HTTP-Methode implementiert. In solch einem Fall ruft das Programm die jeweilige Methode auf (Listing 7).

Die Methode `ServeHTTP()` des ursprünglichen Handlers ist ein Default-Fallback, etwa für eine Fehlermeldung. Der Multiplexer aus Listing 6 ist im Falle des Netzwerkbeispiels um ein Statement zu erweitern. Der Network Handler verfügt über die unterstützten Methoden `ServeGet()` und `ServePost()`:

```
mux.Handle(
    "/network/",
    NewMethodWrapper(NewNetworkHandler()),
)
```

Für den Austausch von Daten mit solchen APIs kommt häufig das JSON-Format zum Einsatz. Im Package io ist das Interface `Writer` definiert, das `http.ResponseWriter` implementiert. Die Encoder enthalten einen JSON-Encoder, der einen `io.Writer`, ein

Es gibt 10 Arten von Menschen.
Die, die iX lesen, und die anderen.
MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK

Listing 8: Definition eines Structs mit JSON-Tags

```
type MeteringPointValue struct {
    Namespace string        `json:"namespace"`
    ID        string        `json:"id"`
    Quantity  int           `json:"quantity"`
    Total     time.Duration `json:"total-duration"`
    Minimum   time.Duration `json:"minimum-duration"`
    Maximum   time.Duration `json:"maximum-duration"`
    Average   time.Duration `json:"average-duration"`
}
```

Listing 9: Daten im JSON-Format versenden

```
func (h MPVHandler) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request,
) {
    w.Header().Set(
      "Content-Type",
      "application/json; charset=ISO-8859-1",
    )
    enc := json.NewEncoder(w)
    mpv := ...
    err := enc.Encode(mpv)
    if err != nil {
      http.Error(
        w,
        err.Error(),
        http.StatusInternalServerError,
      )
    }
    w.WriteHeader(http.StatusOK)
}
```

`os.File` oder einen `bufio.Writer` erwartet. In diesem Anwendungsfall ist es der Response Writer. Die Strukturausgabe lässt sich in JSON durch Tags beeinflussen. Da das Encoding auf Reflection beruht, sind die gewünschten Felder zu exportieren. Der Austausch mit anderen APIs erfordert eine Kleinschreibung und andere Feldnamen (Listing 8). Damit ist es nun ein Leichtes, die Daten im JSON-Format zu versenden. Der Handler instanziiert eine Encoder-Instanz mit dem `ResponseWriter` als Ziel und setzt den entsprechenden `Content-Type`. Das Encoding kann starten (Listing 9).

Wichtig für RESTful APIs sind Pfade mit Identifiern in der Form /networks, /networks/<ID>, /networks/<ID>/interfaces und /networks/<ID>/interfaces/<KIND>. Ein hierfür notwendiger Multiplexer schaltet die Handler in Reihe und verteilt sie entsprechend den Pfaden. Sicherheit mit Techniken wie dem JSON Web Token ist auch als Wrapper möglich. Damit kann sich der eigentliche Handler am Ende auf die fachlichen Aspekte konzentrieren. Um die Infrastruktur kümmern sich die Wrapper und Multiplexer. Gleichzeitig wird so auch das Testen vereinfacht. Hierbei hilft das zum Umfang der Standardbibliothek gehörende Package httptest.

Werkzeuge einsetzen

Ein relevantes Tool, das Go mitliefert, arbeitet über Subcommands. Es trägt bezeichnenderweise den Namen go. Die von ihm gebotenen Unterkommandos reichen vom Übersetzen und Installieren über das unmittelbare Ausführen oder Testen bis zur Formatierung des Quelltexts und der Generierung der Dokumentation im HTML-Format. Weitere Befehle dienen der Analyse des Quelltexts und der sauberen Verwaltung von Modulen. Diese liegen als Packages inklusive einer Versionsnummer vor. Google hat zudem eine Caching- und Verwaltungsinfrastruktur aufgebaut (siehe ix.de/zj85).

Dazu kommen noch zwei für den Einsatz in der Systemprogrammierung spannende Eigenschaften. So entsteht beim Übersetzen ein Binary ohne Abhängigkeiten zu externen Bibliotheken oder Laufzeitumgebungen, was die Verteilung und Installation erleichtert. Gerade in Containerlandschaften werden so Scratch-Container minimaler Größe mit ausschließlich dem Binary als Service betrieben. Und dies wird auch durch die zweite Eigenschaft unterstützt: das Crosskompilieren. Binaries für andere Betriebssysteme und Rechnerarchitekturen lassen sich leicht vom eigenen System aus erzeugen.

Der Einsatzzweck von Go als Systemsprache stand bei Google schnell fest. Doch bis neue Software erschien, dauerte es. Die heute bekanntesten Produkte finden sich zum Großteil im Cloud-Native-Bereich wieder. Einer der bekanntesten großen Vertreter ist Kubernetes. Das Haupt-Repository umfasst mehr als 13 000 Go-Dateien, dazu kommen Komponenten wie client-go zur Kommunikation mit der API. Eine Kernkomponente von Kubernetes ist die verteilte Datenbank etcd für die Speicherung des aktuellen und des jeweils gewünschten neuen Zustands eines Clusters. Sie ist ein hochperformanter Key-Value-Speicher, sicher und schnell – und in Go entwickelt.

Innerhalb von Kubernetes werden Container betrieben. Auch die Containervirtualisierungssoftware Docker ist in Go entwickelt, nicht speziell für Kubernetes, sondern für den sicheren isolierten Betrieb mehrerer Anwendungen auf einem Server. Ebenso verhält es sich mit der Container-Runtime containerd, einer Alternative zu Docker. Ein weiteres DevOps-Tool für die deklarative Verteilung und Konfiguration von Software in verschiedenen Rechnerinfrastrukturen, On-Site wie auch in Clouds, ist Terraform. Es ist in Go entwickelt, ebenso wie auch Juju als Software für die Installation, Skalierung und Vernetzung von in Paketen konfigurierten Anwendungen in Cloud-Umgebungen. Es wird inzwischen auch als JAAS angeboten (Java Authentication and Authorization Service). Zum Verknüpfen, Verwalten und Absichern von Microservices bietet sich Istio an. Andere Tools sind die Datenbank InfluxDB, die Monitoringsoftware Prometheus sowie der schnelle und flexible Name Server CoreDNS. Inzwischen häufiger anzutreffen sind Websites, die das in Go entwickelte Programm Hugo generiert hat.

Fazit

Ein Blick auf die Seite Awesome Go und eine Sprachsuche auf GitHub zeigen, wie viele Projekte auf Go basieren. Eine Community von etwa 2 Millionen Entwickler*innen hat dazu beigetragen (siehe ix.de/zj85). Die Sprache belegte im April 2020 Platz 12 des TIOBE-Index.

Zehn Jahre nach der Ankündigung der Sprache ist der Plan der Go-Entwickler*innen aufgegangen: Sie hat die Domäne der Systementwicklung erobert. Mittlerweile entstehen auch Bibliotheken für die Entwicklung von Microservices. Und mit der zunehmenden Vereinfachung der Programmierung von Kubernetes-Operatoren lassen sich für die Zukunft vielleicht auch native Microservices in dieser Umgebung denken. (nb@ix.de)

Quellen

Listings zum Download, Go-Packages und ermittelte Go-Developer-Zahl: ix.de/zj85



Frank Müller

arbeitet als Solution Engineer beim Kubernetes- und Cloud-Dienstleister Loodse GmbH.

Minds Mastering Machines

Online Konferenz

DIE KONFERENZ FÜR
MACHINE LEARNING UND
KÜNSTLICHE INTELLIGENZ

16. JUNI 2020

- Boosting
- Jupyter Notebooks
- Neuronale Netze
- Modellqualität
- Textanalyse
- ML in Production

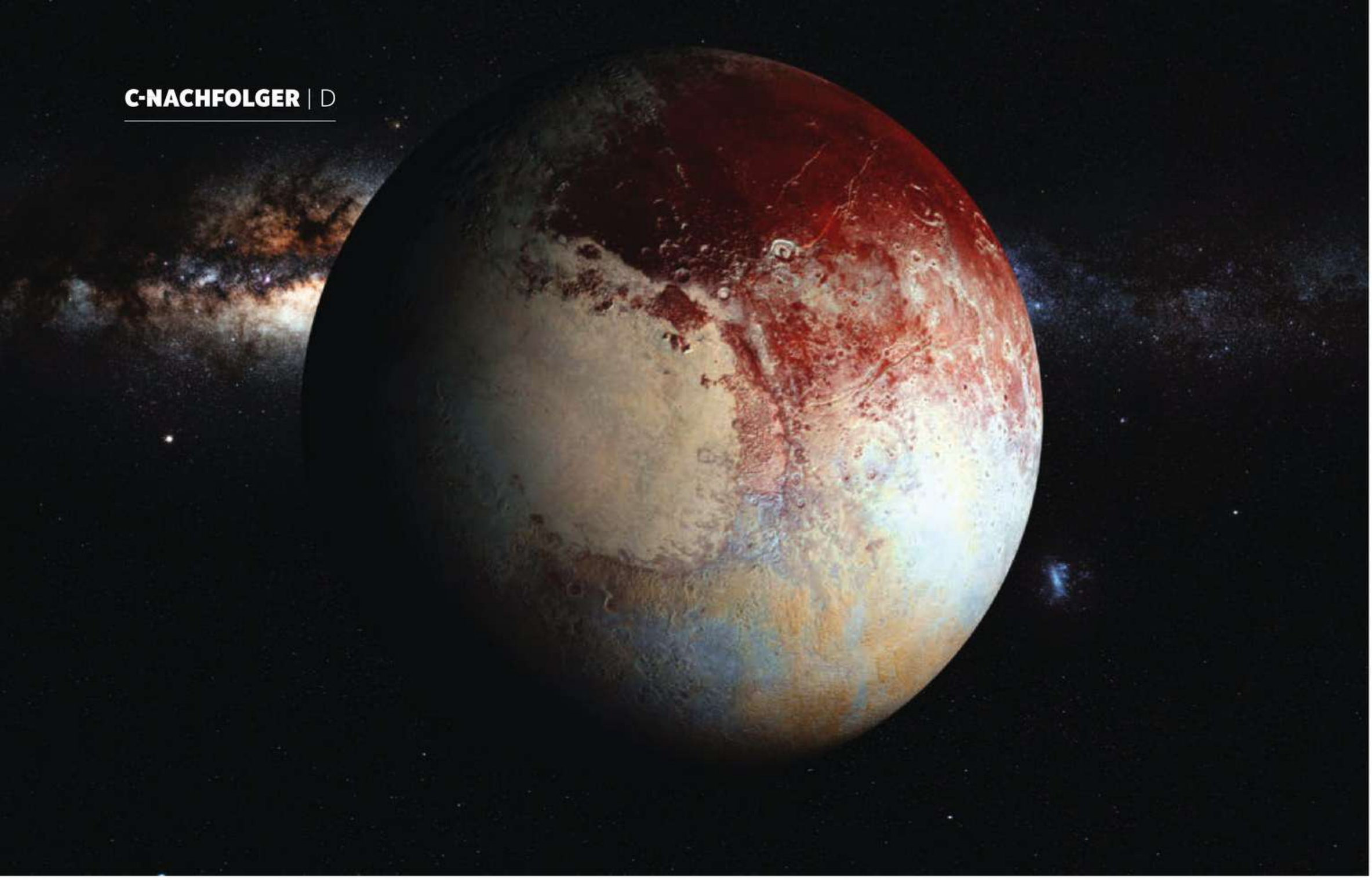
Ticketpreis wird bei der Teilnahme an der klassischen M3 im November angerechnet

WWW.M3-KONFERENZ.DE

Goldsponsor: NOVATEC

Silbersponsor: HMS analytical software

Veranstalter: iX, heise Developer, dpunkt.verlag



Programmiersprache D: die C-Alternative

Gemeinschaftliches Werkeln

Tam Hanna

Die Programmiersprache D möchte C- und C++-Entwickler*innen eine moderne Alternative bieten, die aktuelle Konzepte beispielsweise zur Multithreading-Programmierung mitbringt.

Wer schon länger programmiert, hat von der Sprache D mit Sicherheit schon einmal gehört. Die Arbeiten begannen bei Digital Mars im Jahre 1999, erstmals öffentlich vorgestellt wurde D in der Version 1.0 im Jahr 2001.

Zu diesem Zeitpunkt begann die Entwicklung der Version 2, die im Jahr 2011 präsentiert wurde und zur Vorgängerversion nicht kompatibel war. Die heutige Entwicklung – und auch alle Ausführungen in diesem Artikel – erfolgen ausschließlich in Bezug auf D2. Eine kompakte Liste aller Unterschiede zwischen D1 und D2 gibt es auf der Projektseite (siehe ix.de/zw3h).

Modernisiertes Ökosystem

Die offene Vorgehensweise im Bereich der Entwicklung zeigt sich dadurch, dass es drei weitverbreitete Compiler gibt. Die offizielle Version ist das Programm DMD, das Akronym steht ausgeschrieben für den Namen Digital Mars D Compiler. Daneben existieren mit GDC und LDC zwei weitere Compiler, die die GCC- und die LLVM-Infrastruktur benutzen. Zu guter Letzt liegt auch noch ein eher experimenteller Compiler vor, der für die .NET-Ausführungsumgebung vorgesehenes CIL ausgibt.

iX-TRACT

- Die erste Version der objektorientierten und imperativen Sprache D erschien 2001.
- Mit DMD, GDC und LDC liegen drei weitverbreitete Compiler vor. Damit steht D auf den meisten Zielplattformen zur Verfügung.
- Dank der regen Community ist D in vielen IDEs verfügbar.

Dank dieser breiten Basis steht D für so gut wie alle Zielplattformen zur Verfügung. Im Embedded-Bereich tätige Entwickler*innen müssen allerdings bedenken, dass die Sprachspezifikation die Verwendung von 16-Bit-Zielsystemen untersagt: Kerne wie MSP430 von TI sind also nicht für D geeignet.

Ein weiterer angenehmer Nebeneffekt der breiten Aufstellung ist, dass D mittlerweile in so gut wie allen IDEs Berücksichtigung findet. Die Plug-ins für IntelliJ und Visual Studio erweisen sich dabei als besonders aktiv; mit Dexed gibt es zudem eine speziell für das D-Ökosystem geschaffene Arbeitsumgebung. Das Produkt steht derzeit allerdings nur für Linux als vorkompilierte Binärdatei zur Verfügung.

Der Einstieg in D

Als erste Aktion sollten D-Interessierte diese Arbeitsumgebung herunterladen (siehe ix.de/zw3h). Wer mit einer anderen IDE arbeiten will, kann das auf der Projektseite bereitstehende Plug-in-Portal aufsuchen und ein passendes Tool wählen. Die Umsetzung der nachfolgenden Beispiele erfolgt auf einer Workstation mit Ubuntu 18.04.

Die D-Arbeitsumgebung installiert man mit den folgenden Kommandos:

```
chmod +x dexed.3.9.3.linux64.setup
./dexed.3.9.3.linux64.setup
```

Nach dem Start präsentiert das Programm einen Assistenten, der über das Vorhandensein oder das Fehlen von Werkzeugen informiert (Abbildung 1).

Die D-Webseite stellt die fehlenden Komponenten bereit, wobei der Link „Download DEB File“ anzuklicken ist. Das File lässt sich im Softwarecenter öffnen und wie jede andere Applikation installieren. Lohn der Mühen ist, dass die IDE ab dem nächsten Mal reibungslos startet. Die auf die Klasse `TControl` weisenden Fehler im unten eingeblendeten Messages-Tab sind für die Arbeit mit Dexed nicht weiter relevant. Mit einem Klick auf „File“ wählt man das Menü mit den diversen Optionen zum Erzeugen neuer Dateien. Die Option „New Runnable Module“ generiert ein ausführbares neues Modul.

Die erste Aktion besteht darin, den Code im Tab „runnable“ anzupassen:

```
module runnable;

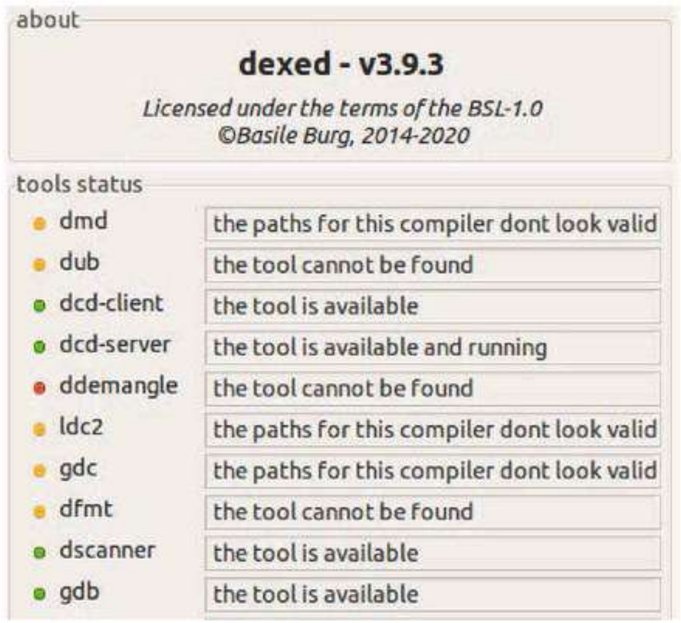
import std.stdio;
import std.complex;
```

Als Erstes fällt hierbei die Deklaration eines Moduls auf. D orientiert sich bei der Unterteilung des Programmcodes an Java. Ein Modul umfasst sämtlichen in einer Datei enthaltenen Code, der analog zum C-Compiler mit Sichtbarkeitssicherungen ausgestattet wird. Zugriffe aus anderen Modulen heraus setzen die Verwendung spezieller Operatoren voraus.

So sehr Präprozessormagie erfahrene Entwickler*innen erfreut, so sehr ist sie für Quereinsteiger schwierig zu durchschauen. Das D-Entwicklerteam begegnet dieser Situation durch das Eliminieren des Präprozessors. Den Import von Headern bewirkt das dedizierte `import`-Statement.

Als Nächstes ist der Einsprungpunkt erforderlich, der in Form einer `main`-Methode vorliegt:

```
void main(string[] args)
{
Complex!(double) myCplxDouble;
double rT, iT;
```



Nach dem Installieren von Dexed sind noch einige Handgriffe erforderlich (Abb. 1).

Listing 1: Informationen in den Datentyp packen

```
myCplxDouble.re = rT;
myCplxDouble.im = iT;
writeln("C lautet " , myCplxDouble);
writeln("Ihre Länge ist ", myCplxDouble.sizeof);
readf(" %s", &iT);
}
```

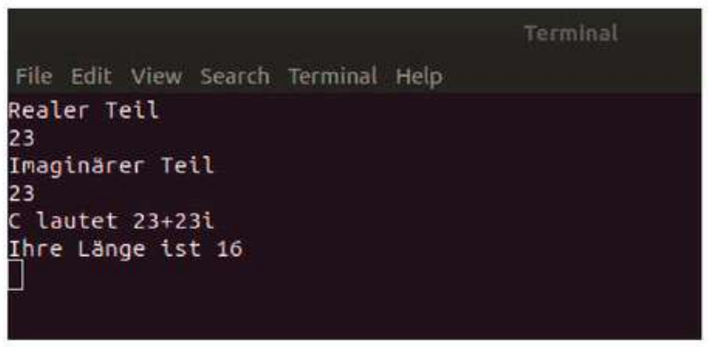
Danach erstellt man eine Templateklasse. D ist eine komplett statisch typisierte Sprache, die aus dem ingenieurwissenschaftlichen Bereich kommende Entwickler*innen mit Komfortdatentypen für die Verarbeitung komplexer Zahlen unterstützt. Im Laufe der letzten Jahre wurden diese allerdings mehr und mehr durch das Modul std.complex ersetzt, das einen Templateparameter entgegennimmt. Ausrufezeichen beschreiben in D den Typ, der zur Instanziierung verwendet wird – eine auf den ersten Blick eigenwillige Besonderheit der Sprache.

Der Rest des Codes ist nach Schema F aufgebaut. Interessant sind allerdings die Leerzeichen vor den beiden Formatstrings:

```
writeln("Realer Teil");
readf(" %s", &rT);
writeln("Imaginärer Teil");
readf(" %s", &iT);
```

Der in D implementierte Eingabeparser ist wesentlich strenger als der von C. Das Quittieren der beiden Zahlen durch Drücken der Entertaste würde aus Sicht des Programms die Eingabe `Zahl1<return>Zahl2<return>` ergeben. Nach dem Einlesen der ersten Zahl ist deshalb ein Whitespace im Format `String` erforderlich, der sich um die Beseitigung des Line Feed kümmert. Die in





Die neue Art der komplexen Zahlenverarbeitung funktioniert problemlos (Abb. 2).

Listing 2: Konstruktor mit dem Schlüsselwort super aufrufen

```
class A { this() { ... } }
class B : A
{
    this(int x)
    {
        ...
        super();  // call base constructor
        ...
    }
}
```

Listing 1 gezeigten Befehle sollen die angelieferten Informationen in den komplexen Datentyp schreiben, um diesen zu guter Letzt in der Konsole auszugeben.

Dexed bietet verschiedene Arten der Programmausführung an. Die bequemste Methode ist „Compile File and Run Outside". Sie lässt sich durch das Drücken der Taste F6 auslösen und startet ein neues Terminalfenster, in dem das Programm läuft (Abbildung 2). Unter Ubuntu kommt es dabei manchmal zu Hakeleien: Das Dexed-Hauptfenster überdeckt in diesem Fall das Terminalfenster mit dem Programm.

Aufmerksame Beobachter bemerken, dass die Beschaffung der Speichergröße der Variablen durch ein Attribut erfolgt. D unterstützt Entwickler*innen bei der Reflexion von Datentypen auf mehrere Arten, um die Anpassung von Algorithmen auf verschiedene Maschinentypen zu ermöglichen. Ein weiterer Klassiker sind die Attribute `max` und `min`, die nach dem folgenden Schema über die Grenzwerte von Variablen informieren:

```
char.max
char.min
ulong.max
double.min
```

Als Kernighan und Ritchie ihren Klassiker zur Programmiersprache C schrieben, steckten Parser und andere Elemente der symbolischen Verarbeitung noch in ihren Kinderschuhen. Zudem stand zur Ausführung von Compilern wenig Rechenleistung zur Verfügung. D bietet eine straffere Syntax, die das Entwickeln erleichtert.

Komfortables Programmieren

Die erste Neuerung ist die Art, wie sich abgeleitete Konstruktoren aufrufen lassen. In D erfolgt dies über das von Java bekannte Schlüsselwort `super`, das allerdings an beliebiger Stelle der hier prinzipiell mit `this` eingeleiteten Konstruktoren stehen darf (Listing 2).

Ein weiterer Bereich, in dem höhere Rechenleistung bequemes Arbeiten erlaubt, sind Arrays. Die Zuweisung eines Werts an den Namen des Arrays bewirkt in D, dass alle Elemente des Feldes denselben konstanten Wert annehmen. Entwickler*innen sparen sich so die in C und C++ notwendige `for`-Schleife:

```
void main() {
int[3] myA;
myA=6;
writeln(myA[0], " ", myA[1]);
}
```

Die Größe eines Arrays ist in D nur dann zur Kompilierzeit festgelegt, wenn man sie als Teil des Datentyps anschreibt. Wer sich stattdessen für die in Listing 3 gezeigte Vorgehensweise entscheidet, kann die Größe des Feldes über das `length`-Attribut zur Laufzeit anpassen. Zu beachten ist allerdings, dass Vergrößerungen eines Arrays im Allgemeinen zu seiner Neuallokation führen.

Strukturen weisen in D den aus C++ erweiterten Funktionsumfang auf und können – für C-Programmierer*innen ist dies neu – auch nach dem in Listing 4 beschriebenen Schema Logik aufnehmen.

Der Unterschied zwischen einer Klasse und einer Struktur besteht in D vor allem darin, dass eine Struktur prinzipiell einen Werttyp darstellt und somit von der Vererbung ausgeschlossen ist. Apropos Vererbung: Die Realisierung mehrstufiger Klassenhierarchien stellt Compilerhersteller laut der Dokumentation des D-Standards seit Jahren vor Probleme. Das D-Entwicklerteam begegnet diesen mit einer Radikalmaßnahme: Mehrstufige Vererbung ist in der Sprachspezifikation nicht vorgesehen und wird von den Compilerherstellern dementsprechend auch nicht angeboten.

Als Alternativvorschlag offeriert der Sprachstandard die Verwendung von Interfaces, die sich nach dem Schema in Listing 5 in eine Klasse einbinden lassen. Eine Klasse darf zwar immer nur von einer anderen Klasse erben, im Bereich der zulässigen Interfaces sind allerdings keinerlei Grenzen vorgesehen.

Der D-Sprachstandard bietet noch weitere interessante Aspekte, die ein von Ali Çehreli verfasstes E-Book im Detail erläutert (siehe ix.de/zw3h). Es stellt einen exzellenten Einstieg in die D-Syntax dar und ist für angehende D-Programmierer quasi Pflichtlektüre.

Mit D in die Parallelisierung

Wer bei der Programmierung auf die Leistung achtet, wird über kurz oder lang Parallelisierung der Ausführung seiner Programme nutzen müssen. Der D-Sprachstandard trägt dieser Situation mit einer Gruppe von Funktionen Rechnung. Das einfachste Beispiel ist dabei die Parallelisierung von `foreach`-Schleifen, die voneinander komplett unabhängige Tasks ausführen (Listing 6).

Listing 3: Größe eines Arrays festlegen

```
void main() {
int[] myA=new int[3];
myA.length = myA.length + 1;
myA[myA.length - 1] = 2;
writeln(myA[0], " ", myA[3]);
}
```

Listing 4: Die Aufnahme von Logik

```
struct S1 {
   int myF;
   this(int _m){ myF = _m; }
   void echoThis(){writeln("Struct enthält ", myF);}
}
```

Listing 5: Interfaces einbinden

```
interface MyIF {
    void echoT();
}

class CErb:C1, MyIF {
. . .
final override void echoThis() {
   write("CErb sagt HALLO ");
   super.echoT();
  }
}
```

Listing 6: foreach-Schleifen parallelisieren

```
import std.parallelism;
. . .

void main() {
    auto s = [ Work(1), Work(2), Work(3), Work(4) ];

    foreach (Work; parallel(s)) {
        s.warte2Sec();
    }
}
```

Listing 7: Gleichzeitiges Einlesen einer Datei

```
import std.file;

void main()
{
    auto file1Task = task!read("foo.txt");
    file1Task.executeInNewThread();
    auto file2Data = read("bar.txt");
    auto file1Data = file1Task.yieldForce;
}
```

Wer das Modul `std.parallelism` importiert und das Kommando `parallel` um das Array stellt, bekommt von D automatisiert einen Executor zugewiesen. Von Haus aus erzeugt er dabei übrigens für jeden Kern des Prozessors einen Thread, weshalb eine Vierkernmaschine dieses Programm in 2 Sekunden ausführen würde.

Das zweite syntaktische Zuckerstück ist die Klasse `Task`, die in der Welt von D ein Taskobjekt ins Leben ruft. Dahinter verbirgt sich eine Hilfsstruktur, die eine auszuführende Payload zwecks einfacherem Dispatching in Threads oder in Richtung eines Threadpools verpackt. Ein schönes Beispiel hierfür ist das Snippet, das am Hauptthread und in einem Nebenthread gleichzeitig eine Datei einliest (Listing 7).

`std.parallelism` bietet Dutzende von Hilfsmethoden an, die das gezielte Ausliefern von Threads und das Einsammeln der von ihnen generierten Ergebnisse erleichtern (siehe ix.de/zw3h).

In der Praxis lässt es sich nicht vermeiden, Informationen zwischen gleichzeitig laufenden Threads auszutauschen, um eine Aufgabe zu erledigen. Die bevorzugte Vorgehensweise dafür ist Message Passing. Die einzelnen Frames legen sich dabei Postkästen an, die auf eingehende Nachrichten warten (Listing 8).

Das hier gezeigte Programm benutzt im ersten Schritt die Funktion `spawn`, um einen neuen Thread mit einer hier nicht abgedruckten Payload ins Leben zu rufen. Anhand des zurückgegebenen Objekts `Tid` können dann durch Verwendung von `send` Nachrichten an den Storage geschickt werden, die hier beispielsweise zur Beendigung der Programmausführung führen würden. Der Arbeiter darf an den Hauptthread ebenfalls eine Nachricht senden. Das Entgegennehmen erfolgt über das Statement `receiveOnly`.

Muss man aus irgendeinem Grund von Hand synchronisieren, so gibt es nach dem folgenden Schema aufgebaute Atomic Operations. Sie lassen sich immer als Ganzes durchführen. Während der Abarbeitung dieser Inkrementierung sind Kontext-Switches aus logischer Sicht ausgeschlossen:

Listing 8: Postkästen warten auf Nachrichten

```
void main() {
    Tid worker = spawn(&workerFunc);
    foreach (value; 1 .. 5) {
        worker.send(value);
        double result = receiveOnly!double();
        writefln("sent: %s, received: %s", value, result);
    }
    worker.send(-1);
}
```

Listing 9: Synchronisationsprimitivum

```
foreach (i; 0 .. 10_000) {
    synchronized {
        int temp = *b;
        *b = *a;
        *a = temp;
    }
}
```

```
void incrementer(shared(int) * value) {
    foreach (i; 0 .. count) {
        atomicOp!"+="(*value, 1);
    }
}
```

Synchronisationsprimitiva sind auch sonst Teil der Sprachspezifikation. Ein gutes Beispiel ist das Snippet, das eine Mutex realisiert. Den innerhalb des Blocks befindlichen Code darf immer nur ein Thread tangieren (Listing 9). D isoliert die in einzelnen Threads lebenden Elemente normalerweise voneinander. Möchte man klassisches Sharing durchführen, so bietet sich die Verwendung des Schlüsselworts `shared` an:

```
shared(int)* p = new int;
```

Dabei gilt allerdings, dass Shared-Elemente nur andere geteilte Elemente aufnehmen dürfen. D beherrscht außerdem das als Green Threads oder Fibers bezeichnete Konzept zur Parallelverarbeitung:

```
import core.thread;
    auto fiber = new Fiber(&fiberFunction);
```

Dabei handelt es sich um eine Art Thread, zwischen dem extrem schnell hin- und hergeschaltet werden kann.

Fazit

Die Programmiersprache D ist ein angenehmes Werkzeug, das Entwickler*innen die Teilhabe an Fortschritten im Bereich der Informatik ermöglicht, ohne zu viele sonstige Neuerungen abzuladen. Insbesondere die Interaktion innerhalb der Community erinnert in vielerlei Hinsicht an die goldene Zeit der Open-Source-Entwicklung. (nb@ix.de)

Quellen

Listings, Hilfsmethoden, Zusatzinfos: ix.de/zw3h

Smarter googeln

Wer Google nach Informationen zum D-Ökosystem befragt, sollte statt D den Suchstring Dlang verwenden. Das Entwicklerteam greift auf diesen syntaktischen Kniff zurück, weil das Kürzel D zu allgemein ist.



Tam Hanna

ist Autor, Trainer und Berater mit den Schwerpunkten Webentwicklung und Webtechnologien. Er leitet die Firma Tamoggemon Holding.

Typsicherheit mit Swift

Vielseitig angelegt

Thomas Sillmann

Swift ist eine hauptsächlich von Apple-Developern genutzte Sprache. Seit 2015 ist sie Open Source und damit auch unter Windows, Linux oder im Web nutzbar.

Die Programmiersprache Swift gehörte zu den Highlights der WWDC im Jahr 2014. Dort stellte Apple sie zum ersten Mal der Weltöffentlichkeit vor und bot damit erstmals eine Alternative zu Objective-C an. Mit Swift verfolgte Apple ganz konkrete Ziele: Nicht nur sollte die Programmiersprache modern sein, sondern auch in puncto Sicherheit und Performance überzeugen. Überdies sollte ein einfacher Zugang sie von Beginn an auszeichnen. Heute, knapp sechs Jahre nach der ersten Präsentation, hat die Programmiersprache Version 5 erreicht und bereits viele große Entwicklungssprünge hinter sich.

Ein Merkmal, das Swift zu einer modernen, sicheren und nutzerfreundlichen Sprache macht, ist die starke Typsicherheit. So sind alle Variablen, Konstanten, Funktionsparameter und sonstigen Elemente einem eindeutigen Typ zuzuordnen. Aber es gibt auch Ausnahmen von dieser Regel – dazu später mehr.

Erstellt man eine Variable oder Konstante, weist man einer solchen entweder selbst explizit den gewünschten Typ zu oder lässt ihn automatisch von Swift ermitteln. Ersteres bezeichnet man als Type Annotation, Letzteres als Type Inference. Ein Beispiel dazu zeigt das folgende Listing. Darin werden zwei Konstanten mithilfe des Schlüsselworts `let` erzeugt, die beide dem Typ `Int` entsprechen. Im Fall von `firstInt` erfolgt die konkrete Angabe des Typs, bei `secondInt` ermittelt ihn Swift automatisch anhand des zugewiesenen Werts.

```
// Type Annotation
let firstInt: Int = 19

// Type Inference
let secondInt = 99
```

Die explizite Typzuweisung ist in Situationen wichtig, in denen der von Swift automatisch abgeleitete Typ nicht dem entspricht, den man verwenden möchte. Listing 1 demonstriert das mithilfe einer Additionsmethode, die zwei Werte vom Typ `Double` als Parameter erwartet (alle Listings siehe ix.de/zt7b). Der `firstValue` zugewiesene Wert kommt einer Fließkommazahl gleich, weshalb Swift ihr den Typ `Double` zuweist. `secondValue` hingegen deklariert Swift als `Int`, obwohl in dem gezeigten Szenario ebenfalls `Double` erforderlich wäre. Dazu hätte `secondValue` explizit mittels Type Annotation als `Double` gekennzeichnet werden müssen.

Noch deutlicher wird die starke Typsicherheit bei der Arbeit mit Arrays. Ein Array kann nur Elemente eines ganz konkreten Typs aufnehmen, also beispielsweise nur Werte vom Typ `Int` oder `String`, aber keine Mischung von beiden (Listing 2).

Dank der starken Typsicherheit lassen sich Logikfehler durch den Compiler verhindern. So ist sichergestellt, dass das gezeigte `numbers`-Array zu jeder Zeit nur über Zahlenwerte verfügt. Auch neu eingefügte Elemente müssen dem Typ `Int` entsprechen, andernfalls macht der Compiler auf diesen Fehler aufmerksam.

Wie eingangs angedeutet, gibt es eine Ausnahme von dieser Regel, und die liegt in den Typen `Any` und `AnyObject`. Sie dienen als eine Art Platzhalter für beliebige Typen und erlauben es, aus der recht starren Typsicherheit von Swift auszubrechen. Ein Developer muss sie aber setzen, Swift leitet sie nicht mit Typinfe-

iX-TRACT

- Die von Apple 2014 veröffentlichte Multiparadigmensprache Swift war eines des Highlights der WWDC 2014 und sollte moderner und sicherer sein als andere Sprachen.
- Version 5 enthält ABI-Stabilität. Damit lässt sich in Swift entwickelter Code auch in künftigen Versionen ausführen.
- Merkmale der Sprache sind eine starke Typsicherheit, flexiblere Enums und Structures sowie Optionals und Generics.

Listing 1: Fehler durch falsche Typzuweisung

```
let firstValue = 19.99
let secondValue = 99

func addition(firstValue: Double, secondValue: Double) -> Double {
    firstValue + secondValue
}

addition(firstValue: firstValue, secondValue: secondValue)
// Fehler: secondValue ist vom Typ 'Int', nicht 'Double'.
```

Listing 2: Typsicherheit von Arrays

```
var numbers = [19, 99, 118] // Integer-Array
var names = ["Thomas", "Michaela", "Mark"] // String-Array

var mixedValues = [19, "Thomas", true]
// Fehler: Elemente eines Arrays müssen demselben Typ entsprechen
```

Listing 3: Eine Enumeration implementieren

```
enum Direction: String {
    case north = "Norden"
    case east = "Osten"
    case south = "Süden"
    case west = "Westen"

    func printInfo() {
      print("Gehe nach \(self.rawValue).")
    }
}

let myDirection = Direction.north
myDirection.printInfo()
// Gehe nach Norden.
```

renz ab. Möchte man also das gezeigte `mixedValues`-Array erfolgreich umsetzen, deklariert man es explizit als Element vom Typ `[Any]`:

```
var mixedValues: [Any] = [19, "Thomas", true]
```

`Any` bezeichnet hierbei eine Instanz jedes beliebigen Typs, ganz gleich ob diese Typen als Enumeration, Structure oder Klasse definiert wurden. `AnyObject` schränkt das ein wenig ein, dieser Typ lässt sich nur im Zusammenspiel mit Klassen einsetzen.

Fortgeschrittene Möglichkeiten

Enumerations und Structures sind vielseitiger als in anderen Programmiersprachen. Enumerations lassen sich ohne Probleme um Computed Properties oder Instanzmethoden ergänzen, um darin zusätzliche Logik unterzubringen. Auch sind die potenziell den Cases zugeordneten Werte nicht auf Integer beschränkt. Sie können auch Strings, Characters oder Fließkommazahlen entsprechen. Wichtig ist, dass alle Cases einen Wert desselben Typs zurückliefern (Stichwort Typsicherheit).

Listing 3 zeigt eine Enumeration, deren Cases ein passender String zugeordnet ist. Zusätzlich verfügt die Enumeration über eine ergänzende Instanzmethode.

Daneben lassen sich die verschiedenen Cases einer Enumeration mit sogenannten Associated Values versehen. Es handelt sich dabei um ergänzende Informationen eines beliebigen Typs, die man beim Erstellen eines Case zusätzlich mit angibt.

Im Beispiel in Listing 4 deklarieren zwei Cases (Kreditkarte und Rechnung) eine Enumeration namens `PaymentMethod`. Die Cases sind mit zusätzlichen Informationen ausgestattet. Bei der Kreditkarte ist das ein String, der die Kreditkartennummer enthalten soll. Die Rechnung erwartet die vollständige Anschrift.

Listing 4: Enumeration mit Associated Values

```
enum PaymentMethod {
    case creditCard(String)
    case invoice(String, String, String)
}
let myPaymentMethod = PaymentMethod.invoice("Max Mustermann", ↵
                                "Musterstraße 19", "12345 Musterstadt")
```

Listing 5: Associated Values einer Enumeration abfragen

```
switch myPaymentMethod {
case .creditCard(let creditCardNumber):
    print("Kreditkartenummer: \(creditCardNumber)")
case .invoice(let name, let street, let city):
    print("\(name), \(street), \(city)")
}
// Max Mustermann, Musterstraße 19, 12345 Musterstadt
```

Listing 6: Unterscheidung zwischen Value und Reference Types

```
// Value Types
struct SomeStruct {
    var value = 0
}
var firstStruct = SomeStruct()
var secondStruct = firstStruct
firstStruct.value = 19
print("firstStruct.value = \(firstStruct.value)")
print("secondStruct.value = \(secondStruct.value)")
// firstStruct.value = 19
// secondStruct.value = 0

// Reference Types
class SomeClass {
    var value = 0
}
var firstClass = SomeClass()
var secondClass = firstClass
firstClass.value = 99
print("firstClass.value = \(firstClass.value)")
print("secondClass.value = \(secondClass.value)")
// firstClass.value = 99
// secondClass.value = 99
```

Auslesen lassen sich solche Associated Values mithilfe von Switch-Abfragen (Listing 5).

Structures ähneln Klassen: Über beide lassen sich Properties, Methoden und Initializer definieren. Des Weiteren können sie konform zu Protokollen sein und mit Extensions erweitert werden. Unterschiede gibt es im Umgang mit der Speicherung der Daten, die Structures und Klassen abbilden. So erschaffen Structures sogenannte Value Types, während Instanzen auf Basis von Klassen Reference Types entsprechen. Weist man die Instanz einer Structure einer anderen Variablen zu, wird diese Instanz kopiert, während bei Klassen jede Variable ein und dieselbe Instanz referenziert. Das Beispiel in Listing 6 verdeutlicht das. Die beiden Variablen `firstClass` und `secondClass` verweisen – weil sie auf einer Klasse basieren – auf dieselbe Instanz. Ändert man eine Eigenschaft dieser Instanz über eine der Variablen, gilt die Änderung auch automatisch für die andere. Bei Value Types tritt dieses Verhalten nicht auf, da deren Instanzen bei Zuweisung zu einer Variablen oder Konstanten immer kopiert werden.

Darüber hinaus besitzen Klassen weitere Eigenschaften. Dazu gehört die Möglichkeit der Vererbung, die bei Structures nicht gegeben ist. Auch lassen sich Techniken wie Type Casting und Deinitializer nur im Zusammenspiel mit Klassen einsetzen.

Ohne Besitz

Ein weiteres Sprachmerkmal sind die sogenannten Optionals. Um deren Funktionalität zu verstehen, muss man wissen, dass

standardmäßig jedes Element – egal ob Variable, Konstante, Methodenparameter oder -rückgabewert – über einen validen Wert verfügen muss. Die aus Objective-C bekannte Zuweisung von `nil`, die einen Verweis auf eine Instanz entfernt, ist unter Swift nicht ohne Weiteres möglich. Sie funktioniert nur, wenn das jeweilige Element als Optional deklariert ist. Denn nur Optionals haben die Möglichkeit, keinen Wert zu besitzen und somit auf `nil` zu verweisen. Nicht-Optionals besitzen immer einen validen Wert. Um ein Element als Optional zu kennzeichnen, setzt man ein `?` nach der Typangabe (Listing 7).

Die Property `address` der Klasse `Person` ist im Gegensatz zur Property `name` als Optional deklariert. `name` muss also immer einen validen Wert zurückliefern, während `address` möglicherweise auch `nil` entsprechen kann.

Aufgrund dieser Regelung ist bei der Arbeit mit Optionals besondere Vorsicht geboten. In der Regel ist beim Zugriff auf ein solches zunächst zu prüfen, ob tatsächlich ein Wert vorliegt. Ist das der Fall, liest man diesen Wert aus dem Optional aus, indem man beim Zugriff auf das Optional ein ! setzt (Listing 8).

Dieses Vorgehen lässt sich mit einer angepassten `if`-Abfrage abkürzen. Hierbei weist man den potenziellen Wert eines Optionals einer temporären Konstante zu. Das Programm führt die `if`-Abfrage nur dann aus, wenn das Optional einen Wert besitzt. Er kann in diesem Fall direkt über die temporäre Konstante ausgelesen werden. Das macht einen Zugriff auf das Optional mit ! überflüssig (Listing 9). Besitzt das Optional keinen Wert, erfolgt auch keine Ausführung der Befehle innerhalb der `if`-Abfrage.

Im Zusammenhang mit Optionals deckt Swift noch ein weiteres Szenario ab. Hierbei geht es um Eigenschaften, die prinzipiell keinen Wert besitzen können, in der Regel aber über einen solchen verfügen. Oder es handelt sich um Eigenschaften, die ihren Wert nicht mehr verlieren, sobald er einmal zugewiesen wurde. In den meisten Fällen ist davon auszugehen, dass diese Eigenschaften einen validen Wert besitzen. Das lässt ein ständiges Prüfen gegen `nil` vor dem Zugriff auf den Wert aufwendig erscheinen.

Abhilfe schaffen hier die sogenannten Implicitly Unwrapped Optionals. Der Zugriff auf die Werte solcher Optionals erfolgt direkt, ohne sie mit ! zu entpacken. Um ein Optional als Implicitly Unwrapped Optional zu deklarieren, fügt man bei der Angabe des Typs statt eines `?` ein ! hinzu. Der Wert lässt sich ohne weitere Anpassungen verwenden:

```
let implicitlyUnwrappedOptional: Int! = 19
let concreteValue: Int = implicitlyUnwrappedOptional
```

Aber Vorsicht: Implicitly Unwrapped Optionals sind – wie der Name schon sagt – im Kern noch immer Optionals. Lediglich der Umgang damit innerhalb eines Projekts ist ein anderer. Doch so kann es sein, dass ein solches keinen Wert besitzt und `nil` entspricht. Greift man dann direkt auf dessen Wert zu und steht kein Ergebnis bereit, kommt es zur Laufzeit zum Absturz der Anwendung.

Daher gilt, im Zweifel auch Implicitly Unwrapped Optionals zunächst auf einen konkreten Wert zu prüfen. Dazu lassen sich dieselben Techniken wie für „herkömmliche“ Optionals einsetzen.

Tauschgeschäfte

Generics erlauben es, generischen und leicht wiederzuverwendenden Code zu schreiben. Das stellt aufgrund der starken Typsicherheit der Sprache eine wichtige Ergänzung dar. Mithilfe von Generics legt man bestimmte Typen, die innerhalb einer Klasse, Structure oder Methode zum Einsatz kommen, nicht explizit fest. Man definiert stattdessen Eigenschaften, zu denen sie konform sein müssen – wie die Implementierung eines bestimmten Protokolls.

Ein typisches Problem, das sich mit Generics lösen lässt, ist in Listing 10 skizziert. Darin finden sich zwei Funktionen namens `swapTwoInts(_:_:)` und `swapTwoStrings(_:_:)`. Beide tauschen die Werte zwischen den Parametern, die man den Funktionen übergibt.

Bereits bei der Implementierung zeigt sich, dass beide Funktionen dieselben Befehle durchführen, lediglich die Parameter und die verwendete Konstante tragen jeweils einen anderen Na-

Listing 7: Deklaration von Optionals

```
class Person {
    var name: String
    var address: String?

    init(name: String, address: String? = nil) {
      self.name = name
      self.address = address
    }
}
```

Listing 8: Prüfen von Optionals gegen nil

```
let somePerson = Person(name: "Max Mustermann", address:
                                                  "Musterstraße 19")
if somePerson.address != nil {
    print("Adresse: \(somePerson.address!)")
}
// Adresse: Musterstraße 19
```

Listing 9: Prüfen von Optionals mit der Syntax if let

```
let somePerson = Person(name: "Max Mustermann", address:
                                                  "Musterstraße 19")
if let address = somePerson.address {
    print("Adresse: \(address)")
}
// Adresse: Musterstraße 19
```

Listing 10: Generische Funktionen mit statischen Typangaben

```
func swapTwoInts(_ firstInt: inout Int, _ secondInt: inout Int) {
    let temporaryFirstInt = firstInt
    firstInt = secondInt
    secondInt = temporaryFirstInt
}

func swapTwoStrings(_ firstString: inout String, _ secondString:
                                                  inout String) {
    let temporaryFirstString = firstString
    firstString = secondString
    secondString = temporaryFirstString
}
```

Listing 11: Tauschfunktion als Generic umsetzen

```
func swapTwoValues<T>(_ firstValue: inout T, _ secondValue: inout T) {
    let temporaryFirstValue = firstValue
    firstValue = secondValue
    secondValue = temporaryFirstValue
}

var firstInt = 19
var secondInt = 99
swapTwoValues(&firstInt, &secondInt)

var firstString = "First"
var secondString = "Second"
swapTwoValues(&firstString, &secondString)
```

men. Dennoch ist die Trennung wichtig, da `swapTwoInts(_:_:)` aufgrund der Typsicherheit nur Integer tauschen kann und `swapTwoStrings(_:_:)` nur den Umgang mit Strings versteht.

Zur Optimierung eines solchen Falls bieten sich Generics an. Denn letztlich gibt es in beiden Funktionen nur eine Bedingung: Der erste und der zweite Parameter müssen demselben Typ entsprechen. Dann ist es möglich, eine Tauschfunktion für jeden verfügbaren Typ umzusetzen.

Die Deklaration von Generics erfolgt innerhalb spitzer Klammern. Pro Typ oder Funktion lassen sich beliebig viele solcher Generics definieren, abhängig davon, wie viele unterschiedliche generische Typen man benötigt. Die Bezeichner für Generics sind frei wählbar. So lässt sich die Tauschfunktion optimieren (Listing 11). Statt eines konkreten Typs versieht man die Funktion mit einem Generic (in diesem Fall trägt es den Namen `T`) und nutzt es als Typ für die beiden Parameter. Welchem konkreten Typ `T` entspricht, entscheidet sich beim Aufruf der Funktion, genauer gesagt bei der Übergabe des ersten Parameters. Da dieser `T` entspricht, wird ab diesem Zeitpunkt der Typ des ersten Parameters für `T` eingesetzt. Damit ist die Tauschfunktion nun für jeden beliebigen Typ nutzbar und es ist unmöglich, zwei Instanzen unterschiedlicher Typen miteinander zu vertauschen.

Ein weiteres Beispiel zu Generics zeigt Listing 12. Statt einer Funktion ist eine Structure namens `Container` mit einem Generic verknüpft. Im Falle von `Container` spricht man auch von einem sogenannten Generic Type, da dieser Typ auf einem Generic basiert. Das Generic besitzt den frei gewählten Namen `Item` und kommt innerhalb einer Property sowie einer Methode der Structure zum Einsatz.

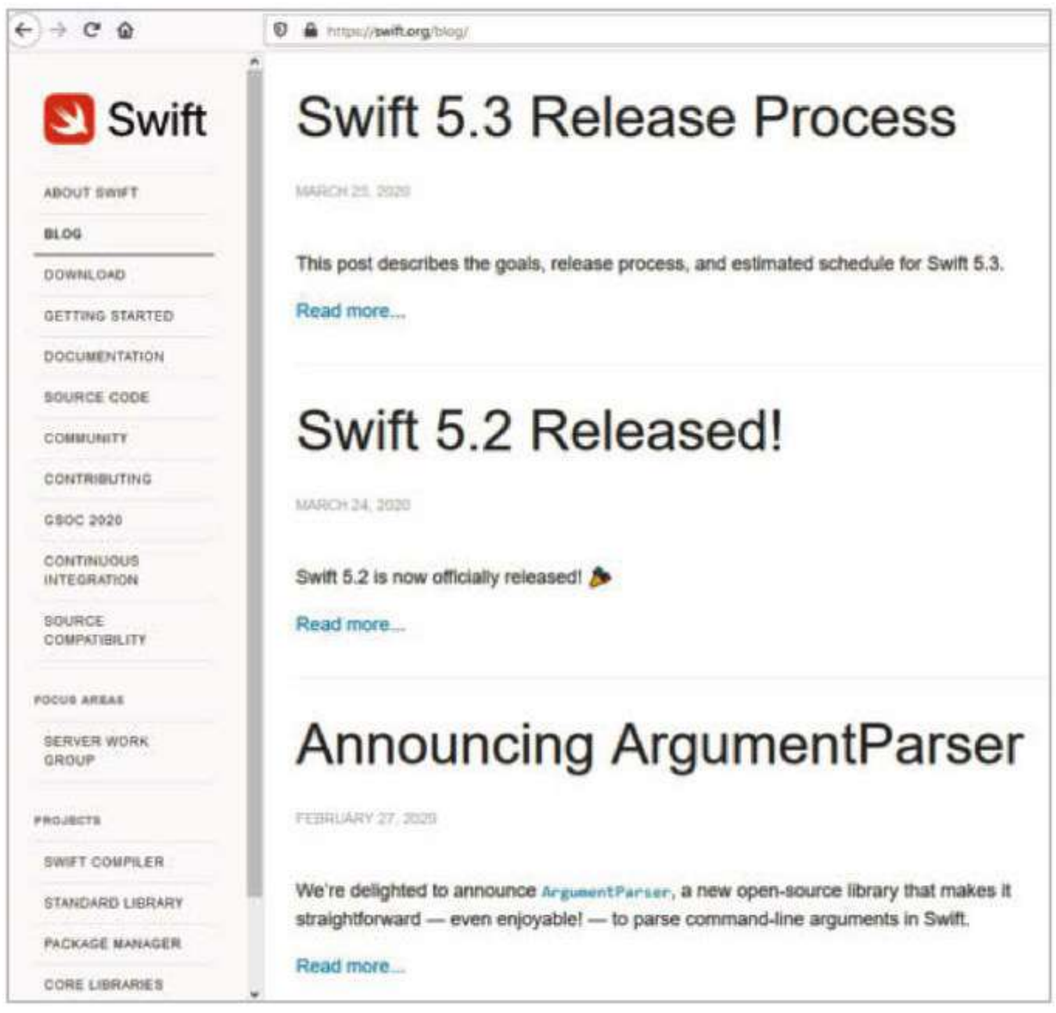
Wie bei der zuvor gezeigten Tauschfunktion entscheidet sich auch hier erst beim Einsatz der Structure `Container`, welchem konkreten Typ `Item` entspricht. Das Listing zeigt zwei erstellte `Container`-Instanzen: In der Instanz `intContainer` entspricht `Item` dem Typ `Int`, bei `stringContainer` dem Typ `String`.

Swift kann den für ein Generic eingesetzten Typ automatisch ableiten. Das ist bei der Umsetzung von `stringContainer` zu sehen. Das Programm übergibt dem `items`-Parameter Strings, die intern dem `Item`-Typ entsprechen. Dadurch wird für `Item` automatisch `String` eingesetzt. Alternativ lässt sich der für das Generic einzusetzende Typ mit eckigen Klammern angeben, wie bei `intContainer` zu sehen.

Listing 12: Einen Generic Type anlegen

```
struct Container<Item> {
    var items: [Item]
    mutating func add(item: Item) {
      items.append(item)
    }
}

var intContainer = Container<Int>(items: [19, 99])
var stringContainer = Container(items: ["First", "Second"])
```



Auf der offiziellen Projektwebsite finden sich aktuelle News sowie Ankündigungen neuer Swift-Frameworks und Sprachfeatures.

Entwicklung und Anwendungsgebiete

Swift hat sich seit der erstmaligen Präsentation im Jahr 2014 stark weiterentwickelt. Mit der 2019 veröffentlichten Version 5 der Programmiersprache folgte die Umsetzung der lange von Entwickler*innen herbeigesehnten ABI-Stabilität (Application Binary Interface). Damit lässt sich mit Swift 5 erstellter Code auch unter neueren Versionen wie Swift 6 ausführen. Zu diesem Zweck ist die Swift Standard Library nun fest in die Betriebssysteme von Apple integriert. Auf anderen Plattformen wie Linux steht die Umsetzung der ABI-Stabilität noch aus.

Die Sprache ist bisher ausschließlich für Apple-Developer relevant. Frameworks wie AppKit, UIKit und WatchKit verstehen sich mit ihr und erlauben die Entwicklung darauf basierender Apps. Zudem bindet auch Apples hauseigene Entwicklungsumgebung Xcode die Sprache in vollem Umfang ein, inklusive Refactoring-Funktionen und Package-Unterstützung.

Eine Vielzahl anderer Codeeditoren, wie Visual Studio Code von Microsoft, unterstützt zwar Swift, dennoch spielt sie außerhalb des Apple-Kosmos bis heute eine kaum nennenswerte Rolle. Auch im Webservice-Bereich schafft es die Sprache bisher kaum, richtig Fuß zu fassen. IBM, eines der Unternehmen, das Swift in diesem Umfeld entwickelt und eingesetzt hat, stellte die zugrunde liegenden Projekte jüngst ein.

Dennoch gibt es Hoffnung für eine stärkere Verbreitung außerhalb der Apple-Plattformen. Dazu gehört allen voran, dass Apple die Sprache Ende 2015 Open Source unter der Apache-2.0-Lizenz veröffentlicht hat. Die Möglichkeit, die Sprache somit auch stärker unter Windows, Linux oder im Web zu nutzen, ist also gegeben.

Apple treibt die Entwicklung mit Swift stark voran und erlaubt es, neue Frameworks nur noch mit Swift zu nutzen. Jüngstes Beispiel ist das neue UI-Framework SwiftUI. Auch deshalb arbeitet ein Großteil von macOS- und iOS-Entwickler*innen ganz oder zumindest teilweise mit Swift, unabhängig von der großen Beliebtheit der Sprache. (nb@ix.de)

Quellen

Listings zum Download: ix.de/zt7b



Thomas Sillmann

ist Autor, Apple-Developer und Trainer. Er hat mehrere Fachbücher, Fachartikel und Onlinekurse veröffentlicht.



C# 8.0: Änderungen bei Schnittstellen und Compiler

Suche nach Null

Holger Schwichtenberg

Die interessanteste Neuerung in C# 8 ist das Warnen vor Null-Referenz-Fehlern. Darüber hinaus bietet die Sprache einige syntaktische Verbesserungen.

Zusammen mit .NET Core 3.0 ist C# 8.0 im September 2019 erschienen. In Xamarin ist C# ebenfalls einsetzbar (siehe ix.de/z56k). Die meisten neuen Sprachfeatures sind auch im klassischen .NET Framework verfügbar. C# 8.0 benötigt als Entwicklungsumgebung Visual Studio 2019 oder eine aktuelle Version von Visual Studio Code.

C# ist konsequent objektorientiert, das heißt, alle Informationen, auch elementare Datentypen wie Zahlen, Zeichenketten sowie die Werte `true` und `false`, sind Objekte, die wiederum Instanzen von .NET-Klassen sind. Intern unterscheidet C# schon immer zwischen Wert- und Referenztypen. In C# 1.0 konnten jedoch nur Referenztypen (Datentyp `string` und eigene Klassen) den Wert `null` annehmen. Erst seit C# 2.0 gilt das auch für Werttypen.

Das Erlauben von `null` erweitert die Ausdrucksfähigkeit eines Typs, weil es damit jetzt eine Repräsentation des Zustandes „kein Wert vorhanden" gibt. Allerdings verursachen Nullwerte in der Praxis Schwierigkeiten, denn der Aufruf eines Attributs oder einer Methode auf einem Objektverweis, der `null` enthält, löst einen Null-Reference-Laufzeitfehler aus (siehe Abbildung).

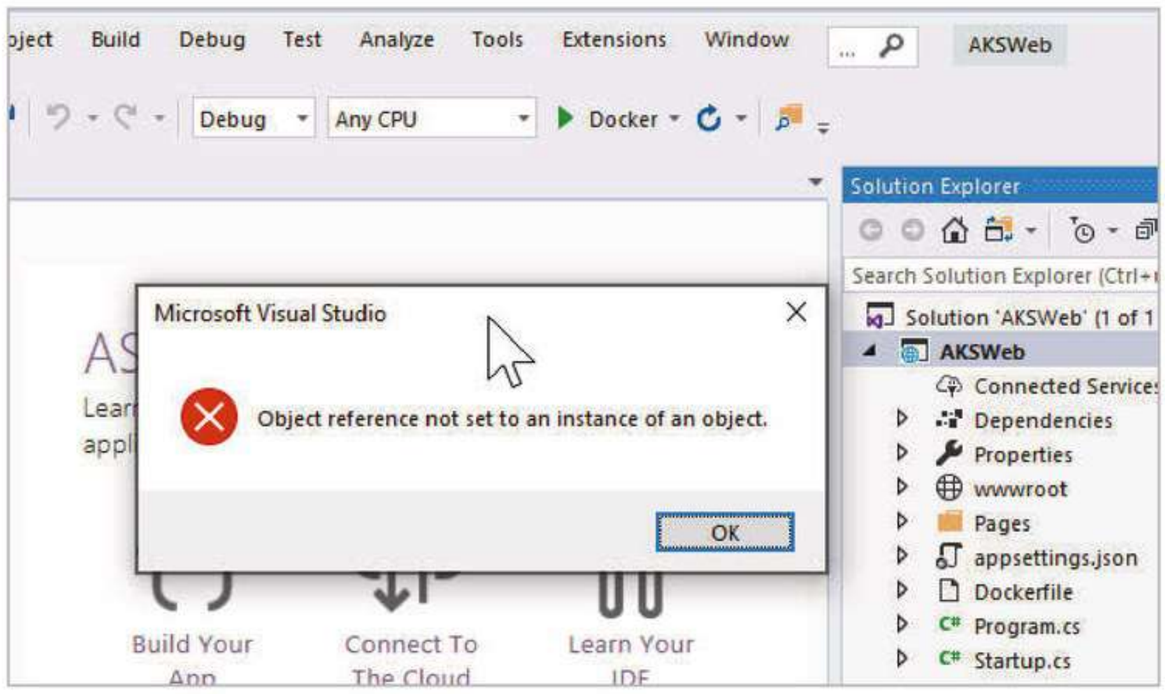
Spezielle Kontexte fangen Fehler ab

Es gibt nun Variablen, Properties, Fields und Methodenrückgabetypen, die zwar Referenztypen sind, den Nullwert aber dennoch nicht mehr erlauben. Microsoft nennt diese neuen Typen Non-Nullable Reference Types. Mit einer Datenflussanalyse kann der Compiler prüfen, ob im Programmablauf eine Zuweisung von `null` an einen Non-Nullable Reference Type existiert. Wenn der Compiler so einen Fall findet, warnt er den Entwickler.

Die Non-Nullable Reference Types und die zugehörige Datenflussanalyse sind nicht generell in jedem C#-8.0-Code ver-

iX-TRACT

- In einem Nullable Context erlauben normale Referenztypen keine Nullwerte mehr.
- C# 8.0-Schnittstellen können Methodenimplementierungen umfassen.
- Mit den Operatoren Range und Index behandelt der Entwickler elegant Teilmengen.
- Switch Expression sind Methoden, die nur aus Switch-Anweisungen bestehen.
- IDisposable ist nun auch für Strukturen auf dem Stack erlaubt.
- Die `using`-Anweisung darf der Entwickler jetzt ohne Codeblock einsetzen.



Ein Fehler der Microsoft-Entwickler erzeugt einen Null-Referenz-Fehler in Visual Studio 2019 nach dem Versuch, einen Menüpunkt anzuwählen.

fügbar, sondern nur in speziellen Bereichen, die Microsoft Kontexte nennt:

- **Nullable Warning Context:** Der Compiler warnt vor dem Auftreten von Null-Reference-Laufzeitfehlern bei allen Zugriffen auf Variablen, bei denen möglich oder nicht sichergestellt ist, dass sie entweder nicht `null` enthalten, oder bei denen der Nullfall nicht abgefangen ist.
- **Nullable Annotation Context:** Referenztypen sind im Standard nicht mehr `nullable` (fähig, den Nullwert anzunehmen). Wenn Nullwerte explizit gewünscht sind, ist dies mit dem Fragezeichen bei der Typdeklaration anzuzeigen, etwa `string?` und `Klasse?` (nicht erlaubt sind `Nullable<string>` und `Nullable<Klasse>` wie bei den Nullable Value Types).
- **Nullable Context** bezeichnet allgemein einen Kontext, der sowohl Warning Context als auch Annotation Context ist, also die Funktionen beider in sich vereint.

Die Tabelle stellt die drei Kontextarten hinsichtlich ihrer Aktivierung und Wirkung gegenüber. Dort sieht man, dass sich ein Kontext über ein ganzes Projekt erstrecken kann. Das Aktivieren eines Kontextes auf Projektebene ist in .csproj-Dateien sowohl für das klassische .NET Framework als auch in den kompakteren .NET-Core-Projektdateien möglich. Listing 1 zeigt an Beispielen die Auswirkungen der drei Kontextarten.

Kontexte verhindern Null-Referenz-Fehler effektiv. Allerdings ist zu bedenken, dass ihr Aktivieren in bestehendem Programmcode aufwendig ist – den meisten Entwicklern dürfte der Compiler viele Warnungen präsentieren. Für den Null-Fall gab es schon zuvor diverse Möglichkeiten zum Prüfen und Behandeln, zum Beispiel den Vergleich (`== null`), den Null Assignment Operator (`??`) und den Null Conditional Operator (`?.`). In C# 8.0 ist das Null Coalescing Assignment (`??=`) hinzugekommen. Mit diesem Operator kann der Entwickler eine Objektzuweisung vornehmen, sofern eine Variable den Wert `null` hat. Im Folgenden wird der Variablen `a` eine neue Instanz nur zugewiesen, wenn die Variable noch keinen Wert hat:

```
a ??= new Autor() { ID = 123, Name = "Holger Schwichtenberg" };
```

Zuvor konnte man dies mit dem Null Assignment Operator ausdrücken:

```
a = a ?? new Autor() { ID = 123, Name = "Holger Schwichtenberg" };
```

oder explizit mit dieser Zeile:

```
if (a == null) a = new Autor() { ID = 123, Name =
                                          "Holger Schwichtenberg" };
```




2020, 266 Seiten
€ 24,90 (D)
ISBN 978-3-86490-737-1



2020, 370 Seiten
€ 29,90 (D)
ISBN 978-3-86490-754-8



2019, 196 Seiten
€ 22,90 (D)
ISBN 978-3-86490-731-9



2. Auflage
2020, 696 Seiten
€ 34,90 (D)
ISBN 978-3-86490-753-1



2020, 268 Seiten
€ 32,90 (D)
ISBN 978-3-86490-708-1

plus+
Buch + E-Book:
www.dpunkt.plus

dpunkt.verlag

Wieblinger Weg 17 · D-69123 Heidelberg · fon: 0 62 21 / 14 83 40
fax: 0 62 21 / 14 83 99 · e-mail: bestellung@dpunkt.de
www.dpunkt.de

Listing 1: Compilerwarnungen im Nullable-Kontext

```
// Normaler Kontext
   string z1 = null;
   Klasse k1 = null;
   int i1 = 1;
   int? i2 = null;

   // Nullable Context einschalten
#nullable enable
   string z2 = null; // Non-Nullable Reference Type -> Warnung!
   string? z3 = null; // Nullable Reference Type
   Klasse k2 = null;// Non-Nullable Reference Type -> Warnung!
   Klasse? k3 = null;  // Nullable Reference Type
   int i3 = 1; // keine Auswirkung auf Value Types!
   int? i4 = null; // keine Auswirkung auf Value Types!
   Console.WriteLine(z2.Trim()); // Warnung: Dereference of a 7
                                    possibly null reference
   Console.WriteLine(z3.Trim()); // Warnung: Dereference of a 7
                                    possibly null reference
   Console.WriteLine(i4.ToString()); // keine Warnung

   // Nullable Context wieder ausschalten
#nullable disable
   z2 = null; // keine Warnung
```

Traditionell dürfen Schnittstellen in objektorientierten Programmiersprachen keine Implementierung enthalten. Das war (abstrakten) Basisklassen vorbehalten. Java hat dieses Prinzip schon 2014 mit seiner achten Version durchbrochen. Nun ist Programmcode auch in C#-8.0-Schnittstellen erlaubt.

Weiterentwicklung von Schnittstellen

Letztere sollen Klassen nicht ablösen, vielmehr möchte man bestehende Schnittstellen weiterentwickeln können, ohne alle abgeleiteten Klassen ändern zu müssen. Das ist sinnvoll, wenn Framework-Anbieter Schnittstellen vorgeben, die andere Softwareentwickler dann implementieren müssen.

Das Implementieren von Code in Schnittstellen erfolgt syntaktisch wie das von Methoden in Klassen, also mit Sichtbarkeitsmodifizierer (`private`, `protected`, `internal`, `public`, `virtual`, `abstract`, `sealed`, `static`, `extern` und `partial`) sowie einem Codeblock in geschweiften Klammern (Block Body) oder einem Lambdaausdruck (Expression Body). Im Standard sind die Implementierungen als `virtual` eingestuft, daher auch der alternative Name in Interfaces: Virtual Extension Methods. Eine Klasse, die eine Schnittstelle mit Implementierung umsetzt, kann folglich jede der Methoden auch wieder überschreiben.

Listing 2 zeigt eine ILogger-Schnittstelle mit einer `Log()`-Methode ohne Implementierung, die die Klasse dann umsetzen muss, sowie einer Überladung von `Log()` und eine weitere `LogDetails()`-Methode, die jeweils eine Implementierung beinhalten. Neben Instanzmethoden können Schnittstellen nun auch statische Methoden sowie statische Properties und Fields enthalten.

Der Aufrufer einer Klasse, die `ILogger` realisiert (etwa `ConsoleLogger`), kann alle drei Methoden verwenden, sofern die Objektvariable auf die Schnittstelle typisiert ist:

```
ILogger l = new ConsoleLogger();
l.Log(LogLevel.Info, "Programmstart");
l.Log(new ApplicationException("Ein Test-Fehler!");
l.LogDetails(new ApplicationException("Ein Test-Fehler!");
```

Microsoft hat in der neuen Sprachversion zwei neue Operatoren für die prägnante Auswahl von Teilmengen eingeführt:

- Der Index-Operator (^) kennzeichnet eine Position relativ zum Ende einer Menge. Der Compiler verwendet dafür die Klasse `System.Index`.
- Der Range-Operator (`..`) legt einen Teilbereich mit Angabe für Start und Ende fest. Der Compiler verwendet dafür die Klasse `System.Range`. Zu beachten ist, dass beim Range-Operator der Startwert inklusive und der Endwert exklusive ist, das heißt, `[2..4]` sind nicht drei Werte, sondern nur zwei (dritter und vierter Wert, da die Zählung bei `0` beginnt).

Range und Index funktionieren mit Zeichenketten und Arrays; Indexe zusätzlich auch mit generischen Listen (Klassen, die die Schnittstelle `IEnumerable<T>` implementieren). Listing 3 zeigt den Einsatz für Range und Index mit einer Zeichenkette als Eingabemenge für Anwendungsfälle, an denen man bisher die `Substring()`-Methode eingesetzt hätte.

Variante für mehrfache Verzweigungen

Die Mehrfachverzweigungen `switch { case: … break; default: … }` gibt es seit der ersten Sprachversion. C# 8.0 hat eine Variante dieses Sprachkonstrukts eingeführt unter Einsatz des Schlüsselwortes `switch` in Verbindung mit Lambdaausdrücken. Listing 4 zeigt eine Methode mit einer Switch Expression.

Der erste Lambdaausdruck `=>` ist syntaktisch ein Expression Bodied Member, den es seit C# 6.0 gibt. Es folgt die Switch Expression, die mit dem Wert beginnt, nach dem differenziert werden soll. Nun kommt das Schlüsselwort `switch`. Die einzelnen Werte mit dem Folgeausdruck sind dann wieder durch `=>` getrennt. Break-Anweisungen sind nicht notwendig. Anstelle des Schlüsselwortes `default` tritt der Discard-Operator (Unterstrich: _). Switch-Anweisungen sind mit elementaren Datentypen, Objek-

Varianten des Nullable-Kontextes

	Nullable Annotation Context	Nullable Warning Context	Nullable Context (entspricht Annotation Context und Warning Context)
Aktivierung auf Projektebene in der .csproj-Datei per XML-Tag	<Nullable>annotations</Nullable>	<Nullable>warnings</Nullable>	<Nullable>enable</Nullable>
Aktivierung im Programmcode mit Compiler-Präprozessor-Anweisungen	#nullable enable annotations	#nullable enable warnings	#nullable enable
Deaktivierung im Programmcode mit Compiler-Präprozessor-Anweisungen	#nullable disable annotations	#nullable disable warnings	#nullable disable
Zurücksetzen auf die Projekteinstellung mit Compiler-Präprozessor-Anweisungen	#nullable restore annotations	#nullable restore warnings	#nullable restore
Bedeutung der Deklaration Klasse k	non-nullable	nullable	non-nullable
Bedeutung der Deklaration Klasse? k	nullable	nicht erlaubt (erzeugt Warnung)	nullable
Warnung vor Null Reference Exceptions	nein	ja	ja

Listing 2: Erweiterte ILogger-Schnittstelle

```
interface ILogger {
  string Prefix { get; set; }
  long Count { get; set; }
  // Methode ohne Implementierung
  void Log(LogLevel level, string message);

  // Methode mit Implementierung mit Block Body
  public void Log(Exception ex)  {
   Log(LogLevel.Error, ex.Message);
  }

  // Methode mit Implementierung mit Expression Body
  public void LogDetails(Exception ex)=> Log(LogLevel.Error, ⤷
                                                    ex.ToString());
 }
```

Listing 3: Range und Index mit einer Zeichenkette

```
string text = "0123456789";
string teilstring1 = text[4..]; // --> ab dem 5. Zeichen bis Ende ⤷
                                                          "456789"
string teilstring2 = text[^4..^0]; // 0 = Ende --> "6789"
string teilstring3 = text[2..4]; // "23"
string teilstring4 = text[0..^0]; // alle
string teilstring5 = text[..]; // .. == alle
System.Range teilstring6 = 1..3; // neue Klasse System.Range, nur in ⤷
                                                    .NET Core und Mono
string teilstring7 = text[r7]; // 12
```

ten (siehe Beispiel der Klasse Person in Listing 4) und Tupeln möglich.

Zwei weitere Verbesserungen betreffen die altbekannte IDisposable-Schnittstelle. Nun können nicht nur Klassen (Schlüsselwort `class`) und normale Strukturen auf dem Heap (Schlüsselwort `struct`), sondern auch Strukturen auf dem Stack (Schlüsselwort `ref struct`) das IDisposable-Pattern umsetzen. Allerdings nicht mit Verweis auf die Schnittstelle `Idisposable`, denn Strukturen auf dem Stack dürfen grundsätzlich keine Schnittstellen implementieren. In diesen Strukturen schreibt man einfach eine `Dispose()`-Methode. Diese lose `Dispose()`-Implementierung ruft die .NET-Laufzeitumgebung am Ende von `using`-Blöcken auf, genauso wie die implementierten `Dispose()`-Methoden der expliziten IDisposable-Schnittstelle:

```
public ref struct Ressource {
public void Dispose() { Console.WriteLine("Ressource vernichtet!"); }
}
```

Sogenannte `using()`-Blöcke sind ein übliches Muster, um Klassen und Strukturen mit IDisposable-Schnittstelle zu verwenden und sicherzustellen, dass am Blockende die `Dispose()`-Methode automatisch ausgeführt wird. Es ist nun möglich, das Schlüsselwort `using` für Klassen mit einer IDisposable-Schnittstelle ohne einen expliziten Codeblock mit geschweiften Klammern zu verwenden. In diesem Fall ist die nach `using` deklarierte Variable gültig bis zum Ende des umgebenden Codeblocks; erst wenn er endet, wird `Dispose()` aufgerufen (Listing 5).

Schon im Sprachstandard C# 7 existierten lokale Methoden, die in andere Methoden eingebettet sind. Lokale Methoden sind aber nicht isoliert, sondern können auf alle Variablen der äußeren Ebenen (der umgebenden Klasse und Funktion) zugreifen. Mit dem Zusatz `static` vor der Deklaration einer lokalen Methode kann der Entwickler diesen Übergriff auf die Umgebung nun unterbinden.

Listing 4: Switch Expression

```
static string HolePersonTyp(Person k) => k switch  {
   // keine weiteren Statements hier erlaubt, z.B. string ausgabe = ⤷
                                                    "{name} ist ein";
   { Status: 'A', Art: PersonArt.Kunde } => $"{k.Name} ist ein ⤷
                                                    guter Kunde",
   { Status: 'A', Art: PersonArt.Lieferant } => $"{k.Name} ist ein ⤷
                                                    guter Lieferant",
   { Status: 'B', Art: PersonArt.Kunde } => $"{k.Name} ist ein ⤷
                                                    durchschnittlicher Kunde",
   { Status: 'B', Art: PersonArt.Lieferant } => $"{k.Name} ist ein ⤷
                                                    durchschnittlicher Lieferant",
   _ => $"{k.Name} ist eine sonstige Person."
  };
```

Listing 5: using als Deklaration statt als Block

```
try   {
    using Dateisystemzugriff d = new Dateisystemzugriff(@"t:\log. ⤷
                                                          txt");
    for (int a = 1; a < 10; a++)    {
     d.Log("Meldung # " + a);
     Console.Write(".");
     System.Threading.Thread.Sleep(10);
    }
    // d ist hier noch gültig
    d3.Log("Ende!");
   }// d ist ab jetzt ungültig, d.Dispose() wird aufgerufen
catch (Exception ex)   {
    Console.WriteLine(ex);
   }
```

Kontinuierliche Weiterentwicklung

C# hat schon immer erlaubt, Attribute in Klassen zu schaffen, deren Wert man nur lesen und nicht schreiben darf. Dafür schafft der Entwickler eine Property nur mit Getter, also ohne Setter. Mit dem neuen Zusatz `readonly` lässt sich zusätzlich erreichen, dass der Compiler jegliche Manipulation des Objektzustandes im Getter verhindert. Auch bei einer Methode kann der Entwickler `readonly` anwenden und dort Methoden erzeugen, die den Zustand nicht beeinflussen dürfen.

Die `foreach`-Schleife gibt es ebenfalls seit C# 1.0. Neu in C# 8.0 ist das Konstrukt `async foreach`. Dabei iteriert der Entwickler über Klassen mit der neuen Schnittstelle `IAsyncEnumerable<T>`. Die Daten treffen dann asynchron ein. Mit dem etablierten Konstrukt des Cancellation Token kann er weitere eingehende Daten unterbinden, sofern die Datenquelle dies zulässt (zu Neuerungen siehe auch ix.de/z65k). (jd@ix.de)

Quellen

Weiterführende Informationen unter ix.de/z65



Dr. Holger Schwichtenberg

leitet das Expertennetzwerk www.IT-Visions.de, das Beratung, Schulungen und Softwareentwicklung im Umfeld von Microsoft-, Java- und Webtechniken anbietet. Er ist Autor des Buchs „Windows PowerShell 5 und PowerShell 7: Das Praxisbuch“, das im Carl Hanser Verlag erschienen ist.



Single-Page Web Applications mit Blazor und C#

Rückkehr in den Browser

Holger Schwichtenberg

Nach dem Ende von Silverlight bietet Microsoft seit .NET Core 3.0 und C# 8.0 wieder die Programmierung interaktiver Browseranwendungen an.

Microsofts Neustart für die Webbrowserprogrammierung heißt ASP.NET Blazor. Auf die erste stabile Version in .NET Core 3.0 am 23.09.2019 folgte am 03.12.2019 die Version 3.1, welche man einsetzen sollte, da der Support für die Version 3.0 bereits im März 2020 wieder enden wird. Dabei ist zunächst nur eine von mehreren geplanten Varianten von Blazor erschienen: Blazor Server. Blazor Server ist hier allerdings nicht die Serverseite einer Client-Server-Lösung, sondern eine eigenständige Softwarearchitektur für die effiziente Entwicklung von Single-Page Web Applications (SPAs).

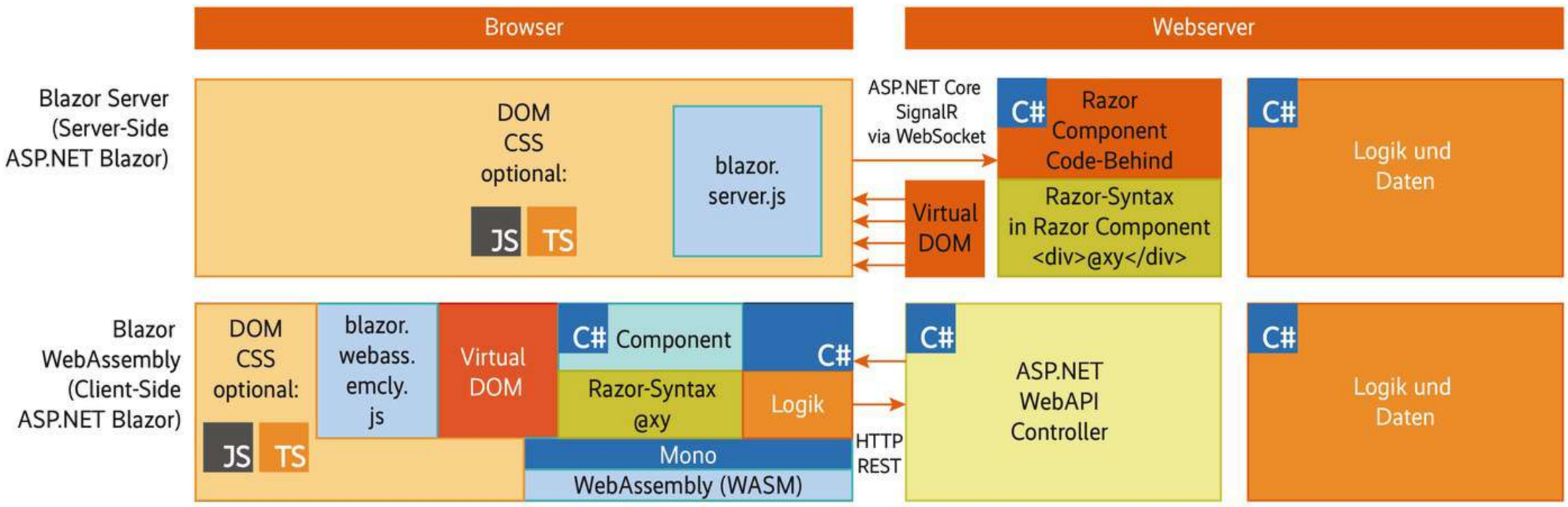
iX-TRACT

- Mit Blazor Server und Blazor WebAssembly lassen sich moderne Webanwendungen in C# schreiben.
- Beim bereits verfügbaren Blazor Server läuft der C#-Code auf dem Server, bei Blazor WebAssembly im Browser. Beide Ansätze haben spezifische Vor- und Nachteile.
- Blazor Server ist bereits produktiv nutzbar, die WebAssembly-Version derzeit als Vorschau verfügbar. Die Portierung von Code zwischen den beiden Ansätzen erfordert nur geringe Anpassungen.

Blazor Server versus Blazor WebAssembly

Bei Blazor Server (alias Server Side Blazor) läuft der C#-Programmcode auf dem Webserver. Die JavaScript-Bibliothek blazor.server.js von Microsoft überträgt jede Benutzerinteraktion zum Webserver und spiegelt jede auf dem Server erzeugte UI-Änderung im Browser wider. Die dafür notwendige kontinuierliche Kommunikation zwischen Webserver und Browser erfolgt über ASP.NET Core SignalR via WebSockets (siehe Abbildung 1).

Die Serverseite hält dabei für jeden angeschlossenen Webbrowser ein Virtual DOM im Hauptspeicher. Das schränkt die Skalierbarkeit ein; laut Microsoft sind dennoch 20 000 gleichzeitige Benutzer mit jeweils einem Klick pro Sekunde auf einem virtuellen Cloud-Server mit 4 CPUs und 14 GB RAM möglich.



Vergleich zwischen Blazor Server und Blazor WebAssembly (Abb. 1)

Das reicht nicht aus, um Google oder Facebook Konkurrenz zu machen, aber es genügt sicherlich für viele in Deutschland entwickelte Webanwendungen.

Anders als klassische, JavaScript-basierte Single-Page-Webanwendungen (SPA) mit Angular, React und Co. ist bei Blazor Server eine Offlinefähigkeit grundsätzlich nicht möglich. Die Bibliothek blazor.server.js sendet periodisch eine kleine 3-Byte-Nachricht an den Webserver. Wenn die Verbindung kurzzeitig abreißt, erscheint die aktuell sichtbare Seite im Browser ausgegraut und der Benutzer sieht eine Nachricht. Ist keine Wiederaufnahme möglich, fordert der Browser den Benutzer nach 30 Sekunden auf, die Seite neu zu laden. Man kann Blazor Server als eine „SPA light" verstehen.

Bei Blazor WebAssembly (alias Client Side Blazor) läuft der C#-Programmcode tatsächlich als echte SPA im Webbrowser in der WebAssembly-VM (siehe Abbildung 1). Es ist noch in der Vorschauphase und soll im Mai 2020 zum ersten Mal stabil erscheinen. Blazor WebAssembly verwendet derzeit im Browser die Mono-Laufzeitumgebung. Im Rahmen der geplanten Zusammenführung der .NET-Entwicklungslinien im Zuge von .NET 5.0 soll sich dies im November 2020 allerdings ändern.

Gleiche APIs und Razor Components

Da Blazor WebAssembly laut Microsoft noch nicht für den produktiven Einsatz reif ist, kommt in diesem Artikel Blazor Server zum Einsatz. Eine Umstellung des Programmcodes auf Blazor WebAssembly ist aber später einfach möglich, da beide Blazor-Varianten – bis auf einige Nuancen – die gleichen Konzepte und APIs verwenden: Die Oberflächenbeschreibung erfolgt mit der aus ASP.NET MVC seit dem Jahr 2010 bekannten Razor-Syntax (HTML mit C#-Inseln), die Benutzerschnittstellensteuerung in C#-Code. Zusammen ergibt das eine Razor Component. Wahlweise kann der Webentwickler beide Teile der Razor Component in eine .razor-Datei legen oder in zwei Dateien trennen (Code-Behind-Modell).

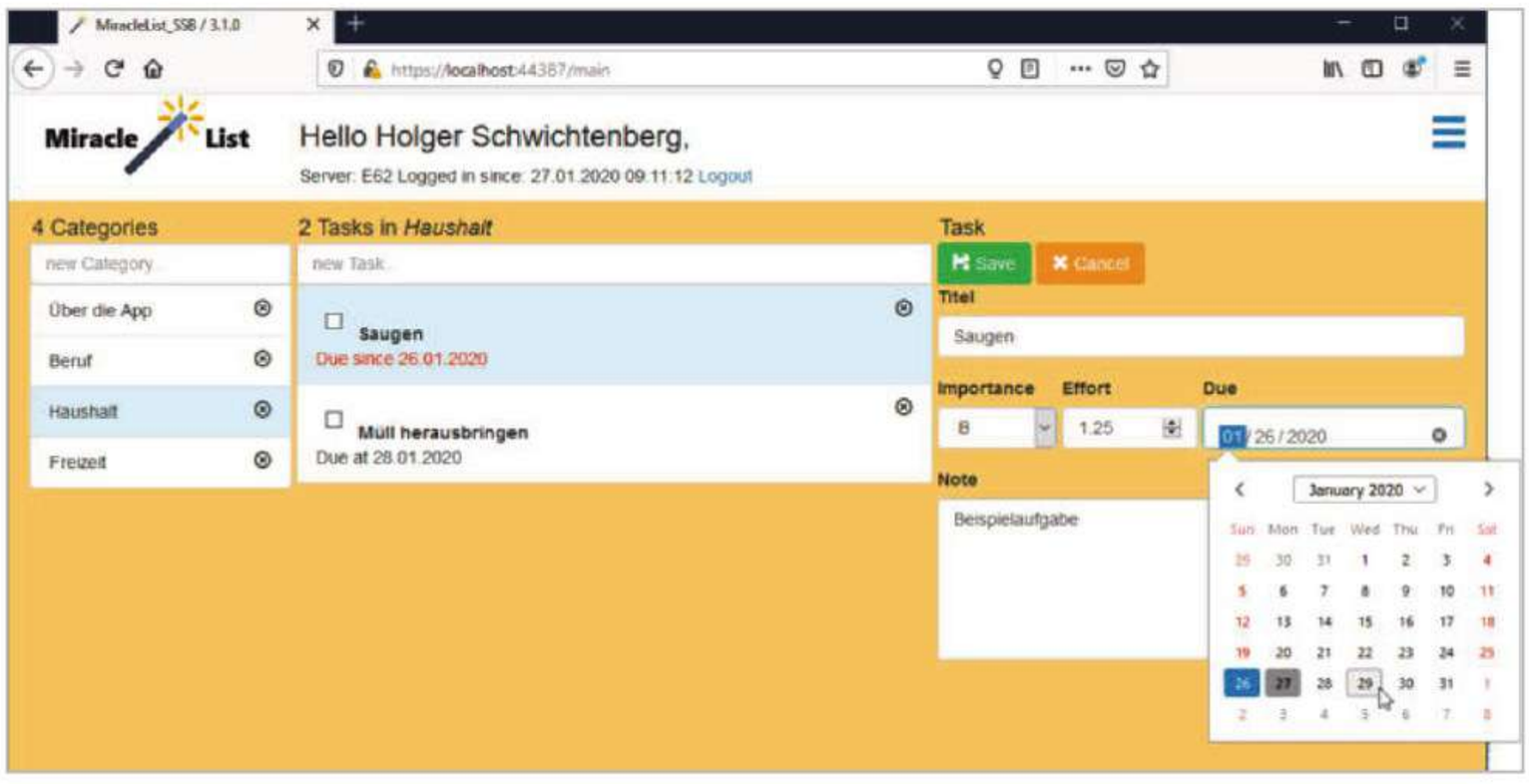
Lediglich im Startcode und bei der Verwendung der APIs auf dem Server gibt es deutliche Unterschiede: Bei Blazor Server kann der Webentwickler direkt mit allen .NET-APIs auf dem Webserver interagieren und so auch Datenbanken direkt verwenden. Blazor WebAssembly läuft hingegen in der Sandbox des Browsers. Es kann folglich viele, aber nicht alle .NET-APIs verwenden. So ist ein Zugriff auf Datenbanken oder Systemressourcen nur über eine Web-API möglich (siehe Abbildung 1).

Für die Zukunft arbeitet Microsoft an weiteren Varianten von Blazor: Blazor Progressive Web Applications (PWA), Blazor Hybrid (Cross-Platform-Anwendungen auf Basis von Electron oder Web Window, siehe ix.de/zxka) sowie Blazor Native. Letzteres wird auch nicht HTML-basierte Oberflächen rendern können. Ein erstes Projekt in diese Richtung sind die Mobile Blazor Bindings zur Entwicklung von iOS- und Android-Apps, wobei die Oberflächen auf Xamarin Forms (XAML) nutzen.

Fallbeispiel MiracleList

Im Rahmen eines dreiteiligen *iX*-Tutorials ist im Jahr 2017 eine Webanwendung zur Aufgabenverwaltung mit TypeScript, Angular und Twitter Bootstrap sowie diversen frei verfügbaren Steuerelementen entstanden [1, 2, 3]. Dieser Beitrag stellt nun eine Implementierung mit Blazor Server vor, die ebenfalls Bootstrap nutzt (siehe Abbildung 2). Aus Platzgründen kann hier nicht der komplette Programmcode abgedruckt oder diskutiert werden; Sie finden ihn als Archiv mit einer Visual-Studio-2019-Projektmappe unter ix.de/zxka.

Mindestvoraussetzung für das Projekt ist die Version 16.4 der Entwicklungsumgebung Visual Studio 2019. In der Entwicklungsumgebung sieht man im Projekt-anlegen-Dialog auf oberster Ebene einen Eintrag „Blazor App". Danach folgt ein Dialog zur Auswahl zwischen Blazor Server und Blazor WebAssembly. Die Archivdatei auf dem *iX*-Listing-Server enthält auch eine Implementierung von MiracleList, die die Vorschauversion von Blazor WebAssembly nutzt.



Die Todo-App MiracleList, implementiert mit Blazor Server (Abb. 2)

Die Blazor-Server-Implementierung besteht aus drei Projekten (siehe Abbildung 3):

- MiracleList_SSB ist das eigentliche Blazor-Server-Projekt.
- ITVBlazorRCL ist eine Hilfsbibliothek, die sowohl im Blazor-Server- als auch im Blazor-WebAssembly-Projekt verwendet wird. Hier ist die Klasse BlazorUtil.cs realisiert, ein Wrapper um einige JavaScript-Funktionen (BlazorUtil.js).
- MLBlazorRCL ist eine weitere gemeinsame Hilfsbibliothek. Sie enthält Grafiken und eine CSS-Datei, die in beiden Varianten zum Einsatz kommt.

RCL steht dabei für Razor Class Library, eine spezielle Form einer DLL-Assembly, in der ein Entwickler wiederverwendbare Elemente (sowohl statische Elemente als auch Razor Components und beliebige C#-Klassen) speichern kann. Hinzu kommen weitere Projekte wie die Geschäftslogik (BL), die Geschäftsobjekte (BO) und die Datenzugriffsschicht (DA), die aber nicht spezifisch sind für die Blazor-Lösung, sondern auch in anderen Anwendungsarten zum Einsatz kommen. Unter „Dependencies“ besitzt das Blazor-Projekt Verweise auf die Geschäftslogik (Projekt „BL“) und die Geschäftsobjekte (Projekt „BO“).

Das Projekt MiracleList_SSB basiert auf der Standardprojektvorlage von Microsoft, es wurden jedoch die Dateien Error.razor, Counter.razor, FetchData.razor und NavBar.razor entfernt. Der /Pages-Ordner enthält stattdessen folgende Dateien:

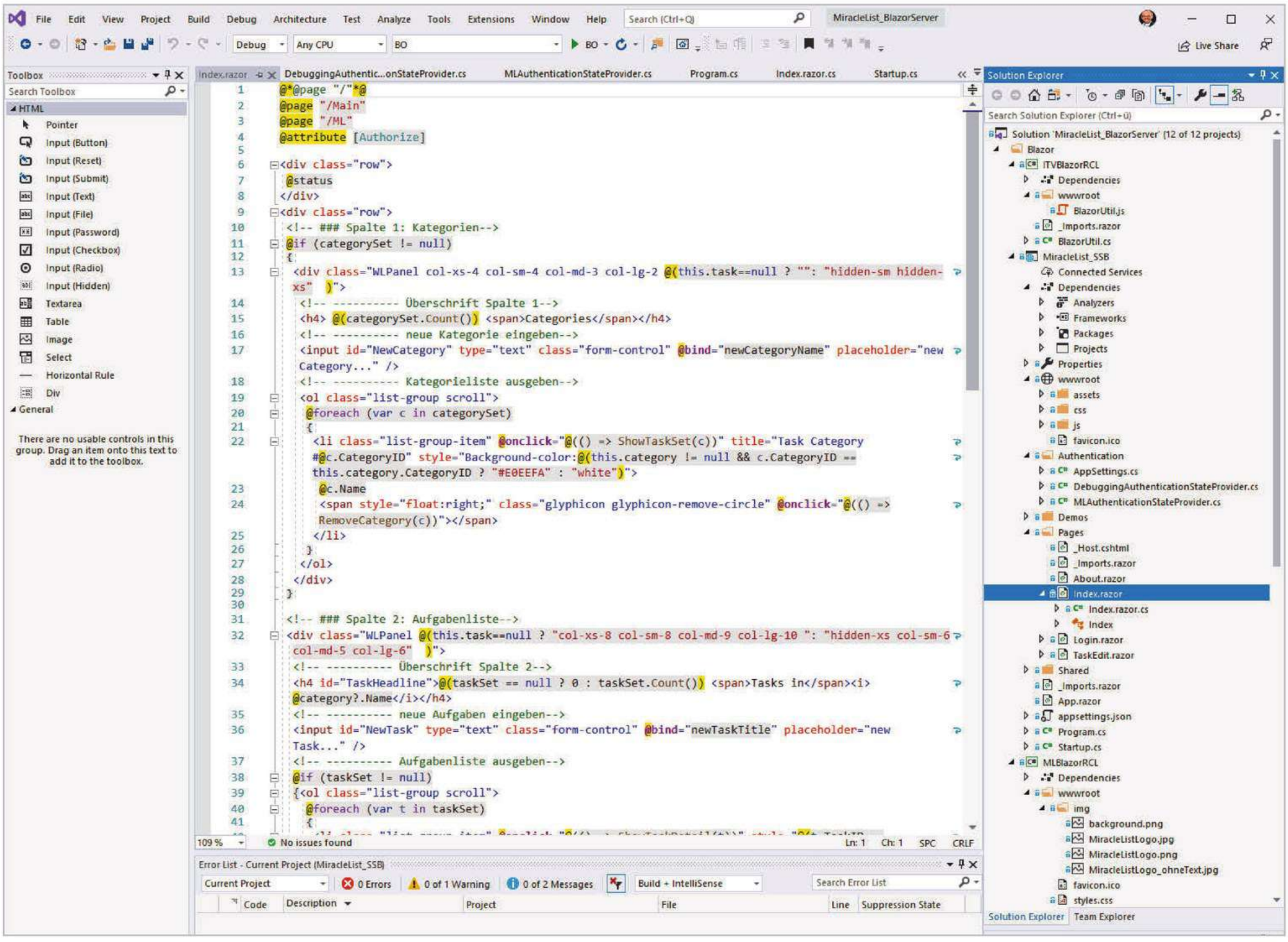
- die Razor Component /Pages/Login.razor (Route /) mit dem Anmeldedialog;
- /Pages/Index.razor (Route /Main) enthält die Hauptansicht mit der Kategorieliste links und der Aufgabenliste rechts;
- /Pages/TaskEdit.razor wird ganz rechts als dritte Spalte einblendet, wenn eine Aufgabe gewählt ist;
- /Shared/MainLayout.razor ist die Masterpage mit Logo und Hamburger-Menü.

Die ersten drei Blazor Components sind mit Code-Behind-Datei realisiert. Die Masterpage verwendet nur eine Datei.

Die Anmeldeansicht

In der Anmeldeansicht der Blazor-Server-Anwendung kann der Benutzer nicht nur Benutzername und Kennwort eingeben, sondern auch eine Datenbank auswählen (Listing 1 und 2). Der Programmcode greift standardmäßig auf eine lokale Microsoft-SQL-Server-Datenbank zu. Wie die Datenbank angelegt werden kann, steht in der Datei readme.md.

Listing 1 beginnt mit den Routen (relative URIs), auf die die Razor Component reagieren soll. Es sind mehrere Routen möglich, eine Route muss aber eindeutig zu einer Razor Component gehören. Ein expliziter Verweis mit @inherits auf die Code-Behind-Klasse ist seit Version 3.1 nicht mehr nötig; die Vorlagen-



Aufbau der Blazor-Server-Implementierung des MiracleList-Webfrontends (Abb. 3)

klasse und die Code-Behind-Klasse sind nunmehr als partielle Klassen miteinander verbunden.

Es folgt der HTML-Code, angereichert mit Razor-Syntax-Elementen: `@xy` zur Ausgabe und `@bind="xy"` zur bidirektionalen Datenbindung an Attribute der Code-Behind-Klasse. Die Ereignisbindung erfolgt mit `@onclick="Methodenname"`.

Die Code-Behind-Klasse in Listing 2 bietet die verwendeten Attribute als C#-Properties an. Zudem erhält sie zwei Dienste per Dependency Injection. Bei der ersten eingehenden HTTP-Anfrage für die Seite führt Blazor die Methode `OnInitializedAsync()` aus, in der man die Ausgangswerte setzen kann. Die beim Klick aktivierte `DoLogin()`-Methode führt die Benutzeranmeldung aus und navigiert bei Erfolg zur Hauptseite unter der Route /Main. Die Methode darf nicht einfach `Login()` heißen, weil dies der Name der umgebenden Klasse ist.

Listing 1: Login.razor

```
@page "/"
@page "/Login"
...
<h2>Benutzeranmeldung</h2>
...
<label for="name">E-Mail-Adresse</label>
<input name="name" @bind="this.Username" id="name" type="text" placeholder="Ihre E-Mail-Adresse" 7
                                                                    required class="form-control">
...
<label for="password">Kennwort</label>
<input name="password" @bind="this.Password" id="password" placeholder="Ihr Kennwort" required 7
                                                                type="password" class="form-control">
...
<label for="password">DB-Server</label>
<select @bind="this.Backend" id="backend" class="form-control">
  <option selected>@DAL.Context.ConnectionString</option>
</select>
...

<button @onclick="DoLogin" class="btn btn-primary" type="submit" id="Anmelden" 7
                                                                    name="Anmelden">Anmelden</button>
<span id="errorMsg" class="text-danger">@ErrorMsg</span>
```

Listing 2: Login.razor.cs

```
...
namespace Web.Pages
{
  public partial class Login
  {
    [Inject]
    public NavigationManager NavigationManager { get; set; }

    [Inject]
    AuthenticationStateProvider mLAuthenticationStateProvider { get; set; } = null;

    string Username { get; set; }
    string Password { get; set; }
    string Backend { get; set; }
    string ErrorMsg { get; set; }

    protected override async System.Threading.Tasks.Task OnInitializedAsync()  {
      Backend = DAL.Context.ConnectionString;
    }

    protected async Task DoLogin()  {
      var u = await ((MLAuthenticationStateProvider)mLAuthenticationStateProvider).LogIn(Username, 7
                                                                        Password, Backend);
      if (u != null)   {
        this.NavigationManager.NavigateTo("/main");
      }
      else   {
        ErrorMsg = "Login Error!";
      }
    }
  }
}
```

Der für die Authentifizierung sorgt

Die Benutzeranmeldung erfolgt in der Klasse `MLAuthenticationStateProvider`, die hier nicht abgedruckt ist, sich aber im Archiv auf dem *iX*-Listing-Server befindet. Für einige Mechanismen wie die Direktive `@attribute [Authorize]` (siehe Listing 3 und 5) ist eine von `AuthenticationStateProvider` abgeleitete Klasse notwendig, die in der Startup-Klasse in der Methode `ConfigureServices()` zu registrieren ist:

```
services.
AddScoped<AuthenticationStateProvider, 7
            MLAuthenticationStateProvider>();
```

Der Entwickler ist innerhalb seiner Implementierung frei hinsichtlich der Benutzeranmeldung und Benutzerkontenverwaltung. Wichtig sind lediglich zwei Details:

- Die Klasse muss die Methode `async Task<AuthenticationState> GetAuthenticationStateAsync()` überschreiben und im zurückgelieferten `AuthenticationState` die Blazor-Infrastruktur jederzeit über den Status (welche Benutzer angemeldet ist oder dass kein Benutzer angemeldet ist) informieren.
- Zudem muss man nach der erfolgreichen Benutzeranmeldung und bei einer Benutzerabmeldung die Blazor-Infrastruktur über einen Aufruf von `this.NotifyAuthenticationStateChanged()` unter Angabe eines `AuthenticationState`-Objekts aktiv benachrichtigen.

Die Hauptansicht

Die Listings 3 und 4 enthalten die Hauptansicht /Pages/Index.razor, die die ersten beiden Spalten aus Abbildung 2 bildet. Für die dritte Spalte wird die Bearbeitungsansichtskomponente TaskEdit.razor mit dem Tag `<TaskEdit>` eingebunden.

Die Razor-Vorlage beginnt mit der Deklaration der Route `/main`. Danach legt `@attribute [Authorize]` fest, dass nur angemeldete Benutzer diese Komponente aufrufen dürfen. Auch wenn ein Benutzer die URI `/main` kennt, kann er die Komponente nicht ohne Anmeldung nutzen.

Im Programmcode findet man eine Reihe von `@if`-Bedingungen, die fallweise ganze Blöcke nur dann rendern, wenn die Daten auch verfügbar sind. Dies ist in Blazor grundsätzlich wichtig, da viele Operationen asynchron sind und es möglich ist, dass die Daten beim ersten Rendern noch nicht verfügbar sind, sodass es zu Null-Referenz-Fehlern kommen kann. Wenn die Daten dann etwas später eintreffen, stößt Blazor automatisch ein erneutes Rendern an.

Fallunterscheidungen mit dem bedingten C#-Ausdruck (`Bedingung ? ja : nein`) findet man auch innerhalb einzelner HTML-Attribute, zum Beispiel bei `style` für die farbliche Hervorhebung (hellblau) der aktuellen Kategorie und der aktuellen Aufgabe sowie auch bei `class` für die fallweise Ausblendung ganzer Spalten. Mit `@foreach` bildet man in Razor eine Schleife, zum Beispiel über die Liste der Kategorien und die Liste der Aufgaben.

Die Code-Behind-Datei in Listing 4 stellt wieder entsprechende Properties sowie Ereignisbehandlungsroutinen bereit, die die Geschäftslogik aufrufen. Dazu gehört auch die Methode `ReloadTasks()`, die beim Ereignis `TaskHasChanged` der `TaskEdit`-Komponente ausgelöst wird. Die an die Texteingabefelder gebundenen Properties `newCategoryName` und `newTaskTitle` lösen bereits in ihrem Setter einen Aufruf der Geschäftslogik aus. So kann der Benutzer komfortabel neue Elemente anlegen, wenn er das Eingabefeld verlässt; er muss nicht noch eine Schaltfläche drücken. Die Zuweisung eines Objekts an die Property `task` in der Methode `ShowTaskDetail()` reicht, um die untergeordnete Komponente `TaskEdit` (Listing 5 und 6) sichtbar zu machen.

Bearbeitungsformular

Die Listings 5 und 6 zeigen das Aufgabenbearbeitungsformular /Pages/TaskEdit.razor inklusive Code-Behind-Datei. Das Formular entsteht mit den normalen HTML-Eingabesteuerelementen wie `<input>`, `<textarea>` und `<button>`. Auch in dieser Razor-Vorlage ist es wieder wichtig, die Blöcke nur zu rendern, wenn die verwendeten Properties (hier: `task`) auch tatsächlich Objekte enthalten.

Die Code-Behind-Klasse deklariert zwei Properties als öffentliche Parameter (annotiert mit `[Parameter]`), eines als Task-Objekt und eines als Ereignis mit einem Zahlparameter (`Action<int>`). Diese beiden Parameter kann der Nutzer (hier: Index.razor) per HTML-Syntax befüllen, was man in Listing 3 sieht:

```
<TaskEdit Task="@task" TaskHasChanged="@ReloadTasks"></TaskEdit>
```

Wenn die Parameter befüllt sind, löst die Blazor-Infrastruktur das Ereignis `OnParameterSet()` in Listing 6 aus. Sowohl beim Speichern von Änderungen als auch beim Abbruch von Änderungen erzeugt der Webentwickler dann manuell das Ereignis `TaskHasChanged()` mit der ID der Aufgabe. Dies führt schließlich zur Ausführung von `ReloadTasks()` in Index.razor.cs, siehe Listing 4.

Zusammenspiel mit JavaScript

Die Aufrufe von `Util.Log()` erzeugen eine Ausgabe in der Debug-Konsole des Browsers. Dies ist in Blazor Server tatsächlich nicht mit einem einfachen `Console.WriteLine()` wie in Blazor WebAssembly zu schaffen, sondern nur über Interoperabilität mit JavaScript (siehe Listing 7), da der C#-Programmcode – wie geschildert – auf dem Server läuft.

Listing 3: Index.razor

```
@page "/Main"
@attribute [Authorize]
...
<div class="row">
<!-- ### Spalte 1: Kategorien-->
@if (categorySet != null)
{
  <div class="WLPanel col-xs-4 col-sm-4 col-md-3 col-lg-2 @(this.task==null ? "": "hidden-sm hidden-xs"  )">
    <!-- ---------- Überschrift Spalte 1-->
    <h4> @(categorySet.Count()) <span>Categories</span></h4>
    <!-- ---------- neue Kategorie eingeben-->
    <input id="NewCategory" type="text" class="form-control" @bind="newCategoryName" placeholder="new Category..." />
    <!-- ---------- Kategorieliste ausgeben-->
    <ol class="list-group scroll">
      @foreach (var c in categorySet)
      {
        <li class="list-group-item" @onclick="@(() => ShowTaskSet(c))" title="Task Category #@c.CategoryID" style="Background-color:@ ↵
                                                    (this.category != null && c.CategoryID == this.category.CategoryID ? "#E0EEFA" : "white")">
          @c.Name
          <span style="float:right;" class="glyphicon glyphicon-remove-circle" @onclick="@(() => RemoveCategory(c))"></span>
        </li>
      }
    </ol>
  </div>
}

<!-- ### Spalte 2: Aufgabenliste-->
<div class="WLPanel @(this.task==null ? "col-xs-8 col-sm-8 col-md-9 col-lg-10 ": "hidden-xs col-sm-6 col-md-5 col-lg-6"  )">
  <!-- ---------- Überschrift Spalte 2-->
  <h4 id="TaskHeadline">@(taskSet == null ? 0 : taskSet.Count()) <span>Tasks in</span><i> @category?.Name</i></h4>
  <!-- ---------- neue Aufgaben eingeben-->
  <input id="NewTask" type="text" class="form-control" @bind="newTaskTitle" placeholder="new Task..." />
  <!-- ---------- Aufgabenliste ausgeben-->
  @if (taskSet != null)
  {
    <ol class="list-group scroll">
      @foreach (var t in taskSet)
      {
        <li class="list-group-item" @onclick="@(() => ShowTaskDetail(t))" style="@(t.TaskID == this.task?.TaskID ? "Background-color:#E0EEFA" : ↵
                                                                                    "Background-color:white")" title="Task #@t.TaskID">
          <span style="float:right;" class="glyphicon glyphicon-remove-circle" @onclick="@(() => RemoveTask(t))"></span>
          <input type="checkbox" name="@("done" + t.TaskID)" id="@("done" + t.TaskID)" checked="@t.Done" class=" MLcheckbox" ↵
                                                          @onchange=@(eventArgs => { t.Done = (bool)eventArgs.Value; ChangeTask(t);  }) />
          <b>@t.Title</b>
...
        </li>
      }
    </ol>
  }
```

Listing 4: Index.razor.cs

```
using BL;
using BO;
...
namespace Web.Pages
{
  public partial class Index
  {
    int userID = 0;
    string status { get; set; } = "Starting...";
    List<Category>  categorySet;
    List<BO.Task>  taskSet;
    Category category;
    BO.Task task;

    string _newCategoryName;
    public string newCategoryName  {
      get { return _newCategoryName; }
      set {
      if (!String.IsNullOrEmpty(value)) CreateCategory(value);
      _newCategoryName = "";
      Util.Log("_newCategoryName");
    }
  }

...

  #region Standard-Ereignisse
  protected async override System.Threading.Tasks.Task OnInitializedAsync()
{
...
    ShowCategorySet();
  }

  protected override async System.Threading.Tasks.Task 7
                                        OnAfterRenderAsync(bool firstRender)  {
    this.status = "";
  }
  #endregion

  #region Reaktionen auf eigene Ereignisse
  public void ShowTaskSet(Category c)  {
    this.category = c;
    this.taskSet = new TaskManager(userID).GetTaskSet(c.CategoryID);
  }
  public void ShowTaskDetail(BO.Task t)  {
    this.task = t;
  }
  public void CreateCategory(string newCategoryName)  {
    if (string.IsNullOrEmpty(newCategoryName)) return;
    var cat = new CategoryManager(userID).CreateCategory(newCategoryName);
    ShowCategorySet();
    ShowTaskSet(cat);
  }
  public void ChangeTask(BO.Task t)  {
    new TaskManager(userID).ChangeTask(t);
  }
...
  public void ReloadTasks(int taskID)  {
    ShowTaskSet(this.category);
    this.task = null;
    this.StateHasChanged(); // notwendig hier!
  }
  #endregion
}
```

Die für die Interoperabilität notwendige Klasse, die die Schnittstelle IJSRuntime realisiert, erhält die BlazorUtil-Klasse via Dependency Injection von der Blazor-Infrastruktur. Da die selbst geschriebene BlazorUtil-Klasse auch per Dependency Injection in die Razor Components kommt, muss der Entwickler diese Klasse in der Methode `ConfigureServices()` in der Klasse Startup beim DI-Container anmelden: `services.AddScoped<BlazorUtil>()`.

Blazor ist zustandsbehaftet

Während klassische serverseitige Webanwendungen zustandslos sind und der Webserver nach jeder Antwort alle Zustände vergisst, muss sich der Webentwickler bei Blazor um die Beibehaltung der Zustände von in Properties gespeicherten primitiven Werten oder komplexen Objekten nicht kümmern. Sie bleiben erhalten, solange die Razor Component zum aktuellen Render-Baum gehört. Eine Komponenteninstanz gehört zum Render-Baum, wenn die Komponente

- entweder über eine Route direkt aktiviert ist,
- in eine andere Komponente eingebettet ist, die über eine Route aktiviert ist,
- in eine andere Komponente eingebettet ist, die in eine Komponente eingebettet ist, die über eine Route aktiviert ist.

Daher hat die Index.razor-Komponente nicht vergessen, was die geladenen Kategorien und Aufgaben sind, selbst wenn der Benutzer zwischenzeitlich mit der TaskEdit.razor-Komponente interagiert hat.

Listing 5: TaskEdit.razor

```
@attribute [Authorize]
@if (task != null)
{
  <div style="padding: 0">
    <div>
      <h4>Task</h4>
      <form>
        <!--Schaltflächen-->
        <button type="button" title="Änderungen speichern" @onclick="Save" class="btn btn-success">
          <span class="glyphicon  glyphicon-floppy-save"></span>

          <span class="hidden-xs" i18n>Save</span>
        </button>
        <button type="button" title="Änderungen verwerfen" @onclick="Cancel" class="btn btn-warning">
          <span class="glyphicon glyphicon-remove"></span>

          <span class="hidden-xs" i18n>Cancel</span>
        </button>
        <!--Titel-->
        <label for="tasktitle">Titel</label>
        <input id="tasktitle" name="tasktitle" type="text" @bind="task.Title" required class="form-control" />
...
        <!--Fälligkeit-->
       <label for="taskDue" i18n>Due</label>
        <input id="taskdue" name="taskdue" type="date" @bind="task.DueNN" class="form-control" />
...
        <!--Notiz-->
        <label for="tasknote" i18n>Note</label>
        <textarea id="tasknote" name="tasknote " rows="5" @bind="task.Note" class="form-control "></textarea>
...
      </form>
    </div>
  </div>
}
```

Weitere Möglichkeiten

Aufgrund des begrenzten Platzes seien hier weitere Möglichkeiten von Blazor nur kurz erwähnt:

- Blazor Server und Blazor WebAssembly laufen in allen modernen Browsern. Der Internet Explorer benötigt bei Blazor Server allerdings ein Polyfill (siehe ix.de/zxka). Blazor WebAssembly läuft im Internet Explorer nicht.
- Das Debugging einer Blazor-Server-Anwendung ist in Visual Studio bei Laufzeitfehlern und manuellen Haltepunkten möglich wie bei jeder anderen .NET-Anwendung auch. Für Blazor WebAssembly arbeitet Microsoft noch an einer einfachen Debugging-Lösung.
- Alternativ zur direkten Verwendung einer Datenbank kann in Blazor Server ein Zugriff auf beliebige APIs und Webservices erfolgen. Blazor WebAssembly kann demgegenüber nur HTTP-/HTTPS-basierte Dienste (beispielsweise REST-Web-APIs, SOAP oder Google RPC) verwenden.
- Neben dem Aufruf von JavaScript aus C# heraus ist auch der umgekehrte Weg von JavaScript zu C# möglich.
- Alternativ zu den normalen HTML-Eingabesteuerelementen kann der Entwickler spezielle Blazor-Steuerelemente verwenden, die auch eine modellbasierte Validierung mit Datenannotationen erlauben.
- Zur Authentifizierung und Benutzerverwaltung kann man wahlweise den vorfertigten Baustein ASP.NET Core Identity verwenden, der nicht nur eine Benutzerdatenbank, sondern auch vorgefertigte, aber anpassbare Formulare für Benutzerregistrierung, Anmeldung, Datenverwaltung und Kennwort-vergessen-Fälle bereitstellt. Allerdings ist ASP.NET Core Identity noch nicht in Blazor, sondern als klassische Multi-Page-Anwendung realisiert, sodass es aus Benutzersicht zu einem Bruch zwischen den Benutzerverwaltungszeiten und der eigentlichen Anwendung kommt.
- Es gibt bereits eine größere Anzahl von Zusatzkomponenten für Blazor, sowohl von kommerziellen Anbietern als auch aus dem Open-Source-Bereich (siehe ix.de/zxka).

Listing 6: TaskEdit.razor.cs

```
...
namespace Web.Pages
{
  public partial class TaskEdit
  {
    [Parameter] public BO.Task Task { get; set; }
    [Parameter] public Action<int> TaskHasChanged { get; set; }
    [Inject] BlazorUtil Util { get; set; } = null;
    // wenn Parameter gesetzt wird
    protected override void OnParametersSet()  {
      Util.Log((nameof(OnParametersSet) + ": " + Task.TaskID));
    }
    protected void Save()  {
      Util.Log(nameof(Save) + ": " + Task.TaskID);
      new BL.TaskManager(userID).ChangeTask(this.Task);
      TaskHasChanged?.Invoke(Task.TaskID);
    }
    protected void Cancel()  {
      Util.Log(nameof(Cancel) + ": " + Task.TaskID);
      TaskHasChanged?.Invoke(Task.TaskID);
    }
  }
}
```

Listing 7: Ausschnitt aus BlazorUtil.cs

```
using Microsoft.JSInterop;
using System;
using System.Threading.Tasks;

namespace ITVisions.Blazor
{
  public class BlazorUtil
  {
    IJSRuntime _jsRuntime { get; set; }
    // JSRuntime via DI
    public BlazorUtil(IJSRuntime jsRuntime)  {
      _jsRuntime = jsRuntime;
    }

    public async void Log(object o)  {
      // Aufbereitung für Ausgabe
      string s;
      if (o.GetType().IsPrimitive || o is string) s = o.ToString();
      else {
        s = o.GetType().Name + " {" + ⤸
                    o.ToNameValueString(attributeSeparator: ";") + "}";
      }

      if (_jsRuntime == null || ⤸
                _jsRuntime.GetType().FullName.Contains("WebAssembly"))
      { // Blazor Webassembly!
        Console.WriteLine(s); return;
      }
      else { // Blazor Server
        try    {
          await _jsRuntime.InvokeAsync<string>("console.log", ⤸
                                                    "LOG: " + s);
        }
        catch (Exception) {
          // macht nichts, es kann in Blazor Server sein,
          // dass die JSRuntime noch nicht verfügbar ist!
        }
      }
    }
...
  }
}
```

Fazit

Blazor Server ist eine neue Softwarearchitektur für die effiziente Entwicklung moderner Webanwendungen mit einer kleinen bis mittleren Nutzeranzahl, wenn Offlinefähigkeit verzichtbar ist. Eine interessante Perspektive ist, diese beiden Beschränkungen ab Mai 2020 durch einen Wechsel zu Blazor WebAssembly überwinden zu können. *iX* wird berichten. (odi@ix.de)

Quellen

[1] Holger Schwichtenberg; Die 100-Millionen-Dollar-App; Webentwicklung mit Angular, Teil 1: Grundgerüst, Zugriff auf Webservices, Templates; *iX* 5/2017, S. 58

[2] Holger Schwichtenberg; Formulargedöns; Webprogrammierung mit Angular, Teil 2; *iX* 6/2017, S. 128

[3] Holger Schwichtenberg; Feinschliff; Webprogrammierung mit Angular, Teil 3; *iX* 7/2017, S. 122

[4] Dokumentation, Erweiterungen, Polyfill: ix.de/zxka



Dr. Holger Schwichtenberg

leitet das Expertennetzwerk www.IT-Visions.de, das Beratung, Schulungen und Softwareentwicklung im Umfeld von Microsoft-, Java- und Webtechniken anbietet. Er schreibt Software in C# und JavaScript/TypeScript, hält Vorträge auf Fachkonferenzen und ist Autor zahlreicher Fachbücher.



© fotolia, Kurt Kleemann

heise Security Tour 2020

Wissen schützt

ONLINE-KONFERENZ 18. Juni

Was tun, wenn's brennt – richtige Vorbereitung ist mehr als die halbe Miete

Auszug aus dem Programm

- *IT-Security im Überblick – damit haben wir es akut zu tun, das kommt auf uns zu*
  (Jürgen Schmidt)
- *IT-Sicherheitsgesetz 2.0 – Grundlagen und Umsetzung in der Praxis*
  (Wilhelm Dolle / Christoph Wegener)
- *Notfall-Planung – so bereiten Sie sich und Ihre Kollegen richtig vor*
  (Manuel Atug / Lukas Reike-Kunze)
- *Anatomie eines Datenschutz-GAUs*
  (Joerg Heidrich)
- *Monitoring und Einbruchserkennung – ein Überblick zu Markt & Techniken*
  (Stefan Strobel)
- *Forensik und Incident Response: Möglichkeiten und Grenzen der Spurensuche*
  (Björn Schemberger)

Richtig Vorbeugen – Sinnvoll Eingreifen – Aus Erfahrung lernen

www.heise-events.de/securitytour

© Adobe Stock/alphaspirit

Partner: F-Secure

Eine Veranstaltung von: heise Security

Organisiert von: heise Events



Javas jüngste Vergangenheit

(R)Evolution von 8 bis 14

Michael Inden

Viele setzen ungeachtet interessanter Neuerungen in den letzten Versionen immer noch auf Java 8. Vermutlich ist das Festhalten dadurch zu erklären, dass, als sie auf Java 9 umsteigen wollten, Oracle eine halbjährliche Releasestrategie angekündigt und damit für Verwirrung gesorgt hat.

Unternehmen scheuten die Komplikationen durch das mit Java 9 neu eingeführte Modulsystem. Die Verunsicherung wuchs, als mit Java 11 auch noch eine neue Lizenzpolitik kam und Java (Oracles JDK) seitdem im produktiven Umfeld kostenpflichtig ist. Dieser Artikel zeigt, dass es trotz dieser Hindernisse einiges gibt, was sich zu entdecken lohnt. Angefangen bei nützlichen Vereinfachungen in der Syntax, gewürzt mit diversen Erweiterungen in den APIs, bis hin zur Möglichkeit, Java-Applikationen in einzelne Module zu gliedern und sauber zu strukturieren. Doch eins nach dem anderen.

Gerade wenn Programme umfangreicher werden, hilft Modularisierung dabei, die Komplexität und Abhängigkeiten der Software besser unter Kontrolle zu halten. Dazu wird ein Programm in einzelne, idealerweise voneinander möglichst unabhängige Bestandteile in Form von Modulen unterteilt. Eine solche saubere Strukturierung ist für das JDK selbst erstrebenswert. Beides wurde mit Java 9 umgesetzt. Was Entwickler*innen für nötig halten, aber für Migrationen vermissen, ist die Unterstützung durch IDEs und Tools, die leider nur rudimentäre Hilfen dabei bieten, neue Module zu erstellen.

iX-TRACT

- Die letzten Java-Versionen sorgten vor allem in den Bereichen HTTP/2 und Concurrency für eine Modernisierung.
- Die Syntaxerneuerungen var und Records halten den Sourcecode kurz und knackig.
- Viele der jüngst eingeführten Features lassen sich gewinnbringend kombinieren.

Was nebenherläuft

Auch im Bereich Concurrency tut sich etwas. Niemand möchte heute mehr auf Low-Level-Konzepten mit `Thread` und `Runnable` direkt programmieren, zumal es nicht einfach wäre, das korrekt umzusetzen. Zur besseren Auslastung von Multi-Core-Prozessoren ist es für Concurrency erstrebenswert, sich von den feingranularen und recht primitiven Ansätzen mit Threads zu lösen.

Vielmehr bieten sich High-Level-Frameworks an. Doug Lea hat hier einen enormen Beitrag geleistet. Executors, das Fork-

Join-Framework, die nützliche, wenn auch mit leicht überfrachteter API versehene Klasse `CompletableFuture` und auch die mit JDK 9 eingeführten Reactive Streams gehen auf sein Konto.

Mittlerweile hat sich HTTP/2 etabliert und löst langsam, aber sicher HTTP 1.1 ab. Lange Zeit fehlte im JDK dazu eine Integration. Nach einer ersten Version in Java 9, jedoch in einem Inkubatorstatus, hat Oracle die HTTP/2-API mit Java 11 vollwertig ins JDK aufgenommen. Die Schnittstelle bringt ein verständliches Programmiermodell mit, das sich an den Konzepten wie Request und Response einer HTTP-Kommunikation orientiert.

Syntax- und API-Erweiterungen

In JDK 9 bis 14 finden sich umfangreiche Erweiterungen in den APIs, etwa bei der Prozesssteuerung, in der `Stream`-API, bei `Optional` sowie in vielen anderen Bereichen. Vor allem die letzten Versionen, also JDK 12 bis 14, haben für diverse Neuerungen in der Syntax gesorgt. Einige davon standen und stehen zunächst als Preview-Feature zur Verfügung. So lassen sich Erfahrungen sammeln und die finale Version weist möglichst viel Praxistauglichkeit auf. Besonders erwähnenswert sind hier die neue, viel klarere Syntax bei `switch` sowie die mehrzeiligen Strings. Daneben lassen sich Datencontainerklassen mithilfe des Schlüsselworts `record` prägnant definieren. Praktischerweise kann man viele APIs kombinieren, wie Beispiel 2 später mit der HTTP/2-API, der `Stream`-API und `Optional` verdeutlicht.

Listing 1: Methoden parallelisieren

```
public void execute()
{
    List<String> data = retrieveData();

    Long value1 = processData1(data);
    Long value2 = processData2(data);
    Long value3 = processData3(data);

    String result = calcResult(value1, value2, value3);
    System.out.println("result: " + result);
}
```

Listing 2: -- CompletableFuture

```
public void execute() throws InterruptedException, ExecutionException
{
    CompletableFuture<List<String>> cFData = CompletableFuture. ↵
                                    supplyAsync(() -> retrieveData());

    // Auswertungen parallel ausführen
    CompletableFuture<Long> cFValue1 = cFData.thenApplyAsync(data ↵
                                                -> processData1(data));
    CompletableFuture<Long> cFValue2 = cFData.thenApplyAsync(data ↵
                                                -> processData2(data));
    CompletableFuture<Long> cFValue3 = cFData.thenApplyAsync(data ↵
                                                -> processData3(data));

    // Einzelergebnisse synchron zusammenführen
    String result = calcResult(cFValue1.get(), cFValue2.get(), ↵
                                                        cFValue3.get());
    System.out.println("result: " + result);
}
```

Beispiel 1: Concurrency

Zwar gibt es seit Java 5 im JDK das Interface Future, allerdings führt es durch seine Methode `get()` oft zu blockierendem Code. Abhilfe schafft seit JDK 8 die Klasse `CompletableFuture`, die die Definition asynchroner Verarbeitungen auf höherer semantischer Ebene als mit `Runnable` oder `Callable` ermöglicht.

Angenommen, man möchte die folgende Methode parallelisieren, die zunächst Daten von einem Server liest, sie dann auf drei unterschiedliche Arten verarbeitet und schließlich zu einem Resultat zusammenführt (Listing 1).

Bevor man an Parallelisierung denkt, sollte man zunächst die Businessfunktionalität sauber in für sich testbaren Methoden kapseln. Wollte man das Ganze auf Low-Level parallelisieren, möchte man sich die ganzen Mühen und Fehlerquellen mit Threadsynchronisierung, Abstimmung der Aufgaben usw. gar nicht vorstellen. Glücklicherweise braucht es das gar nicht, weil `CompletableFuture` erlaubt, auf viel höherer Abstraktionsebene zu arbeiten. Dazu ummanteln Entwickler*innen die ursprünglichen Aufrufe nur mit geeigneten Methoden, um eine parallele Ausführung zu erzielen. `-- CompletableFuture` kümmert sich um die ansonsten notwendigen und schon angedeuteten lästigen Details (Listing 2).

Mit Java 9 lassen sich zusätzlich Time-outs und Exception Handling ergänzen und Fallback-Werte bereitstellen (Listing 3).

Beispiel 2: HTTP/2 + Optional + Streams + Date and Time API + var

Stellen Sie sich vor, Sie sollen die Wechselkurse für einen Zeitabschnitt von einigen Monaten berechnen. Im Netz lassen sich diese unter https://exchangeratesapi.io/ per REST abfragen. Hierzu kann man die HTTP/2-API gewinnbringend nutzen. Dazu schreiben Entwickler*innen eine Methode `performGet()`. Um den Zeitabschnitt zu definieren, nutzt man die Neuerung `datesUntil()` und erzeugt einen `Stream<LocalDate>`. Das geschieht hier durch die Angabe einer `Period` mit monatlicher Schrittweite. Mit einem

mitp-Verlag
IT • Web • Fotografie • Online-Marketing



504 Seiten | 39,99 €
ISBN 978-3-7475-0176-4
www.mitp.de/0176



656 Seiten | 22,99 €
ISBN 978-3-7475-0194-8
www.mitp.de/0194



456 Seiten | 29,99 €
ISBN 978-3-7475-0123-8
www.mitp.de/0123

www.mitp.de

Listing 3: Time-outs und Exception Handling

```
CompletableFuture<List<String>> cFData = CompletableFuture. ⤷
                    supplyAsync(() -> retrieveData()).exceptionally ⤷
                            ((throwable) -> Collections.emptyList());

CompletableFuture<Long> cFValue3 = cFData.thenApplyAsync(data -> ⤷
       processData3(data)).completeOnTimeout(7L, 2, TimeUnit.SECONDS);
```

`TemporalAdjuster` lässt sich jeweils an das Monatsende springen und der GET-Aufruf ausführen. Dabei kommt die Neuerung `var` zum Einsatz und ersetzt die Nutzung eines konkreten Typs. Seit Java 9 bietet `Optional<T>` schließlich die Methode `ifPresentOrElse()`, um sowohl eine Aktion im Positiv- als auch Negativfall ausführen zu können (Listing 4).

Wer das obige Programm ausführt, erhält die Wechselkurse von Schweizer Franken zu Euro von Januar 2019 bis April 2020. Das sieht gekürzt etwa aus wie in Listing 5.

Beispiel 3: Switch

Mit Java 14 und den Switch Expressions vereinfacht sich die Formulierung von Fallunterscheidungen deutlich und bietet eine intuitive Schreibweise (Listing 6).

An diesem Beispiel lassen sich einige syntaktische Neuerungen ablesen. Neben dem offensichtlichen Pfeil statt des Doppelpunkts lassen sich nun auch mehrere Werte hinter dem `case` angeben. Praktischerweise entfällt der `break`. Die hinter dem Pfeil notierten Anweisungen laufen jeweils nur spezifisch für das `case` ab und bei dieser Syntax existiert kein Fall `Through`. Außerdem kann das `switch` nun einen Wert zurückgeben, wodurch sich die Definition von Hilfsvariablen vermeiden lässt. Eine grandiose Vereinfachung, es stellt sich nur die Frage, weshalb das jetzt erst möglich ist.

Beispiel 4: Text Block (Preview-Feature in Java 13 und 14)

Lange hat die Java-Gemeinde auf die einfache Definition mehrzeiliger Strings gewartet. Jetzt ist diese als Preview-Feature in Java 14 enthalten. Dadurch lassen sich mehrzeilige Strings ohne mühseliges Verknüpfen definieren. Man kann außerdem auf ein fehlerträchtiges Escaping verzichten. Das erleichtert unter anderem den Umgang mit SQL-Befehlen sowie JSON in Java-Sourcecode. Intern werden Text Blocks wie normale Strings gehandhabt, weshalb man alle bekannten String-Methoden aufrufen kann. Die mehrzeiligen Strings sind mit je drei Anführungszeichen eingerahmt, wobei der eigentliche Inhalt auf einer neuen Zeile beginnen muss.

```
System.out.println("""
                   -Ich bin ein-
                   -Text Block-
                   """);
```

Das führt zu einer eingerückten Ausgabe, die sich an den unteren drei Anführungszeichen ausrichtet, und einem zusätzlichen Zeilenumbruch am Ende (hier nicht sichtbar).

```
-Ich bin ein-
-Text Block-
```

Das folgende Codebeispiel ist ebenfalls denkbar.

```
System.out.println("""
                   First 'line' simple quotes
                   Second "line" double quotes
                   Third line \""" three quotes
                   Fourth line no quotes, just \\ :-)""");
```

In Bezug auf JSON sähe das Ganze aus wie in Listing 7.

Beispiel 5: Records (Preview-Feature in Java 14)

Mit Java 14 halten als Preview sogenannte Records Einzug. Sie erlauben, auf extrem kompakte Weise spezielle Datencontainerklassen zu erzeugen. Records lassen sich in diversen Anwendungsfällen gewinnbringend einsetzen, etwa für Methoden, die mehrere Werte zurückgeben sollen, Compound Keys für Maps, DTOs, Parameter Value Objects usw. Eine einfache Definition lautet:

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

Listing 4: ifPresentOrElse()

```
public static void main(String[] args) throws Exception
{
    // HTTP/2-API + Date and Time API
    HttpClient httpClient = HttpClient.newBuilder().connectTimeout(Duration ⤷
                                                    .ofSeconds(5)).build();

    // Date and Time API => Stream mit monatlicher Schrittweite
    LocalDate startDay = LocalDate.of(2019, Month.JANUARY, 1);
    LocalDate endDay = LocalDate.of(2020, Month.MAY, 1);
    Period stepOneMonth = Period.ofMonths(1);

    // DATESUNTIL
    startDay.datesUntil(endDay, stepOneMonth).forEach(localDate ->
    {
        // TEMPORALADJUSTERS
        LocalDate endOfMonth = localDate.with(TemporalAdjusters. ⤷
                                                    lastDayOfMonth());

        // var seit Java 10 und Aufruf mit HTTP/2-API
        var optResponseBody = performGet(httpClient, endOfMonth);

        // OPTIONAL
        optResponseBody.ifPresentOrElse((value) -> ⤷
                        System.out.println(endOfMonth + " reported " + value),
                        () -> System.out.println("No data for " + endOfMonth));
    });
}

private static Optional<String> performGet(HttpClient httpClient, LocalDate ⤷
                                                                desiredDate)
{
    try
    {
        HttpRequest httpRequest = HttpRequest.newBuilder().GET().uri(URI. ⤷
                    create("https://api.exchangeratesapi.io/" + desiredDate + ⤷
                                                    "?symbols=CHF")).build();

        var response = httpClient.send(httpRequest, BodyHandlers.ofString());
        String responseBody = response.body();

        return Optional.of(responseBody);
    }
    catch (Exception ex)
    {
        // LOG
    }
    return Optional.empty();
}
```

Listing 5: GET-Aufruf ausführen

```
2019-01-31 reported {"rates":{"CHF":1.1409},"base":"EUR", 7
                                                "date":"2019-01-31"}
2019-02-28 reported {"rates":{"CHF":1.1335},"base":"EUR", 7
                                                "date":"2019-02-28"}
2019-03-31 reported {"rates":{"CHF":1.1181},"base":"EUR", 7
                                                "date":"2019-03-29"}
2019-04-30 reported {"rates":{"CHF":1.1437},"base":"EUR", 7
                                                "date":"2019-04-30"}
…
2020-02-29 reported {"rates":{"CHF":1.0614},"base":"EUR", 7
                                                "date":"2020-02-28"}
2020-03-31 reported {"rates":{"CHF":1.0585},"base":"EUR", 7
                                                "date":"2020-03-31"}
2020-04-30 reported {"rates":{"CHF":1.0582},"base":"EUR", 7
                                                "date":"2020-04-07"}
```

Listing 6: Switch Expression

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

Listing 7: JSON

```
String jsonObj = """
                 {
                     "name"    : "Mike",
                     "birthday": "1971-02-07",
                     "comment" : "Text blocks are nice!"
                 }
                 """;
```

Durch dieses Konstrukt lassen sich implizit aus den Konstruktorparametern die Attribute und die dazu notwendigen Implementierungen von rein lesenden Accessor-Methoden ableiten. Zudem werden die Implementierungen von `equals()` und `hashCode()` automatisch und vor allem kontraktkonform erzeugt. Gleiches gilt für einen passenden Konstruktor und die Methode `toString()`. Listing 8 veranschaulicht, wie viel Sourcecode man schreiben müsste, um eine äquivalente Funktionalität mit den bisherigen Java-Sprachmitteln zu erreichen.

Entwickler*innen können weitere Container erzeugen, die entweder zur Parameterübergabe oder als Rückgabe dienen. Beispielsweise:

```
record ColorAndRgbDTO(String name, int red, int green, int blue) { }
record IntStringReturnValue(int code, String info) { }
```

Die Java-Versionen 9 bis 14 enthalten eine Vielzahl sinnvoller und praktischer Neuerungen und Ergänzungen. Viele alltägliche Programmieraufgaben lassen sich dadurch noch ein wenig eleganter und kürzer als noch mit Java 8 realisieren. Während die Java-Releases 10, 12 und 13 sowie selbst Java 11 als LTS-Version bezüglich ihrer Neuerungen doch eher überschaubar sind, ist vor allem Java 14 ein großer Schritt nach vorn. Als Beispiele haben wir die praktischen Syntaxänderungen bei `switch` sowie die als Preview inkludierten mehrzeiligen Strings, Text Blocks genannt, kennengelernt. Records hingegen überragen diese sogar noch. Sie vereinfachen unter anderem, DTO-Klassen für Parameter Value Objects oder Tupel kombinierte Rückgabetypen zu definieren.

Die einschneidendste Neuerung, die Java mit Version 9 erfahren hat, ist jedoch die Modularisierung. Zwar besitzt Java endlich ein Modulsystem, aber leider findet es weder in der Java-Community noch bei den Toolherstellern richtig Anklang. Dadurch verstärkt sich beides negativ und die Verbreitung hält sich bis dato in Grenzen. Auch einige Inkompatibilitäten durch Änderungen bezüglich Reflection sowie nicht mehr zugänglicher Klassen des JDKs gestalten die Transformation bestehender Applikationen in ein modularisiertes System ziemlich holprig. In der

Listing 8: toString()

```
public final class MyPoint
{
    private final int x;
    private final int y;

    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

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

        MyPoint point = (MyPoint) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode()
    {
        return Objects.hash(x, y);
    }

    @Override public String toString()
    {
        return "MyPoint[x=" + x + ", y=" + y + "]";
    }

    // lesende Zugriffsmethoden auf x und y
}
```

Praxis bietet sich der Einsatz der Modularisierung daher vermutlich eher bei Neuentwicklungen und weniger für Migrationen bereits bestehender Applikationen an. Wer das Modulsystem nicht verwenden will, kann auf einen Kompatibilitätsmodus zurückgreifen. So lassen sich alle Vorteile der Syntaxneuerungen und API-Erweiterungen in vollen Zügen genießen.

Insgesamt sind die neuen Java-Versionen, insbesondere die 14, würdige Nachfolger von Java 8 und sollten die Java-Programmierung auch für die Zukunft für viele Entwickler und Firmen attraktiv machen. Vor allem im privaten Bereich profitiert man von den schnellen Releasezyklen, da in fast jeder neuen Release mindestens eine positive Überraschung steckt.

Hallo Zukunft

Kaum hat Java 14 im März 2020 das Licht der Welt erblickt, sind hinter den Kulissen die Arbeiten an der Septemberrelease von Java 15 in vollem Gange. Derzeit sind fünf JEP (Java Enhancement Proposals) dafür vorgesehen. Drei davon, nämlich Hidden Classes, ZGC und Shenandoah, betreffen Frameworks und die Garbage Collection. Allerdings schenken die meisten Entwickler*innen ihnen wohl kaum Beachtung. Ganz anders sieht es aber mit den zwei anderen JEPs aus. Leider muss das JDK wohl künftig ohne JavaScript-Engine auskommen. Schade, damit ließen sich dynamische Berechnungen recht elegant lösen. Schließen wir mit dem positiven Ausblick, dass die Text Blocks mit Java 15 zum Standard gehören werden. (csc@ix.de)

Quellen

Alle Listings unter ix.de/zxsc



Michael Inden

ist unter anderem als Fachbuchautor tätig. Seine Bücher wie „Java – Die Neuerungen in Version 9–14“ sind alle im dpunkt.verlag erschienen.



Einstieg in Kotlin

Klassischer Ansatz – neu gedacht

René Preißel

Kotlin ist im letzten Jahr zur bevorzugten Sprache in der Android-Entwicklung aufgestiegen und in der Backend-Entwicklung auf der Java Virtual Maschine (JVM) nicht mehr wegzudenken. Das liegt an der kompakten, typsicheren und trotzdem verständlichen Syntax, die genau an den richtigen Schmerzpunkten von Java ansetzt.

Das Kotlin-Team hat insgesamt nur wenige neue Konzepte eingeführt, sodass der Umstieg von Java sehr schnell möglich ist. Entsprechend ist man in wenigen Tagen produktiv und kann mit moderner Syntax arbeiten. Der folgende Artikel dient als Einstieg in Kotlin und konzentriert sich auf die Unterschiede zu Java und einige neue Konzepte. Als Voraussetzung sollten Entwickler*innen mit anderen klassenbasierten Programmiersprachen wie Java, C#, C++, JavaScript oder TypeScript vertraut sein. Begriffe wie Klassen, Interfaces, Properties und Funktionen sollten keine Fremdwörter sein. Als Beispiel fungiert eine kleine Webanwendung für das Spiel Stein, Papier, Schere. Dafür sind grundlegende HTML-/HTTP-Kenntnisse notwendig. Die folgenden Codebeispiele laufen auf der JVM und die vollständigen Quellen sind auf GitLab zu finden.

Kotlin-Dateien enden auf `.kt`. Wie in Java werden Projekte in Paketen (Packages) organisiert, wobei die Paketstruktur nicht mehr zwangsweise mit der Verzeichnisstruktur übereinstimmen muss. Daneben verzichtet Kotlin auf die Java-Restriktion, nur eine öffentliche Klasse pro Datei abzulegen. Vorhandene Java-Klassen lassen sich benutzen, was einen riesigen Pool an Bibliotheken schafft. Auch die Gegenrichtung (Einbindung von Kotlin-Code in Java-Programme) ist in den meisten Fällen möglich. Das bedeutet, dass man innerhalb eines Projektes gleichzeitig mit Java und mit Kotlin arbeiten kann. Namenskonventionen für Klassen (`Person`), Variablen (`name`) und Funktionen (`berechneAlter()`) sind ebenso gleich geblieben. Kotlin verfügt über die gleichen Grunddatentypen wie Java: `Int`, `Float`, `Boolean` etc. Man erkennt

iX-TRACT

- Als knappe und prägnante Alternative zu Java läuft Kotlin auf der JVM.
- Kotlin-Code lässt sich mit Java-Code kombinieren, das ermöglicht Zugriff auf den Riesenfundus bestehender Java-Bibliotheken.
- Mithilfe des Frameworks Ktor lassen sich in Kotlin Webanwendungen erstellen.

Listing 1: SteinPapierSchere.kt

```
package de.e2.sps.start

enum class Wahl {
    STEIN, PAPIER, SCHERE
}

enum class Ergebnis {
    SPIELER_GEWINNT, COMPUTER_GEWINNT, UNENTSCHIEDEN
}

class Spiel(val spielerWahl: Wahl, val computerWahl: Wahl) {
    fun ermittleErgebnis(): Ergebnis {
        return when {
            spielerWahl == computerWahl -> UNENTSCHIEDEN
            spielerWahl == PAPIER && computerWahl == STEIN ->
                                                    SPIELER_GEWINNT
            spielerWahl == STEIN && computerWahl == SCHERE ->
                                                    SPIELER_GEWINNT
            spielerWahl == SCHERE && computerWahl == PAPIER ->
                                                    SPIELER_GEWINNT
            else -> COMPUTER_GEWINNT
        }
    }
}
```

jedoch hier schon am großen ersten Buchstaben, dass es sich um Klassen handelt. Kotlin unterscheidet nicht zwischen primitiven Datentypen und Objekttypen.

Klassen, Variablen und Funktionen

Wie bereits angekündigt geht es in diesem Artikel darum, das Spiel Stein, Papier, Schere zu entwickeln. Dabei sollen menschliche Spieler*innen gegen den Computer antreten. Am Anfang steht die einfache Spielmechanik. Spieler*in und Computer wählen jeweils einen Gegenstand aus. Anschließend gilt es, das Ergebnis zu ermitteln. Hier stehen drei Varianten zur Verfügung: Entweder der Mensch gewinnt, der Computer gewinnt oder das Spiel geht unentschieden aus. In Listing 1 ist der zugehörige Kotlin-Code zu sehen.

Es gibt zwei Aufzählungen (`enum class`) für die Gegenstände (`Wahl`) und das Ergebnis. Nach den Aufzählungen wird eine Klasse für das Spiel definiert. In Kotlin legt man den primären Konstruktor einer Klasse direkt hinter dem Klassennamen fest. Konkret bedeutet das, für das Spiel sollen die Spieler*innenwahl und die Computerwahl übergeben werden. Kotlin ist eine statisch typisierte Sprache. Alle Variablen und Parameter benötigen einen Typ. Typen für Funktionsparameter folgen getrennt durch einen Doppelpunkt auf den Namen. Im Fall des Spiels ist es jeweils der Enum `Wahl`.

Schreibt man vor den Namen eines Konstruktorparameters `val` oder `var`, wird für diesen Parameter automatisch eine Property angelegt. Der Konstruktorparameter wird als initialer Wert direkt zugewiesen. Der Unterschied zwischen `val` und `var` ist, dass `val` eine Read-only Property und `var` eine Read-write Property erzeugt. Der Compiler generiert für diese Properties jeweils ein Attribut, eine `Get`-Funktion und gegebenenfalls eine `Set`-Funktion. Zusätzlich lassen sich mit `var` und `val` weitere Properties ohne Konstruktorparameter im Klassenkörper einrichten.

Methoden (Funktionen) innerhalb des Klassenkörpers definiert das Schlüsselwort `fun`. Für die Klasse `Spiel` soll die Funktion `ermittleErgebnis()` zum Einsatz kommen. Sie berechnet, wer in diesem Spiel gewonnen hat. `ermittleErgebnis()` kommt ohne zusätzliche Parameter aus, da sie nur auf die Properties des Objektes zugreift. Der Returntyp einer Funktion wird nach den Parameterklammern mit einem Doppelpunkt getrennt angegeben.

In diesem Fall soll ein Ergebnis zurückgeliefert werden, das sich einfach ermitteln lässt. Eingangs vergleicht man, ob Mensch und Computer das Gleiche gewählt haben, dann ist es `UNENTSCHIEDEN`. Andernfalls werden die Gewinnerkombinationen für den Menschen geprüft. Treffen sie nicht zu, hat der Computer gewonnen. Für diese mehrfache Fallunterscheidung könnten Programmierer*innen eine `if-else`-Kaskade, wie aus anderen Programmiersprachen bekannt, benutzen. Als kompakte und lesbare Alternative bietet Kotlin allerdings dafür auch die `when`-Expression an. Innerhalb des `when`-Blocks wird auf der linken Seite eine Bedingung definiert und auf der rechten Seite nach dem Pfeil `->` das Ergebnis geliefert. Die gesamte Funktion wird mit dem Statement `return` verlassen.

Um die Spielmechanik zu testen, schreiben Entwickler*innen eine Funktion `main` (Listing 2). Kotlin akzeptiert anders als Java globale Funktionen. Sie lassen sich auf oberster Ebene außerhalb von Klassen mit `fun` definieren, die Funktion `main` markiert dabei wie in allen C-basierten Sprachen den Startpunkt für das Programm. Sie benötigt im hier vorliegenden Fall keine Aufrufparameter und liefert `Unit` zurück. Unit entspricht in etwa `void` aus Java oder C und bedeutet: nichts. Der Returntyp `Unit` kann auch wegfallen, da er der Default-Returntyp ist.

Innerhalb von `main()` definiert man eine lokale Variable `spiel`. Lokale Variablen lassen sich wie Properties mit `val` oder `var` ein-

Listing 2: Mit main die Spielmechanik testen

```
fun main(): Unit {
    val spiel = Spiel(STEIN, PAPIER)
    val spielerWahl = spiel.spielerWahl
    println("Spieler: $spielerWahl. Computer: ${spiel.computerWahl}.")
    println("Ergebnis: ${spiel.ermittleErgebnis()}.")
}
```

Mehr zu Klassen

Kotlin erlaubt Interfaces, abstrakte Basisklassen und generische Typparameter. Klassen verlangen allerdings eine explizite Markierung `open`, um von ihnen erben zu können. Jedes Überschreiben einer Funktion muss explizit mit `override` markiert werden. Die Klasse `Rechteck` zeigt, wie das generische Interface `Comparable<T>` implementiert wird.

Es ist außerdem möglich, Klassen mit `data` zu markieren (siehe `Rechteck`). Für diese Klassen generiert der Compiler weitere hilfreiche Funktionen, beispielsweise eine `copy`-Funktion, um Objekte zu kopieren und dabei nur einzelne Properties zu überschreiben, eine sinnvolle `toString`-Funktion, die alle Konstruktor-Properties ausgibt, sowie eine ==-Operation, die alle Konstruktor-Properties inhaltlich vergleicht.

Listing 3: Comparable<T> implementieren

```
data class Rechteck(val breite: Int, val hoehe: Int):
Comparable<Rechteck> {
    val flaeche: Int = breite * hoehe

    override fun compareTo(other: Rechteck): Int = this.flaeche
                                                   - other.flaeche
}

fun main() {
    val r1 = Rechteck(10, 20)
    val r2 = r1.copy(hoehe = 30)
    println(r2)       // Rechteck(breite=10, hoehe=30)
    println(r1 == r2) // false
}
```

In diesem Fall soll ein Ergebnis zurückgeliefert werden, das sich einfach ermitteln lässt. Eingangs vergleicht man, ob Mensch und Computer das Gleiche gewählt haben, dann ist es `UNENTSCHIEDEN`. Andernfalls werden die Gewinnerkombinationen für den Menschen geprüft. Treffen sie nicht zu, hat der Computer gewonnen. Für diese mehrfache Fallunterscheidung könnten Programmierer*innen eine `if-else`-Kaskade, wie aus anderen Programmiersprachen bekannt, benutzen. Als kompakte und lesbare Alternative bietet Kotlin allerdings dafür auch die `when`-Expression an. Innerhalb des `when`-Blocks wird auf der linken Seite eine Bedingung definiert und auf der rechten Seite nach dem Pfeil `->` das Ergebnis geliefert. Die gesamte Funktion wird mit dem Statement `return` verlassen.

Um die Spielmechanik zu testen, schreiben Entwickler*innen eine Funktion `main` (Listing 2). Kotlin akzeptiert anders als Java globale Funktionen. Sie lassen sich auf oberster Ebene außerhalb von Klassen mit `fun` definieren, die Funktion `main` markiert dabei wie in allen C-basierten Sprachen den Startpunkt für das Programm. Sie benötigt im hier vorliegenden Fall keine Aufrufparameter und liefert `Unit` zurück. Unit entspricht in etwa `void` aus Java oder C und bedeutet: nichts. Der Returntyp `Unit` kann auch wegfallen, da er der Default-Returntyp ist.

Innerhalb von `main()` definiert man eine lokale Variable `spiel`. Lokale Variablen lassen sich wie Properties mit `val` oder `var` ein-

Listing 4: repeat – Funktionen als Parameter

```
fun repeat(times: Int, action: (Int) -> Unit) {

for (index in 1..times) {
        action(index)
    }
}
fun printNumber(number: Int) {
    println(number)
}
fun main() {
    repeat(3, ::printNumber)
}
```

Listing 5: Lambda mit it

```
fun main() {repeat(3) { number ->
        println(number)
    }
    repeat(3) {
        println(it)
} }
```

Listing 6: Hello World mit Ktor

```
fun main() {
    println("Open with: http://localhost:8080/hello?count=3")
    val server = embeddedServer(factory = Netty, port = 8080) {
        //this ist Application
        routing {
            //this ist Routing
            get("/hello") {
                ...
            }
        }
    }
    server.start(wait = true)
}
```

Listing 7: countAsString

```
get("/hello") {
    //this ist PipelineContext
    val countAsString: String? = call.parameters["count"]
    val count = if (countAsString!=null) countAsString.toInt() else 3
    call.respondTextWriter {
        //this ist Writer
        repeat(count) {
            write("Hello World\n")
        }
    }
}
```

richten. Dabei gibt `val` vor, dass der Wert sich nicht ändern lässt (`immutable/final` in Java). Bei lokalen Variablen kann man optional den Datentyp hinter dem Namen spezifizieren. Haben Entwickler*innen keinen Datentyp explizit angegeben, ermittelt der Compiler den Typ automatisch anhand des zugewiesenen Wertes. Im Beispiel bekommt die Variable `spiel` ein neues Objekt vom Typ `Spiel`. In Kotlin fehlt das Schlüsselwort `new`, um Objekte anzulegen, stattdessen lässt sich der Konstruktor einfach wie eine Funktion aufrufen.

Die nächste Zeile demonstriert, wie einer weiteren lokalen Variablen die Property `spielerWahl` zugewiesen wird. Auf Properties lässt sich syntaktisch direkt zugreifen, ohne eine `Get`-Funktion explizit aufrufen zu müssen. Im Hintergrund generiert der Compiler den `Get`-Aufruf. Am Ende von `main` liefert `printIn()` auf der Konsole die Auswahl von Mensch und Computer sowie das Ergebnis. Dabei kann man gut sehen, wie sich innerhalb der Strings auf Variablen mittels `$` zugreifen lässt.

Funktionstypen und Lambdas

Bevor wir mit unserer Spieleentwicklung fortfahren, kommt ein kurzer Einschub, wie Kotlin mit Higher-Order-Funktionen umgeht. Dahinter stecken Funktionen, die wiederum Funktionen als Parameter bekommen oder zurückliefern. Dieses Konzept findet in Kotlin an vielen Stellen Verwendung, um den Code kompakt und lesbar zu definieren.

Listing 4 zeigt das Konzept von Funktionen als Parameter am Beispiel von `repeat`. Diese Funktion soll eine beliebige andere Funktion oder einen Codeblock (Lambda) wiederholt aufrufen. Sie lässt sich mithilfe zweier Parameter definieren. `times` ist einfach eine Zahl (`Int`) und legt die Anzahl der Wiederholungen fest. `action` hingegen weist einen komplizierteren Typ auf: `(Int) -> Unit`. Dieser Funktionstyp steht für eine beliebige Funktion, die eine Zahl (`Int`) übergeben bekommt und nichts (`Unit`) zurückliefert, beispielsweise die Funktion `printNumber`.

Innerhalb von `repeat` wird in einer `for`-Schleife von 1 bis `times` gezählt. In der Schleife wird die übergebene Funktion `action` wie eine normale Funktion mit runden Klammern aufgerufen und der aktuelle `index` übergeben. `printNumber` würde also dreimal mit `1,2,3` aufgerufen werden. Durch die Syntax `::printNumber` in Listing 4 lässt sich die Referenz auf die Funktion ermitteln.

Oft wollen Entwickler*innen Higher-Order-Funktionen nicht mit vorhandenen Funktionen aufrufen, sondern einfach ein Lambda übergeben. Wenn der Funktionstypparameter der letzte Parameter einer Funktion ist, wie bei `repeat`, dürfen Programmierer*innen den Codeblock nach den Parameterklammern angeben. Listing 5 zeigt das anschaulich. `repeat` wird über den Parameter `3` aufgerufen und anschließend ein Lambda nach den runden Klammern übergeben.

Lambdas kennzeichnen geschweifte Klammern `{...}`, ähnlich zu herkömmlichen Codeblöcken. Allerdings können sie nach der öffnenden Klammer ihre Parameter definieren (`number`) und nach einem Pfeil `->` den eigentlichen Code implementieren. Benötigt ein Lambda nur einen Parameter, wie im Beispiel mit `repeat`, können Entwickler*innen auf die explizite Definition des Parameters verzichten und `it` benutzen (Listing 5 unten). Mit diesen sogenannten Trailing Lambdas ermöglicht es Kotlin, eigene Kontrollstrukturen und domänenspezifische Sprachen (DSL) einzubinden. Ein gutes Beispiel für eine DSL ist das Webframework Ktor.

Webanwendungen mit Ktor bauen

Das Stein-Papier-Schere-Spiel soll im Web laufen. Dafür braucht es zunächst einen Webserver und die Möglichkeit, für HTTP-Requests WebHandler zu implementieren. Hier kommt Ktor als Framework für Webanwendungen ins Spiel. Es ist selbst in Kotlin geschrieben und besitzt eine gut lesbare DSL, um WebHandler zu konfigurieren und zu implementieren. Der Code für ein „Hello World"-Beispiel mit Ktor findet sich in Listing 6.

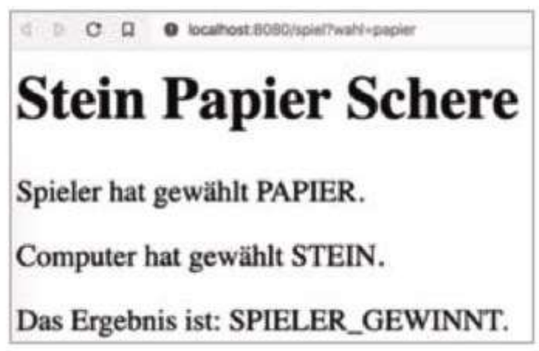
Ktor stellt die Funktion `embeddedServer` bereit, um einen Webserver zu starten. Sie bringt eine ganze Reihe von Parametern für die Konfiguration des Servers mit. Kotlin erlaubt, Funktionsparameter mit Defaultwerten zu versehen und beim Aufruf nur die Parameter zu übergeben, die man anpassen will. Dafür gibt es die Möglichkeit, Parameter mit Namen zu übergeben (Listing 6).

Der Parameter `factory` spezifiziert den zu nutzenden Server (Netty) und der Parameter `port` den Port (8080) des Webservers. Beide lassen sich mit Namen übergeben. `host` dagegen wird nicht gesetzt und bleibt deswegen beim Default: 0.0.0.0. Bei `named`-Parametern wirkt sich die Aufrufreihenfolge nicht aus, da der Name für die richtige Zuordnung sorgt.

Die Funktion `embeddedServer` bekommt noch ein Lambda übergeben, ähnlich wie `repeat` im letzten Beispiel. Entsprechend wird der Codeblock nach dem Funktionsaufruf an die Funktion `embeddedServer` übergeben. Beim Ausführen eines Lambdablockes lässt sich der `this`-Zeiger innerhalb des Blockes auf ein beliebiges Objekt setzen, was an Skriptsprachen wie JavaScript oder Groovy erinnert. In Listing 6 setzt `embeddedServer` für den übergebenen Block den `this`-Zeiger auf ein Objekt vom Typ `Application`.

Nun können Entwickler*innen innerhalb dieses Blockes direkt auf die Funktionen der `Application`-Klasse zugreifen. Beispielsweise ist die in der nächsten Zeile aufgerufene Funktion `routing`

Entwickler*innen müssen für die URL http://localhost:8080/spiel?wahl=papier einen entsprechenden Handler implementieren (Abb. 1).



Listing 8: Strings mit toWahl() erweitern

```
fun String.toWahl() = //this ist String
    when (toLowerCase()) {
        "stein" -> STEIN
        "papier" -> PAPIER
        "schere" -> SCHERE
        else -> null
    }
```

in der Klasse `Application` definiert. `routing` erhält wieder ein Lambda und der `this`-Zeiger wird auf ein `Routing`-Objekt gesetzt.

`Routing` stellt die Funktion `get()` bereit, um WebHandler (GET Request) für URLs (`/hello`) zu registrieren. Durch die Verschachtelung von Lambdas mit eigenen `this`-Zeigern entsteht eine hierarchische Struktur zum typsicheren Konfigurieren von WebHandlern. Schließlich wird der konfigurierte Server gestartet und der aktuelle Thread so lange blockiert, bis der Server beendet wird.

Nie wieder NullPointerExceptions

Im Hello-Beispiel soll der WebHandler für die URL http://localhost:8080/hello?count=3 den Text „Hello World“ so oft ausgeben, wie es der URL-Parameter `count` angibt. Falls er nicht vorhanden ist, soll `3` als Default angenommen werden.

Listing 7 zeigt die Implementierung des WebHandler. Innerhalb der `get`-Funktion steht der `this`-Zeiger auf dem Typ `PipelineContext`. Dieser Typ verfügt über eine Property `call`, um unter anderem auf die URL-Parameter `call.parameters` zuzugreifen. `parameters` ist eine Art Map (Key-Value-Paare) und man kann mit der `[]`-Notation einzelne Parameter per Name ermitteln. Als Ergebnis erhält man entweder einen String oder `null`.

Eine Besonderheit von Kotlin ist, dass die normalen Datentypen keine Nullwerte vertragen. Konkret heißt das, eine Variable vom Typ `String` kann kein `null` aufnehmen. Wer eine Variable `Nullable` definieren möchte, muss ein Fragezeichen `?` hinter den Typ schreiben: `String?`. Dabei entsteht ein neuer Datentyp, der `null` und Strings akzeptiert (Listing 7).

Der Zugriff auf eine Nullable-Variable muss immer mit einem vorhergehenden Nulltest geschehen, sonst meckert der Compiler. Eine Möglichkeit ist in Listing 7 zu sehen. Zuerst überprüft `if`, ob `countAsString` ungleich `null` ist. Die Funktion `toInt()` lässt sich lediglich im True-Zweig nutzen, um den String in eine Zahl umzuwandeln. Im False-Zweig wird der Defaultwert `3` übergeben.

Da man so einen Nulltest häufig beim Zugriff auf Objekte braucht, gibt es den Safe-Call-Operator `?.`. Er ruft die Funktion oder die Property nur auf, wenn das Objekt davor nicht null ist. Andernfalls liefert der ganze Ausdruck selbst `null` zurück, etwa:

```
val count: Int? = countAsString?.toInt()
```

Listing 9: WebHandler implementieren

```
get("/spiel") {
    val spielerWahl = call.parameters["wahl"]?.toWahl()
    if (spielerWahl == null) {
        call.respondText("Ungültige Wahl!")
    } else {
        val computerWahl = Wahl.values().random()
        val spiel = Spiel(spielerWahl, computerWahl)

        call.respondText(
            """
            <h1>Stein Papier Schere</h1>
            <p>Spieler hat gewählt ${spiel.spielerWahl}.</p>
            <p>Computer hat gewählt ${spiel.computerWahl}.</p>
            <p>Das Ergebnis ist: ${spiel.ermittleErgebnis()}.</p>
            """.trimIndent(), ContentType.Text.Html
        )
    }
}
```

Wer im Fall von `null` einen Defaultwert verwenden möchte, kann den Elvis-Operator `?:` nutzen. Dieser liefert den Ausdruck auf der rechten Seite zurück, wenn links ein Nullwert steht. So würde man in Listing 7 typischerweise anstelle des `if-else` schreiben:

```
val count = countAsString?.toInt() ?: 3
```

Mit der direkten Unterstützung von Nullable-Typen und dem Zwang, vor jedem Zugriff auf `null` zu prüfen, gehört eine der am häufigsten auftretenden Exceptions, `NullPointerException`, der Vergangenheit an. Das gilt natürlich nur, wenn man keinen unsicheren Java-Code aufruft.

Der Rest des Codes aus Listing 7 lässt sich schnell erklären. `call.respondTextWriter()` ist eine Funktion mit einem Lambda, in dem ein Writer als `this`-Zeiger fungiert. Alles, was man mit diesem Writer schreibt, landet als Text im Client oder im Browser.

Klassen erweitern

Anschließend gilt es, für das Stein-Papier-Schere-Spiel einen entsprechenden Handler für die URL http://localhost:8080/spiel?wahl=papier zu implementieren (Abbildung 1).

Über den URL-Parameter `wahl` lässt sich der Gegenstand übergeben. Im nächsten Codebeispiel findet sich das Grundgerüst dazu.

```
get("/spiel") {
    val wahlPara: String = call.parameters["wahl"] ?: "" ...
}
```

Anschließend müssen Entwickler*innen den String in den Enum `Wahl` umwandeln. Prinzipiell kann das jeder Enum mithilfe von `valueOf()` automatisch. Als lehrreiches Beispiel implementiert man diese Mapper-Funktion wie nachfolgend selbst.



```
fun stringToWahl(input: String): Wahl? {
    return when (input.toLowerCase()) {
      "stein" -> STEIN

"papier" -> PAPIER
      "schere" -> SCHERE
      else -> null
    }
}
```

Die globale Funktion `stringToWahl` zeigt eine mögliche Umsetzung. `input` ist ein String und das Ergebnis entweder eine `Wahl` oder `null`. Der Returntyp `Wahl?` veranschaulicht das. Intern kommt die Expression `when` zum Einsatz, um einen String in einen Gegenstand zu wandeln. Dieses Mal empfängt `when` einen Parameter, der mit den Strings in den nächsten Zeilen verglichen wird. Bei Gleichheit wird der Ausdruck nach dem `->` zurückgegeben.

Hier lohnt es sich, erneut einen Blick auf Listing 7 und die Stelle zu werfen, an der ein String in einen `Int` umgewandelt wurde: `countAsString.toInt()`. Das ist eleganter als `stringToWahl(wahlPara)`. Außerdem unterstützen IDEs Entwickler*innen besser darin, Funktionen wie `toInt()` zu finden. Typischerweise erscheinen alle Klassenmethoden nach Eingabe eines Punkts. Allerdings muss man `stringToWahl` bereits kennen, um sie aufrufen zu können. Es wäre schön, wenn sich die Klasse `String` um eine Funktion `toWahl()` erweitern ließe. Genau dafür sind Extension-Funktionen da, um beliebige Typen um weitere Methoden zu erweitern.

Listing 8 zeigt, wie man der Klasse `String` die Funktion `toWahl` hinzufügt. Dabei steht der Klassenname vor dem Funktionsnamen. Innerhalb von `toWahl` ist der `this`-Zeiger vom Typ `String`. Viele von den in den letzten Beispielen genutzten Funktionen, etwa `toInt`, sind Extension-Funktionen, keine normalen Klassenmethoden. Mithilfe von Extension-Funktionen lassen sich vorhandene APIs erweitern.

Zusätzlich zeigt `toWahl` zwei weitere syntaktische Besonderheiten von Kotlin. Da der Funktionsrumpf nur aus einer Expression `when` besteht, kann man sich die geschweiften Klammern inklusive `return` sparen und mit dem `=` das Ergebnis der Funktion zurückgeben. Außerdem ermittelt der Compiler den Rückgabetyp einer Single-Expression-Funktion automatisch, weswegen Entwickler*innen diesen nicht explizit angeben müssen.

Im nächsten Schritt veranschaulicht Listing 9 die endgültige Implementierung des WebHandler für unser Spiel. Nachdem der Parameter-String in eine `Wahl` umgewandelt wurde, steht die Prüfung an, ob die `spielerWahl` nicht `null` ist. Daraufhin holt die Funktion `values` alle Werte des Enums `Wahl`. Die Extension-Funktion `Array.random` wählt zufällig einen Gegenstand aus. Danach wird ein neues Spiel angelegt. Zum Schluss wird das Ergebnis als HTML formatiert und an den Client geschickt.

Der HTML-String wird mit drei Anführungszeichen eingeleitet und beendet. Diese sogenannten Raw Strings erlauben es, mehrzeilige Strings in den Code einzubetten.

Typsichere DSLs

Mit kotlinx.html gibt es eine DSL, um mit Kotlin-Syntax HTML zu schreiben. Listing 10 zeigt das anhand der Startseite für unser Spiel (Abbildung 2). Durch Trailing Lambdas und Extension kann man kompiliersicheren HTML-Code schreiben. `kotlinx.html` stellt für jedes HTML-Tag (`body`, `h1`, `form` usw.) eine Funktion bereit, der sich ein Lambda übergeben lässt. Innerhalb dieser Lambdas setzt man den `this`-Zeiger jeweils auf ein zugehöriges Objekt (`BODY`, `H1`, `FORM`).



Auf der Startseite wählen Spieler*innen zwischen den drei Möglichkeiten (Abb. 2).

Listing 10: kotlinx.html

```
get("/start") {
    call.respondHtml {
        body {
            h1 { text("Stein Papier Schere") }
            form(action = "/spiel", method = FormMethod.get) {
                // this ist FORM
                label { +"Deine Wahl:" }
                wahlButton("Stein")
                wahlButton("Papier")
                wahlButton("Schere")
            }
        }
    }
}

fun FORM.wahlButton(wahl: String) {
    button(type = ButtonType.submit, name = "wahl") {
        value = wahl.toLowerCase()
        +wahl
    }
}
```

Bei einem Blick auf das `h1`-Tag fällt in den geschweiften Klammern der eingebettete Text (`text(…)`) auf. Wem das auch noch zu viel Code ist, der kann, wie drei Zeilen darunter im `label`-Tag zu sehen, die Notation `+"..."` nutzen. Was im ersten Moment seltsam aussieht, ist die Möglichkeit, vorhandene Operatoren zu redefinieren. In `kotlinx.html` wurde der unäre Plusoperator für Strings neu definiert. Man kann sich darüber streiten, ob die Syntax in diesem Beispiel schön ist. Doch Kotlin hat viele der bekannten Operatoren (`==`, `<`, `>`) für Standardtypen definiert und ermöglicht damit kompakten und lesbaren Code. In Listing 10 kann man am Beispiel der Extension-Funktion `FORM.wahlButton()` nochmals gut sehen, wie man vorhandene DSLs um neue Funktionen und Komponenten erweitern kann.

Ausblick und Fazit

Kotlin unterstützt nicht nur Android und die JVM als Zielplattform, sondern seit Längerem auch schon JavaScript und noch nicht ganz so lange auch native Plattformen wie Windows, Linux, iOS etc. Insbesondere die Möglichkeit, gemeinsame Bibliotheken für verschiedene Plattformen zu entwickeln und trotzdem plattformabhängigen Code anzusprechen, ist gerade sehr im Kommen (Multiplatform Projects). Dabei geht es nicht darum, überall und immer mit Kotlin zu programmieren, sondern die gleichen Funktionalitäten nur einmal zu entwickeln und dann vom nativen Plattformcode anzusprechen.

Die Sprache ist schnell erlernt und das Tooling dank JetBrains sehr reif. Wenn man langjährigen Java-Entwickler*innen zum ersten Mal Kotlin zeigt, kommt häufig die Aussage: „Ja, es ist kompakter, doch ich kann das alles auch in Java machen, und das bisschen mehr Code ist doch kein Problem.“ Fragt man die gleichen Entwickler*innen nach dem ersten Kotlin-Projekt, ob sie nicht mal wieder Java-Code schreiben könnten, wollen die meisten nicht zurück. (csc@ix.de)

Quellen

Alle Listings unter ix.de/zbea



René Preißel

entwickelt seit über 20 Jahren mit Java, und seit ein paar Jahren hat er noch mehr Spaß beim Programmieren durch Kotlin.

Es gibt 10 Arten von Menschen.

iX-Leser und die anderen.



Jetzt Mini-Abo testen:
3 Hefte + Bluetooth-Tastatur
nur 16,50 €

www.iX.de/testen





Clojure: funktional programmieren auf der JVM

Lisp goes Java

Stefan Kamphausen

Die universelle Programmiersprache Clojure kombiniert die Vorteile der funktionalen Programmierung mit dem Angebot der Java-Welt.

Dieser Artikel soll einen Weg in die Entwicklung mit Clojure aufzeigen, erste Eigenschaften der Sprache erläutern und einen Überblick über Clojures Ökosystem geben.

Wie in jeder Programmiersprache gilt es, zunächst eine schlichte Entwicklungsumgebung einzurichten. Das ermöglicht erste Schritte und kleine Programme. Den einfachsten Weg, mit Clojure zu beginnen, bietet das Programm Leiningen. Die Installation besteht für Anwender*innen unter Linux und macOS aus dem Download eines simplen Bash-Skripts, das im `PATH` auffindbar sein muss. Alternativ lässt sich Leiningen unter macOS via Homebrew installieren (siehe ix.de/z1ah). Für Windows-Anwender*innen existiert eine Batchdatei. Im Anschluss erstellen die folgenden Aufrufe auf der Kommandozeile ein neues Projekt und starten eine REPL:

```
shell> lein new test-project
shell> cd test-project
shell> lein repl
```

Leiningen ist nicht nur ein Programm, das eine REPL starten kann, sondern auch ein vollwertiges Build-Werkzeug, vergleichbar zu Maven aus der Java-Welt. Das Clojure-Ökosystem hält neben Leiningen noch alternative Build-Werkzeuge bereit. Die neuen, aber bereits weitverbreiteten Clojure CLI Tools stellt das Team zur Verfügung, das Clojure entwickelt. Sie lassen sich ebenfalls mittels Homebrew und teilweise aus Linux-Paketquellen installieren (siehe ix.de/z1ah). Weniger verbreitet sind Boot und das Gradle-Plug-in Clojurephant. Während die Entwicklung von Boot derzeit ruht, steht mit Clojurephant Gradles gesamtes Repertoire zur Verfügung.

iX-TRACT

- Clojure ist eine funktionale Programmiersprache auf der JVM.
- Die Lisp-Syntax und die durchdachte Kombination verschiedener Eigenschaften machen Clojure elegant und leistungsfähig.
- Die Sprache eignet sich aufgrund des Zusammenspiels von unveränderlichen Datenstrukturen, Referenztypen und seiteneffektfreien Funktionen für nebenläufige Programmierung.

Interaktives Arbeiten

Das Kürzel REPL steht für Read-Eval-Print Loop, ein Programm, das die Eingabe der Anwender*innen liest, auswertet und das Ergebnis ausgibt. Die REPL ist insofern ähnlich zu einer Shell wie Bash oder PowerShell. Die Arbeit in der REPL erlaubt eine sehr direkte Verbindung zum eigenen Programm. Es ist üblich, in dieser Umgebung

Funktionen so lange manuell aufzurufen, bis sie wie gewünscht funktionieren. Das obligatorische „Hello, world“ sieht in der REPL wie folgt aus:

```
user=> (println "Hello, world")
Hello, world
nil
```

Die Eingabeaufforderung `user=>` enthält den Namen „user“, der sich auf den gerade verwendeten Namensraum bezieht. Clojure nutzt Namensräume (Namespaces), um benannte Dinge wie Funktionen voneinander zu trennen. Das Beispiel zeigt die ungewohnte und gelegentlich belächelte Lisp-Syntax: Die öffnende Klammer steht vor der Funktion. Entgegen der landläufigen Meinung nutzen Sprachen der Lisp-Familie nicht notwendig mehr Klammern im Quellcode als andere Sprachen – besonders Clojure nicht, das mit einigen Konventionen des viel älteren Common Lisp gebrochen hat und nochmals weniger Klammern verwendet.

Strings stellt Clojure wie gewohnt mit doppelten Anführungszeichen dar. Diese Strings entsprechen demselben String-Typ, wie ihn Java verwendet: `java.lang.String`. Das zeigt eine wichtige Eigenschaft von Clojure: Die Sprache ist bewusst in eine Hostumgebung eingebettet – die Java-Virtual-Machine (JVM). Die Interaktion mit der Hostumgebung ist ein wichtiger Aspekt des Clojure-Designs.

Programmierer*innen beenden ihre Eingabe am Prompt durch die Eingabetaste. Daraufhin liest die REPL die Eingabe, wertet sie aus (inklusive der Seiteneffekte) und stellt den Rückgabewert des Ausdrucks dar. Die Ausgabe von `"Hello, world"` im Beispiel ist ein Seiteneffekt, diejenige von `nil` ist der Rückgabewert der Funktion `println`:

```
user=> (+ 2 3)
5
user=> (+ 2 (* 3 4))
14
```

Hier addiert Clojure die Zahlen 2 und 3 und liefert das Ergebnis zurück. Die Ausgabe verlangt nicht den Aufruf von `println`. Das zweite Beispiel zeigt, wie geschachtelte Aufrufe aussehen. Das wirkt ungewohnt, hat jedoch den Vorteil, dass sich niemand eine Präzedenztabelle einprägen muss.

Listing 1: Funktionen definieren und aufrufen

```
user=> (defn hello [name]
         (str "Hello, " name))
#'user/hello
user=> (hello "world")
"Hello, world"
```

Listing 2: Zwei äquivalente Varianten der Objekterzeugung

```
user=> (new java.util.Random)
#object[java.util.Random 0x334d6d67 "java.util.Random@334d6d67"]
user=> (java.util.Random.)
#object[java.util.Random 0x2639e041 "java.util.Random@2639e041"]
```

Clojure verfügt über vergleichsweise wenig Syntax. Auffallend ist die literale Eingabe von Vektoren, Hash-Maps und anderen Datentypen und -strukturen.

Im Fokus: Funktionen

Clojure orientiert sich an funktionalen Programmiersprachen. So stehen Funktionen denn auch im Rampenlicht. Ihre Definition erfolgt mit `defn` (Listing 1).

Listing 1 definiert zuerst eine Funktion mit dem Namen `hello`. In den eckigen Klammern, einem Vektor, folgt die Argumentenliste. Alles Weitere bis zur schließenden Klammer, die den `defn`-Ausdruck beendet, ist der Funktionsrumpf. Er besteht hier aus dem Aufruf der Funktion `str`, die Strings konkateniert.

Der Rückgabewert der Funktionsdefinition ist der spezielle Datentyp `Var`, der die Funktion speichert. Er nimmt Funktionen und – vereinfacht dargestellt – „globale Variablen“ auf. Funktionen treten häufig auch in anonymer Form auf, ohne Bindung an eine `Var`. `"Hello, world"` wird beim Aufruf der Funktion ausgegeben. Ein separater Aufruf von `println` ist nicht notwendig. Die Ausgabe übernimmt hier die REPL, die den Rückgabewert – den neuen String – ausgibt. Damit vereinfacht sich das einleitende Hello-World-Beispiel zu

```
user=> "Hello, world"
"Hello, world"
```




Der Grund dafür ist, dass Strings zu sich selbst evaluieren.

Jedes ernst zu nehmende Clojure-Programm enthält viele Funktionen. Sie an der REPL einzugeben, ist wenig effizient. Abhilfe schafft eine der vielen verfügbaren Entwicklungsumgebungen für Clojure. Die aktuelle Umfrage zum „State of Clojure" zeigt, dass Emacs und Cider nach wie vor die meistgenutzten Entwicklungsumgebungen stellen (siehe ix.de/z1ah). Aber niemand muss Emacs lernen, um Clojure entwickeln zu können. Für viele gängige IDEs und Editoren stehen passende Erweiterungen wie Cursive (IntelliJ), Calva (VS Code) und Fireplace (Vim) bereit (siehe ix.de/z1ah). Es ist obligatorisch, eine solche Umgebung einzurichten. Sie bietet Zugang zur REPL, die inkrementelle Kompilation einzelner Funktionen und weitere IDE-Annehmlichkeiten.

Listing 3: Java-Objekte mit def speichern

```
user=> (def rr (java.util.Random.))
#'de.clojure-buch.heise.2020/rr
user=> (.nextInt rr)
-1468307708
user=> (.nextInt rr)
-1703282749
```

Bausteine

Clojure als gehostete Programmiersprache erbt nicht nur ein Typsystem, Garbage Collection und Ähnliches von der JVM. Clojure-Entwickler*innen stehen die vielen Bibliotheken der Java-Welt zur Verfügung, auf die sie leicht zugreifen können. Die in Listing 2 gezeigten Varianten der Objekterzeugung sind äquivalent.

Mit `def` lassen sich auch Java-Objekte in benannten Variablen speichern, auf denen danach Methoden aufrufbar sind (Listing 3).

Statische Methoden verwenden einen Schrägstrich in ihrem Aufruf:

```
user=> (java.io.File/separator)
"/"
```

Clojure stellt noch weitere Konstrukte für die Interaktion mit Java bereit: das Erzeugen von Klassen, von Objekten, die ein Interface implementieren, und vereinfachte Aufrufe von aneinandergereihten Methodenaufrufen. Somit stehen der Clojure-Gemeinde Tausende frei verfügbarer Bibliotheken zur Verfügung. Die genannten Build-Werkzeuge binden sie durch ihre jeweiligen Projektdefinitionen mitsamt Download ein und lösen transitive Abhängigkeiten auf.

Zudem sind über die Jahre exzellente Clojure-Bibliotheken entstanden, die sich in der von Maven etablierten Manier in öffentlichen Repositories befinden. Wer einen Webserver entwickeln möchte, findet in Ring und den eng verwandten Bibliotheken Compojure und Hiccup eine idiomatische Lösung. Damit ist ein Jetty-Webserver, der verschiedene URLs unterscheiden kann, in nur wenigen Zeilen Clojure geschrieben (eine Demo ungewöhnlicher Routen siehe ix.de/z1ah). Parser für verschiedenste Dateiformate erstellt Instaparse. Im Gegensatz zu etwa ANTLR sind hier keine extra Schritte vor dem eigentlichen Kompilieren notwendig und die Entwicklung von Parsern erfolgt flüssig und interaktiv an der REPL. Reguläre Ausdrücke zum Parsen von Dateien einzusetzen, ist nicht mehr erforderlich.

Das Clojure-Team hat mit core.async und clojure.spec einflussreiche Bibliotheken veröffentlicht, die die Softwareentwicklung in der Clojure-Welt nachhaltig verändert haben. Dabei ermöglicht core.async die asynchrone Kommunikation einzelner Programmabschnitte über Channels. Dieses Konzept ist aus der Programmiersprache Go bekannt, in der die Goroutines einen zentralen Sprachbestandteil bilden. Beide basieren auf demselben theoretischen Unterbau, den Communicating Sequential Processes. In Anlehnung an statische Typsysteme stellt Clojure die Bibliothek clojure.spec bereit. Sie erlaubt es, beliebige Datentypen und -strukturen zu beschreiben. Die Struktur der in Clojure häufig eingesetzten Hash-Maps kann in einem größeren Programm schwer durchschaubar sein. In statisch typisierten Sprachen hilft das Typsystem beim Verständnis der Daten. In Clojure standen dafür lange Zeit nur Kommentare zur Verfügung. Die Bibliotheken Spec und die ältere prismatic/schema gestatten es, Datenstrukturen sowohl zu beschreiben als auch zu validieren. Außerdem können sie zufällige Daten anhand der Beschreibung erzeugen, was bei der Testentwicklung hilfreich ist.

Eine weitere Innovation des Clojure-Teams war die Veröffentlichung von ClojureScript im Jahre 2012. Plötzlich stand die Möglichkeit im Raum, mit Clojure Webapplikationen zu entwickeln. Technisch interessant ist dabei, dass der ClojureScript-Compiler in Clojure geschrieben ist. Seitdem hat sich ClojureScript weiterentwickelt und bildet mittlerweile ein separates Ökosystem. Vor allem im Zusammenhang mit Facebooks React sind viele Projekte wie Reagent und re-frame entstanden. Von Bruce Haumann stammt das Tool Figwheel, das die ClojureScript-Entwicklung auf ein neues Niveau hebt: Es kompiliert und lädt automatisch ClojureScript im Browser. Die Flappy-Bird-Demo ist ähnlich beeindruckend wie Bret Victors Vortrag „Inventing on Principle" (siehe ix.de/z1ah). Das einführende Video zeigt, wie Softwareentwicklung heute vonstattengehen sollte.

Rich Hickey ist nicht nur der Erfinder von Clojure, sondern auch ein unterhaltsamer Redner. Seine Vorträge aus den Jahren 2010 und 2011 waren einflussreich und führten dazu, dass sich der Begriff „to complect", den er verwendet, um das Vermischen wichtiger Aspekte oder Konzepte während der Programmierarbeit zu beschreiben, in der Clojure-Gemeinde eingebürgert hat (siehe ix.de/z1ah).

In Clojures Anfangszeiten war die vereinfachte Entwicklung nebenläufiger Programme ein wesentliches Argument für Clojure. Insbesondere die seinerzeit einzigartige Implementierung von Software Transactional Memory in Clojure wurde oft ins Feld geführt. Im Laufe der Zeit hat sich gezeigt, dass dieses Konstrukt in der Praxis wenig relevant ist. Clojure eignet sich dennoch sehr gut dafür, nebenläufige Programme zu entwickeln. Grund dafür ist die Kombination von Eigenschaften wie unveränderlichen Datenstrukturen, Referenztypen und seiteneffektfreien Funktionen (siehe ix.de/z1ah). Auch die Verfügbarkeit von `pmap` oder `future` in der Standardbibliothek hilft bei ersten Optimierungen durch das Parallelisieren einzelner Programmabschnitte. Etwas unterschätzt ist das Framework Reducers, das parallele Verarbeitung reduzierbarer Datenstrukturen auf Basis von Javas Fork-Join-Framework anbietet. Zusammengefasst ist Clojure eine interessante und lehrreiche Programmiersprache, die sich für die meisten Programmieraufgaben des Alltags eignet. Sie kann auf die Vielzahl frei verfügbarer Java-Bibliotheken zugreifen und erleichtert durch ihren funktionalen Ansatz die Entwicklung nebenläufiger Programme. (nb@ix.de)

Quellen

Listings, Videos und weiterführende Informationen: ix.de/z1ah



Stefan Kamphausen

ist Mitbegründer der :clojureD-Konferenz und Autor des ersten deutschsprachigen Clojure-Buchs. Er arbeitet als VP Engineering bei Acrolinx.



storage2day

22.–24. September 2020 Heidelberg

Die Konferenz zu Speichernetzen und Datenmanagement

PROGRAMM ONLINE

Die **storage2day** ist die neue Fachkonferenz für Systemadministratoren, IT-Leiter und alle mit Storage und Backup befassten Fachleute. Nach ihrer erfolgreichen Premiere 2019 geht sie nun in die zweite Runde.

Das sind die Top-Themen der storage2day 2020:

- Software-defined Storage
- Ceph
- Ransomwäre / Sicherheit
- Backup und Recovery
- Performance
- NVMe
- Speichernetze
- neue RZ-Techniken

Sparen Sie mit dem Very-Early-Tarif!

Bis zum 15.6. zahlen Sie nur 595 Euro zzgl. Mwst. für das zweitägige Konferenzticket.

Zum Vergleich: Der spätere reguläre Preis wird bei 895 Euro liegen.

www.storage2day.de

Goldsponsoren

CLOUDIAN · DATACORE · FAST LTA · FUJIFILM · PURESTORAGE · THOMAS KRENN

Silbersponsoren

point software & systems · SCALITY

Veranstalter

iX · dpunkt.verlag



Eine kurze Geschichte von ECMAScript

Häppchenweise

Golo Roden

Seit dem Jahr 2015 ist wieder Schwung in die Entwicklung von JavaScript gekommen. Der zugrunde liegende Sprachstandard ECMAScript erfährt jährliche Aktualisierungen.

Viele Jahre war es ruhig um JavaScript. Doch seit vor fünf Jahren die Sprachversion ECMAScript 2015 erschienen ist, entwickelt sich die Sprache kontinuierlich weiter (siehe Kasten „JavaScript und ECMAScript: die Zusammenhänge"). Die meisten der jährlich herausgebrachten Änderungen und Erweiterungen sind nur Kleinigkeiten, doch sie verändern das Bild von neu geschriebenem Code. Das merkt man spätestens daran, dass heutiger JavaScript-Code ganz anders aussieht als jener aus dem Jahr 2015. Die großen Neuerungen von ECMAScript 2015 sind den meisten Entwicklern noch bekannt. Die Frage ist, ob das auch für die Nachfolger zutrifft.

ECMAScript 2016 war die bislang am wenigsten spektakuläre Ausgabe von JavaScript. Tatsächlich enthält diese Version gerade einmal zwei neue Features, die zudem noch relativ klein und überschaubar sind, aber einen sehr hohen praktischen Nutzen aufweisen. Zum einen handelt es sich dabei um den neuen Operator ** zum Potenzieren. Will man beispielsweise die dritte Potenz der Zahl 2 berechnen, ist das seit ECMAScript 2016 mit folgender Zeile möglich:

```
const result = 2 ** 3;
```

Zuvor musste man sich dazu der Funktion `Math.pow` bedienen, die zwar prinzipiell den gleichen Zweck erfüllt, aber weitaus umständlicher zu benutzen ist. Außerdem verbessert der Operator gerade in längeren mathematischen Ausdrücken die Lesbarkeit, da nicht immer wieder ein Funktionsaufruf einzuschieben ist.

Zum anderen ist die Funktion `includes` auf Arrays neu, mit der man überprüfen kann, ob ein Wert in einem Array bereits enthalten ist. Damit löst die Funktion den zuvor zweckentfremdeten Einsatz der Funktion `indexOf` ab, was Code ebenfalls kompakter und lesbarer macht:

```
const primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ];
const isPrime = function (number) {
    return primes.includes(number);
};
```

Es ist wichtig, bei `includes` zu beachten, dass der Vergleich nur bei Wertetypen auf Basis des Wertes stattfindet – Referenztypen wie Objekte oder Arrays werden wie üblich anhand ihrer Referenz verglichen. Hier ist also mehr Aufwand erforderlich, um den gleichen Effekt zu erzielen. Am einfachsten geht das mithilfe der bereits seit ECMAScript 2015 bekannten Funktion `some`, der man eine Prädikatsfunktion übergibt.

iX-TRACT

- Die Sprache JavaScript basiert auf dem Standard ECMAScript und ist dessen verbreitetste Implementierung.
- Seit 2015 erscheint jährlich eine neue Version. Zur Kennzeichnung erhält sie statt einer Versionsnummer eine Jahreszahl.
- Eine bahnbrechende Neuerung enthielt die Ausgabe von 2017 mit den Schlüsselwörtern `async` und `await`. ECMAScript 2021 soll das Feature Top-Level `await` enthalten.

JavaScript und ECMAScript: die Zusammenhänge

JavaScript ist eine ursprünglich von Netscape entwickelte Programmiersprache. Allerdings hat Netscape die der Sprache zugrunde liegende Spezifikation bei Ecma International zur Standardisierung eingereicht, woraus die Spezifikation ECMA-262 entstanden ist. Diese wird als ECMAScript bezeichnet.

Deren Versionszählung war zunächst einfach: Auf Version 1 folgte Version 2, darauf dann Version 3. Auf die Eigenheiten von Version 4 konnten sich allerdings die verschiedenen Browserhersteller nicht einigen, weshalb diese Version letztlich entfallen ist. Version 5 war dann als Kompromiss der kleinste gemeinsame Nenner, quasi ein kleines Überbleibsel von dem, was für ECMAScript 4 geplant gewesen wäre.

Nachdem es etliche Jahre Stillstand gab, kam 2010 wieder Bewegung in die Sprache. Geplant war, mit ECMAScript 6 die Versäumnisse von Version 4 nachzuholen, dieses Mal in einem gemeinsamen harmonischen Anlauf aller Beteiligten. Daher wurde ECMAScript 6 auch als ECMAScript „Harmony“ bezeichnet.

Die Veröffentlichung zögerte sich aber bis 2015 hinaus. Zeitgleich beschloss Ecma International, von nun an in jedem Jahr eine neue Version zu veröffentlichen – Maßstab sollte also nicht mehr der Fertigstellungsgrad von Features sein, sondern ein Stichtag: Alles, was bis zu diesem Datum fertig ist, wird in der neuen Ausgabe enthalten sein.

Außerdem ist ECMAScript ab sofort mit einer Jahreszahl statt mit einer Versionsnummer versehen, weshalb ECMAScript 6 als ECMAScript Harmony und zusätzlich auch noch als ECMAScript 2015 bekannt geworden ist. Ab ECMAScript 2016 gibt es nur noch die Jahreszahl als Versionsnummer.

So klein und verhältnismäßig unbedeutend ECMAScript 2016 war, so bahnbrechend war die Version des darauffolgenden Jahres. Die große Neuerung in ECMAScript 2017 schlechthin war die Einführung der beiden neuen Schlüsselwörter `async` und `await`, die den bis dahin gängigen Stil zum Schreiben von asynchronem Code mit Callbacks inzwischen weitgehend verdrängt haben. Aus dem umständlichen verschachtelten, fehleranfälligen Callback-basierten Code zum Auslesen einer Datei (Listing 1) wird dank `async` und `await` der in Listing 2 gezeigte Code (alle Listings siehe ix.de/zmej).

Asynchroner Code liest sich dadurch wie synchroner, behält aber die asynchronen Eigenschaften bei. Auffällig ist, um wie viel kürzer die neue Version im Vergleich zu der vorigen ist. Der Code ist kürzer, nachvollziehbarer und linear lesbar.

Neben dieser großen Neuerung gibt es noch einige nennenswerte Kleinigkeiten. Nur kurz erwähnt sei dabei die Möglichkeit, in Funktionsaufrufen hinter dem letzten Parameter noch ein Komma anzufügen, was in dieser Hinsicht Parität mit der Definition von Arrays und Objekten einführt.

Deutlich bemerkenswerter ist, dass ECMAScript 2017 nun endlich Funktionen vorgibt, um Strings von links oder rechts mit Zeichen aufzufüllen – beispielsweise um eine Formatierung mit Leerzeichen oder Nullen vorzunehmen. Die Funktion `padStart` füllt dabei von links auf, `padEnd` hingegen von rechts (Listing 3).

Gibt man keinen weiteren Parameter an, verwenden die beiden Funktionen das Leerzeichen zum Auffüllen. Wird ein anderes Zeichen gewünscht, kann man es als zweiten Parameter explizit angeben.

Zwei weitere neue Funktionen von ECMAScript 2017 sind `Object.values` und `Object.entries`. Sie geben die Werte beziehungsweise die Schlüssel-Wert-Paare eines Objekts als Arrays zurück und sind in Verbindung mit der `for of`-Schleife hilfreich, um effizient auf alle Schlüssel und Werte eines Objekts zugreifen zu können (Listing 4).

Operatoren auf Objekte statt Arrays anwenden

Nach dem großen Update im Jahr 2017 enthielt ECMAScript 2018 nur zwei neue Features, die beide große Bekanntheit erlangten. Zum einen lassen sich der Rest- und der Spread-Operator nun auch auf Objekte anwenden und nicht wie zuvor nur auf Arrays. Das ist dann praktisch, wenn alle Eigenschaften eines Objekts in ein neues zu kopieren sind, dabei aber noch eine weitere Eigenschaft hinzugefügt werden soll (Listing 5).

Zum anderen wurde die `for of`-Schleife um ihr asynchrones Pendant, die `for await of`-Schleife, ergänzt. Für sie gibt es noch nicht allzu viele praktische Anwendungsfälle – aber sie ist unschlagbar, wenn es um das Iterieren von Streams in Node.js geht. Anstatt wie bislang mühsam mit den verschiedenen Ereignissen `data`, `end` und `error` hantieren zu müssen, lässt sich das nun mit einer einfachen Schleife bewerkstelligen:

```
const fileStream = fs.createReadStream('/etc/hosts');
for await (const data of fileStream) {
    console.log(data);
}
```

Entwirren geht einfacher

ECMAScript 2019 enthält Neuerungen, von denen die mit Abstand praktischste das sogenannte Optional Catch Binding ist. Das Feature ermöglicht das Weglassen der Variablen `Exception` in einem `catch`-Block, in dem man an dem Fehler nicht interessiert ist. In Listing 6 prüft die gezeigte Funktion, ob ein npm-Modul installiert ist. Im Fehlerfall ist die Art des Fehlers irrelevant, es soll nur der Wert `false` zurückgegeben werden.

Listing 1: Umständlicher Code zum Auslesen einer Datei

```
const fs = require('fs');

const getHosts = function (callback) {
    fs.readFile('/etc/hosts', 'utf8', (err, data) => {
      if (err) {
        return callback(err);
      }

      callback(null, data);
    });
};
```

Listing 2: Vereinfachter Code zum Auslesen einer Datei

```
const fs = require('fs');

const getHosts = async function () {
    const data = await fs.promises.readFile('/etc/hosts', 'utf8');

    return data;
};
```

Listing 3: Strings mithilfe der Funktion padStart mit Zeichen füllen

```
const prices = [
    '23.95',
    '42.95',
    '7.99'
];

for (const price of prices) {
    console.log(price.padStart(10));
}
```

Außerdem enthält JavaScript seit ECMAScript 2019 endlich eine `flat`-Funktion für Arrays, mit der sich verschachtelte Arrays auflösen lassen:

```
const array = [[ 1, 2 ], [ 3, 4 ]];
console.log(array.flat());
// => [ 1, 2, 3, 4 ]
```

Standardmäßig entfernt die `flat`-Funktion allerdings nur eine Ebene der Verschachtelung. Will man auf einen Schlag gleich mehrere Ebenen entfernen, lässt sich die gewünschte Anzahl als Parameter übergeben.

Eng verwandt mit `flat` ist die Funktion `flatMap`, die eine Kombination aus `map` und anschließendem `flat` darstellt. Mit ihr lassen sich Werte in Tupel abbilden, die dann aber als eine einzige Liste statt als Liste von Listen zurückgegeben werden:

```
const numbers = [ 1, 2, 3, 4, 5 ];
const result = numbers.flatMap(n => [ n, n * 2 ]);
console.log(result);
// => [ 1, 2, 2, 4, 3, 6, 4, 8, 5, 10 ]
```

Das gleiche Ergebnis erreicht man auch, indem man `map` und `flat` einzeln aufruft:

```
const result = numbers.map(n => [ n, n * 2 ]).flat();
```

Die `flatMap`-Funktion arbeitet aber etwas effizienter und ist daher aus Leistungsgründen vorzuziehen.

Ebenfalls neu in ECMAScript 2019 sind die beiden Funktionen `trimStart` und `trimEnd`, die sich – ähnlich wie `padStart` und `padEnd` – auf Zeichenketten beziehen. Anders als jene Funktionen füllen `trimStart` und `trimEnd` jedoch nicht mit Leerstellen auf eine gewünschte Länge auf, sondern entfernen unnötige Leerstellen am Anfang beziehungsweise Ende eines Strings. Sie stellen daher spezialisierte Varianten der bereits länger existierenden Funktion `trim` dar.

Zu guter Letzt ist in ECMAScript 2019 noch die Funktion `Object.fromEntries` neu enthalten, die das Gegenstück zu `Object.entries` darstellt: Während `Object.entries` die Schlüssel-Wert-Paare eines Objekts als Array zurückgibt, baut `Object.fromEntries` aus ebendiesem Array wieder ein Objekt zusammen. Das ist praktisch, um Funktionen wie `map`, `filter` und `reduce` auf Objektschlüssel oder -werte anwenden zu können, die von Haus aus nur für Arrays zur Verfügung stehen (Listing 7).

Listing 4: Die Funktion Object.entries gibt Werte als Arrays zurück

```
const colors = {
    red: '#ff0000',
    green: '#00ff00',
    blue: '#0000ff'
};

for (const [ name, code ] of Object.entries(colors)) {
    console.log(`${name}: ${code}`);
}
```

Vereinfachtes Vorgehen

Version 2020 von ECMAScript bietet ebenfalls einige Neuerungen. Die bedeutendsten davon sind die aus C# bereits bekannten Operatoren Null-Coalescing und Null-Conditional. In JavaScript tritt an die Stelle des Null-Coalescing Operator häufig der logische Operator `||`. Er reagiert allerdings nicht nur auf `null` und `undefined`, sondern auf alle Werte, die äquivalent zu `false` sind – also `false` selbst, aber auch die Zahl 0 oder die leere Zeichenkette. Der neue Operator, der wie in C# als ?? geschrieben wird, behebt dieses Problem.

Der Null-Conditional Operator ?. vereinfacht den Zugriff auf die Eigenschaften von Objekten, die nicht definiert sind. Greift man mit ihm auf einen Wert eines Objekts zu, das `null` oder `undefined` ist, löst er keinen Fehler aus, sondern gibt `undefined` zurück. Gerade der Zugriff auf verschachtelte Eigenschaften ist damit deutlich lesbarer, da man sich zahlreiche Prüfungen sparen kann. Aus

```
let result = undefined;

if (a && a.b && a.b.c) {
    result = a.b.c;
}
```

wird auf dem Weg das viel kompaktere und einfacher lesbare

```
const result = a?.b?.c;
```

Ebenfalls neu ist der Datentyp `BigInt`. Seit der ersten Version kennt JavaScript zur Darstellung von Zahlen nur den Datentyp `number`, der intern einer Fließkommazahl doppelter Genauigkeit gemäß IEEE 754 entspricht. Diese Spezifikation ist unter anderem der Grund dafür, warum JavaScript Werte wie +0, -0, Infi-

Listing 5: Objekte mit dem Spread-Operator kopieren

```
const source = {
    value: 23,
    otherValue: 42,
};

const target = {
    ...source,
    additionalValue: 7
};
```

Listing 6: catch funktioniert auch ohne Fehlerbindung

```
const isInstalled = function (name) {
    try {
      require(name);

      return true;
    } catch {
      return false;
    }
}
```

Listing 7: Object.fromEntries ermöglicht map, filter und reduce auf Objekten

```
const versions = {
    node: '10.16.0',
    npm: '6.10.2',
    express: '4.17.1'
};

const filteredVersions = Object.fromEntries(
    Object.entries(versions).
      filter(([ key, value ]) => key.startsWith('n'))
);

console.log(filteredVersions);

// => {
//      node: '10.16.0',
//      npm: '6.10.2'
//    }
```

nity und -Infinity abbilden kann und die Division durch null in JavaScript nicht zu einer Ausnahme führt:

```
const result = 23 / 0;

console.log(result);
// => Infinity
```

Allerdings ist der Wertebereich von `number` begrenzt, für Ganzzahlen stehen durch die interne Darstellung als Fließkommazahlen noch nicht einmal die vollen 64 Bit zur Verfügung. So entspricht die größte in JavaScript sicher abbildbare Ganzzahl dem Wert `2 ** 53 - 1`, auf den sich über die Konstante `Number.MAX_SAFE_INTEGER` zugreifen lässt:

```
const value = Number.MAX_SAFE_INTEGER;

console.log(value);
// => 9007199254740991
```

Erhöht man diese Zahl um 1, funktioniert diese Addition. Jedoch lässt sich das Ergebnis nicht mehr von der Zahl unterscheiden, die sich durch die Addition um 2 ergibt. Beide Zahlen sind, obwohl ihr Wert unterschiedlich sein müsste, gleich (Listing 8).

Dieses Problem behebt `BigInt`. Einen Wert dieses Typs erhält man, indem man entweder den Buchstaben n an eine Zahl anhängt oder die Funktion `BigInt` verwendet, um eine `number` oder einen `string` zu konvertieren:

```
const maxSafeInteger = 9007199254740991n;
const maxSafeIntegerFromNumber = BigInt(9007199254740991);
const maxSafeIntegerFromString = BigInt('9007199254740991');
```

Berechnungen mit `BigInt` sind nur mit anderen `BigInt`-Werten möglich. Werte vom Datentyp `number` sind zuvor zu konvertieren, andernfalls erhält man einen Laufzeitfehler:

```
const value = 1n + 2;
// => TypeError: Cannot mix BigInt and other types, use explicit
//conversions
```

Bei der Division findet eine Ganzzahldivision statt, mit einer Rundung gegen null:

```
const notRounded = 4n / 2n;
// => 2n

const roundedTowardsZero = 5n / 2n;
// => 2n
```

Neben Rechenoperatoren für die Grundrechenarten +, -, * und / sind auch die Operatoren ** und % zulässig. Damit lässt sich ein einfacher Algorithmus zum Berechnen und Überprüfen beliebig großer Primzahlen schreiben (Listing 9).

Listing 8: Das Ergebnis der Addition ist nicht mehr zu unterscheiden

```
const value1 = Number.MAX_SAFE_INTEGER + 1;
const value2 = Number.MAX_SAFE_INTEGER + 2;

console.log(value1);
// => 9007199254740992

console.log(value2);
// => 9007199254740992
```

Listing 9: Algorithmus zum Berechnen von Primzahlen

```
const isPrime = function (p) {
    for (let i = 2n; i * i <= p; i++) {
      if (p % i === 0n) {
        return false;
      }
    }

    return true;
};
```

Augen auf

Vorsicht ist beim Einsatz der mathematischen Funktionen geboten, die für das Verwenden mit `BigInt` nicht ausgelegt sind. Aus diesem Grund ist in Listing 9 die obere Grenze der Schleife mit der Multiplikation `i * i` statt `Math.sqrt` berechnet. Diese Funktion beherrscht nämlich nur den Umgang mit dem klassischen Datentyp `number`.

Bei Vergleichs- und logischen Operatoren ist darauf zu achten, dass `===` nicht zwischen `number` und `BigInt` möglich ist, da sich die beiden Datentypen unterscheiden. Ein Vergleich mit dem prinzipiell unsicheren `==` ist aber möglich.

Auch im Bereich des Schlüsselwortes `this` gibt es Neuerungen. Es führt seit jeher zu unerwarteten Schwierigkeiten, wenn man von einer anderen Sprache zu JavaScript wechselt. Obwohl es syntaktisch dem gleichnamigen Schlüsselwort in C# und Java ähnelt, verhält es sich anders. Der Strict Mode und die Einführung der Lambdaausdrücke haben viele der Probleme entschärft. Zwar ist es noch erforderlich zu wissen, wie `this` im Detail funktioniert, aber der Aufwand, verlässlichen Code zu schreiben, ist gesunken.

Schwieriger ist es geworden, auf den globalen Kontext zuzugreifen. Er ist zwar selten und nur mit Bedacht einzusetzen, dennoch aber erforderlich. Im Webbrowser ist er über `window`, `self`, `frames` und eben `this` erreichbar. In Node.js hingegen stehen nur `global` und `this` zur Verfügung. In Shells wie jsc oder d8 funktioniert der Zugriff auf den globalen Kontext ausschließlich über `this`. Allerdings zeigt `this` nur dann auf den globalen Kontext, wenn der Wert nicht auf etwas anderes verweist, beispielsweise bei einem Methodenaufruf.

In der Vergangenheit konnte man sich mit einer Funktion behelfen, die als solche (und eben nicht als Methode) aufgerufen wurde, da `this` in dem Fall auf den globalen Kontext verwiesen hat:

```
const getGlobal = function () {
    return this;
};

const global = getGlobal();
// global verweist auf den globalen Kontext
```

Bei Verwendung des Strict Mode ist das nicht mehr möglich, da `this` in nicht als Methode aufgerufenen Funktionen nun den Wert `undefined` enthält. Daher wurde eine alternative Funktion entwickelt, die `self`, `window` oder `global` verwendet. Je nach Situation enthält man so das korrekte Objekt (Listing 10).

Das Problem löst ECMAScript 2020 mit dem neuen Schlüsselwort `globalThis`, das den globalen Kontext zurückgibt. Dadurch wird der Code nicht nur kürzer und weniger fehleranfällig, sondern vor allem auch expliziter.

Die Schlüsselwörter `import` und `export` stellen die Grundlage für ein modernes Modulkonzept dar, auch wenn die Unterstützung durch Webbrowser und Node.js hat auf sich warten lassen. Allerdings hat `import` einen Nachteil: Das Schlüsselwort darf nur am Anfang einer Datei vorkommen, auf der obersten Ebene des Codes. Diese Einschränkung ist erforderlich, damit die Laufzeitumgebung bereits beim Laden von Modulen analysieren kann, welche Konstrukte von einem Modul zu exportieren und importieren sind.

Listing 10: Die Funktion mit self, window und global

```
const getGlobal = function () {
    if (typeof self !== 'undefined') {
      return self;
    }
    if (typeof window !== 'undefined') {
      return window;
    }
    if (typeof global !== 'undefined') {
      return global;
    }

    throw new Error('Failed to get global context.');
};
```

Listing 11: Das #-Zeichen markiert private Elemente

```
class Stack {
    constructor () {
      this.#data = [];
    }

    push (value) {
      this.#data.push(value);
    }

    // ...
}
```

Gelegentlich ist es aber erforderlich, Code dynamisch zur Laufzeit zu laden. Das kann vorkommen, wenn man ein Plug-in-basiertes System aufbauen möchte und sich die Liste der zu ladenden Plug-ins erst zur Laufzeit ergibt. Bisher war das in Node.js durch den Einsatz der Funktion `require` machbar. Diese Funktion an anderer Stelle als der obersten Ebene aufzurufen, galt zwar schon immer als schlechter Stil, war aus technischer Sicht aber möglich. Hier verhält sich das Schlüsselwort `import` anders.

Die neu in ECMAScript 2020 enthaltene Funktion `import` erfüllt die gleiche Aufgabe wie das Schlüsselwort `import`: das Laden von Modulen. Im Unterschied zum Schlüsselwort darf sie aber an jeder beliebigen Stelle aufgerufen werden, außerdem lässt sich der Parameter zur Laufzeit als String zusammensetzen. Als Ergebnis gibt die Funktion eine Referenz auf das geladene Modul zurück, das Laden erfolgt aber asynchron. Man muss das zurückgegebene Promise also entweder von Hand verwalten oder die Funktion mit `await` aufrufen:

```
const moduleName = `core`;
const module = await import(`./modules/${moduleName}.js`);
```

Was die Zukunft bringt

Die Themenliste des TC39 zeigt die geplanten neuen Features von ECMAScript 2021. Die aufgelisteten Themen sind nach sogenannten Stages sortiert, wobei vor allem die Features in Stage 3 eine gute Chance haben, in der kommenden oder spätestens der darauffolgenden Version von JavaScript enthalten zu sein.

Ein sehr praktisches und schon in Node.js und einigen Webbrowsern zur Verfügung stehendes Feature sind die sogenannten Numeric Separators. Dabei handelt es sich um die Möglichkeit, Zahlen nach Belieben mit Unterstrichen zu versehen, um sie optisch besser strukturieren zu können. Die Struktur ist aber rein visueller Natur, für JavaScript haben die Unterstriche keinerlei Bedeutung. Mit den Numeric Separators lässt sich der Wert von einer Milliarde gut lesbar mit Tausendertrennzeichen schreiben:

```
const billion = 1_000_000_000;
```

Auch Binärzahlen lassen sich damit besser lesbar darstellen:

```
const someBits = 0b1010_0001_1000_0101;
```

Das Gleiche funktioniert auch für hexadezimale Werte:

```
const bytes = 0xA0_B0_C0;
```

Ein weiteres spannendes Feature in Stage 3 ist das sogenannte Top-Level `await`. Hintergrund: Das Schlüsselwort `await` lässt sich derzeit nur in Funktionen verwenden, die mit `async` markiert wurden. Im Normalfall stellt das kein Problem dar, ist aber spätestens auf der obersten Ebene eines Moduls oder einer Anwendung lästig, wenn man sich mit einer IIAFE (Immediately Invoked Asynchronous Function Expression) behelfen muss.

Das Feature ermöglicht den Einsatz von `await` auf oberster Ebene, ohne dort ein zusätzliches `async` zu erfordern, und verbessert damit wiederum die Lesbarkeit des Codes. Zugleich bringt es das Risiko mit sich, den Start von Anwendungen durch eine maßlose Anwendung von `await` stark zu verzögern.

Auch derzeit in Stage 3, aber mit guten Chancen für ECMAScript 2021, ist die Option, Felder und Methoden einer Klasse als privat zu markieren. Bislang ist das in JavaScript nicht möglich, weshalb man sich mit Workarounds behilft – beispielsweise, indem man den Namen aller privaten Elemente einen Unterstrich voranstellt. Das ist allerdings keine technisch saubere Lösung, sondern nur ein visueller Hinweis.

Geplant ist, private Elemente mit einem #-Zeichen zu versehen. Innerhalb der Klasse lässt sich darauf mit `this` zugreifen, von außerhalb sind sie hingegen nicht sichtbar (Listing 11).

Welche dieser Neuerungen es tatsächlich in ECMAScript 2021 schaffen, bleibt abzuwarten. Die größte Wahrscheinlichkeit besteht jedoch für jene Sprachmerkmale, die bereits heute entsprechend dem TC39-Proposal in Webbrowsern oder Node.js implementiert sind, wie die Numeric Separators.

Fazit

Seit ECMAScript 2015 gibt es Änderungen auch abseits der großen Themen. Viele der Neuerungen wirken auf den ersten Blick wie beliebige Kleinigkeiten. In der täglichen Praxis machen sie sich aber rasch bezahlt, da sie lästige Routineaufgaben vereinfachen oder abkürzen. Wie man an der Liste der diskutierten Sprachmerkmale des TC39 sieht, entwickelt sich JavaScript kontinuierlich weiter.

Um bei der ohnehin hohen Entwicklungsgeschwindigkeit des JavaScript-Ökosystems nicht Gefahr zu laufen, abgehängt zu werden, ist es ratsam, sich regelmäßig mit den Neuerungen der Sprachversionen zu befassen. Da die meisten neuen Features nicht sehr komplex sind, lässt sich das mit vertretbarem Zeitaufwand meistern. Neue Sprachmerkmale wie `async` und `await`, die die Sprache grundlegend anders wirken lassen, sind eher selten. (nb@ix.de)

Quellen

Listings zum Download: ix.de/zmej



Golo Roden

ist Gründer, CTO und Geschäftsführer der the native web GmbH, eines auf native Webtechnologien spezialisierten Unternehmens.

Mit allen Wassern gewaschen:

Portofrei ab 15 €



iX Developer Moderne Softwareentwicklung

Verschaffen Sie sich Orientierung im Dschungel der Softwareentwicklung: Machine und Deep Learning, mehr Sicherheit dank DevSecOps, Git-Funktionen ausreizen, Chaos Engineering, Rust, Elixir und Elm - dieses Sonderheft bietet mit einem breiten Themenmix und umfangreichen Praxisartikeln Überblick und vielfältige Anregungen.

Auch komplett digital erhältlich.

shop.heise.de/ix-software20

12,90 € ›

NEU

iX Kompakt Container 2020

Das neue Sonderheft liefert fundiertes Basiswissen unter anderem zu diversen *aaS-Architekturen, Containern und Mikrovirtualisierung. Außerdem jede Menge Tipps und Entscheidungshilfen, um aus der Vielzahl von Tools und Anbietern die passende Container-Software für sich zu finden.

Auch komplett digital erhältlich!

shop-heise.de/ix-container20



14,90 € ›



iX kompakt IT-Sicherheit

Das neue Sonderheft der iX-Experten bringt alle Sicherheitsaspekte ins Bewusstsein und schafft reale Sicherheit: Red Teaming fürs effektive Testing, Malware-Erkennung durch KI, DSGVO und IT-Sicherheit, Anwendung in Containern, Marktübersichten und Tools.

Auch komplett digital erhältlich.

shop.heise.de/ix-security2019

12,90 € ›

Weitere Sonderhefte zu vielen spannenden Themen finden Sie hier: **shop.heise.de/specials-aktuell**

› Generell portofreie Lieferung für Heise Medien- oder Maker Media Zeitschriften-Abonnenten oder ab einem Einkaufswert von 15 €. Nur solange der Vorrat reicht. Preisänderungen vorbehalten.

heise shop

shop.heise.de/specials-aktuell ›



TypeScript: JavaScript-Unterstützung für statische Codeanalyse

Starke Typen

Lenz Weber

JavaScript ist wenig typsicher und daher fehleranfällig. Hier kann das kompatible TypeScript Entwickler mit seinem ausgefeilten Typsystem helfen.

Während JavaScript zwar intern die Typen `string`, `number`, `undefined`, `null`, `boolean`, `symbol` und `object` kennt, spielt dies für Entwickler*innen kaum eine Rolle. Die Sprache ist schwach typisiert und wechselt zur Laufzeit fließend wo nötig zwischen den Typen. Was in der Theorie gut klingt, kann in der Praxis Verwirrung und schwer zu findende Bugs verursachen. Arbeitet der Entwickler nebenher mit anderen schwach typisierten Sprachen, verstärkt sich das Problem. Wer denkt schon daran, dass in JavaScript `1+"2" == "12"` gilt, während PHP hier zum Ergebnis 3 kommt?

Also muss man auf die korrekten Variablentypen achten. Insbesondere bei externen Bibliotheken ist nicht immer klar, welche Datentypen sie erwarten oder zurückliefern. Oft ist auch die Dokumentation veraltet oder schlicht falsch. So schleichen sich Fehler ein. Hier kommt TypeScript ins Spiel und erweitert JavaScript um ein Typsystem, das sich für statische Codeanalyse eignet.

TypeScript bietet jetzt schon alle JavaScript-Features des kommenden ECMAScript2020-Standards wie dynamische `import()`-Aufrufe, den Nullish-Coalescense-Operator `??` und den Optional-Chaining-Operator `?.`. Selbstverständlich erfüllt die Sprache auch alle bisherigen Standards. Erwähnenswert sind Konstrukte wie Klassen mit dem `class`-Schlüsselwort, die Möglichkeit, asynchronen Code iterativ mittels `async .. await` zu schreiben, die Iteration von Iterables mittels `for of` und Generatorfunktionen `function*() {}`. Außerdem wurden einige Features aufgenommen, die sich noch in der Standardisierung befinden.

Dazu gehört, per `top-level await` auch außerhalb von Funktionen das `await`-Schlüsselwort oder private Klassen-Member mit der `myClass.#property`-Schreibweise einzusetzen.

Die TypeScript-Entwickler nehmen nur ausgesuchte TC39-Proposals auf, die sich mindestens in Stage 3 (Candidate) befinden. In diesem Stadium ist die Diskussion um Features größtenteils abgeschlossen und bei der Syntax sind keine weiteren Änderungen mehr zu erwarten.

Die Lektion, mindestens auf Stage 3 zu warten, musste das TypeScript-Team bitter lernen. 2015 führte es Dekoratoren als experimentelles Feature ein. Das dazugehörige Proposal befindet sich heute noch in Stage 2 (Draft) und hat sich in Syntax und Funktionsumfang leicht geändert. TypeScript wird also nachziehen müssen, sollte das Feature es tatsächlich eines Tages in den ECMAScript-Standard schaffen. Das führt unweigerlich zu einem Breaking Change. Darüber hinaus bietet TypeScript Konstrukte,

Listing 1: Die meisten Typen leitet der Compiler situationsbezogen selbst her

```
const druckerzeugnis: Druckerzeugnis = getDruckerzeugnis();
if ('issn' in druckerzeugnis) {
 console.log(druckerzeugnis.issn);
 for (const autor of druckerzeugnis.autoren) {
  console.log(autor)
 }
}
```

Listing 2: Eine von tsc --init erzeugte tsconfig.json

```
{
 "compilerOptions": {
  "target": "es5",
  "module": "commonjs",
  "strict": true,
  "esModuleInterop": true
 }
}
```

zu denen es in JavaScript keine Entsprechung gibt: `enums` und `namespace`. Während Namespaces heutzutage dank besserer Integration von JavaScript-Modulen größtenteils obsolet sind, werden Enums immer noch häufig eingesetzt.

Mit Typen zum Ziel

Doch der Vorteil von TypeScript liegt nicht in der Vereinbarkeit mit JavaScript-Features – das konnte schon ein Transpiler wie babel. Die Stärke der Sprache liegt im Typsystem. Es ist nicht nur Turing-vollständig, sondern schafft es auch, dem Entwickler möglichst oft „aus dem Weg“ zu gehen. Nicht immer ist es nötig, Typen zu definieren. Meist reicht es, Methodenargumente korrekt mit Typen zu annotieren, und der Rest wird automatisch inferiert – also vom Compiler aus dem Kontext des Programmcodes erschlossen. Wer an eine Stelle gerät, an der es unverhältnismäßig kompliziert ist, einen Typ zu beschreiben, dem bietet TypeScript als Notluke das `any`-Schlüsselwort an. Gelegentlich angewendet ist diese Lösung in Ordnung, man sollte es aber nicht übertreiben.

Typen dienen nicht nur dazu, Programmierfehler zu vermeiden, sondern liefern wichtige Metainformationen an Entwicklertools. So profitiert die Autovervollständigung in Werkzeugen wie Visual Studio Code oder WebStorm von den Typinformationen, die der TypeScript-Sprachservice bereitstellt. Er bietet darüber hinaus Refactoring-Möglichkeiten. Mit seiner Hilfe lassen sich beispielsweise Variablen und Funktionen umbenennen, Code und Typen extrahieren oder String-Konkatenierungen zu Template Literals konvertieren. Diese Refactorings stehen in den meisten IDEs direkt zur Verfügung und werden mit jeder neuen Version von TypeScript um neue Features erweitert. Darüber hinaus können auch Linter (Codeanalysetools) auf Typinformationen zugreifen, was neue Regeln zulässt. Mithilfe von Typinformationen lassen sich zudem auch ohne JsDoc-Blocks automatisch Dokumentationen erzeugen.

Die meisten ihrer Fähigkeiten kann TypeScript in JavaScript-Dateien ausspielen. Das kann zum Beispiel bei einer Migration zu TypeScript eine wichtige Rolle spielen, wenn man schon einige Dateien in TypeScript geschrieben hat, ein Teil der Codebasis aber noch in JavaScript vorliegt. TypeScript versteht in JavaScript-Dateien JsDoc-Typannotationen und kann auf deren Basis Typen prüfen. Das ermöglicht einerseits, in JavaScript-Dateien viele der Komfortfunktionen wie Autovervollständigung zu nutzen, andererseits, in TypeScript-Dateien typisierte Werte aus JavaScript-Dateien zu importieren. Bei einer Migration müssen Entwickler*innen nichts im Sinne einer Big-Bang-Modernisierung übers Knie brechen, sondern können Stück für Stück die Codebasis mit Typinformationen anfüttern.

Allgemein folgt in TypeScript eine Typannotation immer auf einen Doppelpunkt. Die wohl häufigste ist die Beschreibung einer Methodensignatur. So kann man den JavaScript-Code

```
function strLen(str) {
 return str.length;
}
```

in TypeScript so schreiben:

```
function strLen(str: string): number {
 return str.length;
}
```

Zuerst annotiert man das Argument `str` mit dem Typ `string`. Dadurch lässt sich die Methode nur noch mit einem String-Argument aufrufen. Innerhalb der Methode geht der Compiler nun fest von einem String-Wert dieser Variablen aus, `str.length` ist demnach also definitiv vom Typ `number`.

Explizit annotieren

Genauso ließe sich im Methodenkörper auch eine Variable oder Konstante mittels `let/var/const variablenname: typ` annotieren. Der Rückgabewert der Methode ist als `number` annotiert. Im gegebenen Fall hätte man es sich jedoch sparen können: Dadurch, dass die Information vorliegt, dass es sich bei `str.length` um den Typ `number` handelt, lautete der Rückgabewert von selbst `number`. Allerdings schützt die explizite Annotation des Rückgabetyps vor Flüchtigkeitsfehlern. Sollte man versehentlich im Methodenkörper an einer Stelle das `return`-Statement vergessen haben oder etwas anderes zurückgeben, warnt TypeScript, dass die Methodensignatur nicht mit ihrem Verhalten übereinstimmt. So würde der Code

```
function strLen(str: string): number {
 if (str.length > 0) {
 return str.length
 }
}
```

den folgenden Fehler provozieren:

```
Function lacks ending return statement and return type does not 7
                                                    include 'undefined'.(2366)
```

Für solche Typannotationen stehen von Haus aus unter anderem die primitiven Typen `string`, `number` und `boolean` zur Verfügung. Zwar existiert auch `object`, das aber jedes mögliche Objekt beschreibt und daher in der Praxis weniger nützlich ist. Ein `interface`

iX-TRACT

- TypeScript ist eine statisch typisierte Programmiersprache, die nach JavaScript übersetzt wird und damit überall einsetzbar ist, wo auch JavaScript eingesetzt werden könnte.
- Ein Parallelbetrieb von JavaScript und TypeScript im selben Projekt ist möglich.
- Für fast alle Bibliotheken des JavaScript-Ökosystems existieren entweder offizielle oder von der Community gepflegte Typdefinitionen.
- TypeScript kann in den meisten Fällen die Typen aus dem Kontext herleiten, der Code wird also nicht durch redundante Typannotationen aufgebläht.

oder einen `type` festzulegen, dürfte hier sinnvoller sein. Zur einfachen Beschreibung der Form eines Objekts lässt sich ein Interface definieren, das überall im selben Scope als Typannotation fungieren kann:

```
interface Buch {
 titel: string;
 autor: string;
 isbn: string;
}
```

Allgemein besteht die Möglichkeit, von jedem Typ ein Array als `typname[]` oder `Array<typname>` zu beschreiben. Darüber hinaus gibt es weitere Basistypen, von denen einige jedoch nicht als Variablen- oder Argumenttyp taugen. Dazu gehört `void`, der beschreibt, dass eine Methode entweder keinen Rückgabewert hat oder dieser nicht verwendet werden sollte. Mit dem Typ `never` kann man eine Funktion annotieren, die niemals auf normale Weise terminiert, etwa weil sie immer eine Exception wirft. Bleibt noch der Top-Typ `any`, dem sich alles zuweisen lässt und der als jeder andere Typ dienen kann. Dem Bottom-Wert `unknown` lässt sich ebenfalls alles zuweisen. Sein Wert ist aber nicht einsetzbar, bis man ihn in einen anderen Typ überführt hat.

Neben Interfaces kennt TypeScript die Definition von Typen mittels `type`. Typen besitzen allgemein deutlich mehr Macht als Interfaces, hier soll es beim häufigsten Einsatz bleiben: der Definition von `intersection types`. So könnte man mittels

```
type Druckerzeugnis = Buch | Zeitschrift;
```

den Typ Druckerzeugnis festlegen, dem entweder ein Buch oder eine Zeitschrift zuweisbar ist. Angenommen, das Interface `Zeitschrift` wäre so beschrieben:

```
interface Zeitschrift {
 titel: string;
 autoren: string[];
 issn: string;
}
```

Hier ließe sich direkt auf die Eigenschaft `titel` eines Druckerzeugnisses zugreifen, da beide Interfaces sie implementiert haben. Ein direkter Zugriff auf `issn` dagegen funktioniert nicht, da diese Eigenschaft nur auf dem Interface `Zeitschrift` definiert ist. Eine von vielen Möglichkeiten, diese Typen wieder aufzulösen, besteht darin, auf die Existenz einer Eigenschaft mit dem `in`-Schlüsselwort zu testen (siehe Listing 1).

Selbstverständlich bietet TypeScript eine ganze Menge mehr Features, aber das Behandelte sollte ausreichen, um ein kleines Programm zu schreiben und mit dem Experimentieren zu beginnen.

TypeScript einrichten

Beim Einrichten von TypeScript muss zunächst ein neuer Ordner angelegt, eine package.json generiert und das Paket typescript installiert werden:

```
$ mkdir test
$ cd test
$ npm init -y
$ npm install typescript
```

Im Anschluss muss der Entwickler die Konfigurationsdatei tsconfig.json erstellen, was er dem TypeScript Compiler tsc überlassen kann:

```
$ npx tsc --init
```

Die dadurch erzeugte tsconfig.json ist (abzüglich vieler Kommentare) in Listing 2 zu finden.

`target` legt fest, für welche Zielplattform der Code transpiliert wird. Vermutlich dürfte man hier zwischen `es5` (sehr alte Browser) oder `esnext` (aktuelle Standards) wählen: `es5`, wenn man sich um nichts mehr kümmern möchte, und `esnext`, wenn man sowieso noch babel einsetzt, um den Code zielgerichtet für bestimmte Browser zu optimieren. In diesem Falle wäre es unsinnig, den Code zweimal zu transpilieren.

`module` beschreibt den Typ des ausgegebenen Moduls. Auch hier fällt die Entscheidung mit ähnlicher Begründung wie bei `target` zwischen `commonjs` und `esnext`.

`strict` stellt die strengstmögliche Typsicherheit ein und sollte immer aktiviert sein. Bei einer Migration kann es helfen, vorübergehend einige Optionen wie `noImplicitAny` abzuschalten. Auf lange Sicht sollte alles im `strict`-Modus laufen, damit TypeScript seine Stärken ausspielen kann.

`esModuleInterop` ist eine Option, Kompatibilität mit älteren Modulen zu schaffen, die statt der ESM-Schreibweise `export default …` die CommonJS-Schreibweise `module.exports = …` verwenden. Im Normalfall sollte diese Option eingeschaltet bleiben.

Legt der Entwickler eine .ts-Datei an und führt mittels

```
$ npx tsc
```

den Compiler aus, erscheint direkt neben der .ts-Datei eine .js-Datei. Da sich diese Vermischung aus Quellcode und Build-Artefakten schnell als störend erweist, erweitert man die tsconfig.json um folgende Werte:

```
{
 "compilerOptions": {
  "outDir": "dist"
 },
"include": ["src/**/*"]
}
```

Listing 3: Das Beispielprogramm holt Daten von einer API ab

```
import fetch from "node-fetch";

interface ApiResult<T> {
 data: T;
}

interface User {
 id: number;
 email: string;
}

function isApiResult(obj: any): obj is ApiResult<unknown> {
 return "data" in obj;
}

function isUser(obj: any): obj is User {
 return obj && typeof obj.id === "number" && typeof obj.email ===
                                                          "string";
}

function assertUserApiResult(obj: any): asserts obj is ApiResult<User>
{
 if (!isApiResult(obj) && isUser(obj.data)) {
  throw new Error("not a valid API result");
 }
}

async function fetchUser(id: number): Promise<ApiResult<User>> {
 const response = await fetch(`https://reqres.in/api/users/${id}`);
 const json: unknown = await response.json();
 assertUserApiResult(json);
 return json;
}

fetchUser(2).then(result => console.log(result.data.email));
```

Jetzt kommen beim Ausführen des Compilers nur noch Quelldateien aus dem Ordner src sowie seinen Unterordnern zum Einsatz und die Build-Artefakte landen im Verzeichnis dist. Im letzten Vorbereitungsschritt startet man den TypeScript-Compiler im `watch`-Modus:

```
$ npx tsc --watch
```

Alle Dateien, die sich ab jetzt im src-Ordner befinden, werden überwacht und automatisch transpiliert. Mittels

```
$ node dist/datei.js
```

lassen sie sich ausführen.

Erste Schritte in TypeScript

Listing 3 veranschaulicht als Beispielprogramm einige Besonderheiten von TypeScript. Zunächst importiert man das Paket node-fetch. Vor dessen Einsatz muss der Entwickler es noch per `npm install node-fetch` installieren. Das genügt in diesem Fall jedoch nicht, der Compiler liefert „Could not find a declaration file for module ‚node-fetch'" zurück. node-fetch ist ein in JavaScript geschriebenes Paket. Einige dieser Pakete beinhalten von Haus aus bereits Typdefinitionen für TypeScript, hier müssen sie noch nachinstalliert werden. Wie für die meisten anderen derartigen Pakete liefert die Community im `npm-scope @types` eine entsprechende Typdefinition. Es gilt also, zusätzlich das Paket @types/node-fetch zu installieren. Arbeitet man mit node als Zielplattform, empfiehlt es sich, zusätzlich das Paket @types/node zu installieren.

Neben der schon bekannten Form der Interface-Definition von `User` richtet der Entwickler ein generisches Interface `ApiResult<T>` ein. `T` dient als Platzhalter, der sich im späteren Gebrauch ausfüllen lässt.

Anschließend bestimmt man `isApiResult` und `isUser`. Sie gehören zu den Type-Predicate-Funktionen, mit deren Hilfe der Programmierer dem Compiler Informationen über den Typ eines Argumentes zur Verfügung stellen kann. Gibt die Funktion mit der Signatur `function isUser(obj: any): obj is User` als Ergebnis `true` zurück, geht der Compiler im sich ergebenden Scope davon aus, dass das Argument `obj` das Interface `User` implementiert.

Listing 4: fetchUser als Promise Chain statt asynchrone Function

```
function fetchUser(id: number): Promise<ApiResult<User>> {
 return fetch(`https://reqres.in/api/users/${id}`)
   .then(response => response.json())
   .then(json => {
     assertUserApiResult(json);
     return json;
   })
}
```

Ähnlich wirkt sich `assertUserApiResult` aus. Dahinter steckt eine neu mit TypeScript 3.7 eingeführte Assertion Function. Bei ihr spielt der Rückgabewert keine Rolle. Der Compiler nimmt an, dass die Funktion einen Fehler wirft, falls die Assertion nicht zutrifft, und setzt im weiteren Verlauf des Codes fest den beschriebenen Typ voraus. Zusätzlich zeigt sich, dass Interfaces in TypeScript lediglich `duck-typed` sind. Ein Objekt muss nicht explizit ein Interface implementieren, um es zu erfüllen. Interfaces beschreiben nur die Schnittstelle oder Form eines Objektes. Die Notwendigkeit für beide Funktionstypen ergibt sich aus einer weiteren Eigenschaft des Typsystems: Es ist ein reines Compile-Time-Typsystem. Ob ein von einer API zurückgelieferter Wert tatsächlich einem Typ entspricht, wird nicht automatisch zur Laufzeit überprüft. Stattdessen muss der Entwickler die entsprechende Logik schreiben.

Selbstverständlich muss das nicht von Hand geschehen. Beispielsweise erlaubt die Bibliothek yup, über eine fluent-API Validierungsschemas mit automatisch inferierten TypeScript-Typen zu definieren. Alternativ lassen sich auch mit diversen Tools Swagger-Definitionen, JSON-Schemas oder GraphQL-Schemas validieren und in TypeScript-Interfaces übersetzen. Ganz Mutige können per `as`-Schlüsselwort den Zieltyp vorschreiben, ohne die Daten vorher zu validieren. `fetchUser` ist eine mit dem Schlüsselwort `async` markierte asynchrone Funktion. Innerhalb dieser Methode lassen sich Promise Chains imperativ ausschreiben, indem man mittels `await` auf den Rückgabewert eines `Promise` wartet. In Promise-Schreibweise sähe diese Funktion aus wie in Listing 4.



2× Make testen und 6 € sparen!

Ihre Vorteile:

- ✓ **GRATIS dazu:** Arduino Nano
- ✓ **Zugriff auf Online-Artikel-Archiv***
- ✓ **NEU:** Jetzt auch im Browser lesen!
- ✓ Zusätzlich digital über iOS oder Android lesen



Für nur 15,60 Euro statt 21,80 Euro.

* Für die Laufzeit des Angebotes.

Jetzt bestellen:

make-magazin.de/miniabo

Das mag im vorliegenden Falle noch recht gut lesbar sein, mündet aber schnell in der altbekannten „promise hell". Asynchrone Funktionen erhöhen die Lesbarkeit des Codes spürbar. Letztendlich nutzen Entwickler*innen die Funktion `fetchUser` aber doch mittels Promise-Schreibweise, da der Modultyp `commonjs` das Nutzen von `top-level-await` nicht zulässt. Hierzu wäre die Transpilation zu einem ESM-Modul mit einem höheren Zielsprachlevel nötig.

Dieses Codebeispiel zielt auf die Node-Plattform ab. Der Einsatz im Browser wäre hier möglich, wenn man das `import`-Statement entfernte, da die `fetch`-Methode in modernen Browsern sowieso schon vorhanden ist. Wollte man mit der Zielplattform „Browser" mit externen Bibliotheken arbeiten, wäre ab diesem Moment der Einsatz eines Bundlers wie parcel oder webpack unabdingbar. Der TypeScript-Compiler bündelt selbst keine externen Abhängigkeiten.

Sobald der erste Code geschrieben ist, sollte man einen Blick in das Transpilat werfen und in der tsconfig.json zwischen den verschiedenen `targets` und `module`-Typen hin- und herwechseln. Während der mit es5/amd transpilierte Code zwar in jedem Browser läuft, aber kaum lesbar ist, zeigt sich bei der Kombination esnext/esnext eine weitere Stärke von TypeScript: Der Code entspricht eins zu eins dem ursprünglichen Code, lediglich bereinigt um Typinformationen. Eine Rückmigration zu purem JavaScript ist auf diesem Wege jederzeit möglich. Man kann TypeScript also gefahrlos im Projekt ausprobieren, ohne sich den Weg zurück zu verbauen.

Der Ursprung der Typen

Wie bereits angedeutet liefern nicht alle Pakete auf npm direkt Definitionen für TypeScript mit. Für viele dieser Pakete findet man Typdefinitionen aus der Community im npm-Scope @types. Diese Typen werden zentral im GitHub-Repository Definitely Typed/DefinitelyTyped verwaltet.

Ursprünglich als Communityprojekt gestartet, handelt es sich bei diesem Repository seit TypeScript 2.0 um eine beeindruckende Kooperation zwischen Microsofts TypeScript-Team und der Community, mit einer Aktivität von über 500 gemergten Pull Requests pro Monat, also Änderungsvorschlägen aus der Community. Hier sollte man für fast jede Bibliothek Typinformationen finden. Falls nicht, muss man in den sauren Apfel beißen und selbst welche für ein externes Modul schreiben. Das ist jedoch heutzutage nur noch sehr selten erforderlich. Wie man dabei vorgeht, steht in der exzellenten Dokumentation zur Sprache. Wer mehr über die Geschichte dieses Projekts erfahren möchte, dem sei der Blogpost „DefinitelyTyped: The Movie" von John Reilly nahegelegt (siehe ix.de/zaf1).

Einsteiger mögen jetzt versucht sein, ihre Applikationen an möglichst vielen Stellen mit möglichst vielen Typinformationen anzufüttern. Davon ist abzuraten. Im besten Fall handelt es sich um eine unnötige Fingerübung, in vielen Fällen kommt es dadurch, dass man einen Wert einer weniger spezifisch typisierten Variablen zuweist, zum Verlust von Typinformationen.

In der Regel sollte man Typen nur dort annotieren, wo dem Compiler keine Typinformationen vorliegen (etwa bei Funktionsargumenten oder bei Variablen, die erst im späteren Programmfluss einen Wert zugewiesen bekommen) oder man einen Typ bewusst beschränken möchte (beispielsweise beim Rückgabewert einer Funktion). Letzteres ist von Interesse, wenn man in der Lage ist, den Typ genauer zu beschreiben, als der Compiler ihn selbst herleiten könnte. Das spielt manchmal in Kombination mit Generics eine Rolle. Nicht zu unterschätzen ist hier auch das Konstrukt der `as-const`-Assertion, das insbesondere bei Arrays mit gemischten Typen einen großen Unterschied ausmachen kann. So würde das Konstrukt

```
function wrap(a: string, b: number) { return [a, b] }
```

den hergeleiteten Rückgabewert `Array<string | number>` haben. Im Gegensatz dazu hätte das Konstrukt

```
function wrap(a: string, b: number) { return [a, b] as const }
```

den hergeleiteten Rückgabewert `[string, number]` – was die API der Funktion deutlich besser beschreibt, da es genau spezifiziert, dass das zurückgegebene Array zwei Werte beinhaltet, von denen der erste ein `string` und der zweite eine `number` ist.

Der Kaninchenbau ist tief

Wer sich wohl fühlt mit TypeScript, kommt irgendwann an den Punkt, an dem er eine neue API nicht mehr prototypisch in JavaScript implementiert, sondern einfach nur noch ihre Schnittstellen beschreibt, um zu testen, wie sie zu bedienen sind. Das könnte man als Type-driven Development bezeichnen. Infolgedessen ergeben sich tatsächlich einfachere und weniger überladene Schnittstellen, denn es ist zwar möglich, fast jede noch so dynamische API mittels TypeScript-Typen abzubilden, die Typdefinitionen können jedoch komplex geraten. Und das ist wiederum ein guter Indikator dafür, dass auch ein Mensch mit dem überladenen Verhalten unter Umständen überfordert wäre, anstatt davon zu profitieren.

Dennoch kann es natürlich am Dreh- und Angelpunkt einer Applikation oder interner Bibliotheken sinnvoll sein, aufwendigere Typen zu schreiben, um komplexes Verhalten abzubilden. TypeScript stellt alles dazu Notwendige zur Verfügung. Dank Generics, Mapped Types und Conditional Types kann das Typsystem, hat man sich einmal mit seinen Besonderheiten angefreundet, auch bei komplexen Typen Spaß machen. Eindrucksvoll demonstriert das beispielsweise der GitHub-Nutzer acutmore, der als Proof of Concept eine eigene 4-Bit-Assembler-Sprache auf Typebene implementiert hat, um dann darin die Fingerübung Fizz Buzz zu programmieren.

Wer Lust bekommen hat, sich intensiver mit der Sprache auseinanderzusetzen, dem sei neben der offiziellen Dokumentation das frei verfügbare Onlinebuch „TypeScript Deep Dive" ans Herz gelegt. Möchte man beginnen, TypeScript in einer React-Applikation zu verwenden, ist zudem das React-TypeScript-Cheatsheet einen Blick wert (siehe ix.de/zaf1). (jd@ix.de)

Quellen

Weitere Informationen unter ix.de/zaf1



Lenz Weber

ist Full-Stack-Developer bei Mayflower und Core-Contributor bei redux-toolkit und fork-ts-checker-webpack-plugin. Er bewegt sich größtenteils im Web- und Securityumfeld und ist auf Twitter unter dem Handle @phry zu finden.

ISDX

INTERNET SECURITY DAYS 2020

Sichern Sie sich jetzt Ihr Ticket! https://isd.eco.de

17. – 18. September
Phantasialand, Brühl

FACHMESSE, KONFERENZ, NETWORKING – DIE PLATTFORM FÜR SECURITY-EXPERTEN

XPERTS
XCELLENCE
XPERIENCE

Der eco – Verband der Internetwirtschaft e. V. und heise Events laden Sie herzlich zu den 10. Internet Security Days (ISD X) ein. Tauschen Sie sich mit führenden IT-Sicherheits-Experten, Sicherheitsverantwortlichen und Anwenderunternehmen aus. Auf der Agenda der Security-Sessions und Workshops stehen Strategien für aktuelle Bedrohungen, praktische Use Cases und Trends der IT-Sicherheit. Das Konferenzprogramm mit begleitender Ausstellung ermöglicht Networking und Spaß. Dafür sorgen schon die fantastische Veranstaltungsumgebung des Phantasialands bei Köln und die Internet Security Night.

Platin Partner

HUAWEI

Silber Partner

IT-SEAL
SOCIAL ENGINEERING ANALYSIS LABS

heise Events
Conferences, Seminars, Workshops

eco 25



Einführung in ClojureScript

Unterwegs in zwei Welten

Maria Geller

ClojureScript ist eine dynamisch typisierte, funktionale Programmiersprache, die nach JavaScript kompiliert wird. Sie will die Vorteile von Clojure, wie persistente Datenstrukturen und datenorientiertes Programmieren, auch in die Welt der Webentwicklung transportieren.

Als ClojureScript in 2011 das Licht der Welt erblickte, sagte Rich Hickey, Erfinder von Clojure: „Why are we doing this? Because Clojure rocks, and JavaScript reaches." Mittlerweile kann man dem Zitat noch mehr Gewicht zuordnen, denn JavaScript läuft heute nicht nur in Browsern, sondern lässt sich dank Node.js auch für die Serverprogrammierung oder mittels React Native für die Entwicklung mobiler Apps einsetzen. ClojureScript greift die Vorteile von Clojure, wie persistente Datenstrukturen und datenorientiertes Programmieren, auf und nutzt JavaScript als Plattform, um Programmierer*innen den Weg in möglichst vielen Bereichen der Softwareentwicklung zu ebnen.

Dieser Artikel stellt die wichtigsten Bausteine zum grundlegenden Verständnis der Sprache vor und greift dann ein Beispiel auf, um die Entwicklung mit ClojureScript zu demonstrieren. Dabei kommen auch Details der Webentwicklung mit ClojureScript zur Sprache.

ClojureScript als Lisp-Dialekt

Da ClojureScript ein Lisp-Dialekt ist, ist der Code in Präfixnotation geschrieben. Entsprechend besetzt die erste Position in der Liste üblicherweise eine Funktion. Ein anderer fester Bestandteil jedes Lisp-Dialekts sind Klammern, die auf den ersten Blick abschreckend wirken können, jedoch den Vorteil haben, die Sprache homoikonisch zu machen. Das heißt, der Code selbst ist in Datenstrukturen der Sprache repräsentiert. Im Fall von Lisp handelt es sich dabei um Listen. Diese Eigenschaft machen sich Makros zunutze. Sie dienen zur Metaprogrammierung.

Die REPL, kurz für Read-Eval-Print Loop, spielt in Clojure und ClojureScript eine wichtige Rolle und ermöglicht interaktives Programmieren. Sie nimmt Ausdrücke entgegen, evaluiert sie und gibt ihren Wert zurück. Nachfolgend steht `=>` für den REPL-Prompt und `==>` für den Rückgabewert.

Jeder Ausdruck in ClojureScript gibt einen Wert zurück. Bei der Auswertung von konstanten Ausdrücken wie Nummern, Strings oder Booleans werden diese unverändert zurückgegeben.

```
=> 1
==> 1
=> true
==> true
```

Eine andere Art von Ausdrücken sind Funktionsaufrufe. Diese haben die Form einer Liste: `(<Funktionsname> <Ausdruck> <Ausdruck>...)`. Entwickler*innen können Funktionsaufrufe schachteln.

```
=> (+ 1 (* 2 3))
==> 7
```

Listing 1: Werte via Vars benennen

```
=> (def x 1)
==> #'cljs.user/x

=> (inc x)
==> 2

=> x
==> 1
```

Listing 2: Projektstruktur in ClojureScript

```
hello-world
      src
          hello_world
              core.cljs
      deps.edn (macOS/Linux)
      cljs.jar (Windows)
      index.html
```

Um die Auswertung einen Ausdrucks zu Unterbinden, kann man diesem einen Apostroph voranstellen.

```
=> '(+ 1 (* 2 3))
==> (+ 1 (* 2 3))
```

Datentypen

In ClojureScript finden sich bekannte Datentypen wieder, deren Implementierungen von der zugrunde liegenden Hostplattform JavaScript abhängen. `Nil` zum Beispiel ist in ClojureScript das Pendant zu JavaScripts `null`. Die Boolean-Werte `true` und `false` sind durch die entsprechenden Booolean-Werte in JavaScript implementiert. Die Implementierungen für ganze Zahlen (Integer) und Gleitkommazahlen (Floating Point) in ClojureScript beruhen beide auf JavaScripts Gleitkommazahlen, da dies der einzige Zahlentyp in JavaScript ist. Neben Zahlen, Booleans, Strings und Characters existieren in ClojureScript auch Keywords und Symbole.

Keywords erkennt man an dem führendem Doppelpunkt, etwa `:keyword`. Werden Keywords evaluiert, geben sie sich selbst zurück. Sie dienen häufig als Schlüssel in assoziativen Datenstrukturen wie Maps und können außerdem als Funktionen fungieren.

```
=> :keyword
==> :keyword
=> (keyword "clojurescript")
==> :clojurescript
```

Symbole sind ein fester Bestandteil eines jeden ClojureScript-Programms. Entwickler*innen nutzen sie, um Einheiten in ClojureScript, etwa Funktionen, einen Namen zu geben.

```
=> 'symbol
==> symbol
=> (symbol "clojurescript")
==> clojurescript
```

Datenstrukturen in ClojureScript

Persistente Datenstrukturen bilden einen fundamentalen Teil von ClojureScript. Werte sind unveränderlich und jede Operation auf einer persistenten Datenstruktur liefert eine neue Datenstruktur zurück. Dabei erzeugt der Compiler keine komplette Kopie, sondern verwendet gleiche Strukturen wieder.

An dieser Stelle lohnt sich zunächst einen Blick auf die zentralen Datenstrukturen in ClojureScript. Listen repräsentieren

iX-TRACT

- Als Lisp-Dialekt ist der ClojureScript-Code in Präfixnotation geschrieben und in Datenstrukturen der Sprache repräsentiert.
- Die funktionale Sprache verfügt über persistente Datenstrukturen, Werte lassen sich nicht verändern und jede Operation auf einer persistenten Datenstruktur liefert eine neue Datenstruktur zurück.
- Ähnlich wie in JavaScript können Entwickler*innen Callbacks und Promises nutzen, um asynchrone Programme zu schreiben.

Listing 3: JavaScript-Datei laden

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
    </head>
    <body>
        <script src="out/main.js"" type="text/javascript"></script>
    </body>
</html>
```

Listing 4: JavaScript-Promises in src/hello_world/promises.cljs

```
(ns hello-world.promises)

(enable-console-print!)

(defn fetch-data+ []
  (println "Fetching data using a promise...")
  (js/Promise. (fn [resolve]
                 (js/setTimeout #(resolve 42) 5000))))

(defn main []
  (-> (fetch-data+)
      (.then #(println "Processing data:" %))))

(main)
```

Programmierer*innen mithilfe runder Klammern. Zusätzlich braucht es allerdings vor der öffnenden Klammer einen Apostroph, um die Auswertung als Funktionsaufruf zu unterbinden. Listen in ClojureScript sind auf sequenziellen Zugriff optimiert.

```
=> '(1 2 3)
==> (1 2 3)
=> (list 1 2 3)
==> (1 2 3)
```

Vektoren nutzen eckige Klammern. Anders als Listen ermöglichen Vektoren einen effizienten indexbasierten Zugriff auf Elemente.

```
=> [1 2 3]
==> [1 2 3]
=> (vector 1 2 3)
==> [1 2 3]
```

Maps bestehen aus Schlüssel- oder Wertpaaren und sind durch geschweifte Klammern gekennzeichnet.

```
=> {:foo "bar" :baz 20}
==> {:foo "bar" :baz 20}
=> (get {:foo "bar" :baz 20} :foo)
==> "bar"
```

Mengen enthalten eindeutige Elemente des gleichen oder unterschiedlichen Typs. Ähnlich wie Maps weisen sie geschweifte Klammern auf, zusätzlich stellt man ihnen eine Raute voran.

```
=> #{1 2 3}
==> #{1 2 3}
=> (set '(1 2 2 3))
==> #{1 2 3}
```

Um Werten einen Namen zu geben, nutzen Entwickler*innen Vars. Sie sind als Referenztyp implementiert und lassen sich mittels `def` erzeugen (Listing 1). Im Beispiel erzeugt man ein Var mit dem Namen x und dem Wert 1. Ruft man die Funktion `inc` mit der Var auf, wird der Wert 2 zurückgegeben. Der Wert von x bleibt dabei unverändert.

Funktionen

Funktionen in ClojureScript lassen sich mit `defn`, `fn` oder `#()` definieren.

```
=> (def add-two (fn [x] (+ x 2)))
==> #'cljs.user/add-two
=> (add-two 1)
==> 3
```

Im Beispiel erzeugt man eine Variable mit dem Namen `add-two` und weist ihr eine Funktion zu. Dieser Prozess lässt sich mit `defn` verkürzen.

```
=> (defn subtract-one [x] (- x 1))
==> #'cljs.user/subtract-one
=> (subtract-one 2)
==> 1
```

Wie üblich für eine funktionale Sprache sind Funktionen in ClojureScript First-Class-Objekte. Das heißt, sie können als Übergabe- oder Rückgabewert einer Funktion auftreten oder lassen sich einer Variablen zuweisen.

```
=> (map inc [1 2 3 4])
==> (2 3 4 5)
```

`map` ist eine Funktion, die als ersten Parameter eine andere Funktion entgegennimmt, im Beispiel `inc`, und diese auf jedes Element der Sammlung anwendet.

Während man in den meisten Fällen ohne Seiteneffekt auskommt, ist es jedoch manchmal nötig, den Wert einer Variablen zu verändern. Hierfür stellt ClojureScript Atome zur Verfügung. Zugriffe auf Atome sind synchron und atomar, das heißt, der Wert wird entweder vollständig überschrieben oder gar nicht.

```
=> (def state (atom {:clojure "rocks"}))
==> #'cljs.user/state
=> @state
==> {:clojure "rocks"}
```

Atome erzeugen Entwickler*innen mit `atom` und dereferenzieren sie mittels `deref` oder `@`. Um den Wert eines Atoms zu ändern, nutzt man `swap!`. `swap!` wendet die Funktion auf den momentanen Wert des Atoms an, um den neuen Wert zu erhalten.

```
=> (swap! state (fn [current-value] 7
                           (assoc current-value :javascript"reaches")))
==> {:clojure "rocks", :javascript "reaches"}
```

Zusätzlich setzt `reset!` den Wert zurück.

```
=> (reset! state {})
==> {}
```

Mit JavaScript Interop JavaScript-Funktionen aufrufen

ClojureScript gibt Entwickler*innen die Möglichkeit, von der Hostsprache Gebrauch zu machen und direkt JavaScript-Funktionen aufzurufen. JavaScript-Namensräume sind mit dem Präfix `js` zugänglich. Um auf Objekteigenschaften zuzugreifen, lässt sich eine Punktnotation nutzen. Zusätzlich stellt man dem Eigenschaftsnamen einen Bindestrich voran. Folgender Code fragt die Breite des Browserfensters ab:

```
=> (. js/window -outerWidth)
==> 1831
```

Die Punktnotation dient außerdem dazu, Funktionen aufzurufen. Programmierer*innen können die Punktnotation außerdem dazu einsetzen, Funktionen aufzurufen.

```
=> (. js/console log "Hello ClojureScript")
"Hello ClojureScript"
==> nil
```

Das Beispiel zeigt, dass zunächst `"Hello ClojureScript"` als Seiteneffekt ausgegeben wird. Der Ausdruck `(. js/console log "Hello ClojureScript")` liefert den Rückgabewert `nil`.

JavaScript-Bibliotheken und Codeoptimierung mit Google Closure Tools

Der ClojureScript-Compiler macht in mehrfacher Hinsicht Gebrauch von Google Closure Tools. Google Closure Tools ist eine Sammlung von Hilfsmitteln, die Programmierer*innen beim Entwickeln von Webanwendungen mit JavaScript unterstützen.

Jedes ClojureScript-Projekt kann die Google-Closure-Bibliothek nutzen. Dahinter steckt eine umfangreiche JavaScript-Bibliothek, die Funktionen für DOM-Manipulation, Serverkommunikation, Animationen, Datenstrukturen und vieles mehr bereitstellt.

```
=> (require '[goog.math :as math])
==> nil
=> (math/modulo 5 3)
==> 2
```

Zusätzlich lässt sich der JavaScript-Code, der vom ClojureScript-Compiler erzeugt wird, mit dem Google-Closure-Compiler optimieren. Dabei entfernt dieser zum Beispiel Leerzeichen, Kommentare und gibt Variablen kürzere Namen. In einem weiteren Schritt kann er den Sourcecode analysieren mit dem Ziel, ungenutzten Code zu entfernen, was die Größe der JavaScript-Dateien zusätzlich reduziert.

Hello, World

Nach der groben Übersicht über die Sprache schließt sich jetzt ein erstes „Hello World"-Beispiel an. Linux- oder macOs-Nutzer*innen reicht dafür eine aktuelle Installation von Clojure. Windows-Nutzer*innen brauchen Java 8 und müssen sich eine alleinstehende ClojureScript-Jar herunterladen. Ein minimales Beispiel folgt typischerweise der Projektstruktur wie in Listing 2.

In der deps.end-Datei sind Abhängigkeiten für das Projekt definiert. Statt JSON oder XML nutzt man EDN als Datenformat. EDN steht für Extensible Data Notation und ist eine Untermenge von Clojure. Die Abhängigkeiten lassen sich also in einer bekannten Syntax definieren. Der nachstehende Codeausschnitt beschreibt den Inhalt der Datei:

```
{:deps {:org.clojure/clojurescript {:mvn/version "1.10.520"}}}
```

Den ClojureScript-Code für das „Hello, World!" schreibt man in die Datei src/hello_world/core.cljs:

```
(ns hello-world.core)
(enable-console-print!)
(println "Hello, World!")
```

Listing 5: core.async Beispiel in src/hello_world/async.cljs

```
(ns hello-world.async
    (:require  [cljs.core.async :refer [chan put! take!]]))

(enable-console-print!)

(defn fetch-data [ch]
    (println "Fetching data using core.async...")
    (js/setTimeout #(put! ch 42) 5000))

(defn main []
    (let [ch (chan)]
      (fetch-data ch)
      (take! ch #(println "Processing data:" %))))

(main)
```

Listing 6: Startpunkt der Anwendung in src/cljs/counter/core.cljs

```
(ns counter.core
  (:require [reagent.core :as reagent]
            [re-frame.core :as re-frame]
            [counter.events :as events]
            [counter.views :as views]))

(defn ^:dev/after-load mount-root []
  (re-frame/clear-subscription-cache!)
  (reagent/render [views/main-panel] (.getElementById js/document
                                                      "app")))

(defn init []
  (re-frame/dispatch-sync [:initialize-db])
  (mount-root))
```

In der ersten Zeile erzeugen Entwickler*innen mit `ns` einen neuen Namensraum. Die Anweisung `(enable-console-print!)` setzt die dynamische Variable setzt die dynamische Variable `print-fn` auf `console.log`. Das bewirkt, dass Ausgaben in der Browserkonsole erscheinen. In der letzten Zeile ruft man die Funktion `println` auf, um den String `"Hello, World!"` auszugeben. In der Datei index.html lässt sich die JavaScript-Datei laden, die der ClojureScript-Compiler erzeugt hat (Listing 3). Um den Code nach JavaScript zu kompilieren, geben Programmierer*innen im Terminal ein:

```
clj --main cljs.main --compile hello-world.core
```

Und für Windows:

```
java -cp "cljs.jar;src" cljs.main  --compile hello-world.core
```

`--main` ruft eine Clojure-Funktion auf, in diesem Fall `cljs.main`. `cljs.main` nimmt unterschiedliche Befehle entgegen, einer davon ist `--compile`. Mit `--compile` spezifiziert man, dass man den `hello-world.core`-Namespace nach JavaScript kompilieren will. Sobald der Befehl fertig ist, lässt sich der kompilierte JavaScript-Code im Ordner `out` inspizieren. In der Datei out/hello_world/core.js zeigt sich zum Beispiel, dass

```
(println "Hello, World")
```

nach

```
cljs.core.println.call(null,"Hello, World!");
```

übersetzt wurde. Um das kompilierte Projekt auszuführen, öffnen Programmierer*innen die Datei index.html im Browser. In der Browserkonsole sollte `"Hello, World!"` als Ausgabe erscheinen.

Nebenläufigkeit

JavaScript ist historisch single-threaded, das heißt, Multithreading ist nicht Teil der Programmiersprache. Mittlerweile ist es aber möglich, JavaScript in mehreren Threads auszuführen. Das geschieht über die Web-Workers-API im Browser oder über das Worker-Threads-Modul in Node.js. Im Folgenden konzentrieren wir uns jedoch auf die Konzepte, die JavaScript im Rahmen der ECMAScript-Spezifikationen hinzugefügt wurden, um asynchrone Programme zu schreiben: Callbacks und Promises. Diese gleichen Konstrukte kann man ebenfalls in ClojureScript verwenden. Zur existierenden „Hello World"-Projektstruktur fügt man eine zweite Datei src/hello_world/promises.cljs hinzu (Listing 4).

Die Funktion `fetch-data+` gibt als Rückgabewert ein Promise zurück. Das Promise wird nach einem Time-out von fünf Sekunden mit dem Wert 42 erfüllt, indem man `resolve` aufruft. Sobald das Promise erfüllt ist, läuft in der `main`-Funktion der Callback, der sich mittels `then` registrieren lässt.

Um das Beispiel zu kompilieren, gibt man im Terminal ein:

```
clj --main cljs.main --compile hello-world.promises
```

Nachdem man die index.html-Datei im Browser geladen und die Browserkonsole geöffnet hat, sollte diese Ausgabe zu sehen sein:

Listing 7: Den App-Zustand initialisieren in src/cljs/counter/db.cljs

```
(ns counter.db)

(def default-db
  {:counter 0})
```

Listing 8: Event Handler in src/cljs/counter/events.cljs

```
(ns counter.events
  (:require [re-frame.core :refer [reg-event-db]]
            [counter.db :as db]))

(reg-event-db :initialize-db
  (fn [_ _]
    db/default-db))

(reg-event-db :set-counter
(fn [db [_ counter]]
   (assoc db :counter counter)))
```

```
:  Fetching data...
Processing data: 42
```

Neben den JavaScript-Konzepten zur asynchronen Programmierung können Entwickler*innen die ClojureScript-Library `core.async` verwenden. Sie richtet sich nach dem Standard Communicating Sequential Processes (CSP). Das tragende Konzept in `core.async` sind `channels`. Sie dienen als eine Art Förderband, um Nachrichten zu senden und zu empfangen (Listing 5).

Um `core.async` zu nutzen, wird der deps.end-Datei eine Abhängigkeit hinzugefügt:

```
{:deps {:org.clojure/clojurescript {:mvn/version 1.10.520"}
                :org.clojure/core.async {:mvn/version 1.1.587"}}}
```

`chan` erzeugt einen ungepufferten Channel, den man als Parameter in die `fetch-data`-Funktion reicht. In `fetch-data` lässt sich der Wert 42 nach einem Time-out von fünf Sekunden mittels `put!` in den Channel übertragen. Mit `take!` können Entwickler*innen Werte aus dem Channel nehmen und in einer Callback-Funktion verarbeiten. Wie zuvor kompiliert man den Code und kann das Ergebnis in der Browserkonsole nach einem Refresh sehen.

```
clj --main cljs.main --compile hello-world.async
```

Anstatt Projekte direkt mithilfe des `cljs.main`-Namespace zu bauen, verwenden Programmierer*innen häufig Build-Tools wie `Leiningen` oder `Boot`. Diese ermöglichen unter anderem auch den Gebrauch von Projektvorlagen. Sie vereinfachen den Start und das Set-up eines neuen Projektes stark. Im weiterführenden Beispiel kommt `Leiningen` zum Einsatz.

Funktional und reaktiv programmieren

Reagent ist als Bibliothek in ClojureScript-Projekten weit verbreitet und bildet eine Schnittstelle zwischen ClojureScript und der JavaScript-Bibliothek React.

Mittels React schreiben Entwickler*innen wiederverwendbare Komponenten für Webanwendungen. React kümmert sich im Hintergrund darum, Änderungen an Daten innerhalb der Komponenten abzubilden. Daneben sorgt es dafür, dass nur die Teile des User Interface, die sich geändert haben, neu gerendert werden.

Reagent taucht oftmals in Kombination mit re-frame auf, einem ClojureScript-Framework, um Single-Page Applications zu bauen. re-frame gibt Entwickler*innen eine Vorlage, wie sich der Zustand einer Anwendung verwalten lässt und der Datenfluss innerhalb der Anwendung aussieht.

Um ein neues Projekt mit Reagent und re-frame zu starten, lässt sich die Projektvorlage re-frame in Leiningen heranziehen:

```
lein new re-frame counter
```

Im erstellten counter-Verzeichnis liegt das fertige Gerüst, um eine Webanwendung zu schreiben. Diese können Programmierer*in-

nen mit `lein dev` ausführen. Navigieren sie im Browser zu http://localhost:8280, bekommen sie `"Hello from re-frame"` angezeigt.

Das re-frame-Konzept lässt sich als eine Schleife betrachten, mit unterschiedlichen Phasen, durch die die Daten der Anwendung fließen. Die erste Phase heißt Event Dispatch. Ein Event wird gesendet, wenn etwas in der Anwendung passiert, zum Beispiel wenn Nutzer*innen auf einen Knopf drücken oder eine Eingabe tätigen. Nach dem Event Dispatch folgt das Event Handling. Hier entscheidet man, wie mit dem Event umgegangen werden soll. Dies stößt man mithilfe von Funktionen an, auch als Event Handler bezeichnet.

Das Ergebnis eines jeden Event Handlings sind Seiteneffekte. Zu Seiteneffekten gehören zum Beispiel HTTP-Anfragen, aber auch das Aktualisieren des App-Zustandes. Mit re-frame können Entwickler*innen Funktionen schreiben, die die Daten im App-Zustand abfragen. Diese Datenabfragen heißen in re-frame Subscriptions. Subscriptions nutzt man im View-Teil der Anwendung. Um die richtige UI anzuzeigen, braucht es Daten aus dem App-Zustand, die Entwickler*innen über Subscriptions abfragen können.

Ist der Seiteneffekt eines Handlers ein neuer App-Zustand, löst das eine Kettenreaktion aus. Die Subscriptions werden neu berechnet, was wiederum dazu führt, dass neue Daten in der UI angezeigt werden müssen und die relevanten Komponenten neu gerendert werden. Um dieses Konzept zu unterstützen, unterteilt man ein Projekt mit re-frame in bestimmte Namespaces. Die hier wichtigen Dateien befinden sich in src/cljs/counter: core.cljs, db.cljs, events.cljs, subs.cljs, views.cljs.

Hinter der Beispielanwendung steckt ein einfacher Zähler. Die UI wird aus zwei Knöpfen bestehen, einem zum Hochzählen, einem anderen zum Zurücksetzen. Zusätzlich ist der momentane Wert des Zählers zu sehen. `core.cljs` enthält üblicherweise den Startpunkt einer Anwendung (Listing 6). In Zeile 9 des Codebeispiels fügt man das HTML, das man in `views/main-panel` erstellt, dem DOM-Knoten mit der ID `app` hinzu. In Zeile 12 initialisiert man den App-Zustand. `:initialize-db` ist dabei der Name eines Event Handlers, der ausgerufen wird, wobei `db` den App-Zustand bezeichnet. Den App-Zustand speichert man in einer Map. Hier setzen Programmierer*innen den initialen Wert des Zählers auf 0 (Listing 7).

In `events.cljs` sind die Event Handler definiert, in diesem Fall `:initialize-db` und `:set-counter`. Der erste Parameter beider Funktionen ist der aktuelle App-Zustand. Als zweiten Parameter übergibt man den Event-Vektor, der relevante Daten zu dem Event enthalten kann. Für `:set-counter` ist es der neue Wert des Zählers. Beide Handler geben einen aktualisierten App-Zustand zurück (Listing 8).

In `subs.cljs` definieren Entwickler*innen die Datenabfragen. In diesem Fall handelt es sich dabei um eine Subscription, die den aktuellen Wert des Zählers zurückliefert (Listing 9).

`view.cljs` nimmt die UI für die Zähleranwendung auf. Dabei nutzt man Hiccup, eine Clojure-Bibliothek, um HTML zu generieren. Vektoren repräsentieren HTML-Elemente. Die erste Position im Vektor bestimmt, um welches Element es sich handelt. Die Attribute für die Elemente lassen sich in einer Map definieren (Listing 10).

Die `main-panel`-Komponente gibt ein `div`-Element mit drei Kindelementen zurück. In Zeile 6 fragt man den momentanen Wert des Zählers durch die `:counter`-Subscription ab. Der Aufruf zu `subscribe` gibt ein `re-frame/atom` zurück. `@` dereferenziert es, um an den Wert des Zählers zu kommen. Ein `re-frame/atom` verhält sich wie ein normales ClojureScript-Atom.

Der Unterschied zu einem herkömmlichen Atom: Sobald sich der Wert eines dereferenzierten `re-frame/atom` verändert, läuft der Codeblock um dieses `re-frame/atom` erneut ab. Das heißt, die `main-panel`-Komponente wird neu gerendert, sobald sich der Wert des Zählers verändert. In Zeile 9 und 11 fügen Entwickler*innen den `button`-Elementen Event Handler für das `on-click`-Event hinzu. Wenn Nutzer*innen in der UI klicken, wird das Event `:set-counter` mit dem jeweiligen Wert gesendet. Die Event Handler für `:set-counter` aktualisieren den Wert im App-Zustand. Daraufhin wird die Subscription für `:counter` neu berechnet und gibt einen neuen Wert zurück. Dies hat zur Folge, dass die Komponente neu gerendert wird und der aktuelle Wert des Zählers erscheint.

Listing 9: Subscriptions auf den App-Zustand definieren in src/cljs/counter/subs.cljs

```
(ns counter.subs
  (:require [re-frame.core :refer [reg-sub]]))

(reg-sub :counter
  (fn [db]
    (:counter db)))
```

Listing 10: HTML generieren mit Hiccup in src/cljs/counter/views.cljs

```
(ns counter.views
  (:require [re-frame.core :refer [dispatch subscribe]]
            [counter.subs :as subs]))

(defn main-panel []
  (let [counter @(subscribe [:counter])]
    [:div
     [:div counter]
     [:button {:on-click #(dispatch [:set-counter (inc counter)])}
      "Count"]
     [:button {:on-click #(dispatch [:set-counter 0])}
      "Reset"]]))
```

Clojure Reaches

Rückblickend erscheint der Schritt, Clojure auf JavaScript zu portieren, sehr erfolgreich. Durch ClojureScript erweitert Clojure seine Reichweite und lässt sich nicht nur wie ursprünglich auf der JVM ausführen, sondern kann auch JavaScript als Plattform nutzen. Entwickler*innen können das umfangreiche JavaScript-Ökosystem und gleichzeitig Clojures Konzepte nutzen, um robuste Programme zu schreiben. Beide Sprachen weisen gemeinsame Features auf (dynamische Typisierung, Fokus auf Funktionen und interaktive Entwicklung über eine REPL), die dazu führen, dass sich Entwickler*innen mit Erfahrungen in jeder der Sprachen, ob Clojure oder JavaScript, schnell zu Hause fühlen und zielstrebig ihre Projekte umsetzen können.

Da ClojureScript selbst mittlerweile sehr stabil ist und Featureparität mit Clojure in den meisten Bereichen erreicht hat, wird der Fokus für die nächsten Jahre eher auf Tooling liegen. Ein wichtiger Bereich bleibt weiterhin, die Interoperabilität mit JavaScript zu verbessern, um den Wechsel für Entwickler*innen von JavaScript auf ClojureScript so problemfrei wie möglich zu gestalten. (csc@ix.de)

Quellen

Alle Listings unter: ix.de/zs7r



Maria Geller

ist Softwareentwicklerin bei Pitch und fühlt sich, seitdem sie 2015 im Rahmen von Googles Summer of Code am ClojureScript-Compiler mitgewirkt hat, in der Clojure-Community zu Hause.

software quality lab
Innovation Meets Quality

powered by iX

SEMINARE FÜR MEHR SOFTWARE QUALITÄT

MANAGEMENT & PROZESSE

Lernen Sie die Herausforderungen des Managements mit diesen Kursen zu meistern!

Change Management Cookbook für Führungskräfte	**06.08.2020**	**München**

PROGRAMMIERUNG & CODE

Sichern Sie nachhaltig das technische und wirtschaftliche Überleben Ihres Softwaresystems!

Code Review praktisch betrachtet	**30.07.2020**	**München**
Clean Code	**11.08.2020**	**München**

REQUIREMENTS ENGINEERING

Gute Requirements sind der Grundstein für ein erfolgreiches Projekt!

Anforderungen klar und deutlich schreiben	**29.07.2020**	**München**

TESTEN

Durch erfahrene Test- und QS-Experten unterstützen wir Sie bei der Durchführung Ihrer Qualitätssicherungs- und Test-Aktivitäten.

ISTQB Certified Tester - Advanced Level: Test Manager (CTAL-TM)	ISTQB International Software Testing Qualifications Board	**15.06.2020 - 19.06.2020**	**München**
Architekturzentriertes Testen		**01.09.2020-03.09.2020**	**Frankfurt**

E-ACADEMY

Weiterbildungen bequem vom Büro oder vom Homeoffice aus!

Infos unter https://www.software-quality-lab.com/leistungen/e-academy/

AKTION

Online-Kurse mit einem Spezial-Rabatt von 15%!

Infos unter https://www.software-quality-lab.com/aktuelles/aktion/



Einstieg in WebAssembly

Nachbrenner fürs Web

Sven Kölpin

WebAssembly ist seit 2019 ein offizieller W3C-Standard. Wasm-Anwendungen sind performanter als JavaScript-Code und eröffnen zudem den Zugriff auf die Ökosysteme von Sprachen wie C/C++ oder Rust.

Die Idee, andere Programmiersprachen als JavaScript in den Browser zu bringen, ist fast so alt wie die Programmiersprache selbst. Zahlreiche Versuche, andere Ökosysteme in die Webplattform zu integrieren, sei es über Browser-Plug-ins wie Flash oder mit alternativen Sprachen wie Google Dart, haben sich jedoch nicht durchgesetzt. Erfolgreicher sind JavaScript-Sprachaufsätze wie TypeScript oder Compile-to-JavaScript-Frameworks wie Vaadin. Mit ihnen lassen sich diverse Sprachen und Anwendungen nach JavaScript kompilieren und somit indirekt in den Browser bringen. Unter dem Strich sind diese Vorgehensweisen jedoch nicht optimal, weil sie entweder proprietäre Softwarekomponenten benötigen oder versuchen, etwaige Unzulänglichkeiten von JavaScript durch Werkzeuge zu beheben.

Mit WebAssembly (Wasm) haben die Browserhersteller einen Weg geschaffen, der es zumindest prinzipiell erlaubt, beliebige Programmiersprachen abseits von JavaScript auf eine standardisierte Weise für das Web zu verwenden. WebAssembly definiert ein Binärformat, dessen Ausführung eine Stack-basierte virtuelle Maschine übernimmt. Wasm ist keine neue Programmiersprache, sondern vielmehr ein Kompilierungsziel für diverse Sprachen. Für den Einsatz ist eine Laufzeitumgebung (virtuelle Maschine) vonnöten, die WebAssemblys Binärfor-

iX-TRACT

- WebAssembly ist ein Binärformat zum Ausführen von Code im Browser.
- Seit Dezember 2019 gilt WebAssembly als offizieller W3C-Standard.
- Primär lässt sich derzeit C/C++- und Rust-Code übertragen, aber prinzipiell ist WebAssembly für beliebige Sprachen offen.

Listing 1: wat-Repräsentation eines Wasm-Moduls

```
;; this module defines function
;; that calculates the sum of two 32-bit
;; numbers and returns the  result

;;module declaration
(module
        ;;import function "log" from browser
           (import "env" "log" (func $log))

           ;;declaration of function "sum"
    (func $sum (param $firstNo i32) (param $scndNo i32) (result i32)
            ;;add first param to stack
     get_local $firstNo
     ;;add second param to stack
            get_local $scndNo
     ;;add the numbers on the stack
     i32.add
     ;;call the imported log-function
     call $log
            ;;return the value from the current stack
     return
           )

           ;;export function "sum" to browser
           (export "sum" (func $sum))
)
```

Listing 2: wat im Zusammenspiel mit Docker

```
Windows:
docker run -it --rm -v %cd%:/data -w /data jungomi/wabt wat2wasm ↵
                                              ./example.wat  -o ./example.wasm
Unix:
docker run -it --rm -v $PWD:/data -w /data jungomi/wabt wat2wasm ↵
                                              ./example.wat  -o ./example.wasm
```

Listing 3: Compile API

```
(async () => {
    const myAgeModule = await WebAssembly.compileStreaming ↵
                                                  (fetch('./my_age.wasm'));
    //module can now be instantiated, cached, passed to a worker etc.

    //get meta-information about the module
    WebAssembly.Module.exports(myAgeModule); // [{name: "getMyAge", ↵
                                                      kind: "function"}]
    WebAssembly.Module.imports(myAgeModule); // []
})();
```

mat in Maschinencode übersetzt und ausführt. Die virtuelle Maschine für WebAssembly kann als Teil des Browsers oder auf beliebigen anderen Plattformen implementiert sein. Wasm ist nicht auf die Webplattform beschränkt, auf die das Binärformat ursprünglich ausgelegt war. Dass WebAssembly künftig außerhalb des Webs eine Rolle spielen könnte, zeigt der vor Kurzem ins Leben gerufene Standard WebAssembly System Interface (WASI). Er definiert eine API für die Interaktion von Wasm mit dem jeweiligen Betriebssystem und kommt in Grundzügen beim Cloud-Anbieter fastly zum Einsatz. Die aktuelle Version 1.0 von WebAssembly lässt sich seit 2018 in allen relevanten Browsern verwenden. Zudem ist Wasm seit Dezember 2019 offizieller W3C-Standard und somit fester Bestandteil der Webplattform.

Wasm 1.0 ist ein Minimum Viable Product (MVP) mit vornehmlich grundlegenden Funktionen. Der Einsatz ohne passende Hilfsmittel findet daher noch auf einem vergleichsweise geringen Abstraktionslevel statt. Zudem liegt der Fokus der ersten Version hauptsächlich auf den Sprachen C, C++ und Rust, obgleich zahlreiche Projekte für das Zusammenspiel mit anderen Sprachen existieren.

Flexible und schnelle Webanwendungen

WebAssembly öffnet die Türen des Webs nicht nur für andere Sprachen, sondern bietet weitere Vorteile. Vor allem bei der Performance kann das Format in bestimmten Fällen punkten. Im Gegensatz zu JavaScript, das als textuelles Format über die Leitung geht, sind die Daten bei Wasm binär und benötigen daher weniger Bandbreite. Im Browser laufen Wasm-Anwendungen zudem in fast nativer Geschwindigkeit auf dem jeweiligen Zielsystem. Besonders geeignete Anwendungsszenarien für WebAssembly sind deshalb ressourcenintensive Applikationen wie Video-/Audioanwendungen oder Spiele. Der reale Geschwindigkeitsvorteil gegenüber JavaScript hängt freilich vom konkreten Anwendungsfall ab.

Weiterhin ermöglicht WebAssembly das Portieren ganzer Anwendungen und Bibliotheken aus diversen Sprachen in das Web. Derzeit lassen sich vor allem Applikationen und Libraries, deren Codebasis in C oder C++ geschrieben ist, mit Wasm in den Browser bringen. Ein bekanntes Beispiel hierfür ist das Projekt AutoCAD, dessen Entwickler*innen Teile einer über dreißig Jahre alten Codebasis in den Browser portiert haben. eBay setzt ebenfalls auf WebAssembly und nutzt dafür eine von C++ nach Wasm portierte Bibliothek für Barcodes. Für Entwickler*innen bietet WebAssembly eine Vielzahl neuer Möglichkeiten, da das gesamte Ökosystem der jeweiligen Programmiersprache im Web zur Verfügung steht.

Friedliches Nebeneinander

Auf den ersten Blick scheint WebAssembly der erste ernst zu nehmende Konkurrent für JavaScript zu sein. Beim genaueren Hinschauen zeigt sich jedoch, dass Wasm noch nicht ohne JavaScript auskommt.

Zum Laden und Initialisieren von WebAssembly-Modulen ist eine standardisierte JavaScript-API notwendig. Zudem erlaubt die aktuelle MVP-Version keine direkte Interaktion zwischen WebAssembly und dem Document Object Model (DOM) oder anderen Web-APIs. Zum Verwenden von Objekten des Hostsystems (konkret des Browsers) ist die Integration eines Garbage Collector (GC) erforderlich, den der WebAssembly-Standard noch nicht vorsieht. In Wasm implementierte Module müssen deshalb für die Kommunikation mit der Nutzeroberfläche oder für das Absetzen von Requests derzeit den Umweg über JavaScript nehmen.

Die Macher des Standards betonen zudem, dass sie Wasm nicht entwickelt hätten, um JavaScript zu ersetzen. Vielmehr soll er die Webplattform vervollständigen und als Ergänzung zu JavaScript, HTML und CSS existieren, um unter anderem perfor-



mancekritische Anwendungsfälle im Browser umzusetzen. Das spiegelt sich in der Architektur wider: JavaScript und WebAssembly sind stark gekoppelt. Sie teilen sich einen Call Stack und bei WebAssembly greifen die gleichen Sicherheitsfunktionen wie in JavaScript. Unter anderem läuft Wasm wie die Skriptsprache in einer Sandbox und hat somit keinen Zugriff auf das Dateisystem, die Hardware oder beliebige Speicherregionen. Darüber hinaus greifen weitere Sicherheitsmechanismen von JavaScript wie die Same-Origin-Policy bei WebAssembly.

Obwohl WebAssembly viele Schwachstellen heutiger Webanwendungen behebt, wird der Standard JavaScript und dessen riesiges Ökosystem in absehbarer Zeit kaum verdrängen. Vielmehr ist eine Koexistenz beider Welten, in der JavaScript und Wasm gemeinsam ihre jeweiligen Stärken ausspielen können, ein vielversprechendes Szenario für zukünftige Webapplikationen.

Unter der Haube

In WebAssembly stehen derzeit vier nummerische Datentypen zur Verfügung:

- 32-Bit und 64-Bit Integer (i32 / i64),
- 32-Bit und 64-Bit Float (f32 / f64).

Boolesche Ausdrücke sind über die Werte 0 und 1 modelliert. Komplexere Datentypen wie Strings lassen sich nur über das lineare Memory-Modell abbilden.

Ein WebAssembly-Modul hat unterschiedliche Bestandteile. Neben der Definition von Funktionen und globalen Variablen, die die eigentliche Programmlogik des Moduls beschreiben, enthält es die Deklaration von Imports und Exports. Mit Imports kann WebAssembly-Code indirekt über JavaScript auf einzelne Funktionen des Browsers zugreifen. Die Export-Definition stellt in Wasm kompilierte Funktionen und Variablen zur Verwendung in JavaScript zur Verfügung. Zu den importier- beziehungsweise exportierbaren Werten zählen unter anderem Funktionen und globale Variablen.

Manche WebAssembly-Module benötigen expliziten Speicher in Form eines linearen Memory-Modells. Im Kern ist das ein ArrayBuffer und somit ein nicht veränderbares Bytearray. Es verhält sich grundsätzlich wie der Heap in C/C++, mit dem Unterschied, dass das Modul nur mit Speicheradressen interagieren kann, die sich innerhalb der Grenzen des übergebenen ArrayBuffers befinden. Damit ist WebAssembly vor ungültigen Speicherzugriffen und Speicherüberläufen geschützt.

Ein Wasm-Modul kann ausschließlich über die vier genannten numerischen Datentypen mit dem Hostsystem kommunizieren. Daher lassen sich Funktionsparameter, Rückgabewerte und Variablen nur als Zahlenwerte in ein WebAssembly-Modul hinein- beziehungsweise herausreichen. Der Austausch komplexerer Daten muss über das lineare Memory-Modell erfolgen.

Von wat zu wasm

Der WebAssembly-Standard definiert neben dem binären Format des Assemblercodes eine menschenlesbare textuelle Repräsentation von Wasm. Der in .wat-Dateien abgelegte Code ist hauptsächlich für Debugging-Zwecke oder das Umsetzen simpler Anwendungen geeignet. Es ist theoretisch aber denkbar, ganze Programme im wat-Format zu entwickeln. Es basiert auf einer S-Expression-Syntax, die eine Baumstruktur aufweist. Entwickler*innen müssen .wat-Dateien in das Wasm-Binärformat kompilieren, um sie ausführen zu können. Direkt im Browser lässt sich wat nicht verwenden.

Listing 1 zeigt ein im wat-Format definiertes Wasm-Modul. Es beschreibt eine Funktion, die die Summe zweier 32-Bit-Zahlen bildet, eine vom Browser übergebene Logfunktion aufruft und das Ergebnis zurückgibt. Die äußerste Klammer bildet die Deklaration des Moduls. Zeile 8 importiert eine Funktion mit dem Namen `log`. Es folgt die Definition der `sum`-Funktion (ab Zeile 11) und deren Export in Zeile 25.

Um das im wat-Format erstellte Wasm-Modul im Browser zu verwenden, müssen Entwickler*innen es zunächst in die binäre

Listing 4: Instantiate

```
(async () => {
    const myAgeModule = await WebAssembly.compileStreaming(fetch('./ ↵
                                                        my_age.wasm'));

    //returns the instance
    const instanceA = await WebAssembly.instantiate(myAgeModule);
    console.log(instanceA.exports); // {getMyAge: Function}

    // returns the compiled module AND the instance
    const {instance, module} = await WebAssembly.instantiateStreaming ↵
                                        (fetch('./wasm/my_age.wasm'));
    console.log(instance.exports); // {getMyAge: Function}

    //call the exported function
    console.log(instance.exports.getMyAge());
})();
```

Listing 5: Instantiate with imports

```
(async () => {
    const {instance} = await WebAssembly.instantiateStreaming ↵
                                        (fetch('./example.wasm'), {
        env: {
            //env-object contains all imports
            log: () => console.log('HI!!!')
        }
    });
    console.log(instance.exports.sum(30, 2)); // HI!!! 32
})();
```

Listing 6: check_primes.c

```
int isPrime(int num) {
  for (int i = 2; i < num; i++) {
    if (num % i == 0) {
      return 0;
    }
  }
  return (num != 1 && num != 0) ? 1 : 0;
}


int checkPrimes(int maxNumber) {
  int count = 0;
  for (int i = 0; i < maxNumber; i++) {
    if (isPrime(i) == 1) {
      count++;
    }
  }
  return count;
}
```

Listing 7: emscripten

```
Windows:
docker run --rm -v %cd%:/data -it trzeci/emscripten emcc -g /data/check_ ↵
     primes.c -s SIDE_MODULE=1 -Oz -s EXPORTED_FUNCTIONS=['_checkPrimes'] -o ↵
                                                        /data/checkPrimes.wasm"
Unix:
docker run --rm -v $PWD:/data -it trzeci/emscripten emcc -g /data/check_ ↵
     primes.c -s SIDE_MODULE=1 -Oz -s EXPORTED_FUNCTIONS=['_checkPrimes'] -o ↵
                                                        /data/checkPrimes.wasm"
```

Listing 8: JavaScript initialize

```
(async () => {
    const {instance} = await WebAssembly.instantiateStreaming ↵
                                    (fetch('./checkPrimes.wasm'), {
        env: {
            __memory_base: 0
        }
    });
    console.log(instance.exports._checkPrimes(123));
})();
```

Listing 9: emscripten js code

```
//compile with emscripten:
docker run --rm -v %cd%:/data -it trzeci/emscripten emcc -g ↵
/data/c/check_primes.c -s -Oz -s EXPORTED_FUNCTIONS=['_checkPrimes'] ↵
                                                -o /data/checkPrimes.js",

<script src="./checkPrimes.js"></script>
<script>
    Module.onRuntimeInitialized = () => {
        console.log(Module._checkPrimes(123));
    };
</script>
```

Wasm-Repräsentation umwandeln. Dafür existieren einige Werkzeuge, die im Browser ausführbar sind, wie WebAssembly Studio, oder lokal einzusetzen, wie das WebAssembly Binary Toolkit (WABT). Für die lokale Variante empfiehlt sich der Einsatz von Docker, um etwaige Installationsschritte zu umgehen. Auf der Basis von Docker lässt sich eine .wat-Datei wie in Listing 2 in das Wasm-Format verwandeln.

Hello World mit JavaScript und Wasm

Zum Ausführen einer Wasm-Datei im Browser ist JavaScript erforderlich. Der WebAssembly-Standard definiert dazu eine JavaScript-API zum Laden, Initialisieren und Verwenden von Wasm-Modulen im Browser.

Auf die API lässt sich von JavaScript aus über das globale WebAssembly-Objekt zugreifen. Sie stellt unter anderem Funktionen zur Verfügung, die ein Wasm-Modul kompilieren und Instanzen eines Moduls erzeugen: `instantiate` beziehungsweise `instantiateStreaming`. Wie die Funktionsnamen vermuten lassen, setzt sich das Ausführen von WebAssembly-Applikationen im Groben aus zwei Schritten zusammen: dem Kompilieren eines Moduls und dem Erzeugen von Instanzen.

Das Kompilieren eines WebAssembly-Moduls geschieht wahlweise über `WebAssembly.compile` oder `WebAssembly.compileStreaming`. Letztere Funktion kompiliert ein Modul direkt aus einem Stream heraus, der unter anderem ein HTTP-Response-Objekt sein kann. Die Funktion `compileStreaming` ist aufgrund der besseren Effizienz der von Mozilla empfohlene Weg.

Das Kompilieren erzeugt eine Instanz des Prototypobjekts `WebAssembly.Module`. Es ist statuslos und lässt sich anschließend mehrfach instanziieren, für eine spätere Verwendung beispielsweise in der IndexedDB cachen oder für Multithreading-Anwendungen an WebWorker übergeben. `WebAssembly.Module` liefert zudem statische Methoden, mit denen sich Metainformationen über ein kompiliertes Modul abfragen lassen. Entwickler*innen können darüber unter anderem alle Imports und Exports eines Wasm-Moduls auslesen. Listing 3 zeigt die Anbindung einer `compileStreaming`-API und der statischen Methoden. Der Code nutzt ein aus der im nächsten Codebeispiel gezeigten .wat-Datei erzeugtes WebAssembly-Modul. Es stellt eine `myAge`-Methode zur Verfügung, die einen festgelegten Wert von 102 zurückgibt.

```
(module
  (func $myAge (result i32)
    i32.const 102)
  (export "getMyAge" (func $myAge)))
```

Eigene Instanzen

Für das Erzeugen von Instanzen existiert die Funktionen `WebAssembly.instantiate` und `WebAssembly.instantiateStreaming`. Mit ersterer Funktion lassen sich Instanzen aus einem zuvor kompilierten WebAssembly-Modul oder aus den Wasm-Binärdaten (in Form eines TypedArrays oder ArrayBuffers) erstellen.

`WebAssembly.instantiateStreaming` ermöglicht das Kompilieren und direkte Instanziieren eines Moduls in einem einzelnen Aufruf. Damit darf der im vorherigen Abschnitt beschriebene Schritt des expliziten Kompilierens entfallen. In den meisten Fällen ist das Erstellen einer WebAssembly-Applikation mit der `WebAssembly.instantiateStreaming`-API der effizienteste Weg.

Der Rückgabewert beider Instanziierungs-APIs enthält stets ein Objekt des Typs `WebAssembly.Instance`. Dabei handelt es sich um eine statusvolle und ausführbare Instanz eines `WebAssembly.Module`-Objektes. Es enthält unter anderem alle vom Modul exportierten Funktionen, die sich aus JavaScript heraus aufrufen lassen. Listing 4 zeigt die Verwendung beider APIs. Das `exports`-Objekt der Instanz ermöglicht den Zugriff auf die vom Wasm-Modul exportierte Funktion `getMyAge`.

Import mit Ansage

Derzeit kann WebAssembly noch nicht direkt mit den APIs des Browsers interagieren. Ein Wasm-Modul kann jedoch über die Import-Definition (vgl. Listing 2, Zeile 8) gewisse Erwartungen an den instanziierenden Code stellen. Beim Erzeugen der Instanz validiert die Laufzeitumgebung, ob alle geforderten Import-Objekte mit den erwarteten Signaturen und Rückgabewerten zur Verfügung stehen. Ist das nicht der Fall, bricht der Prozess ab und das Wasm-Modul lässt sich nicht verwenden.

Im Beispiel aus Listing 2 erwartet das WebAssembly-Modul die Übergabe einer Funktion mit dem Namen `log`. Entwickler*innen können der `WebAssembly.instantiateStreaming`-Funktion dazu ein JavaScript-Objekt als zweiten Parameter übergeben. Darin sind unter anderem Import-Objekte definierbar. Listing 5 zeigt die Instanziierung mit Imports.

Das Importieren ist ein vielseitiges Feature von WebAssembly. Neben der Übergabe eigener Funktionen lassen sich damit unter anderem ganze Bibliotheken des Hostsystems für Web-

Listing 10: primes_with log

```
//...
extern void logPrime(int prime);

int checkPrimes(int maxNumber) {
  int count = 0;
  for (int i = 0; i < maxNumber; i++) {
    if (isPrime(i) == 1) {
      logPrime(i);
      count++;
    }
  }
  return count;
}
```

Listing 11: js usage

```
(async () => {
    const {instance} = await WebAssembly.instantiateStreaming ↗
                            (fetch('./dist/checkPrimes_and_log.wasm'), {
        env: {
            __memory_base: 0,
            _logPrime: currentPrime => console.log('Found prime', ↗
                                                    currentPrime)
        }
    });
    console.log(instance.exports._checkPrimes(123));
})();
```

Assembly bereitstellen. Außerdem lassen sich exportierte Funktionen anderer WebAssembly-Module importieren und damit Dynamic Linking realisieren. Vor allem für große Wasm-Applikationen ist dieses Vorgehen sinnvoll, um sie in eine Vielzahl kleinerer Module zu unterteilen und damit die Netzwerklast und Kompilierzeit im Browser zu reduzieren.

Gut kompiliert

Der größte Mehrwert des WebAssembly-Binärformats liegt darin, dass es ein Kompilierungsziel für beliebige Sprachen ist. Die Übersetzung einer Programmiersprache in das Binärformat ist allerdings nicht im Wasm-Standard definiert. Die Umsetzung muss innerhalb der jeweiligen Sprachökosysteme erfolgen.

Das Hauptaugenmerk von WebAssembly liegt momentan auf den Sprachen C und C++, die keinen Garbage Collector benötigen. Zusätzlich bieten Rust und der TypeScript-zu-WebAssembly-Compiler AssemblyScript eine gute Anbindung. Andere höhere Sprachen wie Java oder C# sind im Hinblick auf Wasm noch recht eingeschränkt. Für das Übersetzen auf WebAssembly ist neben dem eigentlichen Code das Kompilieren und Mitliefern der eigenen Runtime beziehungsweise des GC zu Wasm erforderlich. Diese Vorgehensweise kommt zum Beispiel bei C# mit Blazor oder bei Go zum Einsatz.

Von C/C++ zu WebAssembly

C und C++ lassen sich mit dem Emscripten-Toolkit nach WebAssembly kompilieren, einem der ausgereiftesten Werkzeuge, um C/C++ in die Wasm-Welt zu bringen. Emscripten hat bereits beim Portieren der Codebasis von AutoCAD, der Unreal Engine 3 und von SQLite nach WebAssembly geholfen.

Neben dem eigentlichen WebAssembly-Modul kann Emscripten JavaScript- und HTML-Dateien generieren. Sie schaffen eine zusätzliche Abstraktionsschicht, indem sie die Initialisierungslogik für WebAssembly kapseln und die Wasm-Module automatisch kompilieren und instanziieren. Die von Emscripten generierte JavaScript-Datei enthält zusätzlich eine Vielzahl an Hilfsfunktionen, die die Kommunikation zwischen JavaScript und WebAssembly vereinfachen. Dieser JavaScript-Glue-Code ist besonders sinnvoll für den Austausch komplexer Datentypen über numerische Werte mit dem WebAssembly-Modul. Der Preis dafür ist allerdings eine große JavaScript-Glue-Code-Datei, die zusätzlich zum Wasm-Modul bereitstehen muss. Emscripten erlaubt die Übergabe unterschiedlicher Parameter, mit denen unter anderem steuerbar ist, ob es neben dem WebAssembly-Modul zusätzlich den JavaScript-Glue-Code generieren soll.

Für den lokalen Einsatz von Emscripten empfiehlt es sich erneut, ein Docker-Image zu verwenden, um den sonst recht aufwendigen Installationsprozess zu beschleunigen. Beispielsweise steht das Image trzeci/emscripten zur Verfügung.

Hello World in C und WASM

Als Beispielanwendung soll das Kompilieren eines in C geschriebenen Algorithmus zum Ermitteln von Primzahlen mit Emscripten zu WebAssembly dienen. Im ersten Schritt soll Emscripten zunächst keinen Glue-Code generieren. Listing 6 zeigt die Datei check_primes.c, die sich mit den in Listing 7 für Windows und Unix gezeigten Befehlen über das Docker-Image zu WebAssembly kompilieren lässt. Listing 8 zeigt schließlich die Initialisierung in JavaScript. Mit Emscripten generierte WebAssembly-Module erfordern zwingend den Import der globalen Variablen `__memory_base`. Sie ist vornehmlich für das Dynamic Linking von Modulen wichtig und lässt sich im konkreten Fall schlicht mit `0` initialisieren.

Wie erwähnt kann Emscripten die Initialisierungslogik und den JavaScript-Glue-Code generieren. Listing 9 zeigt den für die Anbindung erforderlichen JavaScript-Code. Wie weiter oben bereits erwähnt kann Emscripten sowohl die Initialisierungslogik als auch den JavaScript-Glue-Code generieren. In Listing 9 findet sich der erforderliche JavaScript-Code, den Entwickler*innen für eine Anbindung benötigen.

Listing 12: C Code + JS Code String return

```
//hello.c
const char* sayHello() {
    return "Hello world from Wasm";
};

//javascript
(async () => {
    const memory = new WebAssembly.Memory({initial: 256, maximum: 512});
    const readableBuffer = new Uint8Array(memory.buffer);

    const {instance} = await WebAssembly.instantiateStreaming(fetch('./dist/hello.wasm'), {
        env: {
            __memory_base: 0,
            memory
        }
    });

    console.log(readableBuffer); // [72, 101, 108, 108, 111,...]
    const pointer = instance.exports._sayHello(); //points to specific index in the memory-view, e.g. 0
    console.log(readString(readableBuffer, pointer)); //convert the bytes to a string using some helper function
})();
```

Richtlinien für den Import

Imports sind ebenfalls beim Kompilieren einer Anwendung aus einer anderen Sprache nach WebAssembly erlaubt. Die sprachliche Syntax für Imports ist dabei von der eingesetzten Programmiersprache abhängig. In C/C++ sind externe Funktionen über das `extern`-Keyword zu definieren. Emscripten stellt beim Kompilieren sicher, dass in der Wasm-Datei entsprechende Import-

Definitionen existieren. Zur Demonstration dient eine erweiterte Fassung der Primzahlenberechnung, die Listing 10 zeigt. Sie gibt die jeweils gefundenen Zahlen direkt im Log aus. Listing 11 zeigt die Anbindung in JavaScript.

Austausch komplexer Datentypen zwischen JavaScript und WebAssembly

Um komplexe Datentypen wie Strings zwischen WebAssembly und JavaScript zu teilen, ist der Umweg über das lineare Memory-Modell unabdingbar. Beim Initialisieren einer Wasm-Instanz lässt sich dazu ein `WebAssembly.Memory`-Objekt als Import-Parameter übergeben, das sich sowohl aus der JavaScript-Welt als auch aus Wasm heraus ändern lässt. Das Vorgehen ermöglicht den indirekten Austausch komplexer Daten. Mehrere WebAssembly-Module können sich zudem ein Memory-Objekt teilen, was unter anderem für Dynamic Linking wichtig ist.

Die Initialisierung des `WebAssembly.Memory`-Objektes erfolgt über einen Konstruktor. Er erfährt über den Parameter `Pages`, wie viel Speicher er initial reservieren soll. Eine Page ist 64 KByte groß. Über den optionalen Parameter `maximum` ist zudem definierbar, bis zu wie viele Pages das Wasm-Modul maximal anfordern darf. Das Memory-Objekt enthält einen ArrayBuffer – ein schreibgeschütztes Bytearray – für den Datenaustausch. Über eine darauf mit Typed-Arrays wie `Int8Array` oder `UInt8Array` definierte Ansicht lässt sich der Inhalt des linearen Memory-Modells in JavaScript auslesen und verändern.

Das ermöglicht den Austausch von Datentypen wie Strings zwischen JavaScript und WebAssembly, obwohl die beiden Welten eigentlich nur mit numerischen Werten kommunizieren können. Bei Zeichenketten findet der Austausch im Groben durch das Schreiben eines bestimmten String-Werts als Array von Char-Codes in das Memory-Objekt statt. Eine Funktion, die einen String zurückliefern sollte, gibt stattdessen einen Pointer zurück. Er definiert den Index im Memory-Array, in dem die Char-Codes des Strings beginnen. Zum Auslesen ist das Rückübersetzen der Char-Codes in einen String erforderlich. Listing 12 zeigt ein einfaches Beispiel.

Das Vorgehen ist recht aufwendig und ohne Hilfsfunktionen nicht zu empfehlen. Glücklicherweise schafft der JavaScript-Glue-Code von Emscripten Abhilfe, wie Listing 13 zeigt.

Listing 13: String-Binding mit Emscripten

```
<script src="./hello_glue.js"></script>
<script>
    Module.onRuntimeInitialized = () => {
        //wrap the function. Second parameter defines the return value
        const sayHello = cwrap('sayHello', 'string');
        console.log(sayHello());
    };
</script>
```

JavaScript versus WebAssembly

Das WebAssembly-Binärformat läuft in nahezu nativer Geschwindigkeit im Browser. In bestimmten Fällen ist Wasm deshalb um ein Vielfaches schneller als der gleiche Code in JavaScript. Die folgenden beiden Beispiele zeigen, in welchen Szenarien Wasm seine Stärken ausspielen kann. Obwohl es keine wissenschaftlich korrekten Versuche oder Messungen sind, lässt sich gut erkennen, für welche Anwendungsfälle Wasm besonders gut geeignet ist.

Der gezeigte Algorithmus zur Primzahlenberechnung lässt sich ebenso in JavaScript umsetzen. Ein Blick auf die Ausführungszeiten der in C und in JavaScript geschriebenen Algorithmen verrät: WebAssembly ist deutlich flotter als JavaScript. Besonders bei rechenintensiven Anwendungsfällen kann WebAssembly deutliche Vorteile bringen. Den gezeigten Algorithmus, mit dessen Hilfe sich Primzahlen berechnen lassen, können Programmierer*innen ebenso in JavaScript umsetzen. Wer direkt vergleicht, wie viel Zeit die Ausführung der in C und JavaScript geschriebenen Algorithmen in Anspruch nimmt, stellt fest: Die WebAssembly-Version ist ab einer bestimmten Suchgröße um bis zu 30 Prozent schneller als die JavaScript-Variante. Entsprechend kann WebAssembly vor allem bei Anwendungsfällen, in denen besonders viel Rechenleistung gefragt ist, seine Vorteile deutlich ausspielen.

Flinke Gesichtserkennung

Ein weiterer interessanter Anwendungsfall für einen direkten Vergleich ist die Umsetzung einer Webcam-basierten Gesichtserkennung im Browser – inspiriert von dem auf GitHub verfügbaren WebSight-Projekt. Dafür können Entwickler die openCV-Bibliothek, eine der bekanntesten für Bildverarbeitung und maschinelles Sehen, zu WebAssembly kompilieren. Die zu großen Teilen in C/C++ geschriebene Codebasis können sie mit Emscripten konvertieren. Eine Anleitung ist auf der OpenCV-Site als Emcripten-Tutorial verfügbar. Erneut empfiehlt sich der Weg über Docker.

Für die Umsetzung in JavaScript lässt sich die Bibliothek HAAR.js verwenden. Sie bietet sich als JavaScript-Portierung des openCV-HAAR-Detection-Algorithmus für einen groben Vergleich an. Im Schnitt ist die openCV-WebAssembly-Portierung deutlich performanter und erzielt eine höhere Bildwiederholfrequenz als der Algorithmus im reinen JavaScript-Code. Dafür ist die WebAssembly-Anwendung mit über 6 MByte Wasm-Code und 200 KByte JavaScript-Glue-Code nicht gerade schlank. Das liegt unter anderem daran, dass sie die gesamte konvertierte openCV-Bibliothek enthält. Die reine JavaScript-Version benötigt lediglich 15 KByte.

Gekommen, um zu bleiben

WebAssembly ist bereits ein fester Bestandteil der Webplattform und wird sich mit großer Wahrscheinlichkeit weiter etablieren. Mit Wasm öffnet sich das Web für zusätzliche Ökosysteme, performancekritische Anwendungsfälle und das Portieren bestehender Codebasen. Zahlreiche vielversprechende Vorschläge für eine Erweiterung des teilweise noch rudimentären Featuresets von WebAssembly sind auf GitHub aufgeführt, darunter Reference Types und Garbage Collection. Bis dahin helfen Werkzeuge wie Emscripten mit dem Generieren von JavaScript-Glue-Code dabei, Wasm alltagstauglich zu gestalten – leider zum Preis einer erhöhten Netzwerklast. (rme@ix.de)

Quellen

Alle gezeigten Beispiele und Anwendungen finden sich unter ix.de/zvyr.



Sven Kölpin

ist Softwareentwickler bei der CEWE Stiftung und Autor von Fachbeiträgen zum Thema Webanwendungen.



PHP: Programmiersprache fürs Web

Gekommen, um zu bleiben

Michael Hofer

Kaum eine Programmiersprache beeinflusste die Evolution des World Wide Web mehr als PHP. Trotz der Konkurrenz durch JavaScript und Node.js hält sie sich hartnäckig an der Spitze der Webentwicklung.

Momentan stehen die Zeichen bei der Webentwicklung zwar auf JavaScript, Node.js und davon abhängigen Frameworks, den Spitzenplatz können sie der Programmiersprache PHP jedoch immer noch nicht streitig machen. Befragt man die einschlägigen Auguren der Szene wie TIOBE, Stack Overflow, PYPL oder RedMonk, ergibt sich immer dasselbe Bild: PHP hat einen Dauerplatz unter den Top Ten.

Die Geschichte von PHP begann 1995. Der Programmierer Rasmus Lerdorf benötigte ein Tool, das Formularinhalte einer Webseite übernehmen und bearbeiten kann. Er nannte sein Werk Personal Homepage Tools/Form Interpreter (PHP/FI). Dieser erste Entwurf wies nur rudimentäre Ansätze einer Programmiersprache auf, wichtige Elemente wie Schleifen fehlten. Lerdorf arbeitete weiter und brachte zwei Jahre später die Version PHP/FI 2 heraus.

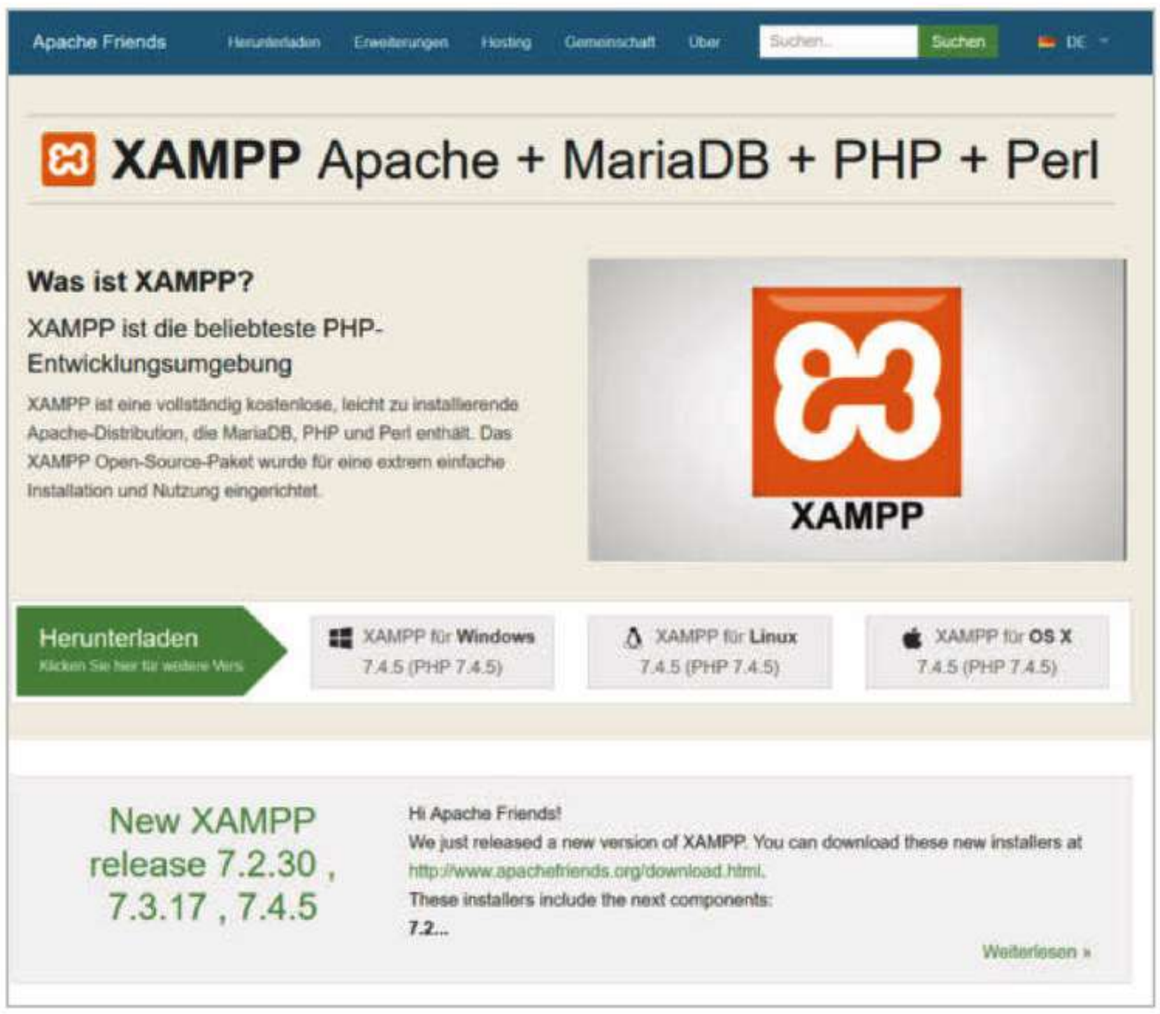
Bei der Umsetzung eines Universitätsprojekts stießen die beiden Studenten Andi Gutmans und Zeev Suraski auf das Tool, erkannten dessen Potenzial und beschlossen, die Skriptsprache zusammen mit Lerdorf völlig neu zu schreiben. Das Ergebnis war im Juni 1998 die Version PHP 3. Zeitgleich erhielt die Sprache einen neuen Namen: PHP Hypertext Processor. Damit wollten die Entwickler unterstreichen, dass die neue Sprache nicht nur den kleinen Homepage-Bastler ansprechen sollte, sondern durchaus ein professionelles Publikum. Der Erfolg gab ihnen recht. Mit PHP 3 stieg die Zahl der Installationen auf über eine Million.

Gutmans und Suraski gründeten die Firma Zend Technologies, die maßgeblich an der Weiterentwicklung von PHP beteiligt war. Deutlich erkennen ließ sich das erstmals bei PHP 4. Den Kern der Version bildete die von Zend entwickelte Zend Engine, eine Art Compiler, der den Skriptcode zur Laufzeit in Bytecode umwandelt und ausführt.

Dieses Prinzip trug wesentlich zu einer gesteigerten Performance der Sprache bei. Mit zahlreichen Verbesserungen und Funktionserweiterungen erschien PHP 4 im Mai 2002 und verdrängte sukzessive den bislang dominierenden Sprachstandard

iX-TRACT

- PHP folgt der klassischen Client-Server-Architektur, dabei bestehen verschiedene Varianten der Kombination Webserver, PHP und Datenbank für alle gängigen Betriebssysteme.
- Die Sprache verfügt über eine Reihe von Datenbankschnittstellen, darunter PDO, und weitere Abstraktionsebenen, etwa DBA oder ODBC.
- Datentypen einer Variablen in PHP brauchen keine explizite Definition. Die Sprache nimmt Typumwandlungen automatisch vor.



XAMPP stellt eine umfangreiche Entwicklungsplattform für PHP- Entwickler*innen zur Verfügung (Abb. 1).

Listing 1: Hello World <html>

```
<html>
<head>
<title>PHP-Test</title>
</head>
<body>
<?php echo '<p>Hallo Welt</p>'; ?>
</body>
</html>
```

Listing 2: Typenumwandlung <?php

```
<?php
$foo = True; // Boolean
$a = 1234; // Float
$a = 1.234; //
$a = 'dies ist ein einfacher String';

// Beispiel für ein Array
$array = array(
    "foo" => "bar",
    "bar" => "foo",
    100   => -100,
    -100  => 100,
);
?>
```

Listing 3: while

```
<?php
$aktuelleZahl = 'Aktuelle Zahl: ';
$i = 1;
while ($i <= 50) {
    echo $aktuelleZahl
    echo $i;
    echo "<br>";
    $i++;
}
?>
```

Perl von der Bühne. Die starke Präsenz von PHP im Web ist vor allem auf den einfachen Einstieg und den schnellen Lernerfolg zurückzuführen. Als die Sprache immer stärker im Enterprise-Umfeld zum Einsatz kam, wuchs die Nachfrage nach neuen Möglichkeiten des objektorientierten Programmierens. Für komplexe Projekte sollte die Version PHP 5 umfangreiche neue Optionen der Objektorientierung bieten. PHP Data Objects (PDO), eine neue Methode für den Zugriff auf Datenbanken, und XML-Erweiterungen zählen zu den Highlights von PHP 5.

Wer auf die Version 6 als Nachfolger von PHP 5 gewartet hatte, sah sich enttäuscht. Wegen Problemen bei der Implementierung (zum Beispiel der Unterstützung verschiedener Unicode-Standards) stellte man die Entwicklung 2010 ein. Im Jahr 2014 entschied die Entwicklercommunity, dass PHP 7 der neue Nachfolger von PHP 5 werden soll. Ende 2015 folgte die erste 7er-Version und die derzeit aktuelle Release 7.4 im November 2019.

PHP im Einsatz

Das Marktforschungsunternehmen W3Techs hat ermittelt, dass 78,4 Prozent (Mai 2020) aller Websites PHP als serverseitige Programmiersprache verwenden. Damit lässt PHP alle Mitbewerber weit hinter sich. Auch der gegenwärtige Trend zu JavaScript, das mithilfe von Node.js eine serverseitige Programmierung zulässt, berührt diese Spitzenposition nicht. Viel zum Erfolg hat sicher die Open-Source-Lizenz der Sprache beigetragen, die es Hostern erlaubt, PHP auf ihren Webservern und Hostingpaketen zu implementieren.

Die Funktionsweise von PHP basiert auf der klassischen Client-Server-Architektur. Der PHP-Interpreter läuft in einer Serverumgebung wie Apache oder IIS, gegebenenfalls ergänzt um eine Datenbankkomponente wie MySQL, MariaDB etc. Der Client (Browser) sendet Befehle, Anfragen oder Formularergebnisse zur Verarbeitung an den Server. Die Ergebnisse wandern wieder zurück an den Client. Die Konstellation Webserver, PHP und Datenbank gibt es in verschiedenen Varianten und für alle gängigen Betriebssysteme. LAMP steht beispielsweise für die Linux-Variante, WAMP für das Windows-Pendant und MAMP für Systeme auf Basis eines Mac. Vorkonfigurierte Pakete, etwa XAMPP von www.apachefriends.org, erleichtern das Installieren eines Systems zusätzlich (Abbildung 1).

Programmieren mit PHP

PHP-Programmierer*innen steht eine riesige Auswahl an Befehlen und Funktionen zur Verfügung, die sie beim Bau einer Webapplikation verwenden können. Das bekannte „Hello World“ sieht in PHP aus wie in Listing 1.

Das Beispiel zeigt, wie einfach sich HTML- und PHP-Code mischen lassen. Die beiden Klassiker des Webs verstehen sich problemlos. In PHP existieren Datentypen, die man auch von anderen Programmiersprachen kennt. Dazu gehören zum Beispiel die skalaren Datentypen Boolean (Wahrheitswerte `true`, `false`), Integer (Ganzzahl), Float (Gleitkommazahl) und String (Zeichenketten). Darüber hinaus steht für Listen von Werten der Datentyp Array zur Verfügung. Anders als bei anderen Sprachen müssen PHP-Entwickler*innen Datentypen einer Variablen nicht explizit definieren. PHP erkennt den Typ von sich aus bei der ersten Zuweisung von Werten. Allfällige Typumwandlungen nimmt PHP automatisch vor (Listing 2).

Für Vorgänge, die sich wiederholen sollen, bis ein bestimmtes Ergebnis eintritt, bietet PHP die Schleifenoperationen `while`, `do while` und `for`. Den Klassiker `while` zeigt Listing 3. Die Schleife zählt bis 50 hoch und gibt den String `Aktuelle Zahl:` und den jeweiligen Zähler `$i` aus. Auch Kontrollstrukturen stehen PHP-Entwickler*innen zur Verfügung. Neben `if/then/else` gibt es beispielsweise als Alternative `switch` (Listing 4).

Breit ist auch das Angebot an Funktionen. Manche davon setzen allerdings voraus, dass bestimmte PHP-Extensions bei der Installation einkompiliert wurden. Zum Beispiel gilt das für bestimmte Grafik- und Datenbankfunktionen. Die meisten String-, Array- und Variablenfunktionen sind bereits in jeder PHP-Version enthalten. Ein Aufruf von `phpinfo()` zeigt eine aktuelle Liste der vorhandenen Extensions (Abbildung 2).

Listing 4: switch

```
<?php
if ($i == 0) {
    echo "i ist gleich 0";
} elseif ($i == 1) {
    echo "i ist gleich 1";
} elseif ($i == 2) {
    echo "i ist gleich 2";
}

switch ($i) {
    case 0:
        echo "i ist gleich 0";
        break;
    case 1:
        echo "i ist gleich 1";
        break;
    case 2:
        echo "i ist gleich 2";
        break;
}
?>
```

Als Beispiel für eine leistungsfähige PHP-Funktion sei `SimpleXML` angeführt, mit der sich XML-Daten manipulieren lassen (Listing 5). Entwickler*innen müssen dem Funktionsaufruf die Webadresse eines RSS-Feeds und die Anzahl der anzuzeigenden Items übergeben. Das Ergebnis ist eine Liste mit den verlinkten Headlines dieses Feeds. In der Funktion taucht auch noch der Befehl `foreach` auf. Damit kann man elegant über Arrays iterieren.

Datenbankoperationen

Keine halbwegs professionelle Webapplikation kommt ohne Datenbank aus. PHP-Entwickler*innen stellt dies in vielen Fällen vor Herausforderungen. Die Dimensionen einer Shopapplikation mit ihrer unübersehbaren Anzahl an Datenbanken für Artikel, Lieferanten, Kunden, Bestellungen, Lagerbestände, offene Posten etc. wirft manche komplizierte Problemstellung auf. Wobei das riesige Angebot an Datenbanken und deren Auswahl noch erschwerend hinzukommt.

PHP unterstützt Programmierer*innen hier mit einer Palette an Datenbankschnittstellen, von denen PDO die flexibelste ist. Daneben gibt es weitere Abstraktionsebenen, beispielsweise DBA (für den Zugriff auf Datenbanken vom Typ Berkeley DB) oder ODBC (Open Database Connectivity). Zusätzlich weist PHP eine große Zahl anbieterspezifischer Datenbankerweiterungen auf. Darunter befinden sich alle großen Namen der DB-Szene wie Firebird, IBM Db2, Informix, MongoDB, MSSQL, Oracle OCI8, PostgreSQL und SQLite3. Für PHP-Entwickler*innen im Enterprise-Bereich sind die Datenbankschnittstellen unerlässliche Voraussetzungen für professionelle Applikationen. Das Codefragment in Listing 6 zeigt am Beispiel der SQLite3-Schnittstelle, wie einfach sich damit Datenbankoperationen umsetzen lassen.

Der einfache Zugang zur Sprache ist eine Quelle des Erfolgs von PHP, der Riesenfundus an fertigen Applikationen, Bibliotheken und Sammlungen von Codeausschnitten eine andere. Ob bei GitHub, SourceForge oder Stack Overflow, in jedem Fall wirft eine Suchanfrage mit dem Stichwort PHP eine sechsstellige Ergebnisliste aus.

Erfolgreiche PHP-Applikationen

Der Renner unter den PHP-Applikationen ist sicherlich das bekannte Open-Source-CMS WordPress. Laut Statista hatte diese Applikation im CMS-Segment im April 2020 einen Marktanteil von 63 Prozent. Die Konkurrenten Joomla und Drupal lagen bei unter fünf Prozent. WordPress selbst wirbt auf seiner Website damit, dass derzeit 36 Prozent aller Websites weltweit WordPress nutzen – eine beeindruckende Statistik im stark umkämpften Markt der CMS. Die Anzahl der Blogposts von WordPress-Usern belief sich in diesem Zeitraum auf rund 75 Millionen und die Anzahl der Page Views von WordPress-Artikeln auf 28 Milliarden. WordPress selbst schätzt, dass 36 Prozent aller Websites das CMS einsetzen (siehe ix.de/zpge).

Die Liste der populären PHP-Anwendungen ist damit nicht zu Ende. Im E-Commerce-Sektor kann die Sprache Namen wie Magento (Adobe), PrestaShop, Shopware und OXID eShop vorweisen. Im Bereich Forensoftware sind Invision Power Board, phpBB und vBulletin bekannte Vertreter mit PHP-Background. Auch im CRM-Bereich hat PHP mit SugarCRM einen prominenten Vertreter. Und nicht zu vergessen: Die berühmte Onlineenzyklopädie Wikipedia basiert auf PHP.

In diesem Zusammenhang sollte man Facebook erwähnen. Auch das weltgrößte soziale Netzwerk arbeitet mit PHP. Allerdings hat Facebook dabei einen Sonderweg beschritten, mit dem es sich vom Community-PHP weitgehend abgrenzt. Beispielsweise hat das Unternehmen eine mit PHP interoperable Entwicklungssprache namens Hack und eine quelloffene Laufzeitumgebung namens HHVM, kurz für HipHop Virtual Machine, entworfen (siehe ix.de/zpge).

Tools für PHP-Developer

Ein Programmiersprachenökosystem ist immer nur so gut wie das Instrumentarium, das in Form von Frameworks, IDEs und

PHP Version 7.0.33-0+deb9u7

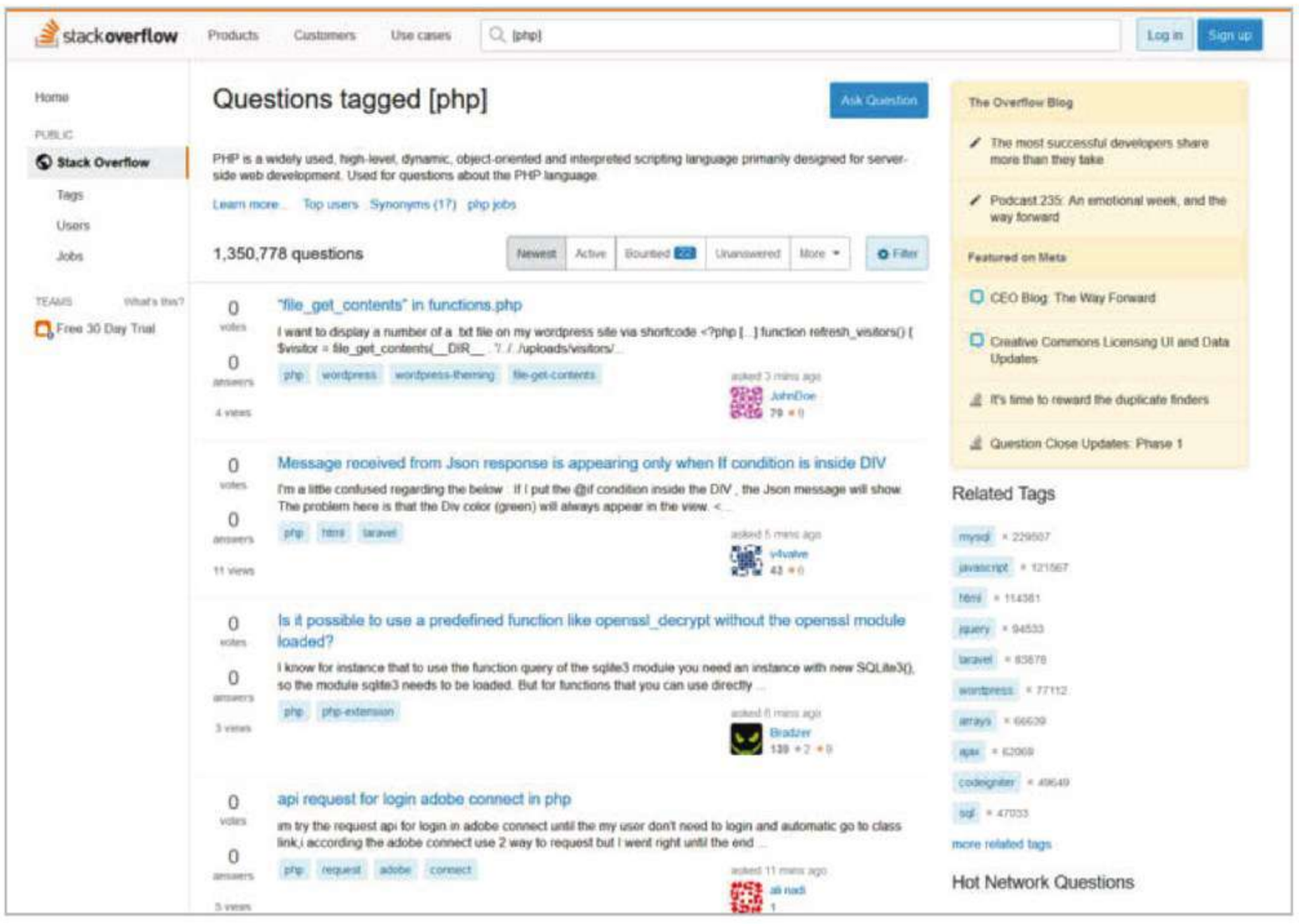
System	Linux pi-camera 4.19.66+ #1253 Thu Aug 15 11:37:30 BST 2019 armv6l
Build Date	Feb 16 2020 15:11:40
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php/7.0/apache2
Loaded Configuration File	/etc/php/7.0/apache2/php.ini
Scan this dir for additional .ini files	/etc/php/7.0/apache2/conf.d
Additional .ini files parsed	/etc/php/7.0/apache2/conf.d/10-opcache.ini, /etc/php/7.0/apache2/conf.d/10-pdo.ini, /etc/php/7.0/apache2/conf.d/20-calendar.ini, /etc/php/7.0/apache2/conf.d/20-ctype.ini, /etc/php/7.0/apache2/conf.d/20-exif.ini, /etc/php/7.0/apache2/conf.d/20-fileinfo.ini, /etc/php/7.0/apache2/conf.d/20-ftp.ini, /etc/php/7.0/apache2/conf.d/20-gettext.ini, /etc/php/7.0/apache2/conf.d/20-iconv.ini, /etc/php/7.0/apache2/conf.d/20-json.ini, /etc/php/7.0/apache2/conf.d/20-pdo_sqlite.ini, /etc/php/7.0/apache2/conf.d/20-phar.ini, /etc/php/7.0/apache2/conf.d/20-posix.ini, /etc/php/7.0/apache2/conf.d/20-readline.ini, /etc/php/7.0/apache2/conf.d/20-shmop.ini, /etc/php/7.0/apache2/conf.d/20-sockets.ini, /etc/php/7.0/apache2/conf.d/20-sqlite3.ini, /etc/php/7.0/apache2/conf.d/20-sysvmsg.ini, /etc/php/7.0/apache2/conf.d/20-sysvsem.ini, /etc/php/7.0/apache2/conf.d/20-sysvshm.ini, /etc/php/7.0/apache2/conf.d/20-tokenizer.ini
PHP API	20151012
PHP Extension	20151012
Zend Extension	320151012
Zend Extension Build	API320151012,NTS
PHP Extension Build	API20151012,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	disabled
IPv6 Support	enabled
DTrace Support	available, disabled
Registered PHP Streams	https, ftps, compress.zlib, php, file, glob, data, http, ftp, phar
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv2, tls, tlsv1.0, tlsv1.1, tlsv1.2
Registered Stream Filters	zlib.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk, convert.iconv.*

This program makes use of the Zend Scripting Language Engine:
Zend Engine v3.0.0, Copyright (c) 1998-2017 Zend Technologies
with Zend OPcache v7.0.33-0+deb9u7, Copyright (c) 1999-2017, by Zend Technologies

phpinfo liefert umfangreiche Informationen zu einer vorhandenen PHP-Installation (Abb. 2).

Codebibliotheken zur Verfügung steht. Die Großen der Branche, Google, Apple und Microsoft, investieren viel Geld und Ressourcen, um ihren Way of Development Entwickler*innen näherzubringen. Für ein communitygetriebenes Modell wie PHP ist es schwer mitzuhalten. Trotzdem können auch Programmierer*innen auf ein Set an leistungsfähigen Frameworks zurückgreifen. Diese helfen beim strukturierten Programmieren, vermeiden redundanten Code und erhöhen die Sicherheit der Applikationen. Großer Beliebtheit erfreut sich Laravel, das vor Kurzem in Version 7 erschienen ist. Weitere Vertreter sind Symfony, CodeIgniter, Yii, CakePHP und schließlich Zend Framework, das jetzt Laminas Project heißt.

PHP-Programme lassen sich mit jedem einfachen Editor wie Notepad++ erstellen. Zur Effizienzsteigerung gibt es jedoch integrierte Entwicklungsumgebungen (IDE). Zu nennen sind hier vor allem Zend Studio, NetBeans PHP, die Eclipse PHP Development Tools und PHPStorm. Während es sich bei NetBeans und Eclipse um kostenlose Open-Source-Projekte handelt, sind Zend Studio und PHPStorm kommerzielle Produkte.



Stack Overflow: Wer Lösungen für ein Dev-Problem sucht, dem wird hier geholfen (Abb. 3).

Support für PHP

Klassischer Support, wie er im kommerziellen Bereich existiert, ist im PHP-Umfeld die Ausnahme. Die agile PHP-Community springt an dieser Stelle ein. Das klassische PHP-Manual beispielsweise bietet nicht nur eine umfangreiche Sprachreferenz, sondern liefert mittels Kommentarfunktion auch Erfahrungsberichte, Beispielcode und Tipps zu jedem aufgeführten Item. Stack Overflow dient ebenfalls als Anlaufstelle für Hilfesuchende. Wie bei Wikipedia findet sich alles, was in diesem Forum gefragt, geschrieben und kommentiert wird, unter der Creative-Commons-Lizenz und darf daher frei verwendet werden. Momentan weist die Plattform über 1,3 Millionen Einträge mit dem Tag php auf (Abbildung 3). Weitere informative Anlaufstellen sind PHPBuilder, Tutorialzine oder das PHP-Tutorial von w3schools.com.

Listing 5: SimpleXML

```
<?php
function rssfeed ($webadresse, $anzahl) {
    $xmlfile = $webadresse;
    $xml = simplexml_load_file($xmlfile);
    if ($xml) {
      echo '<li>';
      $i=0;
      foreach ($xml->channel->item as $a) {
        $i=$i+1;
        if ($i<$anzahl) {
          $hyperlink= '<a href="'.$a->link.'">'. $a->title.'</a>';
         echo $hyperlink;
         }
      }
      echo "</li>";
      }
      else {
      echo "Kann Verbindung nicht herstellen";
      }
}
?>
```

Listing 6: SQLite3-Schnittstelle

```
<?php
$dbname="usersb.db";
$db = new SQLite3($dbname);
$query = "CREATE TABLE IF NOT EXISTS user_table(user_id integer ,
                        primary key, user_name text, user_email text)";
$db->exec($query);
?>
```

Ausblick auf PHP 8

Innerhalb der PHP-Gemeinde spekuliert man schon viel über die nächste Major-Version der Sprache. Genaues lässt sich bislang aber weder zum Releasedatum noch zu den signifikantesten Änderungen sagen. Nicht einmal eine Pleite wie bei PHP 6 lässt sich ausschließen. Ein paar Dinge sind jedoch schon durchgesickert. In PHP 8 soll ein Just-in-Time-Compiler (JIT) implementiert sein. Das dürfte die Performance von PHP-Programmen und -Websites signifikant steigern und damit die User Experience nachhaltig beeinflussen. Weiterhin sind Verbesserungen in Sachen OOP geplant sowie die Einführung von Union Types. (nb@ix.de)

Quellen

Alle Listings und Links finden sich unter https://ix.de/zpge



Michael Hofer

ist Autor und Entwickler mit dem Spezialgebiet HTML, XML und CSS. Außerdem programmiert er in PHP.



Der JavaScript-Compiler Svelte

Kompiliert statt gepackt

Daniel Frank

Svelte ist ein Framework zum Erstellen von JavaScript-Komponenten für Web-Frontends. Allerdings geht Svelte einen anderen Weg als die etablierten Frameworks: Es kompiliert den Svelte-Code zu einfachem JavaScript. Das sorgt für schlanke und schnelle Web-Apps.

Svelte ist ein JavaScript-Framework zum Erstellen von Interface-Komponenten für den Browser. Dabei verfolgt Svelte einen Ansatz, der sich grundlegend von etablierten Frameworks unterscheidet: Mit Svelte erstellte Komponenten enthalten JavaScript, HTML und CSS in einer Datei und werden zu Code kompiliert, den der Browser direkt ausführen kann. Dadurch entstehen schlanke Komponenten mit wenig Overhead. Mit Frameworks wie Angular, Vue und React erstellte Anwendungen müssen hingegen immer den Framework-Core einbinden, der Grundfunktionen beispielsweise für das Rendern von Komponenten enthält. Durch den Übersetzungsschritt benötigen Svelte-Apps keinen solchen Core.

Der Kopf hinter Svelte ist Rich Harris, der auch für das JavaScript-Framework Ractive und den Modul-Bundler Rollup verantwortlich zeichnet. Der Grafikeditor bei der New York Times nutzt JavaScript, um Daten im Web zu visualisieren. 2016 begann er mit der Arbeit an Svelte, weil er aus verschiedenen Gründen mit den verfügbaren Frontend-Frameworks unzufrieden war: Die Performance der Webseiten war ihm zu schlecht, der Anteil an JavaScript zu hoch, der Komfort für Entwickler zu gering (siehe ix.de/z88f). Seit 2019 steht Svelte in Version 3 zur Verfügung.

iX-TRACT

- Svelte ist ein junges JavaScript-Framework, das in JavaScript, HTML und CSS erstellte Komponenten in Code übersetzt, den der Browser direkt ausführen kann.
- Das Ergebnis sind schlanke, schnelle Webanwendungen, die ohne Virtual DOM auskommen.
- Die Svelte-Syntax ist einfach zu erlernen und das Framework nimmt Entwicklern viel Arbeit ab, sodass sie nur wenig Code schreiben müssen.

Es geht auch ohne Virtual DOM

Das Virtual DOM (VDOM) ist eine gängige Technik der Webentwicklung mit JavaScript, bei der die Struktur einer Webseite zunächst in JavaScript-Objekten abgebildet wird. Sobald die Darstellung der Daten erforderlich ist, setzt das Framework die Struk-

tur aus dem VDOM in das reale DOM des Browsers ein. Der Browser rendert nun die Daten. Verändern sich die Daten, passt das Framework zuerst das VDOM an und tauscht dann die im DOM gerenderten, veralteten Daten gegen die aktuellen im VDOM aus.

Das verbessert die Performance, da der Browser seltener neu rendern muss – würde man die neuen Daten direkt ins DOM schreiben, wäre für jede kleine Veränderung ein Update der sichtbaren Seite notwendig. Das bedeutet mehr Arbeit für den Browser, was sich insbesondere bei Animationen oder Webseiten, die große Datenmengen darstellen, in einer schlechteren Performance äußert.

Die wichtigste Frage bei der Verwendung von VDOM lautet: Was hat sich verändert und welche Teile der Seite müssen erneut gerendert werden? Um dies zu beantworten, erfolgt laufend ein Abgleich zwischen den gerenderten und den Daten im VDOM. Leider kostet dieser Abgleich ebenfalls Rechenzeit und führt zu Einbußen in der Performance. Entwickler müssen den Abgleich zwischen VDOM und DOM selbst optimieren, falls die Performance für ihren Anwendungsfall nicht ausreicht.

Die meiste Logik für das VDOM steckt im Kern der Web-Frameworks. Ein Server sendet diesen, zusammen mit dem eigentlichen JavaScript-Code einer Seite, an den Browser. Die Gesamtgröße der Webseiten erhöht sich und führt zu mehr Last für den Browser, der das ganze JavaScript verarbeitet.

Svelte nutzt kein VDOM. Stattdessen wird der geschriebene Programmcode (wie auch bei Ionic und Angular) kompiliert und zu einem Bundle zusammengefasst. Durch den Kompilierungsschritt ist kein Framework-Core notwendig.

Ein Gruß an die Welt

Zur Verdeutlichung dient das klassische „Hello World"-Beispiel. Node.js und der Node Package Manager NPM müssen zur Installation auf dem verwendeten Computer eingerichtet sein. In einer Kommandozeile sind folgende Befehle auszuführen:

```
npx degit sveltejs/template svelte-app
cd svelte-app
npm install
```

Der erste Befehl kopiert die aktuelle Version des Svelte-Templates von GitHub in den Ordner svelte-app, der letzte Befehl installiert die Abhängigkeiten des Node-Projektes. In der Datei package.json findet sich in den devdependencies neben Svelte der JavaScript-Bundler Rollup, der nötig ist, um Svelte-Komponenten in Code zu übersetzen. Dazu nutzt Rollup verschiedene Plug-ins, darunter rollup-plugin-svelte, das hilft, die Svelte-Komponenten zu verstehen und zu verarbeiten.

Mit dem Befehl `npm run dev` kompiliert Rollup den Code und überwacht die Quelldateien auf Änderungen. Ein Testserver stellt die Seite auf localhost:5000 bereit. Änderungen am Quellcode lösen eine erneute Kompilierung aus.

Listing 1: App.svelte

```
<script>
  export let name;
</script>

<main>
  <h1>Hello {name}!</h1>
  <p>...</p>
</main>

<style>
  ...
</style>
```

Listing 2: main.js

```
import App from './App.svelte';

const app = new App({
  target: document.body,
  props: {
    name: 'world'
  }
});

export default app;
```

Listing 3: App.svelte mit Zähler

```
<script>
  export let name;
  let count = 0;
  setInterval(() => count += 1, 1000);
</script>

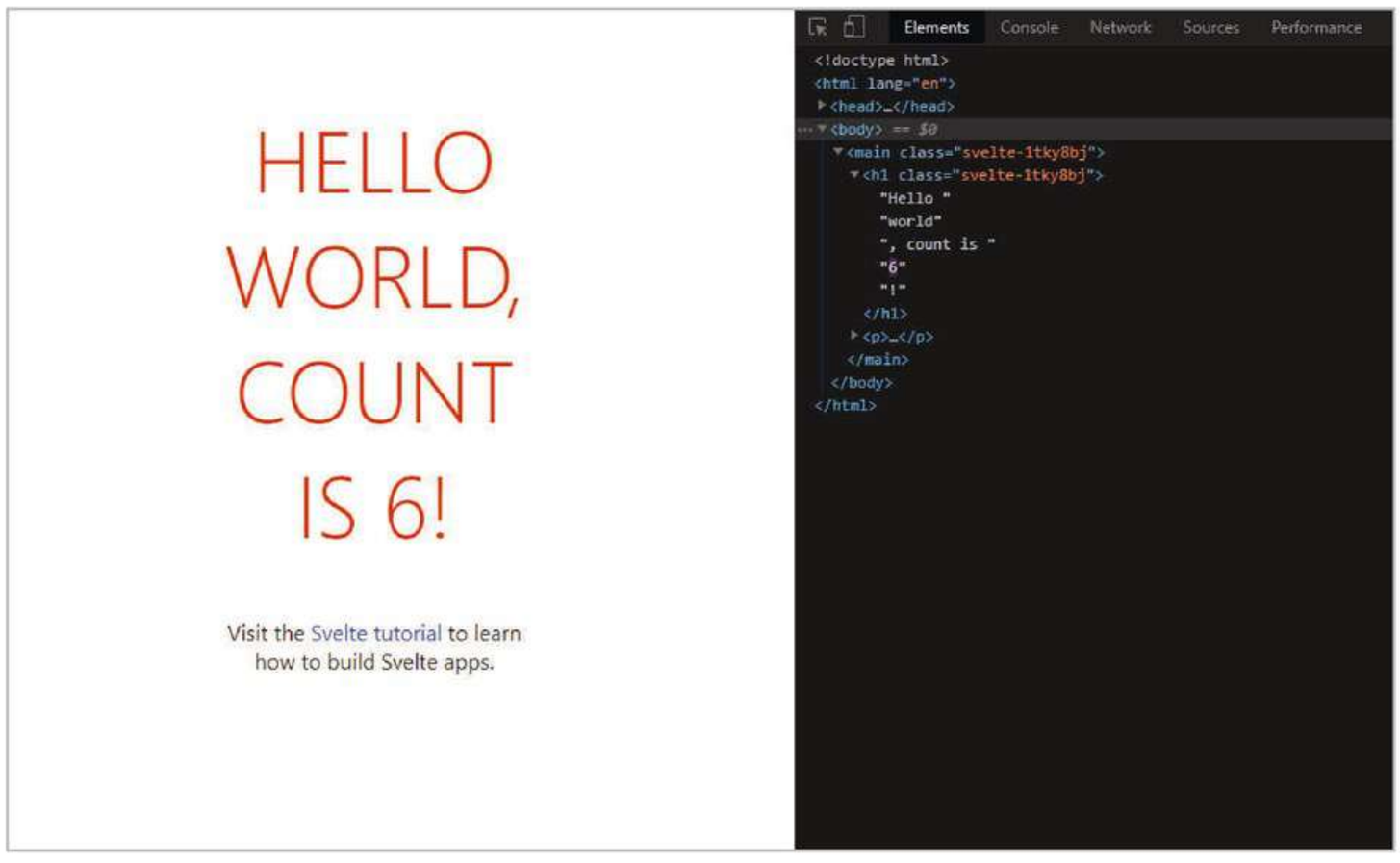
<main>
  <h1>Hello {name}, count is {count}!</h1>
  <p>...</p>
</main>

<style>
  ...
</style>
```

Im Ordner svelte-app sind hauptsächlich die Unterordner src mit dem Quelltext und public mit dem kompilierten Code und der HTML-Startseite interessant. Die Datei App.svelte kombiniert JavaScript, CSS und HTML in einer Datei (Listing 1). Das Styling gilt dabei nur für die jeweilige Komponente, eine Vermischung mit Styles aus anderen Komponenten ist ausgeschlossen. Das HTML definiert die Ausgabe einer Überschrift und eines Absatzes. Die Überschrift umfasst den Text „Hello" und den Inhalt der Variablen `name`.

Die geschweiften Klammern im HTML-Template können Variablen, aber auch Anweisungen enthalten. Variablen in Svelte sind reaktiv: Erfolgt eine Zuweisung an eine Variable, erfährt die Komponente ein Update und wird gerendert.

Listing 2 zeigt die Datei main.js. Der Aufruf des Konstruktors erzeugt eine Instanz der App-Komponente und initialisiert



Svelte fügt dem HTML keine eigenen Elemente hinzu, ergänzt jedoch Klassen, um das CSS zu kapseln (Abb. 1).

Listing 4: Bar.svelte

```
<script>
  export let name = '';
  export let value = 0;
</script>

<div class="bar">
  <p>{name}</p>
  <div>
    <div style="width:{value * 100 / 375}%;"></div>
  </div>
  <p>{value}</p>
</div>
```

Listing 5: main.js für Bar.svelte

```
const bar = new Bar({
  target: document.body,
  props: {
    name: 'Max',
    value: 20
  }
});
```

die Variable `name` mit dem Text „world“. Die neue Instanz der App-Komponente hängt am Body der Seite.

Gebündelt und gerendert

Wie sieht nun das Bundle zu dieser Komponente aus? Die Datei /build/bundle.js enthält das kompilierte JavaScript. Der Code umfasst drei Klassen (`SvelteComponent`, `SvelteComponentDev`, `App`) und viele Einzelfunktionen. Von den rund 500 Zeilen Code dient etwa die Hälfte dem Debugging und ist im Production-Build nicht mehr enthalten – der Production-Build des Beispiels ist etwa drei Kilobyte groß. Die Klasse `App` besteht im Wesentlichen aus einem Konstruktor. Die Klasse wird benötigt, um den Scope der Instanzen festzulegen.

Einzelfunktionen außerhalb der Klassen initialisieren Komponenten (`init`, `create_fragment`), weisen Variablen Werte zu (`update$$`), verändern das DOM (`mount_component`, `set_data_dev`) und regeln das Scheduling von Updates (`make_dirty`, `schedule_update`, `flush`). Verwendet man Einzelfunktionen, vermeidet man die Dopplung von Funktionalität innerhalb der Klassen.

Das erste Rendern läuft folgendermaßen ab:

- Der Konstruktor einer Komponente ruft den Konstruktor der Basisklassen und danach die Funktion init auf.
- Die `init`-Funktion erstellt das Grundgerüst der neuen Instanz und verlinkt es mit den notwendigen Einzelfunktionen.
- Der Instanz werden Funktionen für das Rendering der Instanz zugewiesen (`create_fragment`).
- Die DOM-Knoten der Instanz werden erstellt und mit dem DOM verknüpft (`mount_component`).
- `init` ruft als Letztes die Funktion `flush` auf. Sie aktualisiert bei Bedarf weitere Komponenten.

Den Ablauf für das Update einer Komponente versteht man, wenn App.svelte um einen Zähler erweitert wird. Das hinzugefügte Intervall inkrementiert die Variable `count` einmal pro Sekunde (Listing 3).

Svelte ersetzt im Bundle die Callback-Funktion durch diese Codezeile:

```
setInterval(() => $$invalidate(1, count += 1), 1000);
```

Beim Update von `count` ergibt sich folgender Ablauf:

- Der Browser ruft den Callback von setInterval jede Sekunde auf.
- `$$invalidate` markiert die Variable mithilfe der Funktion `make_dirty` als verändert und sorgt für den Aufruf von `flush` am Ende des aktuellen Event-Loop-Zyklus.
- `flush` aktualisiert den Zähler im DOM mit der Funktion `set_data_dev`.
- Der Event-Loop-Zyklus ist zu Ende und der Browser rendert die Veränderungen des DOM.

Bundles sind schlank und performant

Der Vorteil dieser Vorgehensweise liegt darin, dass der Code im Bundle Variablen und Komponenten ganz gezielt aktualisiert. Dies ist möglich, da der Compiler alle Komponenten untersucht, Zuweisungen an Variablen erkennt und sie durch Funktionsaufrufe ersetzt. Ein VDOM ist daher nicht notwendig, was die Performance von Anwendungen erhöht. Der Compiler weiß zudem, welche Features von Svelte eine Anwendung nutzt, und übernimmt nur diese Funktion in das Bundle, sodass die Bundles sehr schlank bleiben. Erst bei einer sehr hohen Zahl von Komponenten erreicht ein Bundle die Größe des Core anderer JavaScript-Frameworks.

Der Compiler macht außerdem neue Anweisungen in Svelte-Komponenten verfügbar. Er wandelt beispielsweise das Schlüsselwort `export` im Script-Bereich der Komponente so in Code um, dass man von außerhalb der Komponente auf die Variable zugreifen kann. Die Komponente wird aktualisiert, sobald eine

Listing 6: Styling für den Balken

```
<style>
  :root {
    --bar-height: 10px;
  }

  .bar {
    align-items: center;
    display: flex;
    justify-content: space-between;
  }

  .bar p {
    text-align: right;
  }

  .bar p:first-child {
    margin-right: 1em;
    min-width: 4em;
    width: 10%;
  }

  .bar p:last-child {
    margin-left: 1em;
    min-width: 2.5em;
  }

  .bar div {
    height: var(--bar-height);
  }

  .bar > div {
    background: lightgrey;
    width: 100%;
  }

  .bar > div > div {
    background: rgba(62, 144, 84, 1);
    display: block;
  }
</style>
```

Zuweisung erfolgt – das ist keine Standardfunktionalität von JavaScript (auch wenn JavaScript das Schlüsselwort `export` im Kontext von JavaScript-Modulen kennt). Svelte besitzt noch mehr solche speziellen Anweisungen und stellt somit Entwicklern Funktionen zur Verfügung, die sie sonst selbst schreiben müssten. Ein weiterer Vorteil des Compilers: Es lassen sich Webkomponenten generieren. Man erweitert hierfür den Svelte-Block der Datei rollup.config.js um das Flag `customElement: true`. Jede Komponente braucht außerdem eine Kennzeichnung, um den Namen der Webkomponente festzulegen (`<svelte:options tag="my-component" />`). Schließlich verwendet man die generierten Elemente anstatt der Svelte-Komponente (`<my-component/>` anstatt `<MyComponent/>`). Es steht also mehr als ein Zielformat für die Kompilierung zur Verfügung.

Nun noch ein Blick auf die DOM-Knoten der App-Komponente. In Abbildung 1 sieht man, dass dem Markup der Elemente `<main>` und `<h1>` jeweils eine CSS-Klasse hinzugefügt wurde. Diese ermöglichen die Kapselung des CSS innerhalb der Komponente, ohne andere DOM-Knoten einer Anwendung zu beeinflussen. Der Ansatz von Svelte ist viel simpler als ein Shadow-DOM (Browser-API zur Kapselung von Elementen) und steckt ebenfalls im Compiler.

Server Side Rendering – Seiten werden vom Server gerendert und als HTML an das Frontend geschickt – ist eine weitere Besonderheit von Svelte. Der Compiler verfügt nämlich über zwei Renderer: einen für den Browser und einen zweiten für das Rendern auf der Serverseite. Zur Aktivierung benötigt die Konfiguration des Compilers ein anderes Flag, nämlich `generate: "ssr"` (Default ist `generate:"dom"`). Die generierten Komponenten füttert man serverseitig mit Daten und erhält als Output HTML und CSS für den Browser.



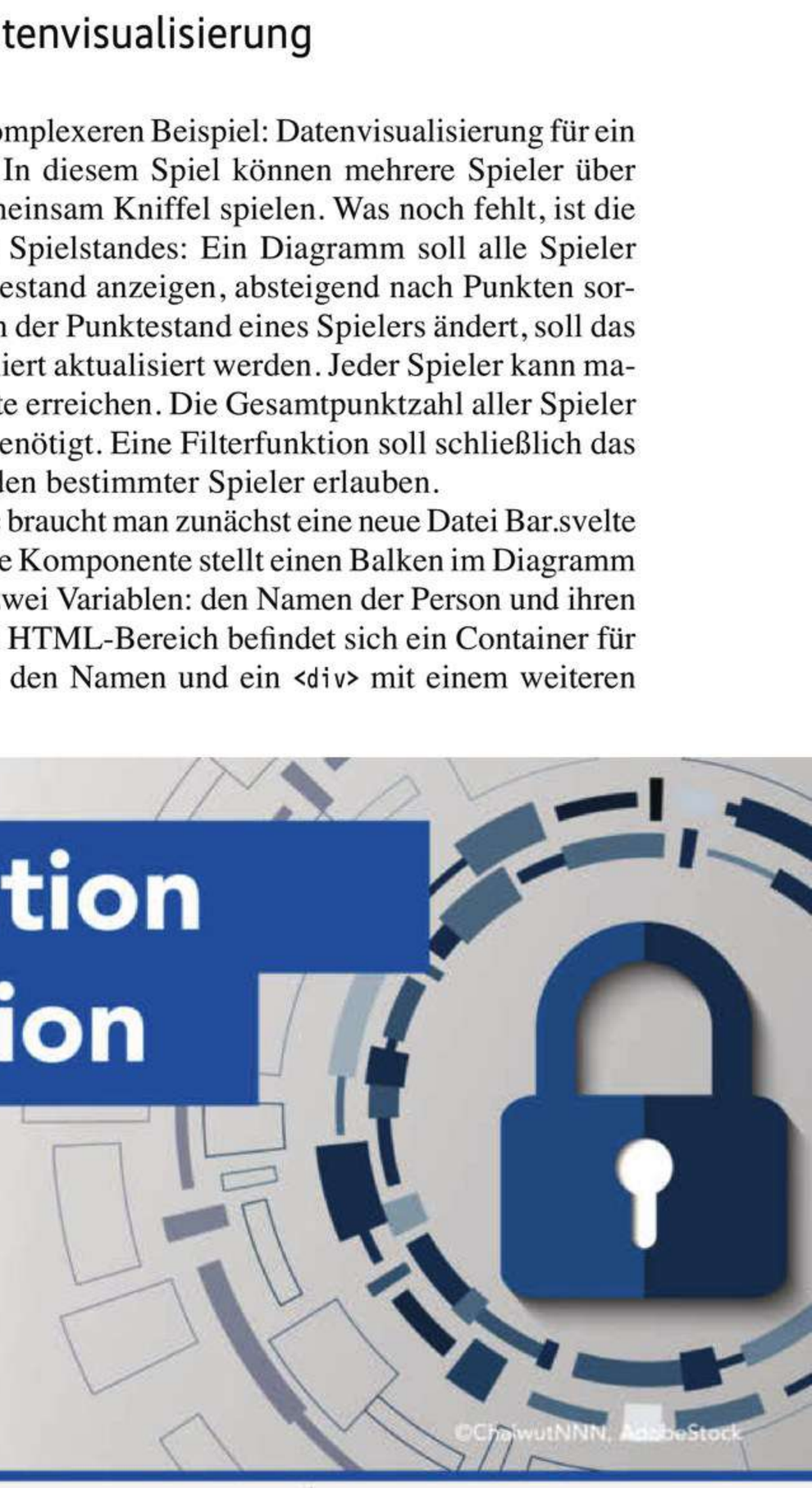
Darstellung der Balkenkomponente im Browser (Abb. 2)

Knifflige Datenvisualisierung

Nun zu einem komplexeren Beispiel: Datenvisualisierung für ein Online-Kniffel. In diesem Spiel können mehrere Spieler über das Internet gemeinsam Kniffel spielen. Was noch fehlt, ist die Darstellung des Spielstandes: Ein Diagramm soll alle Spieler mit deren Punktestand anzeigen, absteigend nach Punkten sortiert. Sobald sich der Punktestand eines Spielers ändert, soll das Diagramm animiert aktualisiert werden. Jeder Spieler kann maximal 375 Punkte erreichen. Die Gesamtpunktzahl aller Spieler wird ebenfalls benötigt. Eine Filterfunktion soll schließlich das schnelle Auffinden bestimmter Spieler erlauben.

Im Ordner src braucht man zunächst eine neue Datei Bar.svelte (Listing 4). Diese Komponente stellt einen Balken im Diagramm dar. Sie enthält zwei Variablen: den Namen der Person und ihren Punktestand. Im HTML-Bereich befindet sich ein Container für den Balken, der den Namen und ein `<div>` mit einem weiteren

Offensive Web Application Pentesting & Exploitation



Live-Webinar

am 22.06. um 10 Uhr

Preis: 150,00 € inkl. MwSt.

Aufbauend auf zu Beginn kurz aufgefrischten Basis-Techniken des Web Pentesting werden Sie anschließend die Nutzung offensiverer Techniken erlernen. Ziel wird es sein, tiefer in Systeme einzudringen (Post-Exploitation) sowie Rückkanäle sinnvoll aufzubauen. Folgende Schwerpunkte werden dabei behandelt und in Live-Demos bzw. wer mag auch gerne an vom Trainer bereitgestellten Übungszielen eigenständig in der Cloud ausgeführt:

- Aufspüren von Sicherheitslücken
- Eskalieren zu Remote Code Executions
- das Persistieren im System und Legen von Remote- und Reverse-Shells
- die Exfiltration von Daten über Tunnel
- das Ausnutzen von Blind- sowie Super-Blind-Lücken mittels Out-of-Band-Seitenkanäle (Time und DNS)

sowie vieles weitere.



Referent:
Christian Schneider

Whitehat Hacker & Trainer

heise Events
Conferences, Seminars, Workshops

www.heise-events.de/webinare/pen_exploit

Listing 7: BarChart.svelte

```
<script>
  import Bar from './Bar.svelte';
  export let title;
  let data = [];
  export function addPerson(name) {
    data = [...data, { name, value: 0, id: data.length }];
  }
</script>

<article>
  <div>
    <h3>{title}</h3>
  </div>
  {#each data as entry}
    <Bar name={entry.name} value={entry.value}></Bar>
  {/each}
</article>
```

`div`-Element enthält, aus dem später der Balken wird. Die Berechnung der Balkenbreite geschieht inline. Das innere `div`-Element enthält ein dynamisches Style-Attribut: Ändert sich der Punktestand, ändert sich auch die Breite. Als Letztes kommt der Punktestand. Listing 5 zeigt die zugehörige main.js.

Nach dem Speichern zeigt sich das Ergebnis: Name und Betrag sind sichtbar. Allerdings ist noch kein Balken zu sehen, da das entsprechende Styling fehlt. Listing 6 enthält ein Flexbox-Layout, ein paar Abstände und die Hintergrundfarbe für den Balken. Dieser erscheint im Browser, sobald das Styling in die Datei Bar.svelte kopiert und gespeichert wurde.

Das Balkendiagramm

Der Code für das komplette Diagramm kommt in die neue Datei BarChart.svelte (Listing 7). Die Variable `title` speichert den Namen des Diagramms, das Array `data` den Namen einer Person zusammen mit ihrem Punktestand. Das Array ist von außen nicht zuweisbar, da das `export`-Statement fehlt. Die Funktion `addPerson`, der ein Name übergeben wird, übernimmt seine Initialisierung. Sie weist der Variablen `data` ein neues Array zu, das die Daten aus dem bestehenden Array und ein neues Objekt mit drei Properties enthält: einer ID (mehr dazu später), dem Namen und dem Punktestand. Das bestehende Array zu erweitern, funktioniert hier nicht, da Svelte die Änderung des Arrays nur registriert, wenn eine Zuweisung an die Variable erfolgt.

Damit alle Personen dargestellt werden, enthält das HTML der Komponente eine `{#each}`-Schleife, der ein iterierbares Objekt – hier das Array `data` – übergeben wird. Für jedes Objekt im Array werden alle Tags gerendert bis zum Erreichen von `{/each}`. Pro Person rendert die Komponente also einen Balken. Jeder Balken bekommt den jeweiligen Namen und Punktestand der Person zugewiesen. Dazu wird am Anfang die Balkenkomponente importiert. Die Datei main.js benötigt eine Anpassung, damit man das Ergebnis betrachten kann (Listing 8). Zusammen mit dem Namen einer Person wird im Diagramm der Punktestand abgebildet. Um diesen im Verlauf des Spiels aktualisieren zu können, ist eine weitere Funktion in BarChart.svelte erforderlich (Listing 9). Sie erwartet den Namen einer Person und den neuen Punktestand.

Die Variable `index` speichert die Position des Namens, falls dieser im Array enthalten ist. An die entsprechende Stelle im Array kommt der Inhalt des vorhandenen Objektes, lediglich der Punktestand wird mit dem neuen Wert überschrieben. Danach folgt die Sortierung des Arrays, da die Darstellung der Personen immer in absteigender Reihenfolge ihres Punktestandes erfolgen soll.

Listing 8: Erweiterung von main.js

```
import BarChart from './BarChart.svelte';

const players = ['Johanna', 'Laura', 'Max', 'Peter'];

const chart = new BarChart({
  target: document.body,
  props: {
    title: 'Spielstand',
  }
});

players.forEach(name => chart.addPerson(name));
```

Listing 9: Anpassung des Punktestandes (BarChart.svelte)

```
export function update(name, value) {
  const index = data.findIndex((d) => d.name === name);
  data[index] = { ...data[index], value };
  data.sort((a, b) => b.value - a.value);
}
```

Den Punktestand aktualisieren

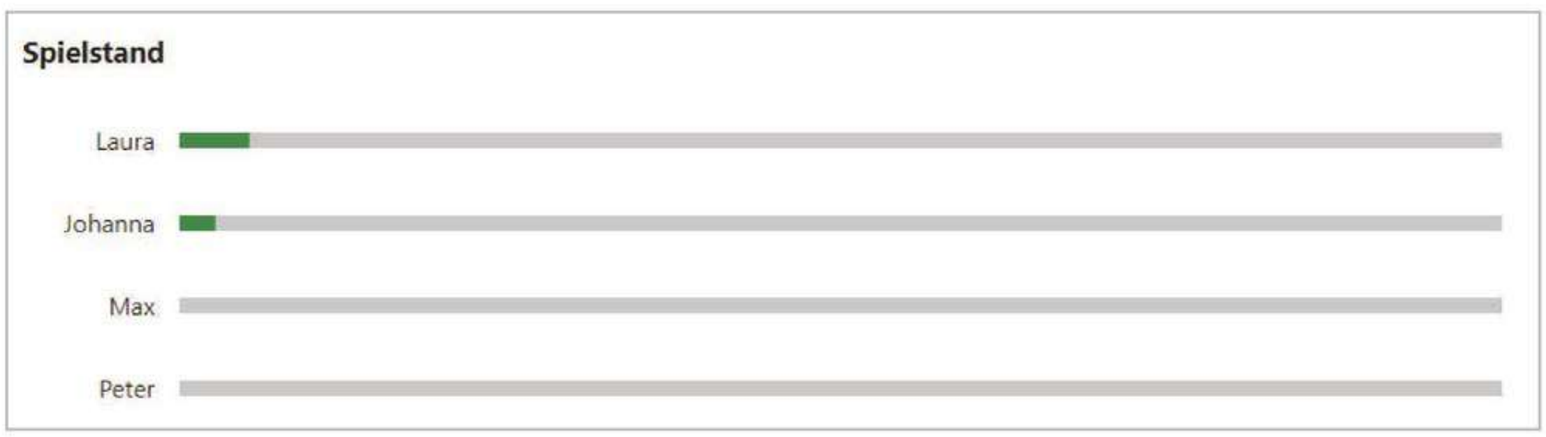
Der Aufruf der neuen Funktion erfolgt in main.js:

```
chart.update('Johanna', 10);
chart.update('Laura', 20);
```

Schließlich erhalten Laura und Johanna grüne Balken im Diagramm. Der Balken von Laura ist nun oben im Diagramm, da sie die meisten Punkte hat. Leider springen die Balken einfach auf den neuen Wert. Eine Animation schafft hier Abhilfe. Dazu muss das Styling in Bar.svelte um eine Zeile ergänzt werden, die die Balken bei Änderung der Punktestände animiert:

```
.bar > div > div {
  ...
  transition: width .3s ease-out;
}
```

Die Datei main.js erhält zwei neue Funktionen zum Testen des Diagramms (Listing 10). Die erste generiert zufällige Zahlen von 0 bis 375; ihr kann man Parameter für den Minimal- und Maximalwert übergeben. Die zweite Funktion aktualisiert nach Ablauf einer übergebenen Zeit eine zufällige Person im angegebenen Balkendiagramm. Der Aufruf dieser Funktion erfolgt am Ende der Datei.



Das Diagramm enthält jetzt einen animierten Balken pro Spieler (Abb. 3).

Reactive Statements

Bisher fehlt die Darstellung der Gesamtpunktzahl im Diagramm. Dafür eignet sich ein anderer Mechanismus von Svelte:

Listing 10: Testfunktionen in main.js

```
import BarChart from './BarChart.svelte';

const players = ['Johanna', 'Laura', 'Max', 'Peter'];;

const getRandom = (max = 375, min = 0) => {
  min = Math.floor(min);
  max = Math.ceil(max);
  return Math.round(Math.random() * (max - min)) + min;
};

const updateData = (chart, people, time) => {
  setTimeout(() => {
    const index = getRandom(people.length - 1);
    const name = people[index];
    const value = getRandom();

    chart.update(name, value);

    updateData(chart, people, time);
  }, time);
};

const chart = new BarChart({
  target: document.body,
  props: {
    title: 'Spielstand',
  }
});

players.forEach(name => chart.addPerson(name));
updateData(chart, players, 2000);
```

Listing 11: Ermittlung der Gesamtpunktzahl per Reactive Statement

```
// im <script> Tag
$: total = data
  .reduce((acc, actual) => acc + actual.value, 0);

// nach der Schleife im HTML
<div>
  <p>Gesamtpunkte:</p>
  <p>{total}</p>
</div>
```

Listing 12: Ein Reactive Statement macht das HTML in Bar.svelte übersichtlicher

```
<script>
  const MAX_VALUE = 375;

  export let name = '';
  export let value = 0;

  $: width = Math.round(100 * value / MAX_VALUE);
</script>

<div class="bar">
  <p>{name}</p>
  <div>
    <div style="width: {width}%;"></div>
  </div>
  <p>{value}</p>
</div>
```

Reactive Statements. Zur Kennzeichnung solcher Anweisungen verwendet Svelte die Kombination von Dollarzeichen und Doppelpunkt. Die Syntax ist valides JavaScript und wird als Label bezeichnet (siehe ix.de/z88f).

Das Kompilieren erzeugt daraus den Code für die automatische Aktualisierung der Variablen. Er führt die Anweisungen vor dem Rendern aus, falls sich Variablen dabei ändern. Das Balkendiagramm erhält eine neue Variable namens `total` und neue HTML-Elemente für die Ausgabe. Die JavaScript-Funktion `Array.reduce` addiert alle Beträge, um das Ergebnis in `total` zu speichern. Sobald sich ein Punktestand ändert, passen sich die Gesamtpunkte automatisch an (Listing 11). Ein Reactive Statement macht auch das HTML in der Komponente für die Balken in Bar.svelte übersichtlicher: Die Konstante `MAX_VALUE` speichert die

Docker und Container in der Praxis

16. und 17. Juni 2020, je 9 bis 12.30 Uhr

Der **Online-Workshop** „Docker und Container in der Praxis“ richtet sich an Entwickler und Administratoren, die neu in das Thema einsteigen wollen oder sollen. Im Anschluss an die zwei Vormittage wissen Sie, wobei Container Sie in Ihrer Arbeit unterstützen können und sind in der Lage, eine eigene Infrastruktur in Betrieb zu nehmen oder eigene Docker-Abbilder zu verpacken.

Themenschwerpunkte

- Docker-Grundbegriffe
- Images finden und auswählen
- Eigene Software verpacken
- Dienste mit TLS veröffentlichen
- Fehler finden und Logs auswerten
- Hands-On in einer vorbereiteten Test-Umgebung

Preis: 499,00 Euro inkl. MwSt.





Referent:
Jan Mahn,
Heise Medien GmbH & Co. KG

www.heise-events.de/workshops/dockon

Maximalpunktzahl im Kniffel (375), die Variable `width` berechnet sich neu, sobald sich der Punktestand ändert (Listing 12).

Noch eine Animation

Das Diagramm wird nun alle paar Sekunden aktualisiert. Allerdings ist es schwer, den Überblick zu behalten, wenn sich die Reihenfolge der Spieler ändert. Die Lösung ist eine Animation der vertikalen Balkenposition. Svelte hilft dabei: Eine `import`-Anweisung macht eine passende Animation in BarChart.svelte verfügbar. `flip` berechnet Start- und Endposition eines Elements und animiert den Zwischenweg. Die Animation gehört in die Schleife des HTML-Bereichs (Listing 13).

Bei jedem Update der Daten kann sich die Sortierung des Arrays ändern, was dann eine Animation auslöst. Dazu muss jedes Element im Array eindeutig identifizierbar sein. Dies ist der Grund für die ID, die mit jedem Namen und Punktestand gespeichert wird. Die Dauer der Animation ist auf 500 Millisekunden festgelegt.

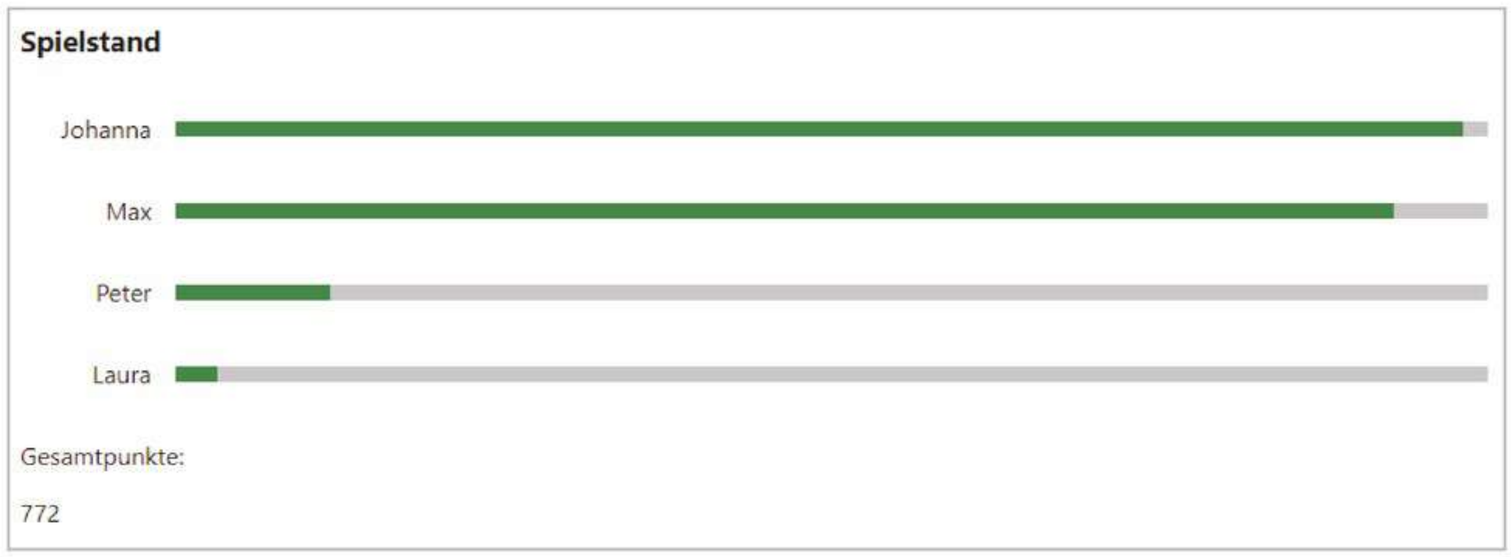
Damit Balken und Position nicht gleichzeitig animiert werden, ist eine Verzögerung notwendig. Diese gehört zur CSS-Transition in Bar.svelte:

```
transition: width .3s .5s ease-out;
```

Die Transition der Balkenlänge erhält eine Verzögerung von einer halben Sekunde. Da die Animation der Balkenposition eine halbe Sekunde dauert, erfolgen die Animationen nun hintereinander statt gleichzeitig. Zwischen der Änderung des Punktestandes und der Animation des Balkens liegt eine kurze Verzögerung, damit haben Betrachter einen besseren Überblick.

Svelte-Bindings: Spieler filtern

Voraussetzung für einen Filter zur Auswahl einzelner Spieler ist ein Input-Feld in BarChart.svelte. Der Script-Bereich erhält eine weitere Variable namens `filter` und das HTML ein neues Input-Element. Entwickler können Svelte-Bindings nutzen, um Eingabe und Variable zu verknüpfen. Sobald ein User etwas in das Eingabefeld schreibt, ändert sich der Inhalt der Variablen. Ein gefiltertes Array enthält nur noch die Namen, die dem Filter entsprechen. Dieses Array wird für das Rendern der Daten verwendet. Damit die Gesamtpunkte ebenfalls angepasst werden, kommt das Array auch dort zum Einsatz.



Ein Reactive Statement berechnet den Gesamtpunktestand (Abb. 4).

Die folgenden Codezeilen ergänzen BarChart.svelte:

```
// im <script>-Tag
let filter = '';
// im HTML, direkt nach der Überschrift <h3>
<input bind:value={filter} placeholder="Filtertext eingeben">
```

Nun muss nur noch das Array mit den Daten gefiltert werden, sobald etwas eingegeben wird. Dazu erstellt man ein gefiltertes Datenarray in BarChart.svelte:

```
$: filteredData = data.filter(entry => {
  const lcName = entry.name.toLowerCase();
  const lcFilter = filter.toLowerCase();
  return lcName.includes(lcFilter);
});
```

Man nimmt also die Daten und filtert sie anhand der Namen der Personen. Sobald ein Name den Inhalt des Input-Feldes enthält (Groß-/Kleinschreibung wird dabei ignoriert), schreibt man das entsprechende Objekt in das neue Array. Dieses Array setzt man in die Schleife ein, um die herausgefilterten Personen nicht mehr anzuzeigen. Damit die Gesamtpunkte ebenfalls angepasst werden, kommt das Array auch dort zum Einsatz. Listing 14 zeigt die Codezeilen.

Optisch aufpoliert

Durch Filtern der Liste verliert das Diagramm an Höhe. Das liegt daran, dass sich die Höhe aus dem Inhalt berechnet. Damit die Höhe erhalten bleibt, definiert man einen Wert dafür (Listing 15):

Listing 13: Animierte Veränderung der Balkenposition (BarChart.svelte)

```
// im <script> Tag
import { flip } from 'svelte/animate';

// im HTML-Bereich
{#each data as entry, index (entry.id)}
  <div animate:flip="{{ duration: 500 }}">
    <Bar name={entry.name} value={entry.value}></Bar>
  </div>
{/each}
```

Listing 14: Filterfunktion in BarChart.svelte

```
// im <script> Tag
$: total = filteredData
  .reduce((acc, actual) => acc + actual.value, 0);

// Anpassung der Schleife im HTML
{#each filteredData as entry, index (entry.id)}
  <div animate:flip="{{ duration: 500 }}">
    <Bar name={entry.name} value={entry.value}></Bar>
  </div>
{/each}
```

Listing 15: Grafik mit konstanter Höhe (BarChart.svelte)

```
// im <script> Tag
const BAR_HEIGHT = 55;

// im HTML
<div style="height: {data.length * BAR_HEIGHT}px;">
  {#each filteredData as entry, index (entry.id)}
    <div animate:flip="{{ duration: 500 }}">
      <Bar name={entry.name} value={entry.value}></Bar>
    </div>
  {/each}
</div>
```

Listing 16: Styling für das Balkendiagramm

```
<style>
  :root {
    --padding-side: 2em;
  }

  article {
    border-radius: .3em;
    box-shadow:
      4px 4px 2px -2px rgba(0, 0, 0, 0.1),
      0px 0px 2px 0px rgba(0, 0, 0, 0.2);
    margin: 1em 2em 2em 2em;
    width: 30em;
  }

  article > div:first-child {
    align-items: baseline;
    background: rgba(240, 240, 240, 1);
    display: flex;
    flex-wrap: wrap;
    justify-content: space-between;
    padding: 0 1em;
  }

  article > div > div {
    padding: 0 var(--padding-side) 0 var(--padding-side);
  }

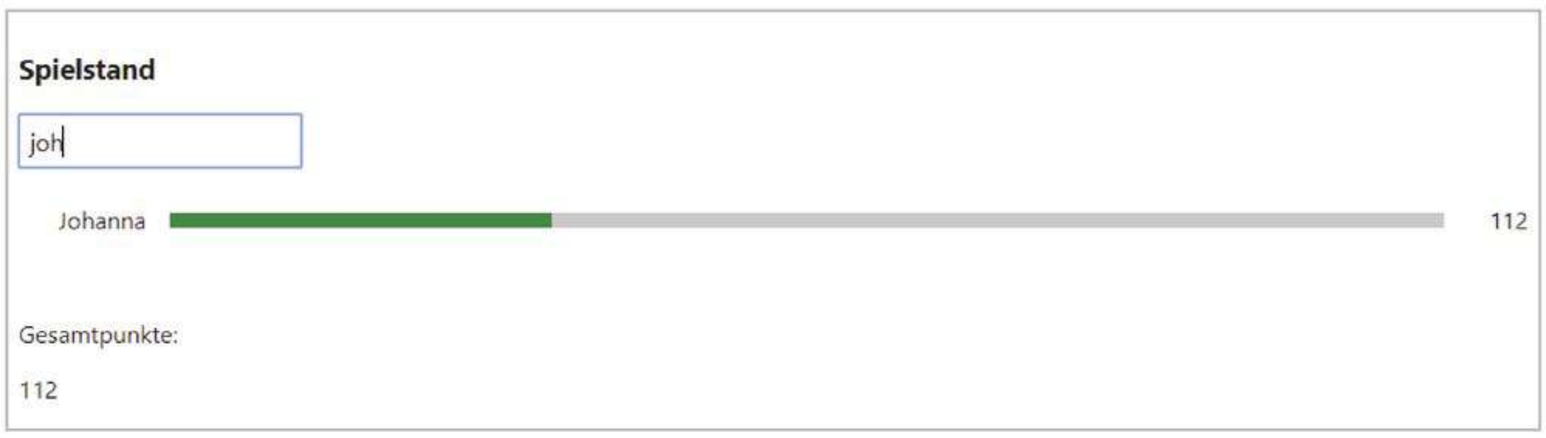
  article > div:last-child {
    align-items: center;
    display: flex;
    justify-content: flex-end;
    padding: 0 var(--padding-side) .5em var(--padding-side);
  }

  article > div > p:last-child {
    min-width: 2.4em;
    text-align: right;
  }

  input {
    background: rgba(250, 250, 250, 1);
    border: 0;
    font-size: .9em;
    padding-left: .5em;
    max-width: 10em;
  }

  p:first-child {
    margin: 0 1em;
  }
</style>
```

Die Liste der angezeigten Spieler lässt sich über den Text im Eingabefeld filtern (Abb. 5).

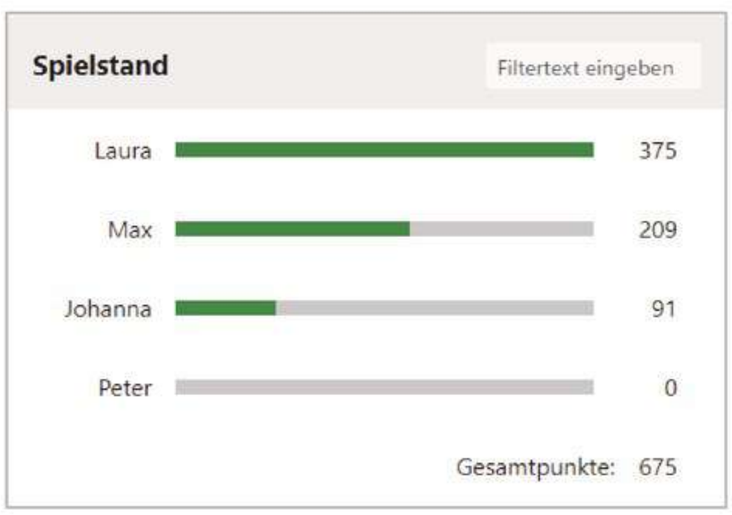


Die neue Variable `BAR_HEIGHT` mit dem Wert 55 legt die Höhe eines Balkens fest. Dem zusätzlichen `div`-Tag in der Schleife wird die Höhe für alle Balken mittels CSS zugewiesen. Der Wert für die Höhe der einzelnen Balken ist zwar statisch, dies reicht für das Beispiel jedoch aus. Bei der ansprechenden Gestaltung der Balkendiagramme hilft zu guter Letzt das Styling aus Listing 16. Abbildung 6 zeigt das Ergebnis.

Fazit

Die hier vorgestellten Möglichkeiten sind noch lange nicht alles was, Svelte zu bieten hat – das Beispiel zeigt lediglich einige der Features. Entwickler müssen mit Svelte nur wenig Code schreiben; das Framework übernimmt das Erstellen von Event Handlern, die direkte Interaktion mit dem DOM und vieles mehr. Als Entwickler kann man sich darauf konzentrieren, *was* passieren soll, das *Wie* tritt in den Hintergrund.

Zudem ist die Syntax von Svelte recht einfach zu erlernen. Tutorials und eine ausführliche Beschreibung aller Features hält die offizielle Svelte-Website svelte.dev bereit. Wer sich damit einen Tag beschäftigt, sollte in der Lage sein mit dem Framework zu arbeiten. Die kompilierten Bundles sind sehr kompakt. Wer sich für Vergleiche zwischen JavaScript-Frameworks in Bezug auf Menge des Codes, Performance oder Größe der Bundles interessiert, wird im Web fündig (siehe ix.de/z88f).



Das fertige Balkendiagramm mit Styling (Abb. 6)

Für Routing, Testen, Skalierung und die Verwendung von TypeScript gibt es momentan noch keinen Standardweg. Die Arbeit an diesen und anderen Themen läuft aber bereits. Die Webseiten von Svelte und GitHub sind gute Anlaufstellen für weitere Informationen. Zwei nennenswerte Projekte auf Basis von Svelte sind Sapper und Svelte Native (siehe ix.de/z88f). Sapper ist ein Framework zum Bau von Webapplikationen, Svelte Native richtet sich an Entwickler von Apps für Smartphones.

Für Projekte in den Bereichen Embedded, Datenvisualisierung oder Animationen ist Svelte jetzt schon eine interessante Alternative zu den etablierten Frameworks. Ob man Svelte einsetzen kann und will, hängt auch stark von den eigenen Ansprüchen und dem Umfeld ab, in dem man arbeitet. Es lohnt sich auf jeden Fall, Svelte kennenzulernen. (odi@ix.de)

Quellen

Videos, Software und Tools zu Svelte: ix.de/z88f



Daniel Frank

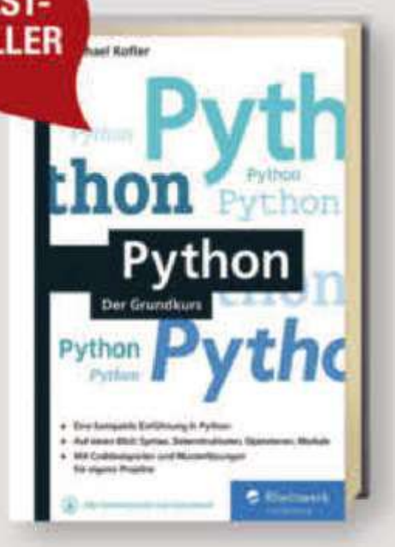
arbeitet als Senior Software Engineer bei Method Park im Team Internet of Things an den Themen Webtechnologie, Prototyping und Benutzerschnittstellen. Sein persönliches Interesse gilt den Herausforderungen von Agile und User Experience.

Für Wissenshungrige

Ausgewählte Fachliteratur

shop.heise.de/buecher

BEST-SELLER



Michael Kofler

Python - Der Grundkurs

Diese Python-Einführung konzentriert sich auf das Wesentliche und zeigt Ihnen, wie Sie die Sprache in eigenen Projekten einsetzen. Erfahren Sie praxisgerecht, wie Sie mit Python Daten verarbeiten, den Raspberry Pi ansteuern, wiederkehrende Aufgaben automatisieren und vieles mehr.

ISBN 9783836266796
shop.heise.de/python-buch

14,90 € ›



Christian Solmecke, Sibel Kocatepe

DSGVO für Website-Betreiber

Ihr Leitfaden für die sichere Umsetzung der EU-Datenschutz-Grundverordnung. Experten erklären Schritt für Schritt, wie Sie Ihren Webauftritt vollständig rechtskonform gestalten – gut verständlich auch für Nichtjuristen.

ISBN 9783836267120
shop.heise.de/dsgvo-websites

39,90 € ›



Jörg Frochte

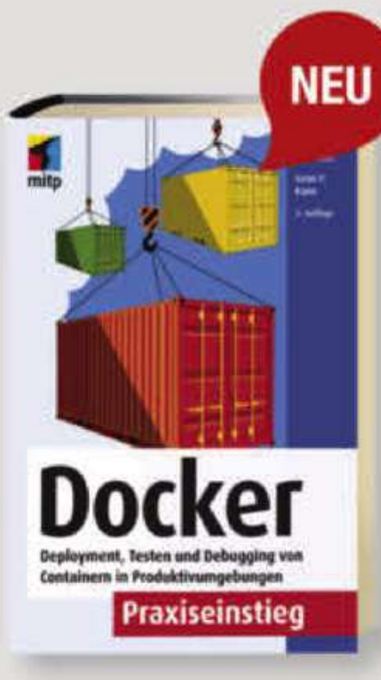
Maschinelles Lernen (2. Aufl.)

Maschinelles Lernen ist ein interdisziplinäres Fach, das die Bereiche Informatik, Mathematik und das jeweilige Anwendungsgebiet zusammenführt.
In diesem Buch werden alle drei Teilgebiete gleichermaßen berücksichtigt.

ISBN 9783446459960
shop.heise.de/maschinelles-lernen

38,00 € ›

NEU



Karl Matthias, Sean P. Kane

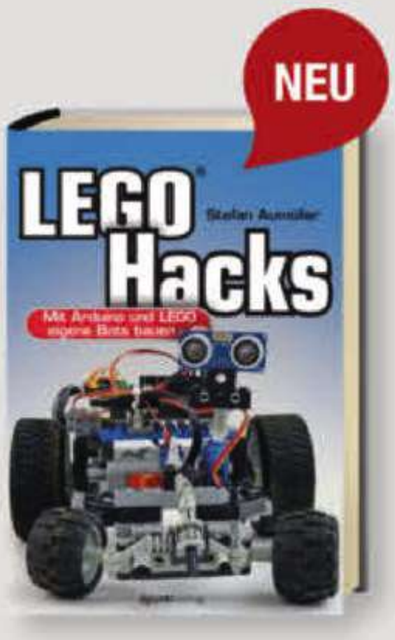
Docker Praxiseinstieg (2. Aufl.)

Lernen Sie, wie Sie Docker-Images Ihrer Anwendungen erstellen, testen und deployen sowie skalieren können, und wie Sie die Container in der Produktivumgebung pflegen und warten. Die Einrichtung und das Testen von Docker-Anwendungen kommen ebenso zur Sprache wie das Debugging eines laufenden Systems.

ISBN 9783958459380
shop.heise.de/docker-praxis2

25,99 € ›

NEU



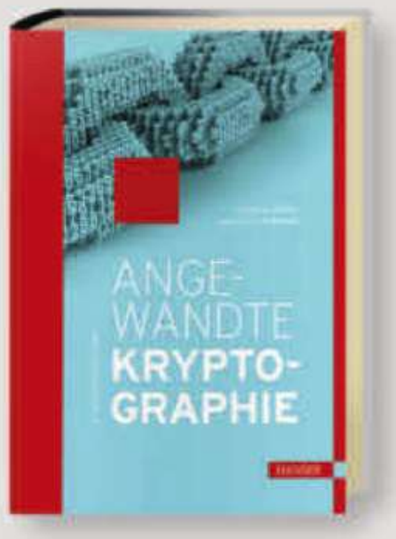
Stefan Aumüller

LEGO® Hacks

Dieses Buch zeigt, wie aus LEGO und dem Arduino faszinierende Modelle werden. Lernen Sie, Sensoren und Aktoren, mit LEGO-Elementen zu verbinden und daraus viele neue spannende Projekte aufzubauen. Auch Grundlagen der Elektronik werden Ihnen vermittelt

ISBN 9783864906435
shop.heise.de/buch-legohacks

29,90 € ›



Wolfgang Ertel, Ekkehard Löhmann

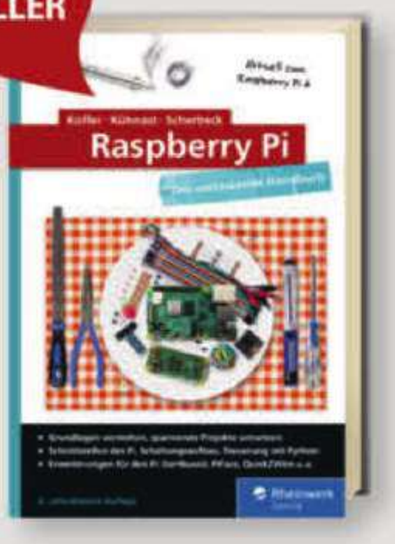
Angewandte Kryptographie (6. Aufl.)

Ziel des Buches ist es, Grundwissen über Algorithmen und Protokolle zu vermitteln und kryptographische Anwendungen aufzuzeigen. Mit so wenig Mathematik wie nötig, aber vielen Beispielen, Übungsaufgaben und Musterlösungen.

ISBN 9783446463134
shop.heise.de/kryptographie6

34,99 € ›

BEST-SELLER



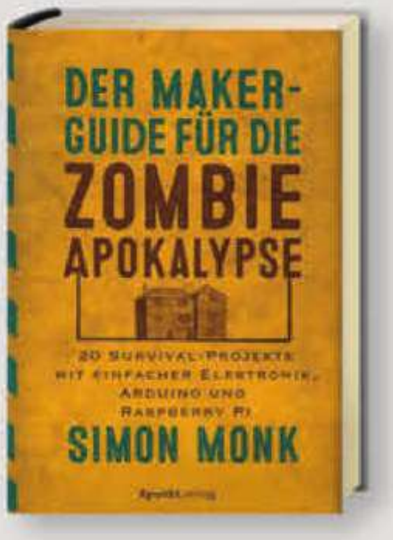
Michael Kofler, Charly Kühnast, Christoph Scherbeck

Raspberry Pi (6. Aufl.)

Das umfassende Handbuch mit über 1.000 Seiten komplettem Raspberry-Wissen, um richtig durchstarten zu können. Randvoll mit Grundlagen und Kniffen zu Linux, Hardware, Elektronik und Programmierung.
Aktuell für alle Versionen, inkl. Raspberry Pi 4!

ISBN 9783836269339
shop.heise.de/raspberry-6

44,90 € ›



Simon Monk

Der Maker-Guide für die Zombie-Apokalypse

Bereiten Sie sich vor: mittels 20 Survival-Projekten mit einfacher Elektronik, Arduino und Raspberry Pi werden Sie Ihren eigenen Strom erzeugen, unverzichtbare Bauteile vor dem Zombie-Zugriff retten und lebensrettende Elektronikschaltungen bauen, um Untote aufzuspüren.

ISBN 9783864903526
shop.heise.de/zombies

24,90 € ›

PORTOFREI
AB 15 €
BESTELLWERT

› Generell portofreie Lieferung für Heise Medien- oder Maker Media Zeitschriften-Abonnenten oder ab einem Einkaufswert von 15 €.
Nur solange der Vorrat reicht. Preisänderungen vorbehalten.

und Maker!

Zubehör und Gadgets

shop.heise.de/gadgets

NEU



ParkLite

ParkLite denkt mit. Die elektronische Parkscheibe stellt automatisch nach ca. 20 Minuten die Parkzeit ein. Damit ist Schluss mit Bußgeldern! Hitze- und kältebeständig, inklusive Reinigungstuch und Klebepads.

shop.heise.de/parklite **29,90 € ›**



Aluminium-Case FLIRC

Das hochwertige Gehäuse aus stabilem Aluminium ist ideal, um den Raspberry Pi 4 als Media Center zu verwenden. Das elegante Design integriert sich optimal in jede Wohnumgebung. **Auch im Set mit Raspi 4 Model B 2GB erhältlich.**

shop.heise.de/flirc **23,90 € ›**

NEU



musegear® finder Version 2

Finden Sie Schlüssel, Handtasche oder Geldbeutel bequem wieder statt ziellos zu suchen. Mit dem Finder können Sie z.B. das Smartphone klingeln lassen oder Wertgegenstände einfach tracken und noch mehr.

shop.heise.de/musegear **24,90 € ›**



Raspberry Pi-Kameras

Aufsteckbare Kameras, optimiert für verschiedene Raspberry Pi-Modelle mit 5 Megapixel und verschiedenen Aufsätzen wie z. B. Weitwinkel für scharfe Bilder und Videoaufnahmen.

shop.heise.de/raspi-kameras **ab 18,50 € ›**

NEUER PREIS!



ArduiTouch-Set

Setzen Sie den ESP8266 oder ESP32 jetzt ganz einfach im Bereich der Hausautomation, Metering, Überwachung, Steuerung und anderen typischen IoT-Applikationen ein!

~~69,90 €~~

shop.heise.de/arduitouch **36,90 € ›**

NEU



PokitMeter – Multimeter, Oszilloskop und Logger

PoKit misst, zeigt und protokolliert eine Vielzahl von Parametern wie Spannung, Strom, Widerstand und Temperatur mittels Verbindung via Bluetooth mit Ihrem Smartphone oder Tablet.

shop.heise.de/pokit **94,90 € ›**

NEUER PREIS!



Komplettset Argon ONE Case mit Raspberry Pi 4

Das Argon One Case ist eines der ergonomischsten und ästhetischsten Gehäuse aus Aluminiumlegierung für den Raspberry Pi. Es lässt den Pi nicht nur cool aussehen, sondern kühlt auch perfekt und ist leicht zu montieren. Praktisch: alle Kabel werden auf der Rückseite gebündelt ausgeführt – kein Kabelsalat!

~~117,60 €~~

shop.heise.de/argon-set **99,90 € ›**

NEU



NVIDIA Jetson Nano B01

Die neue Revision B01!
Die Leistung moderner KI für Millionen Geräte. Mit dem Jetson Nano von NVIDIA können Sie als Heimbastler oder Entwickler platzsparend und effizient in die Welt der KI eintauchen. Ideale Voraussetzung für die Programmierung neuronaler Netze dank vier A57-Kerne und einem Grafikprozessor mit 128 Kernen.
Inklusive Netzteil!

shop.heise.de/jetson **134,90 € ›**

28% RABATT



Make Family + Makey-Paket

Darüber freut sich die ganze Familie: „Make Family" - das vollgepackte PDF-Magazin mit 21 Anleitungen zum kreativen Basteln mit Kids auf über 200 Seiten.
Dazu: der knuffige Makey-Plüschroboter und der Makey-Lötbausatz mit LEDs und Batterie.

~~27,70 €~~

shop.heise.de/makey-paket **19,90 € ›**



„No Signal" Smartphone-Hülle

Passend für Smartphones aller Größen bis 23cm Länge blockt diese zusammenrollbare Hülle alle Signale von GPS, WLAN, 3G, LTE, 5G und Bluetooth, sowie jegliche Handy-Strahlung. Versilbertes Gewebe im Inneren der Tasche aus recycelter Fallschirmseide bildet nach dem Schließen einen faradayschen Käfig und blockiert so alles Signale.

shop.heise.de/no-signal-sleeve **29,90 € ›**

› Bestellen Sie ganz einfach online unter **shop.heise.de** oder per E-Mail: **service@shop.heise.de**

heise shop

shop.heise.de ›



Elm-Apps als Web Components

Funktional integriert

Martin Grotz

Mit Elm lassen sich Webanwendungen ohne Laufzeitfehler erstellen. Die Sprache eignet sich zudem als Einstieg in die funktionale Programmierung.

Die Sprache Elm dient der Entwicklung browserbasierter Webanwendungen. Sie kommt mit einem geringen Sprachumfang aus und vermeidet unerwartete Laufzeitfehler. Die Webanwendungen bleiben fehlerfrei, ein sicheres Refactoring des bestehenden Codes ist gewährleistet. Elm ist zugleich Programmiersprache, Architektur, Framework und Laufzeitumgebung.

Am Beispiel einer Komponente zur Anzeige von Bewertungssternen gibt der Artikel einen Einstieg in die Entwicklung mit Elm. Der Benutzer wählt dabei per Mausklick eine Wertung zwischen einem und fünf Sternen aus. Zuerst wird diese Komponente alleinstehend angefertigt und später als Web Component für die einfache Nutzung in Webseiten bereitgestellt.

Der Aufbau jedes Elm-Programms folgt einer fest vorgegebenen Architektur (Pattern „The Elm Architecture“), die auf einem unidirektionalen Datenfluss aufbaut (Abbildung 1). Daten fließen dabei vom Model in die darstellende `view`-Funktion. Die `update`-Funktion verarbeitet alle in der `view`-Funktion erzeugten Nachrichten und übergibt eine veränderte Kopie des Models zur Aktualisierung der Darstellung an die `view`-Funktion.

Der Hauptvorteil dieser Architektur: Der Datenfluss ist leicht nachvollziehbar und alle Daten liegen an einer zentralen Stelle. Außerdem kann man innerhalb der Anwendung zu Debug-Zwecken „zurückspulen“, da der Elm-Debugger alle Zustände und Nachrichten protokolliert und wieder rückgängig machen kann. Allerdings gibt es auch Nachteile: Die verbreitete, komponentenbasierte Art, Anwendungen zu bauen, ist damit kaum möglich. Komponenten können keine eigene, nach außen hin unsichtbare Datenhaltung haben. Zudem kann die Integration externer Bibliotheken kompliziert sein, da deren Daten und Nachrichten explizit in die eigene Anwendung zu integrieren sind.

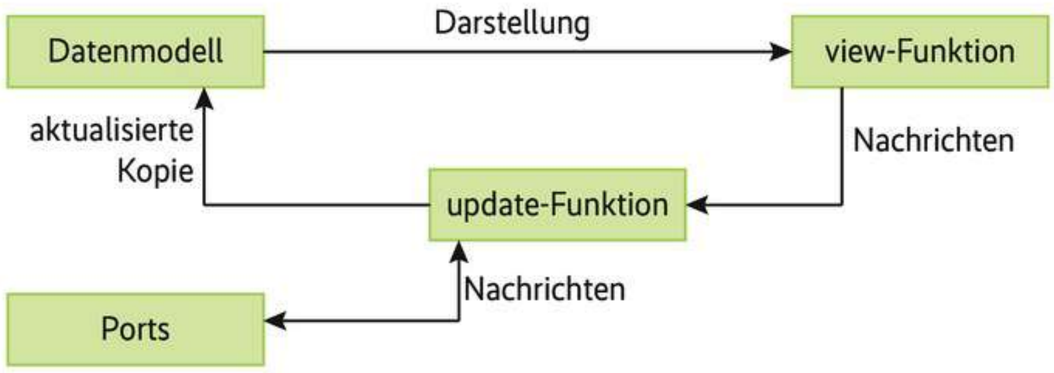
Die Elm-Architektur hat mittlerweile über Redux aus der React-Welt größere Verbreitung gefunden. Es gibt sie aber auch für Web-Frameworks wie Angular und Vue.js. Weitere Projekte bringen diese Architektur in andere Programmierwelten, zum Beispiel über die Bibliothek Elmish-WPF in die Windows-Desktop-UI-Entwicklung mit .NET.

Ungewohnte Datentypen

Für den Einstieg in Elm benötigt man keine lokale Installation, der Onlineeditor Ellie stellt alles Notwendige im Browser bereit.

iX-TRACT

- Elm ist eine funktionale Programmiersprache zur Erstellung von Webanwendungen.
- Die Sprache gibt zusammen mit der Laufzeitumgebung zahlreiche Garantien. Die wichtigste: Es gibt keine unerwarteten Laufzeitfehler.
- Elm-Anwendungen lassen sich als Web Components in bestehende Seiten integrieren, was den Einstieg in die funktionale Programmierung erleichtert.



Die unidirektionale Elm-Architektur zentralisiert den Anwendungszustand im Datenmodell und vereinheitlicht alle Zustandsübergänge durch das Auslösen und Verarbeiten von Nachrichten (Abb. 1).

Die Sprache enthält zwei in der objektorientierten Welt nicht häufig verwendete Datentypen: Records und Custom Types.

Records bilden die Grundlage der Datenmodellierung. Ein Record kann beliebige Felder enthalten und ist immer vollständig zu erstellen. Records sind, wie alle anderen Daten auch, unveränderbar (immutable). Einzig eine veränderte Kopie lässt sich erzeugen (Listing 1; alle Listings siehe ix.de/zbg3). Vergleiche zwischen zwei Records finden auf Strukturebene statt (Structural Equality), nicht auf Basis der Referenz (Reference Equality). Die Eigenschaften eines Record passen zur Welt des Domain-driven Design: Es lassen sich damit schnell vollwertige Value Objects erstellen, ohne Boilerplate-Code schreiben zu müssen.

Der zweite Datentyp sind die Custom Types, in anderen Programmiersprachen auch Union Types oder Algebraic Data Types genannt. Custom Types ähneln Enums aus anderen Sprachen, können aber mehr. Daten lassen sich jeder möglichen Ausprägung zuordnen (Listing 2), bei einer Abrechnungsschnittstelle sind das die jeweils zum Zahlungsmittel passenden Informationen: Kreditkartennummer und Ablaufdatum bei Kreditkartenzahlung, die IBAN bei Zahlung per Lastschrift oder die Accountdaten bei einer Bezahlung über PayPal.

Da Elm eine funktionale Programmiersprache ist, spielen Funktionen eine große Rolle. Sie sind sowohl als Argument als auch als Rückgabe einer Funktion erlaubt. Elm gestattet dabei auch die teilweise Anwendung von Funktionen (Partial Application). Viele Funktionen erwarten eine Funktion als Argument (Higher-Order Functions). Ein Beispiel dafür ist die Iteration über alle Elemente einer Liste mit anschließender Filterung. Hierbei kommt auch der Pipelineoperator zum Einsatz, der dazu dient, die Rückgabe einer Funktion direkt an die nächste Funktion zu übergeben (Listing 3). Die Programmiersprache implementiert außerdem anonyme Funktionen, die sich an Ort und Stelle definieren lassen.

In Elm ist es üblich, die Typsignatur bei Funktionen auf Modulebene mit anzugeben. Wie in funktionalen Sprachen üblich, sind einzelne Parameter mit einem -> getrennt. Der letzte Typ in dieser Kette ist der Rückgabetyp:

```
add : Int -> Int -> Int
add a b = a + b
```

Muster erkennen

Wichtiges Hilfsmittel zur Verarbeitung von Daten ist das Pattern Matching. Die wohl häufigste Form ist die Unterscheidung je nach Ausprägung eines Custom Type. Dabei lassen sich auch direkt die dem Custom Type zugehörigen Daten auspacken, zum Beispiel der Benutzername eines fiktiven PayPal-Accounts (Listing 4).

Listing 1: Immutable Records als Basis der Datenmodellierung

```
-- Typ definieren
type alias MyRecord =
    { text : String
    , count : Int
    }

-- Record erstellen
aRecord =
    { text = "Lorem Ipsum"
    , count = 42
    }

-- einen Record-Konstruktor erzeugen
anotherRecord = MyRecord "Lorem Ipsum" 42

-- veränderte Kopie erzeugen
changedCopy = { aRecord | count = 7 }
```

Listing 2: Custom Types zur Strukturierung unterschiedlicher Daten

```
type alias CardInfo = { holder : String, number : String }
type alias Account = { username : String, email : String }

type PaymentMethod
    = Cash
    | CreditCard CardInfo
    | PayPal Account
```

Für die weitere Entwicklung ist eine lokale Installation notwendig. Sie erfolgt am einfachsten mit npm:

```
npm install --global elm elm-format elm-test elm-analyse create-elm-app
```

Danach legt man ein neues Projekt an:

```
create-elm-app elm-rating
```

Unter Windows treten gelegentlich Fehlermeldungen der Art auf, dass der Befehl `elm` in den `node_modules` nicht existiert oder dass die Elm-Packages nicht installiert werden konnten. Nach einem Wechsel in den neu erstellten Ordner elm-rating und dem Ausführen von `elm-app start` sollte sich aber trotzdem eine Demoseite mit der URL http://localhost:3000/ im Browser öffnen. Für die Entwicklung empfiehlt sich Visual Studio Code mit dem Plug-in Elm. Achtung: Es gibt auch ein älteres Plug-in namens elm, das nicht mehr weiterentwickelt wird.

Die einzig relevante Datei ist Main.elm. Alles andere, was `create-elm-app` erzeugt hat, ist unverändert weiterverwendbar. Der Import häufig benötigter Namespaces aus der Elm-Standardbibliothek erspart das spätere Nachtragen der verwendeten HTML-Elemente und -Events:

Listing 3: Partial Application, Higher-Order Functions und Funktionsverkettung

```
-- Addition zweier Zahlen
add a b = a + b

-- Partial Application, "a" ist als 2 festgelegt, "b" ist noch offen
add2 = add 2

filteredList =
    -- 2 zu jeder Zahl in der Liste addieren
    List.map add2 [ 1, 2, 3, 5, 8, 13 ]
      -- Neue Liste an List.filter übergeben und via anonymer
      -- Filterfunktion nur Zahlen kleiner 10 behalten
      |> List.filter (\input -> input < 10)
      -- Ergebnis: [3, 4, 5, 7]
```

Listing 4: Pattern Matching zur Verarbeitung von Custom Types

```
pay : Amount -> PaymentMethod -> Bool
pay amount paymentMethod =
    case paymentMethod of
      Cash ->
        payWithCash amount

      CreditCard cardInfo ->
        payWithCard amount cardInfo

      PayPal { username, _email } ->
        payViaPayPal amount username
```



Die Bewertungskomponente enthält den Titel sowie eine Wertung von einem bis fünf Sterne (Abb. 2).

```
import Browser
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
```

Außerdem lässt sich die Funktion `view` stark vereinfachen, da der generierte Democode nicht mehr benötigt wird:

```
view : Model -> Html Msg
view model =
    div [] []
```

Jetzt beginnt die Umsetzung der Bewertungskomponente. Sie besteht aus den zu bewertenden (Film-)Titeln mit einer Bewertungsskala von bis zu fünf Sternen (Abbildung 2). Dabei werden nur ganze Sterne vergeben und die geringstmögliche Wertung ist ein Stern. Gibt es noch keine Bewertung, erscheinen alle Sterne inaktiv. Ein Klick auf einen Stern füllt die Sterne von links nach rechts farbig aus. Dabei lassen sich keine Sterne überspringen.

Das Datenmodell ist einfach, da nur zwei Werte zu speichern sind. Einer davon ist optional:

```
type alias Model =
    { title : String
    , rating : Maybe Rating
    }
type Rating = One | Two | Three | Four | Five
```

Der Typ `Maybe` ist ein bereits in Elm eingebauter Custom Type, der das mögliche Fehlen eines Wertes codiert. Elm beinhaltet kein `null` oder `undefined`, fehlende Werte sind daher immer vom Typ `Maybe`. Die möglichen Bewertungen beschreibt ein eigener Custom Type, der die Werte von eins bis fünf explizit unterscheidet. Das spart zahlreiche Eingabeprüfungen ein, da ungültige Werte gar nicht mehr vorkommen können – der Compiler lässt sie nicht mehr zu. Ungültige Werte durch ein Typsystem unmöglich zu machen, ist ein wichtiger Baustein beim Entwickeln fehlerfreier Programme. Das starke Typsystem von Elm trägt dazu bei. Bevor das Programm wieder kompiliert, ist noch die Funktion `init` anzupassen, um dem Model sinnvolle initiale Werte zuzuweisen:

```
init : ( Model, Cmd Msg )
init =
    ( { title = "A fancy title"
      , rating = Nothing
      }
    , Cmd.none)
```

Eine Frage der Ansicht

Ein Datenmodell alleine macht aber noch keine sinnvolle Webanwendung: Erst durch die Darstellung in der `view`-Funktion bekommt der Anwender etwas zu sehen und kann mit der Benutzeroberfläche interagieren.

Da die View nur Funktionen enthält und es keine Trennung in Template und restlichen Code gibt, steht der volle Sprachumfang zur Verfügung. Teile lassen sich einfach in eigene, wiederverwendbare Funktionen auslagern.

Alle HTML-Elemente stehen als Funktionen mit zwei Argumenten zur Verfügung: Das erste Argument ist eine Liste von Attributen, das zweite eine Liste von Kindelementen. Der Einfachheit halber werden zur Darstellung der Bewertungssterne passende Unicode-Zeichen verwendet. Eine Aufteilung des Codes in Hilfsfunktionen für die einzelnen Bestandteile verbessert die Lesbarkeit und erleichtert das spätere Auslagern in eigene Funktionen (Listing 5).

`onClick` registriert einen Click Handler in den HTML-Elementen der Sterne. Die Browser-Event-Handler-Funktion wandelt das JavaScript-Event in eine Elm-Nachricht zur späteren Behandlung in der `update`-Funktion um. Die Hilfsfunktion `ratingToStarCounts` auf Modulebene berechnet aus dem Rating die notwendige Menge der gefüllten und leeren Sterne (Listing 6).

Die beiden in der View verwendeten CSS-Klassen sind dann noch in der Datei main.css nachzutragen (Listing 7).

Listing 5: HTML-Rendering in der View-Funktion

```
view : Model -> Html Msg
view model =
    let
      -- HTML-Element für leeren Stern
      emptyStar index =
          span [ style "cursor" "pointer", onClick (StarClicked index) ] [ text "☆" ]

      -- HTML-Element für gefüllten Stern
      filledStar index =
          span [ style "cursor" "pointer", style "color" "yellow", onClick (StarClicked index) ] [ text "★" ]

      -- Wie viele Sterne von welcher Art werden benötigt?
      { numberOfFilledStars, numberOfEmptyStars } =
          ratingToStarCounts model.rating

      -- alle DOM-Elemente für die gefüllten Sterne
      renderedFilledStars =
          -- Liste mit Zahlen: [1..n]
          List.range 1 numberOfFilledStars
              -- Für jede Zahl einmal die HTML-Element-Funktion mit der
              -- Zahl als zusätzliches Argument aufrufen
              |> List.map filledStar

      -- alle DOM-Elemente für die leeren Sterne
      renderedEmptyStars =
          List.range (numberOfFilledStars + 1) (numberOfFilledStars + numberOfEmptyStars)
              |> List.map emptyStar

      -- Liste mit allen notwendigen Stern-DOM-Elementen erzeugen
      allRenderedStars =
          renderedFilledStars ++ renderedEmptyStars
    in
    div [ class "elm-rating__wrapper" ]
      [ h2 [] [ text model.title ]
      , div [ class "elm-rating__star-container" ] allRenderedStars
      ]
```

Listing 6: Nachrichtentyp für Benutzerinteraktion

```
type Msg
    = StarClicked Int
type alias StarCounts =
    { numberOfFilledStars : Int
    , numberOfEmptyStars : Int
    }

ratingToStarCounts : Maybe Rating -> StarCounts
ratingToStarCounts rating =
    case rating of
      Nothing -> StarCounts 0 5
      Just One -> StarCounts 1 4
      Just Two -> StarCounts 2 3
      Just Three -> StarCounts 3 2
      Just Four -> StarCounts 4 1
      Just Five -> StarCounts 5 0
```

Listing 7: Die notwendigen Styledefinitionen

```
.elm-rating__wrapper {
    background-color: lightgrey;
    width: 15rem;
    display: flex;
    flex-direction: column;
    align-items: center;
}

.elm-rating__star-container {
    display: flex;
    font-size: 3rem;
}
```

Listing 8: Aktualisierung des Models

```
numberToRating : Int -> Maybe Rating
numberToRating number =
    case number of
      1 -> Just One
      2 -> Just Two
      3 -> Just Three
      4 -> Just Four
      5 -> Just Five
      _ -> Nothing

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
      StarClicked clickedIndex ->
        let
          newRating =
            numberToRating clickedIndex
        in
        ( { model | rating = newRating }, Cmd.none )
```

Listing 9: Einbettung der Bibliothek in der index.js-Datei

```
import elmWebComponents from '@teamthread/elm-web-components'

elmWebComponents.configure('0.19');

elmWebComponents.register('elm-rating', Elm.Main, {
    setupPorts: ports => {
      ports.ratingChanged.subscribe(data => {
        alert(JSON.stringify(data));
      })
    },
    onSetupError: error => {
      alert('elmWebComponents -> Something went wrong', error)
    }
});
```

Die Verarbeitung der Klicknachricht findet in der Funktion `update` statt. Diese Funktion bekommt das aktuelle Datenmodell sowie die zu verarbeitende Nachricht übergeben (Listing 8). Rückgabe ist ein Tupel aus einer gegebenenfalls veränderten Kopie des Datenmodells sowie einem Command, dessen Datentyp kurz `Cmd` heißt. Commands signalisieren der Runtime, einen Seiteneffekt auszuführen. Diese löst danach eine Nachricht aus, die erneut ins Update eingespeist wird.

Damit ist die erste Version der Rating-Komponente abgeschlossen. Jede weitere Änderung kann jetzt auf die Unterstützung durch den Elm-Compiler zählen. Dessen hilfreiche Fehlermeldungen, die jeden Anpassungsprozess Schritt für Schritt begleiten, haben dazu geführt, dass die Elm-Community von Compiler-driven Development spricht.

Mit der Bibliothek elm-web-components lassen sich in Elm geschriebene Controls als Web Components in eine bestehende Webseite integrieren. Dazu sind im Projektordner zwei Befehle auszuführen: `npm init -y`, um eine package.json-Datei anzulegen, und

```
npm install --save-dev @teamthread/elm-web-components
```

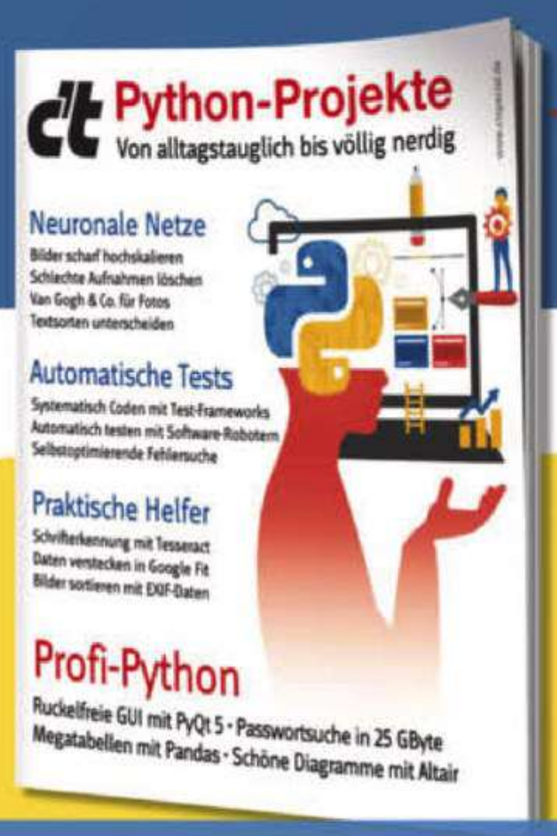
um die eigentliche Bibliothek zu installieren. Anschließend ist im Ordner src in der index.js-Datei der alte Code zur Initialisierung, `Elm.init(...)`, zu löschen und die Zeilen für die Konfiguration

Einfach loslegen mit Python

NEU

Auch Heft + PDF erhältlich mit 29% Rabatt





„Learning by Doing" ist das Motto der Neuauflage des unverzichtbaren c't Sonderheftes - vollständig überarbeitet und aktualisiert, darüber hinaus mit komplett neuen Beiträgen auf 150 Seiten.
Stürzen Sie sich in die Python-Projekte für alle Schwierigkeitsgrade – dabei lernen Sie ganz automatisch. Vom blutigen Anfänger bis zum Berufsprogrammierer findet jeder passende Projekte für sich, die ganz nebenbei auch konkrete Probleme lösen."

shop.heise.de/ct-python20

14,90 € ›

heise shop

shop.heise.de/ct-python20 ›

› Generell portofreie Lieferung für Heise Medien- oder Maker Media Zeitschriften-Abonnenten oder ab einem Einkaufswert von 15 €. Nur solange der Vorrat reicht. Preisänderungen vorbehalten.

des Plug-ins und das Registrieren der Komponente sind einzufügen (Listing 9).

Ports bilden die Schnittstelle zu JavaScript. Im konkreten Fall reagiert die Ausgabe der über den `ratingChanged`-Port empfangenen Daten auf Ereignisse.

Daten übersetzen

Im HTML lässt sich die Komponente jetzt bereits verwenden, auch wenn die Seite ohne entsprechende Anpassungen im Elm-Programm zunächst einmal einen Fehler wirft. In index.html im public-Ordner ist die Web Component drei Mal mit unterschiedlichen Daten einzubinden:

```
<elm-rating title="The best movie ever" rating="5"></elm-rating>
<elm-rating title="The worst movie ever" rating="1"></elm-rating>
<elm-rating title="The most unrated movie ever" rating=""></elm-rating>
```

Die angepasste Funktion `init` in Main.elm übersetzt das übergebene Datenpaket aus `title` und optionalem `rating` in eine Elm-taugliche Datenstruktur (Listing 10).

Die Funktion `andThen` für `Maybe` packt den übergebenen Wert aus `Maybe` aus, hier also den übergebenen Rating-String. Dann ist entweder die gewünschte Funktion `String.toInt` oder `numberToRating` auszuführen. Da beide Funktionen `Maybe` als Rückgabetyp haben, kann `andThen` die Rückgabe des Funktionsergebnisses unverändert übernehmen. Ist die Eingabe leer, wird die übergebene Funktion gar nicht erst ausgeführt, sondern stattdessen `Nothing` zurückgegeben.

Sind komplexere Daten zu verarbeiten, kommt in der Regel JSON als Übergabeformat zum Einsatz. JSON-Decoder und -Encoder übernehmen die Verarbeitung, für die hier übergebenen Daten sind Flags ausreichend.

Listing 10: Web-Components-Attribute entgegennehmen

```
type alias Flags =
    { title : String
    , rating : Maybe String
    }

init : Flags -> ( Model, Cmd Msg )
init flags =
    let
      parsedRating =
        let
          parsedString =
            Maybe.andThen String.toInt flags.rating
        in
        Maybe.andThen numberToRating parsedString
    in
    ( { title = flags.title
      , rating = parsedRating
      }
    , Cmd.none
    )
```

Listing 11: Änderungen nach außen geben

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        StarClicked clickedIndex ->
            let
                newRating =
                    numberToRating clickedIndex

                ratingAsString =
                    Maybe.map ratingToString newRating

                ratingChangedPortCmd =
                    ratingChanged (Flags model.title ratingAsString)
            in
            ( { model | rating = newRating }, ratingChangedPortCmd )
```

Listing 12: Alle Bausteine miteinander verknüpfen

```
main : Program Flags Model Msg
main =
    Browser.element
        { view = view
        , init = init
        , update = update
        , subscriptions = always Sub.none
        }
```

Eine Ergänzung in der ersten Zeile des Elm-Programms erlaubt den Einsatz von Ports:

```
port module Main exposing (..)
```

Der eigentliche Port ist dann schnell geschrieben:

```
port ratingChanged : Flags -> Cmd msg
```

Er lässt sich in der Funktion `update` bei jeder Änderung des Ratings wie eine Funktion aufrufen und gibt ein `Cmd` zurück (Listing 11). Die Übergabe an die Elm-Runtime im Rückgabe-Tupel der `update`-Funktion soll einen Seiteneffekt auslösen.

Die Hilfsfunktion `ratingToString` konvertiert die geänderten Daten in ein für JavaScript verständliches Format und ermöglicht so, dass diese auch wieder in Richtung JavaScript fließen können. Da sie keine Flags erwartet, muss man die Initialisierung des Programms im letzten Codeblock von Main.elm anpassen (Listing 12).

Jetzt steht der Einbindung der Elm-Applikation als Web Component nichts mehr im Weg. Die Daten fließen von der HTML-Seite in die Applikation hinein und über Ports zurück in JavaScript.

Fazit

Kommt man von Sprachen wie JavaScript oder TypeScript zu Elm, gibt es einige Anfangshürden zu überwinden: eine fest vorgegebene Programmarchitektur, die fremdartige Syntax und Konzepte der funktionalen Programmierung. Lässt man sich aber darauf ein, bietet Elm einen guten Einstieg ins funktionale Paradigma. Bei der Entwicklung von Webseiten und Webelementen lassen sich schnell erste Erfolge verzeichnen.

Mit der Option, ein Elm-Programm als Web Component in eine bestehende Seite zu integrieren, sind erste Schritte ohne großes Risiko möglich. Fortgeschrittene können eine Single-Page Application ausschließlich in Elm umsetzen oder größere JavaScript-Bibliotheken mithilfe von Ports und JSON-Decodern und -Encodern anbinden.

Dass die Beschäftigung mit Konzepten wie Custom Types, Pattern Matching oder monadischen Datentypen wie `Maybe` den Einstieg in umfangreichere funktionale Sprachen wie F# oder Haskell erleichtert, ist dabei ein angenehmer Nebeneffekt. (nb@ix.de)

Quellen

Alle Listings zum Download: ix.de/zbg3



Martin Grotz

ist Softwareentwickler bei Redheads und beschäftigt sich mit der Entwicklung moderner Webanwendungen und funktionaler Programmierung.

WIR MACHEN
KEINE WERBUNG.
WIR MACHEN EUCH
EIN ANGEBOT.

ct.de/angebot

Jetzt gleich bestellen:
ct.de/angebot
+49 541/80 009 120
leserservice@heise.de

ICH KAUF MIR DIE c't NICHT. ICH ABONNIER SIE.

Ich möchte c't 3 Monate lang mit 35 % Neukunden-Rabatt testen.
Ich lese 6 Ausgaben als Heft oder digital in der App, als PDF oder direkt im Browser.

**Als Willkommensgeschenk erhalte ich eine Prämie nach Wahl,
z. B. einen RC-Quadrocopter.**





Elixir: pragmatisch und leicht zu erlernen

Robuste Prozesse

Martin Grotz

Mit der funktionalen Programmiersprache Elixir lassen sich skalierbare und robuste Programme schreiben. Dank des Erlang-Unterbaus stehen erprobte Konzepte zur Verfügung, um auch komplizierte Laufzeitabhängigkeiten einfach auszudrücken.

Die funktionale Programmiersprache Elixir ist auf Nebenläufigkeit ausgerichtet. Sie ist dynamisch typisiert, verwendet eine an Ruby angelehnte Syntax und ist noch recht jung – Version 1.0 erschien im September 2014. Der Elixir-Compiler erzeugt Erlang-Bytecode. Es besteht eine hundertprozentige Kompatibilität: Erlang-Code lässt sich problemlos aus Elixir heraus nutzen und umgekehrt. Dies gilt auch für alle Sprachbibliotheken. Dadurch profitiert Elixir vom Erlang-Ökosystem und erbt durch die Erlang-Basis zugleich zahlreiche Laufzeitvorteile.

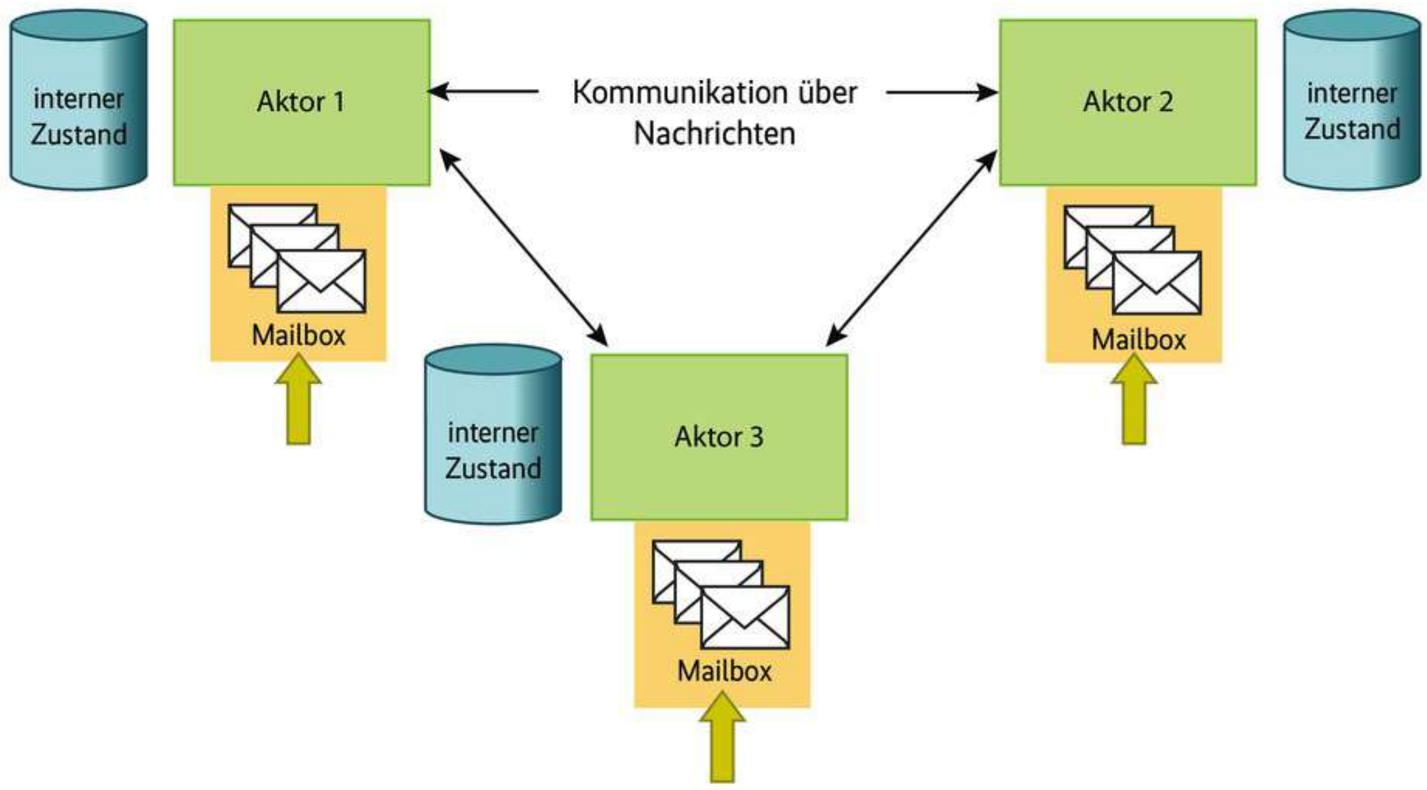
Die Basis jedes Elixir-Systems sind Aktoren. Das theoretische Aktormodell entstand allerdings unabhängig von der Implementierung in der Erlang-Runtime, deshalb unterscheiden sich viele Bezeichnungen leider in Theorie und Praxis. Aktoren sind jeweils vollständig voneinander unabhängige Laufzeitkomponenten, die sich keinerlei Zustand mit anderen Aktoren teilen. Die Kommunikation erfolgt nachrichtenbasiert (Abbildung 1).

Dazu besitzt jeder Aktor eine Eingangswarteschlange, um alle Nachrichten von anderen Aktoren für die spätere Bearbeitung zu sammeln. Die anschließende sequenzielle Verarbeitung der Nachrichten erleichtert die Programmierung, da im Code keine Parallelverarbeitung zu berücksichtigen ist. Innerhalb eines Aktors wird also immer nur genau eine Nachricht gleichzeitig verarbeitet. Für die Skalierung des Gesamtsystems muss das jeweilige Programm zusätzliche Aktoren starten. Da die Kommunikation nur über Nachrichten funktioniert, ist es nicht erforderlich, die Aktoren auf dem gleichen System auszuführen, damit sie miteinander kommunizieren können. Eine Verteilung über beliebig viele Rechnersysteme ist problemlos möglich.

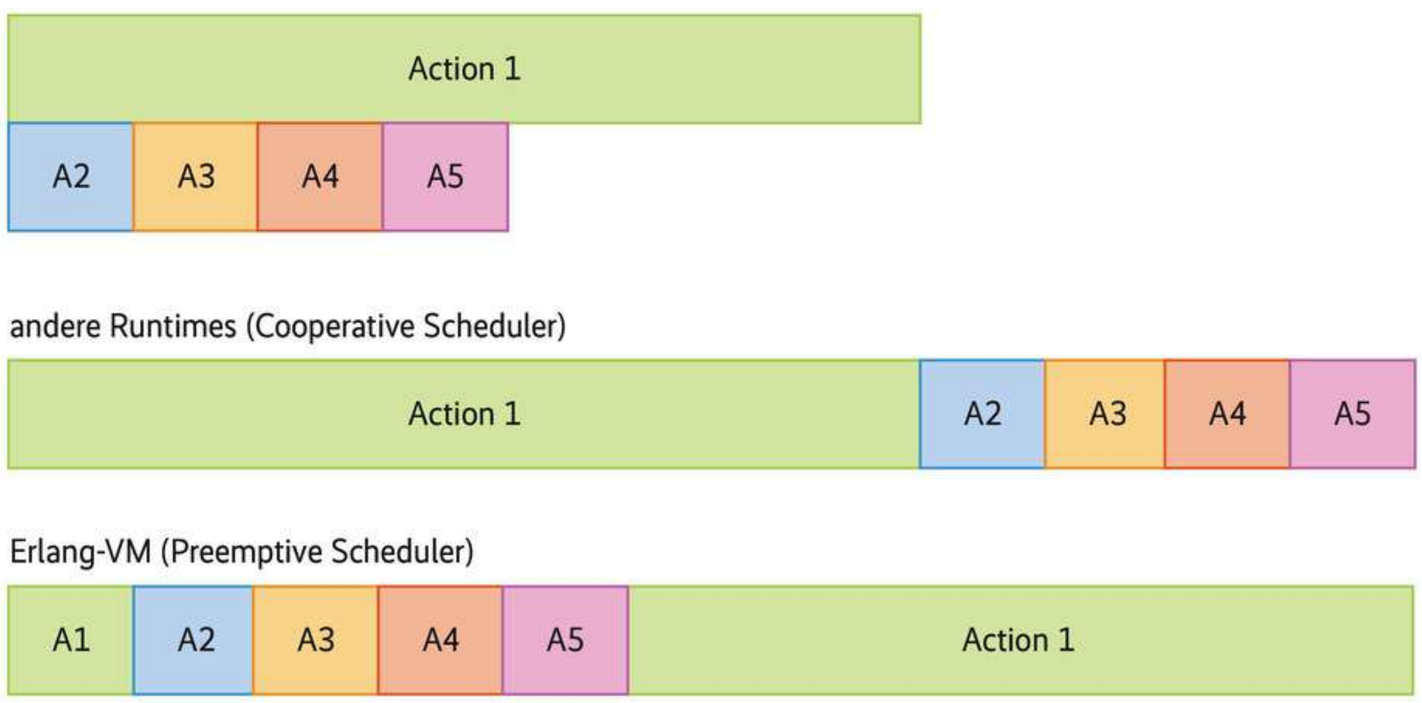
iX-TRACT

- Elixir ist eine auf Nebenläufigkeit optimierte moderne funktionale Programmiersprache, die zu Erlang-Bytecode kompiliert wird.
- Die Erlang-VM dient als Laufzeitumgebung mit geringer Latenz und hoher Zuverlässigkeit.
- Das Aktormodell ermöglicht zusammen mit Abstraktionen aus der Elixir-Standardbibliothek die einfache Orchestrierung komplexer Prozessabhängigkeiten.

Da der Compiler den Elixir-Programmcode in Erlang-Bytecode umwandelt, gibt es keine eigene Elixir-Laufzeitumgebung. Stattdessen kommt hier die über Jahrzehnte erprobte Erlang-VM BEAM zum Einsatz. Erlang bezeichnet Aktoren als Prozesse. Im Folgenden werden beide Begriffe austauschbar verwendet. Die Erlang-Prozesse sind nicht mit schwergewichtigen Betriebssystemprozessen zu verwechseln, deren Erzeugung teuer ist und die eine nicht unerhebliche Menge Arbeitsspeicher pro laufendem Prozess brauchen. Erlang-Prozesse benötigen wenig Arbeitsspeicher (circa 2 KByte) und lassen sich schnell erzeugen (in circa 10 Mikrosekunden). Daher gehen Erlang-Programme geradezu verschwenderisch mit Prozessen um. Hunderte oder Tausende gleichzeitig ausgeführte Prozesse sind selbst bei kleinen Systemen absolut üblich und bereiten keinerlei Probleme.



Im Aktormodell kommunizieren voneinander unabhängige Aktoren ausschließlich über Nachrichten (Abb. 1).



Lange Rechenoperationen blockieren die Erlang-VM nicht, da sie sich jederzeit unterbrechen lassen (Abb. 2).

Belastbar

Die VM verteilt die jeweiligen Prozesse automatisch auf Scheduler, um eine gleichmäßige Auslastung des Rechnersystems zu erreichen: Standardmäßig gibt es einen Scheduler pro logischem Prozessor. Da die Aktoren sich keinen Speicher teilen, kann eine fast beliebige Anzahl an Prozessoren ausgelastet werden – und das ohne zusätzliche Komplexität bei der Programmierung!

Eine weitere Besonderheit ist die Art des Schedulings innerhalb der Erlang-VM: Im Gegensatz zu den Laufzeitumgebungen der meisten anderen Programmiersprachen verwendet sie kein kooperatives Scheduling, das darauf angewiesen ist, dass gerade laufende Rechenoperationen regelmäßig freiwillig eine Unterbrechung anbieten. Stattdessen findet, ähnlich wie in modernen Betriebssystemen, ein präemptives Scheduling statt: Jeder Prozess bekommt eine Zeitscheibe zugewiesen. Ist diese Zeit abgelaufen, wird die Ausführung unterbrochen und der nächste Prozess ist an der Reihe. Dadurch blockiert ein in Erlang erstelltes System auch bei Überlast nie vollständig (Abbildung 2).

Erlang bietet fertige Grundbausteine zum Erstellen eines Systems an, die sich auch in Elixir nutzen lassen: OTP (Open Telecom Platform). Beim Namen dieser Bausteinsammlung zeigt sich der historische Ursprung Erlangs in der Telekommunikationsbranche, obwohl die erstellten Systeme längst alle Softwarebereiche erobert haben.

OTP erspart die händische Implementierung des Spawn-Receive-Send-Loops für jeden einzelnen Prozess. Stattdessen gibt es drei verschiedene Abstraktionen: Tasks, die die einmalige Ausführung einer Aufgabe in einem Prozess ermöglichen, Agents, die den Anwendungszustand in einem Prozess vorhalten und für andere Prozesse verfügbar machen, und `GenServer`, die dann zum Einsatz kommen, wenn Zustand und Logik in einem Prozess zu vereinen sind.

Ein weiterer wichtiger Baustein sind Supervisors. Diese verwalten den Lebenszyklus aller anderen Prozesse. Sie kümmern sich um das Starten und Stoppen, garantieren die richtige Reihenfolge bei logisch voneinander abhängigen Prozessen und starten abgestürzte Prozesse neu. Dabei bilden sie eine Baumstruktur, da ein Kindprozess eines Supervisors auch ein weiterer Supervisor sein kann. Ergänzend dazu existieren noch `DynamicSupervisors`, die mit einer variablen Menge an Kindprozessen umgehen können – beim einfachen Supervisor müssen Entwickler*innen die verwalteten Prozesse explizit bei der Programmierung angeben.

Nach der Installation von Elixir ist das erste Projekt schnell erzeugt: Die Eingabe von

```
mix new filesize-elixir-ix --app filesize --module Filesize --sup
```

erstellt ein leeres Projekt inklusive Supervisorstruktur. Der eigentliche Programmcode liegt dabei im Unterordner lib/filesize. Dort sind auch alle weiteren Dateien anzulegen. Als Entwicklungsumgebung hat sich VS Code mit der Erweiterung ElixirLS bewährt. Ziel ist eine Anwendung, die ausgehend von einem Ordner rekursiv das Dateisystem durchläuft und die Größe aller Dateien ermittelt und zum sortierten Abruf bereitstellt.

Datenhaltung

Den Anfang macht ein Prozess, der sich die bereits gesammelten Ergebnisse merkt. Da außer dem Vorhalten des Zustands keine weitere Funktionalität nötig ist, kommt ein Agent zum Einsatz (Listing 1; alle Listings siehe ix.de/z7d5). Zu Beginn des Moduls wird über das Makro `use` eine Standardimplementierung eines Agents eingefügt. Makros dienen in Elixir zur Erweiterung

des Sprachumfangs. OTP und zahlreiche Bibliotheken machen sich das zunutze, um die Menge an sich wiederholendem Code zu reduzieren. Die Modulkonstante `@me`, über das Makro `__MODULE__` zur Kompilierzeit mit dem Modulnamen belegt, dient zur eindeutigen Benennung des Agents.

In Agents ist nur eine Funktion nötig, um den Prozess später durch einen Supervisor überwachen zu können: `start_link`. Diese delegiert die eigentliche Arbeit an die Standard-OTP-Agent-Implementierung. Einziges Argument ist eine Funktion, die den Anfangszustand festlegt – eine durch [] angezeigte leere Liste.

Die anderen beiden Funktionen bilden die gewünschte Schnittstelle nach außen ab: `add` fügt ein neues Ergebnistupel vorne an die bestehende Ergebnisliste an, `get_all_sorted` hingegen erledigt die sortierte Rückgabe aller bisherigen Ergebnisse. Für Lambdafunktionen gibt es die Kurzschreibweise `& (meine_funktion(&1, &2))`, die gleichbedeutend ist mit

```
fn arg1, arg2 -> meine_funktion(arg1, arg2) end
```

Das &, gefolgt von einer benannten Funktion inklusive Stelligkeit, referenziert hingegen die entsprechende Funktion. Funktionen lassen sich durch ihren Namen und die Stelligkeit, also die Anzahl der Parameter, eindeutig festlegen.

Der zweite Baustein der Datenhaltung ist eine Art Warteschlange für die noch zu durchsuchenden Ordner. `GenServer` ermöglicht Funktionalität über die reine Zustandsverwaltung hinaus. `GenServer` benötigen für eine `Supervisor`-Überwachung zwei Funktionen: `start_link` und `init`, wobei `start_link` die `init`-Funktion implizit aufruft (Listing 2).

`GenServer`-Implementierungen teilen sich in zwei Bereiche: erstens direkt aufrufbare Funktionen (Client), die andere Module nutzen und im Kontext des aufrufenden Prozesses ausgeführt werden. Diese Funktionen schicken Nachrichten an den eigentlichen `GenServer`-Prozess. Zweitens Callback-Funktionen, die die empfangenen Nachrichten innerhalb des `GenServer` bearbeiten (Server).

```
handle_call(payload, from, current_state)
```

behandelt dabei eine vorher von `GenServer.call(to, payload)` erzeugte Nachricht. Calls sind synchron, der sendende Prozess wartet auf die Antwort und blockiert so lange. `handle_call` endet immer mit einem Antworttupel, das ein Statusatom (`:reply`), direkte Antwortdaten und den neuen inneren Zustand des Prozesses enthält. Der letzte Ausdruck in einer Funktion ist gleichzeitig der Rückgabewert.

Musterbasierte Suche

Das aus vielen anderen funktionalen Programmiersprachen bekannte Pattern Matching, quasi ein deutlich mächtigeres `switch-case`-Statement, gibt es auch in Elixir. In Funktionen leitet das Schlüsselwort `case` ein Pattern Matching ein. Die Auswertung der Muster erfolgt von oben nach unten: Das Programm durchläuft den Codepfad des ersten zutreffenden Musters. Das erste Muster im Beispiel erkennt eine leere Liste. Der Rückgabewert `nil` im Codepfad dieses Musters bedeutet, dass kein Ordner mehr zu bearbeiten ist. Enthält die Liste mindestens einen Eintrag, trifft das zweite Muster zu. Dann ist die Antwort der erste Eintrag der Liste und der neue innere Zustand des Prozesses die entsprechend um den Kopf gekürzte Ordnerliste.

Pattern Matching kommt außerdem im Funktionskopf zum Einsatz: Jedes implementierte `handle_call` bestimmt über Pattern Matching, für welche Nachrichten es zuständig ist. Die Laufzeitumgebung durchsucht beim Aufruf die Funktionen in der Reihenfolge, in der sie im Quellcode stehen, und führt die erste Funktion mit passendem Muster aus. Achtung: Passt keine

Listing 1: Für die Sammlung der Ergebnisse ohne weitere Funktionalität reicht ein Agent

```
# Datei: lib/filesize/results.ex
defmodule Filesize.Results do
    use Agent
    @me __MODULE__

    def start_link(_init_arg), do: Agent.start_link(fn -> [] end, name: @me)

    def add(new_result), do: Agent.update(@me, fn existing_results -> [new_result | ,
                                                                       existing_results] end)


    def get_all_sorted() do
      Agent.get(@me, fn existing_results ->
        Enum.sort_by(
          existing_results,
          &elem(&1, 0), # erstes Element aus Tupel dient als Sortierschlüssel
          &>=/2 # Vergleichsfunktion für absteigende Sortierung
        )
      end)
    end
end
```

Listing 2: Der GenServer verwaltet die Warteschlange noch zu bearbeitender Ordner

```
# Datei: lib/filesize/queue.ex
defmodule Filesize.Queue do
    use GenServer
    @me __MODULE__

    def start_link(init_arg), do: GenServer.start_link(__MODULE__, init_arg, name: @me)
    def init(_init_arg), do: {:ok, []}

    # CLIENT
    def add_directory(directory) do
      GenServer.call(@me, {:add, directory})
      Filesize.WorkerManager.start_workers()
    end

    def get_next_directory(), do: GenServer.call(@me, {:get_next_directory})
    def directory_done(directory), do: GenServer.call(@me, {:directory_done, directory})

# SERVER
    def handle_call({:add, directory}, _from, state) do
      {:reply, nil, [directory | state]}
    end

    def handle_call({:get_next_directory}, _from, state) do
      case state do
        [] -> {:reply, nil, state}
        [next_dir | rest] -> {:reply, next_dir, rest}
      end
    end

    def handle_call({:directory_done, directory}, _from, state) do
      {:reply, :ok, List.delete(state, directory)}
    end
end
```

der Funktionen, bleibt die Nachricht unbearbeitet in der Prozesswarteschlange hängen. In größeren Systemen gibt es daher zumeist eine letzte `handle_call`-Implementierung, die alle unbearbeiteten Nachrichten sammelt und deren Empfang protokolliert. Dies verhindert, dass unbearbeitete Nachrichten die Eingangswarteschlange überfluten, und ermöglicht eine spätere Fehlersuche.

In jedem System muss es jemanden geben, der die eigentliche Arbeit erledigt. Dieser `Worker` hat allerdings keine Schnittstelle nach außen – stattdessen holt er sich selbst von der Queue den nächsten Ordner ab und durchsucht ihn nach neuen Ordnern. Diese meldet der `Worker` an die Queue zurück. Bei gefundenen Dateien ermittelt er die Größe und gibt sie an den Results-Prozess durch (Listing 3). Der `Worker` gibt außerdem Meldung an einen noch zu implementierenden `WorkerManager` über einen erfolgten Start beziehungsweise über das Prozessende, damit dieser die Anzahl der gerade aktiven Arbeitsprozesse im Blick behalten kann.

Durch `:transient` erfolgt ein Neustart des Prozesses nur, wenn dieser unerwartet endet. Bei regulärem Ende – wenn keine weiteren Ordner mehr in der Warteschlange sind – gibt es keinen automatischen Wiederanlauf. Andere mögliche Verhalten wären hier `:temporary` für „niemals neustarten“ und der Standardwert `:permanent` für einen Neustart, ganz gleich wie der Prozess endet.

Der Pipelineoperator `|>` in der Funktion `traverse` übergibt den ausgewerteten Ausdruck als erstes Argument an die nachfolgende Funktion. Viele Funktionen liegen in zwei Varianten vor: Bei der ersten endet der Funktionsname in einem Ausrufezeichen. Dann wirft die Funktion bei einem Problem einen Ausnahmefehler. Die gleiche Funktion ohne Ausrufezeichen gibt hingegen ein Tupel zurück: entweder `{:ok, data}` oder im Fehlerfall `{:error, error_message}`. Das ist bei der Verkettung von Funktionen oder zur manuellen Fehlerbehandlung praktisch. Außerdem kommen Funktionen vor, deren Namen mit einem Fragezeichen enden. Diese Prüffunktionen geben immer einen booleschen Wert zurück.

Ein `WorkerManager` startet die gewünschte Anzahl neuer `Worker`-Prozesse (Listing 4). Der `GenServer` funktioniert nach dem gleichen Muster wie die anderen: Lifecycle-Funktionen kümmern sich um den Start, Clientfunktionen bieten eine einfache Schnittstelle

Listing 3: Entbehrliche Worker-Prozesse erledigen die eigentliche Arbeit im Dateisystem

```
# Datei: lib/filesize/worker.ex
defmodule Filesize.Worker do
    use GenServer, restart: :transient

    def start_link(init_arg), do: GenServer.start_link(__MODULE__, init_arg)

    def init(_init_arg) do
      Filesize.WorkerManager.started()
      Process.send_after(self(), :traverse_one_directory, 0)
      {:ok, nil}
    end

    def terminate(_reason, _state), do: Filesize.WorkerManager.done(self())

    def handle_info(:traverse_one_directory, _from) do
      work_dir = Filesize.Queue.get_next_directory()
      traverse(work_dir)
      Filesize.Queue.directory_done(work_dir)

      {:noreply, nil}
    end

    # kein Ordner mehr zu bearbeiten -> Worker ist fertig und lässt sich beenden
    defp traverse(nil), do: Filesize.WorkerManager.done(self())

    defp traverse(directory) do
      File.ls!(directory)
      |> Enum.map(fn path ->
        absolute_path = Path.join(directory, path)

        if File.dir?(absolute_path) do
          Filesize.Queue.add_directory(absolute_path)
        else
          file_info = File.stat!(absolute_path)
          Filesize.Results.add({file_info.size, absolute_path})
        end
      end)

      # nächsten Ordner aus der Warteschlange holen
      send(self(), :traverse_one_directory)
    end
end
```


NEU: c't DOCKER & CO – Container leicht gemacht

NEU

c't DOCKER & CO 2020

Die Arbeit mit **Kubernetes, Docker & Co.** hält auch für erfahrene Programmierer einige typische Fallen bereit. Das Sonderheft **c't wissen DOCKER & CO** enttarnt mögliche Stolpersteine und zeigt Möglichkeiten, diesen auszuweichen. **Profis und Einsteiger** im Container-Kosmos finden Tipps und Hintergrundinfos, die selbst in offiziellen Dokumentationen fehlen.

Auch digital für 12,99 € erhältlich!

shop.heise.de/docker-co20

14,90 € ›

NEU-AUFLAGE 2020



› Generell portofreie Lieferung für Heise Medien- oder Maker Media Zeitschriften-Abonnenten oder ab einem Einkaufswert von 15 €. Nur solange der Vorrat reicht. Preisänderungen vorbehalten.

heise shop

shop.heise.de/docker-co20 ›

Listing 4: WorkerManager ist für die richtige Anzahl an aktiven Worker-Prozessen verantwortlich

```
# Datei: lib/filesize/worker_manager.ex
defmodule Filesize.WorkerManager do
    use GenServer
    @me __MODULE__

    def start_link(max_workers), do: GenServer.start_link(__MODULE__, max_workers, name: @me)
    def init(max_workers), do: {:ok, %{max_workers: max_workers, current_workers: 0}}

    def start_workers(), do: GenServer.call(@me, :start_workers)
    def done(pid), do: GenServer.call(@me, {:stop_worker, pid})
    def started(), do: GenServer.cast(@me, :worker_started)

    def handle_cast(:worker_started, state) do
      {:noreply, %{state | current_workers: state.current_workers + 1}}
    end

    def handle_call(:start_workers, _from, state) do
      if state.current_workers < state.max_workers do
        state.current_workers..(state.max_workers - 1)
        |> Enum.each(fn _ -> Filesize.WorkerSupervisor.add_worker() end)
      end

      {:reply, nil, state}
    end

    def handle_call({:stop_worker, pid}, _from, state) do
      Filesize.WorkerSupervisor.stop_worker(pid)
      {:reply, nil, %{state | current_workers: state.current_workers - 1}}
    end
end
```

Listing 5: StateSupervisor propagiert den Fehlerfall sofort nach oben

```
# Datei: lib/filesize/state_supervisor.ex
defmodule Filesize.StateSupervisor do
    use Supervisor
    def start_link(init_arg), do: Supervisor.start_link(__MODULE__, init_arg)

    def init(_init_arg) do
      children = [Filesize.Results, Filesize.Queue]
      Supervisor.init(children, strategy: :one_for_all, max_restarts: 0)
    end
end
```

nach außen und Message Handler verarbeiten die tatsächlich an den benannten Prozess geschickten Nachrichten. Einzige Besonderheit: Der `Worker` soll sofort loslegen können und nicht erst auf eine Antwort des `WorkerManager` warten müssen. Daher erfolgt die Rückmeldung bei Worker-Start über die Funktion `started` nicht als synchroner Call, sondern als Fire-and-Forget-Cast. Im Gegensatz zum Call blockiert der Cast den sendenden Prozess nicht bis zum Eintreffen einer Antwort.

Damit sind alle Grundbausteine der Applikation vorhanden. Ein spannender Teil fehlt allerdings noch: der Supervision Tree. Er ersetzt die explizite Fehlerbehandlung im Programm. Anstatt zu versuchen, jede mögliche Fehlerquelle im Voraus zu erkennen, setzt Erlang – und damit auch Elixir – auf das Motto „Let it crash“: der betroffene Prozess stürzt ab, die Fehlermeldung wird gegebenenfalls protokolliert und der Prozess in einem definierten Zustand neu gestartet.

Überwachung

Das Design des Supervision Tree richtet sich nach den Abhängigkeiten der einzelnen Prozesse voneinander (Abbildung 3). Im vorliegenden Fall soll die gesamte Ausführung sofort enden, wenn entweder die Queue mit den noch zu bearbeitenden Ordnern oder aber der Results-Agent mit den bisherigen Ergebnissen abstürzt, da ein Weiterarbeiten mit unvollständigen Ergebnissen sinnlos wäre. Ein Absturz von `WorkerManager` beeinflusst Warteschlange und Ergebnisse nicht, muss aber die gerade laufenden Worker-Prozesse beenden. Denn sonst gerät die Zählung der aktiven Arbeiter aus dem Tritt, da der innere Zustand des Prozesses bei einem Neustart verloren geht. Gibt es dagegen nur Probleme innerhalb eines `Workers`, reicht ein Neustart genau dieses Aktors aus.

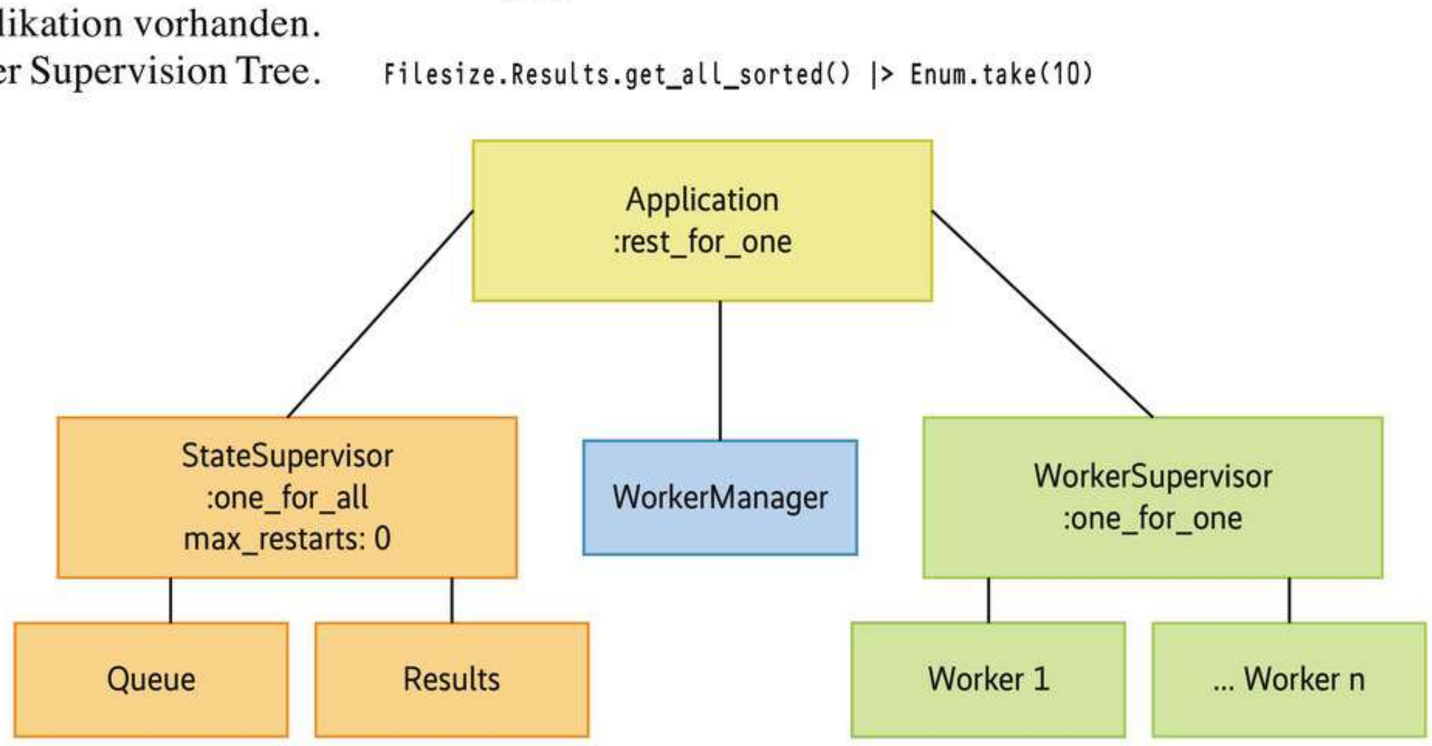
Die erste Anforderung, also den Umgang mit Problemen in Queue und Results, erledigt ein Supervisor-Tandem: `StateSupervisor` bewirkt mit der Strategie `:one_for_all` und einem Restart-Limit von 0, dass ein Crash von Queue oder Results sofort den anderen der beiden Prozesse mit beendet und das Beenden eine Ebene höher meldet (Listing 5).

Den zweiten Teil des Tandems bildet der Application-Supervisor, der immer ganz oben in jedem Elixir-System angesiedelt ist (Listing 6). Darin startet die Strategie `:rest_for_one` alle nach einem abgestürzten Prozess in der Liste `children` verzeichneten Kindprozesse neu. Die zweite Anforderung bei Problemen mit `WorkerManager` ist damit erfüllt.

Zuletzt gibt es noch einen minimalistischen `DynamicSupervisor`, der die Verwaltung der `Worker` übernimmt. Er startet und stoppt Kindprozesse auf Kommando. Im Gegensatz zum normalen Supervisor ist keine abgeschlossene Liste von Kindern anzugeben (Listing 7).

Endlich ist alles bereit für einen Testlauf: Die Eingabe von `iex -S mix` in der Konsole im Projektordner startet erst den Kompiliervorgang und anschließend das Programm in der Interactive-Elixir-Umgebung `iex`. Der Befehl `Filesize.Queue.add_directory(".")` wirft den implementierten Mechanismus an und ein nach kurzer Zeit eingegebenes

```
Filesize.Results.get_all_sorted() |> Enum.take(10)
```



Der Supervision Tree bildet das gewünschte Laufzeitverhalten im Fehlerfall ab: Weiter rechts stehende Äste hängen hier von denen weiter links ab (Abb. 3).

Listing 6: Die Gesamtanwendung verbindet alle Äste miteinander

```
# Datei: lib/filesize/application.ex
defmodule Filesize.Application do
    use Application

    def start(_type, _args) do
      children = [
        Filesize.StateSupervisor,
        {Filesize.WorkerManager, 4},
        Filesize.WorkerSupervisor
      ]

      opts = [strategy: :rest_for_one, name: Filesize.Supervisor]
      Supervisor.start_link(children, opts)
    end
end
```

Listing 7: Die Worker sind austauschbar

```
# Datei: lib/filesize/worker_supervisor.ex
defmodule Filesize.WorkerSupervisor do
    use DynamicSupervisor
    @me __MODULE__

    def start_link(init_arg), do: DynamicSupervisor.start_link ↵
                                    (__MODULE__, init_arg, name: @me)
    def init(_init_arg), do: DynamicSupervisor.init(strategy: ↵
                                                        :one_for_one)

    def add_worker(), do: {:ok, _pid} = DynamicSupervisor.start_ ↵
                                            child(@me, Filesize.Worker)
    def stop_worker(pid), do: DynamicSupervisor.terminate_ ↵
                                                    child(@me, pid)
end
```

ruft die ersten zehn Ergebnisse ab. Spannend ist auch ein Blick in die grafische Prozessüberwachung über die Eingabe von `:observer.start()`. Der Karteireiter „Applications“ enthält eine Übersicht aller aktiven Prozesse.

Fazit

Elixir macht es als pragmatische funktionale Programmiersprache mit leicht erlernbarer Syntax einfach, Anwendungen zu erstellen. Der Erlang-Unterbau sorgt für ein stabiles und skalierbares Laufzeitsystem. OTP als Baukasten stellt alles Notwendige bereit, um die gewünschte Funktionalität schnell in Code abzubilden. Supervisorbäume sorgen für Stabilität, fördern das Nachdenken über das Laufzeitverhalten des Systems und lenken den Blick auch auf Ereignisse abseits des „Happy Path“. Das Aktormodell erlaubt eine Entkoppelung der einzelnen Programmteile und erleichtert durch die unabhängigen Prozesse die Programmierung von Nebenläufigkeit.

Durch eine aktive und freundliche Community ist in den wenigen Jahren seit Version 1.0 ein großes Ökosystem um die Sprache herum entstanden. Im Gegensatz zur behutsam weiterentwickelten Programmiersprache entwickeln sich die Bibliotheken in hoher Geschwindigkeit in viele Richtungen weiter. Projekte wie Nerves (Elixir für IoT), Lumen (Elixir via WebAssembly im Browser) oder LiveView (Elixir statt JavaScript für dynamische Webseiten) erweitern die Einsatzmöglichkeiten der Sprache weit über reine Serveranwendungen hinaus und machen Elixir zu einer guten Wahl in verschiedensten Einsatzbereichen. (nb@ix.de)

Quellen

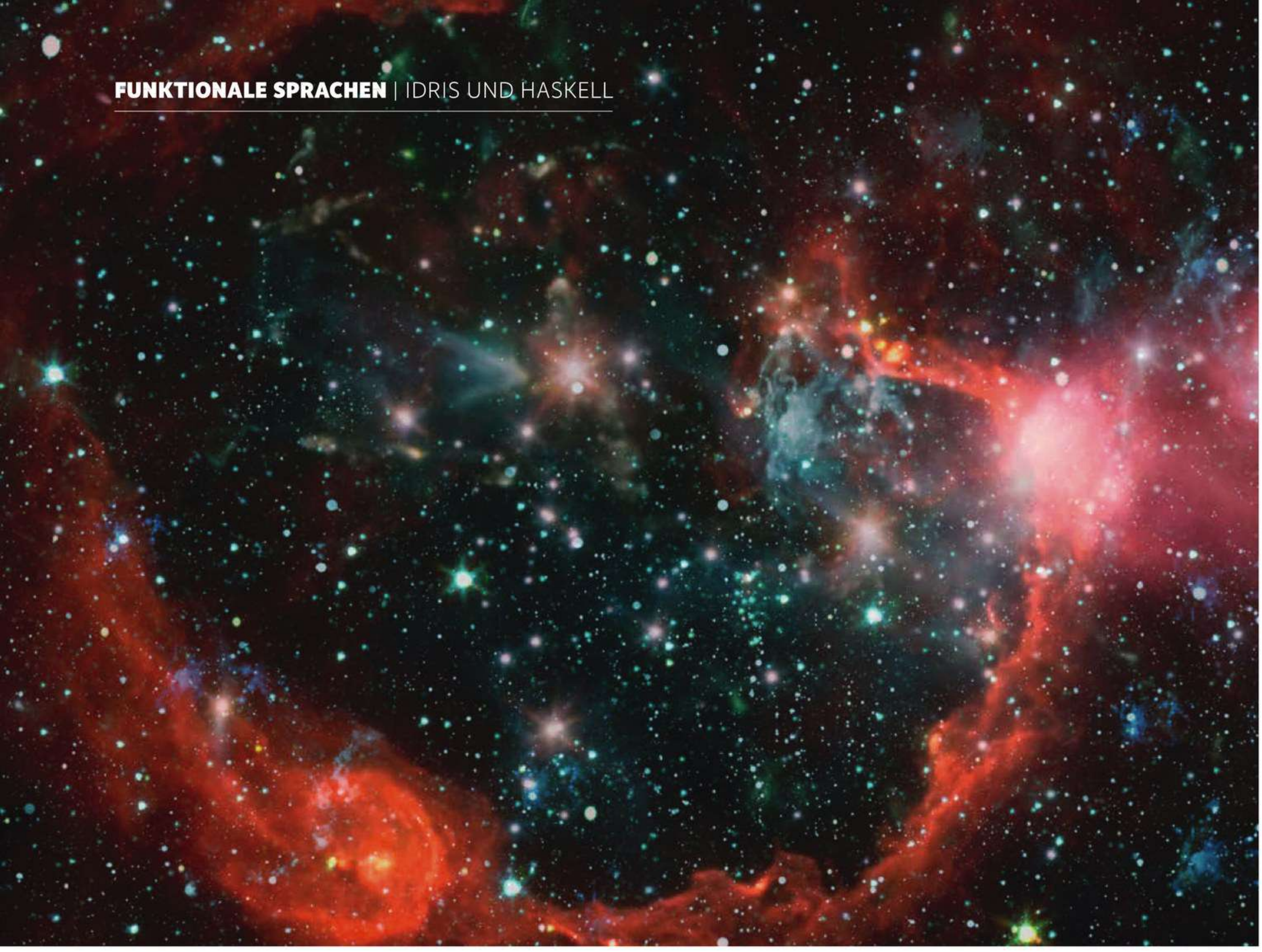
Alle Listings zum Download: ix.de/z7d5



Martin Grotz

ist Softwareentwickler bei Redheads und beschäftigt sich mit der Entwicklung moderner Webanwendungen und funktionaler Programmierung.





Funktionale Programmierung mit Haskell und Idris

Effekte kontrollieren

Michael Sperber

Der funktionalen Programmiercommunity hängt vielerorts der Ruf an, esoterischen Code zu schreiben, der wenig bis gar nichts mit der „Realität“ zu tun hat. Das ist insbesondere in der funktionalen Programmiersprache Haskell möglich, und der noch junge Abkömmling Idris setzt bei der Esoterik noch eins drauf.

Dieser Artikel ist eine kleine Reise durch einige ausgewählte Elemente der funktionalen Sprachen Haskell und Idris, die das Programmieren mit „Effekten“ in den Griff bekommen – also den Aspekt der Programmierung, der in der Praxis am meisten Probleme macht. Vorkenntnisse in Haskell oder Idris sind für die Lektüre zwar hilfreich, aber nicht erforderlich; ein wenig Geduld und Fantasie sind allerdings notwendig.

Effekte, was soll das überhaupt sein?

Der Begriff des „Seiteneffekts“ ist allgemein bekannt: Die Auswertung eines Ausdrucks hat in vielen Programmiersprachen nicht nur ein Ergebnis, sondern „verändert die Welt“, also den Zustand des Programms, und zwar durch eine Zuweisung an eine Variable oder ein Objektattribut:

```
x = 5;
...
x = 10;
```

Funktionale Programmierer*innen meiden Anweisungen mit Seiteneffekten, denn sie führen zu versteckter Kopplung im Programm: Zwischen den Zuweisungen mit jeweils 5 und 10 wird die Variable x wahrscheinlich ausgelesen. Und wenn das zufällig zum falschen Zeitpunkt geschieht, kann der Wert falsch sein. Unzählige Bugs haben als Ursache, dass irgendwie der Zustand falsch ist, weil er unbewusst oder ungewollt verändert wurde. Aber es gibt noch andere Effekte als Zuweisungen. In Java gibt

Listing 1: Beispieltiere mit Gleichungen benennen

```
a1 = Dillo Alive 10 -- lebendiges Gürteltier, 10kg
a2 = Parrot "Hello!" 2 -- Papagei mit Begrüßung, 2kg
a3 = Dillo Dead 12 -- totes Gürteltier, 12kg
a4 = Parrot "Goodbye!" 1 -- Papagei mit Verabschiedung, 1kg
```

Listing 2: Folge von Zustandstransformationen

```
feedAndDie d =
    let d2 = feed 2 d
        d3 = feed 3 d2
        d4 = runOver d3
    in d4
```

es den `throw`-Befehl, der eine Exception signalisiert, die aktuelle Ausführung abbricht und einen Exception Handler ansteuert. Er verändert zwar nicht den Zustand, auf den das Programm direkt zugreifen kann, aber den Kontrollfluss – Exceptions sind mithin ein „Kontrolleffekt". Für Programmierer*innen, die versuchen nachzuvollziehen, wie ein Programm funktioniert, ist mitunter verwirrend, dass bei einem Aufruf wie dem folgenden einfach nichts herauskommt:

```
x = object.method(...)
```

Stattdessen wird während der Ausführung von `object.method` eine Exception ausgeführt, mit der Folge, dass dann aber eine andere Stelle des Programms aktiv ist – und die Zuweisung an x wird nie passieren. Weil das so überraschend sein kann, sind bestimmte sogenannte „Checked" Exceptions in Java Teil des Typsystems:

```
class Class {
    T method (...) throws RemoteException {
      ...
    }
}
```

Eine `throws`-Annotation ist nützlich, weil sie etwas darüber verrät, welche Änderungen `method` vornimmt, außer ein Ergebnis vom Typ `T` zu berechnen. In dem Fall verdeutlicht sie, dass `method` möglicherweise mit einem anderen Rechner kommuniziert.

Leider haben `throws`-Annotationen auch Nachteile: Wenn beispielsweise eine annotierte Methode `Stream.map` aus der Java-Standardbibliothek aufrufen will, gibt es die Fehlermeldung „Unhandled Exception". Letztere lässt sich nur durch umständliche Maßnahmen vermeiden, zum Beispiel durch Verwenden von „Unchecked Exceptions" und damit den Verzicht auf die `throws`-Annotation. Dementsprechend gibt es viele Effekte, die in keiner Typsignatur auftauchen. Sie können Entwickler*innen zum

iX-TRACT

- Funktionale Entwickler*innen meiden Anweisungen mit Seiteneffekten. Sie dennoch umzusetzen, bedeutet für sie, ganz andere Mittel zu nutzen, als in objektorientierten Sprachen zur Verfügung stehen.
- Für den produktiven Einsatz ist die Programmiersprache Idris noch nicht geeignet, sie setzt aber richtungsweisende Ideen für den Umgang mit Typen um.
- Über Erweiterungen ist auch Haskell in der Lage, ähnliche Effektsysteme wie das in Idris zu implementieren.

Listing 3: Vorrat beim Füttern mitführen

```
type Stash = Weight
feedFromStash :: Weight -> Animal Weight -> Stash -> (Animal
                                                      Weight, Stash)
feedFromStash amount animal stash =
    if amount <= stash
    then (feed amount animal, stash - amount)
    else (feed stash animal, 0)
```

Listing 4: Futter aus Vorrat entfernen

```
removeFromStash :: Weight -> Stash -> (Weight, Stash)
removeFromStash amount stash =
   let realAmount = min amount stash
   in (realAmount, stash - realAmount)
```

Fluchen bringen, wenn ein unschuldiges Stück Code plötzlich auf eine Datenbank zugreifen möchte, die nötige Verbindung dafür aber noch nicht initialisiert ist oder der Webserver auf dem Desktop der Servermaschine einen Dialog anzeigt.

Haskell-Programmierer*innen wollen so nicht arbeiten: Sie wollen, dass „alle" Effekte in den Typsignaturen ablesbar sind, und außerdem, dass die Programmiersprache beim Einsatz von Effekten einem nicht ständig Steine in den Weg legt. Beides konsequent durchzuziehen, erfordert ganz andere Mittel, als sie in objektorientierten Sprachen zur Verfügung stehen.

Man könnte meinen, dass dadurch primär die Programmierung eingeschränkt wird – beliebige Effekte sind ja nicht mehr so einfach erlaubt. Die Einschränkung bei den Effekten bietet aber auch neue Freiheiten: Code mit wenig Effekten ist universeller einsetzbar, weil er nicht mit anderen Aspekten des Programms interferiert, einfacher parallelisierbar ist und vor allem leichter zu verstehen.

Ein bisschen Haskell

Zur Einführung hier ein Beispielklassiker für Haskell – es geht um Tiere auf dem texanischen Highway, die wahlweise überfahren oder gefüttert werden. Es gibt Gürteltiere („`dillo`", kurz für „armadillo") und Papageien („`parrot`"):

```
data Animal =
    Dillo Liveness Weight
    | Parrot String Weight
```

Die Definition (ein sogenannter algebraischer Datentyp) entspricht einem Datentyp `Animal` mit zwei Klassen namens `Dillo` und `Parrot`. Der senkrechte Strich `|` kann als „oder" gelesen werden: Ein `Animal` ist ein `Dillo` oder ein `Parrot`. Ersteres hat zwei Attribute der Typen `Liveness` und `Weight`, Letzteres einen Text und ein `Weight`-Attribut. `Liveness` und `Weight` sind folgendermaßen definiert:

```
data Liveness = Alive | Dead
type Weight = Int
```

Der Einfachheit halber ist `Weight` ein Typsynonym für `Int`. In Listing 1 gibt es ein paar Beispieltiere, die auch mit Gleichungen Namen bekommen (alle Listings siehe ix.de/zfzh).

Zur Simulation, wie ein Tier überfahren wird, dient eine Funktion `runOver`. Vor die eigentliche Funktion kommt eine Typsignatur, die beschreibt, dass die Funktion ein `Animal`-Objekt als Eingabe akzeptiert und wieder eines zurückliefert:

```
runOver :: Animal -> Animal
```

Anders als ein typisches Java-Programm für denselben Zweck verändert die Funktion nicht einfach zum Beispiel das `Liveness`-Attribut der Gürteltiere. Stattdessen ist die Eingabe ein Objekt, das den Zustand vor dem Überfahren repräsentiert, heraus kommt ein neues Objekt für den Zustand danach. Das alte – sozusagen die Erinnerung an das niedliche Tier vor dem Überfahren – bleibt unverändert.

Definiert wird die Funktion über zwei Gleichungen, die mit der Form der `Dillo`- beziehungsweise `Parrot`-Objekte definiert sind:

```
runOver (Dillo liveness weight) = Dillo Dead weight
runOver (Parrot sentence weight) = Parrot "" weight
```

Das heißt, aus einem Gürteltier mit „Lebendigkeit“ (`liveness`) und Gewicht (`weight`) wird eines, das tot ist und als Gewicht wieder `weight` hat. Analog geht das für den Papagei. (Dass Haskell bei Funktionsaufrufen ohne Klammern auskommt, ist anfänglich etwas verwirrend, hat aber einen guten Grund.)

Eine Funktion für das Füttern von Tieren geht ähnlich:

```
feed :: Weight -> Animal -> Animal
feed amount (Dillo liveness weight) = Dillo liveness (weight + amount)
feed amount (Parrot sentence weight) = Parrot sentence (weight + amount)
```

Die Typsignatur sieht mit den zwei Pfeilen etwas merkwürdig aus. Zunächst muss man wissen, dass der Pfeil von rechts geklammert wird, da steht also äquivalent:

```
feed :: Weight -> (Animal -> Animal)
```

Obwohl `feed` konzeptuell zwei Eingaben hat, wird sie in Haskell realisiert als sogenannte „currifizierte“ (nach dem Logiker Haskell B. Curry) oder „geschönfinkelte“ Funktion (nach Moses Schönfinkel, der die Idee noch früher, aber leider einen Umlaut im Namen hatte). Das heißt, `feed` akzeptiert zunächst die erste Eingabe – das Gewicht des Futters –, um dann eine Funktion zurückzugeben, die aus dem einen `Animal`-Objekt ein anderes macht. Die geklammerte Schreibweise hat den Vorteil, dass man sieht, dass `feed` eine Funktion liefert mit der gleichen Signatur wie `runOver` – nämlich eine Funktion, die „Zustand transformiert“.

Um `feed` aufzurufen, reiht man die Argumente aneinander: `feed 2 a1`. `(feed 2) a1` wird geklammert, was anzeigt, dass es sich nicht wirklich um einen Funktionsaufruf mit zwei Argumenten handelt, sondern die Funktion nur auf das erste Argument und anschließend das Ergebnis auf das zweite Argument angewendet wird. Ein Typsynonym gibt der Idee der Zustandstransformation einen Namen:

```
type AnimalTransformer = Animal -> Animal
```

Die beiden Typsignaturen lassen sich nun refaktorisieren, um die Idee zu verdeutlichen:

```
runOver :: AnimalTransformer
feed :: Weight -> AnimalTransformer
```

Listing 5: Futter aus Vorrat entnehmen

```
removeFromStashST :: Weight -> Stash ->(Weight, Stash)
removeFromStashST amount =
      let realAmount = min amount stash
      in (realAmount, stash - realAmount))

removeFromStashST :: Weight -> ST Weight
removeFromStashST amount =
ST (\ stash ->
      let realAmount = min amount stash
      in (realAmount, stash - realAmount))
```

`Animal` ist nicht der einzig mögliche Typ für einen Zustand. Genauso könnte es `AccountTransformer`, `ScoreTransformer` et cetera geben. Folgende Typdefinition abstrahiert entsprechend über Animal:

```
type Transformer state = state -> state
```

Das `state` ist ein Typparameter, vergleichbar mit einem Generic in Java, nur ohne spitze Klammern. Damit lässt sich jetzt `AnimalTransformer` durch `Transformer Animal` ersetzen.

Typparameter spielen übrigens eine zentrale Rolle in diesem Artikel. Haskell-Programmier*innen mögen sich die Frage stellen, ob die Definition von `Animal` nicht noch einen verträgt. Unter den insgesamt vier Attributen von Animal taucht der Typ `Weight` zweimal auf, er ist deshalb ein guter Kandidat:

```
data Animal weight =
Dillo Liveness weight
    | Parrot String weight
```

Die Typen von `runOver` und `feed` müssen sich entsprechend leicht ändern. Ein Vorteil von Haskell ist, dass die Typsignatur fehlen darf – der Compiler berechnet sie in dem Fall automatisch. Bei `runOver` ergibt sich folgender Code:

```
runOver :: Transformer (Animal weight)
```

Aus diesem Typ lässt sich etwas Wesentliches erkennen: Da für das Gewicht weiterhin ein Typparameter existiert (zu erkennen an der Kleinschreibung), funktioniert `runOver` für alle Typen, die für `weight` eingesetzt werden. Mit anderen Worten: `runOver` kann am Gewicht nichts ändern, da es nichts darüber weiß. Anders ist es bei `feed`:

```
feed :: Num weight => weight -> (Transformer (Animal weight))
```

Diese Signatur führt zwar ebenfalls `weight` als Typvariable – erlaubt also unterschiedliche Typen –, aber auch einen Constraint, nämlich `Num weight` vor dem Doppelpfeil. Er besagt, dass `weight` ein „numerischer Typ“ sein muss, um etwas zum Gewicht addieren zu dürfen. Das gilt unabhängig vom eigentlichen Typ `Double` oder `Integer`: Es ließen sich ebenso neue numerische Typen definieren, um zum Beispiel die Maßeinheit mitzuführen.

Typparameter bieten noch weitere Abstraktionsmöglichkeiten: Gibt es einen, kann man schauen, ob man einen Funktor definieren kann. Funktoren finden sich auch in Java in Form einer Methode namens `map`, die es zum Beispiel in `java.util.Stream` und `java.util.Option` gibt. Die Methode wendet eine Funktion auf alle Elemente eines Streams an oder den einen Wert, der möglicherweise in `Optional` drin ist.

In Haskell gibt es ebenfalls `map`, zum Beispiel auf Listen:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (first:rest) = (f first) : (map f rest)
```

Da steht: Gegeben ist eine Funktion von `a` nach `b` (wieder Typvariablen – das können beliebige Typen sein), und eine Liste von `a`-Objekten (die eckigen Klammern erzeigen Listen) liefert `map` eine Liste von `b`-Objekten. Listen in Haskell sind (wie in

Listing 6: Komposition von Stash Transformer

```
composeST :: ST result1 -> (result1 -> ST result2) -> ST result2
composeST transformer next =
ST (\ stash ->
    let (result1, stash1) = applyST transformer stash
    in applyST (next result1) stash1)
```

allen funktionalen Sprachen) so aufgebaut: Es gibt die leere Liste `[]`; eine nicht leere ist eine sogenannte `Cons`-Liste, die in ihr erstes Element und den Rest zerfällt, also die Liste aller Elemente ohne das erste Element. Eine `Cons`-Liste wird als `first:rest` geschrieben, der Doppelpunkt trennt das erste Element und den Rest. Die erste Gleichung von `map` besagt demnach, dass `map` über die leere Liste wieder die leere Liste liefert. Die zweite Gleichung sagt, dass `map` die Funktion `f` auf das erste Element anwendet, das Ergebnis von `map` auf dem Rest der Liste berechnet und beides wieder zu einer Liste zusammenfügt. Das geht auch für `animalMap`:

```
animalMap :: (a -> b) -> Animal a -> Animal b
```

Die Funktion macht aus einer Funktion von `a` nach `b` eine von `Animal a` nach `Animal b`. Der Unterschied zu `map` ist nur, dass statt `=[...]=` hier `Animal` steht. Warum die Nonsensnamen `a` und `b` statt `weight1` und `weight2`? Weil es schwieriger wäre, die Verbindung zu anderen Funktoren zu sehen, bei denen es nicht um Gewicht geht. So sieht die Definition von `animalMap` aus:

```
animalMap f (Dillo liveness weight) = Dillo liveness (f weight)
animalMap f (Parrot sentence weight) = Parrot sentence (f weight)
```

Das ist unter anderem sinnvoll, um `feed` etwas kompakter zu definieren:

```
feed amount animal =
    animalMap (\ weight -> weight + amount) animal
```

Der Backslash \ ist die Haskell-Schreibweise für Lambda: Da steht eine Funktion, die auf ein Gewicht `weight` eine Menge `amount` addiert. Hardcore-Haskell-Programmierer*innen würden sogar das `animal` – das links und rechts am Ende steht und darum redundant ist – weglassen:

```
feed amount = animalMap (\ weight -> weight + amount)
```

Genau wie sich `Animal` mit der Definition einer `map`-Funktion zum Funktor machen lässt, geht das bei vielen anderen Typen, solange sie nur einen Typparameter haben. Die „Idee“ Funktor kann man in Haskell auch direkt ausdrücken, als eine sogenannte Typklasse:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Diese Typklasse ist in der Haskell-Standardbibliothek enthalten. Typklassen sind etwas anderes als Klassen in objektorientierter Programmierung, sie sind eher mit Interfaces vergleichbar. Aber besser, man löst sich da vom Bedürfnis, das analoge Java-Konzept zu finden. Hier steht: Ein `f` – etwa `[]` oder `Animal` – ist dann ein Funktor, wenn es eine Funktion namens `fmap` gibt mit der gezeigten Signatur. Der Einsatz von `f` in `f a` und `f b` zeigt an, dass `f` ein Typkonstruktor ist, also selbst einen Typparameter hat. Das lässt sich benutzen, um `Animal` offiziell zum Funktor zu erklären:

Listing 7: Aus Vorrat füttern

```
feedFromStashST :: Weight -> Animal Weight -> ST (Animal Weight)
feedFromStashST amount animal =
composeST (removeFromStashST amount)
    (\ realAmount -> feedST realAmount animal)
```

Listing 8: Stash Transformer mit Monadensyntax

```
feedFromStashST :: Weight -> Animal Weight -> ST (Animal Weight)
feedFromStashST amount animal =
    do realAmount <- removeFromStashST amount
       feedST realAmount animal
```

```
instance Functor Animal where
    fmap = animalMap
```

Anschließend kann man `fmap` statt `animalMap` schreiben und muss nicht darüber nachdenken, wie die spezielle `map`-Operation für einen bestimmten Typ heißt.

Wie gehen Effekte in Haskell wirklich?

Eine ganze Folge von Zustandstransformationen könnte wie in Listing 2 aussehen. `let` bindet lokale Variablen: Da durch jede Teiltransformation wieder ein neues `Animal`-Objekt entsteht, bekommt jedes davon einen eigenen Namen. Statt des umständlichen Wegs ist es möglich, die drei Funktionen direkt hintereinander zu verwenden, ohne den Zwischenergebnissen Namen zu geben:

```
feedAndDie d = runOver (feed 3 (feed 2 d))
```

Das geht sogar noch kompakter ohne explizite Erwähnung von `d`:

```
feedAndDie = runOver . (feed 3) . (feed 2)
```

Der Punkt `.` steht für eine sogenannte Funktionskomposition und wendet die einzelnen Funktionen hintereinander an, eine jeweils auf das Ergebnis der vorherigen, und zwar von rechts nach links. Der Punkt ist in Haskell lediglich eine Funktion, die allerdings in Infixschreibweise zwischen ihren Argumenten steht. Er hat folgende Typsignatur, die anzeigt, wie der Zustand von rechts nach links „durchgefädelt“ wird:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Listing 9: Tiere auf dem texanischen Highway, Idris-Version

```
data Liveness = Dead | Alive

Weight : Type
Weight = Int

data Animal : Type -> Type where
 Dillo : Liveness -> weight -> Animal weight
 Parrot : String -> weight -> Animal weight

a1 : Animal Int
a1 = Dillo Alive 10
a2 : Animal Int
a2 = Parrot "Hello!" 2
a3 : Animal Int
a3 = Dillo Dead 12
a4 : Animal Int
a4 = Parrot "Goodbye!" 1

Transformer : Type -> Type
Transformer state = state -> state

feed : Num weight => weight -> (Transformer (Animal weight))
feed amount (Dillo liveness weight) = Dillo liveness (weight + amount)
feed amount (Parrot sentence weight) = Parrot sentence (weight + amount)

runOver : Animal weight -> Animal weight
runOver (Dillo liveness weight) = Dillo Dead weight
runOver (Parrot sentence weight) = Parrot "" weight
```

Aber was ist, wenn die Anforderungen an das Programm sich ändern? Die texanische Highway-Behörde könnte zum Beispiel verfügen, dass ab sofort die Futtermenge pro Tag begrenzt wird. Es gibt also jeden Tag einen Vorrat, einen `stash`. Dann reicht die Signatur `Animal weight -> Animal weight` nicht mehr aus, um die Fütteroperation zu beschreiben. Die Funktion muss deshalb den Vorrat mitführen: sowohl als Eingabe als auch – weil sie den Vorrat vermindert – als Ausgabe. Der Vorrat ist `Zustand`. Das sieht wie in Listing 3 aus.

Die Klammern um `(Animal Weight, Stash)` bedeuten, dass die Funktion beide Werte – das neue `Animal`-Objekt und den neuen Vorrat – zusammen in ein Tupel gepackt zurückgibt. Dass `Weight` wie `Stash` erneut ein konkreter Typ ist und keine Typvariable, vereinfacht den Code. Das `if-then-else` entscheidet, ob entweder die gewünschte Futtermenge, wenn noch genug Vorrat da ist, oder der restliche Vorrat verfüttert wird.

Leider funktioniert bei der neuen Funktion der Punkt nicht mehr, um mehrere solcher Operationen zu komponieren – Ein- und Ausgabetypen passen nicht genau zueinander. Das wird sichtbar, wenn man aus `feedFromStash` eine Hilfsfunktion herauslöst, die eine Anfrage an den Vorrat bearbeitet (Listing 4).

Diese Funktion ermittelt, wie viel von der angefragten Menge im Vorrat verfügbar ist, und gibt diese Menge heraus. Das Muster ist das gleiche, aber wieder passen Ein- und Ausgabetypen nicht zusammen. Es wird eine neue Kompositionsoperation für Operationen benötigt, die einen Vorrat mitführen. Um da systematisch heranzugehen, werden „Operationen, die einen Vorrat mitführen“, als eigener Datentyp definiert. In `feedFromStash` und `removeFromStash` ist das der hintere Teil:

```
data ST result = ST (Stash -> (result, Stash))
```

Die Typvariable `result` abstrahiert über `Animal Weight` in `feedFromStash` und `Weight` in `removeFromStash`. Diese Typdefinition definiert einen Wrapper namens `ST` („stash transformer“), der bei Gelegenheit an- und wieder abzuwickeln ist. Das ist erst mal umständlich, kann aber den konkreten Inhalt verstecken. An der Stelle ist erwähnenswert, dass `ST` einen Typparameter hat und damit wie `Animal` ein heißer Kandidat für einen Funktor ist. Passend zum Typ lässt sich eine ST-Variante von `removeFromStash` definieren. Folgender Code refaktorisiert zunächst, um die Currifizierung durch ein explizites Lambda sichtbar zu machen:

```
removeFromStash amount = \ stash ->
    let realAmount = min amount stash
    in (realAmount, stash - realAmount)
```

Durch das Wrapping in `ST` entsteht im Anschluss eine neue Typsignatur (Listing 5).

Listing 10: Aus Vorrat füttern (Effekt in Idris)

```
feedFromStash : Weight -> Animal Weight -> Eff (Animal Weight) [STASH]
feedFromStash amount animal =
   do realAmount <- removeFromStash amount
      feedF realAmount animal
```

Listing 11: Debug-Output mit STDIO-Effekt

```
feedFromStash : Weight -> Animal Weight -> Eff (Animal Weight) ,
                                                [STASH, STDIO]
feedFromStash amount animal =
    do realAmount <- removeFromStash amount
      putStr ("feeding " ++ show realAmount)
      feedF realAmount animal
```

Direkt anwenden lässt sich die Funktion nicht, ohne zuvor `ST` wieder abzuwickeln. Das erledigt folgende Hilfsfunktion:

```
applyST :: ST result -> Stash -> (result, Stash)
applyST (ST transform) stash = transform stash
```

Die neue Funktion für das Komponieren von `ST` sieht etwas merkwürdig aus (Listing 6).

Anders als `.` läuft `composeST` von links nach rechts ab: Erst geschieht die Transformation `transformer`; das Resultat wird `result1` genannt und der Vorrat danach `stash1`. `result1` wird benutzt, um mit `next` die nächste Transformation zu berechnen, die danach läuft. Jetzt war ja die Idee, `removeFromStash` mit `feed` zu komponieren. Dazu wird noch eine ST-Version von `feed` benötigt:

```
feedST amount animal =
ST (\ stash -> (feed amount animal, stash))
```

Diese Funktion reicht `stash` einfach durch, ändert den Zustand also überhaupt nicht. Die Funktion wird deshalb „pure“ genannt. Weil es viele solcher reinen Funktionen gibt, die sich in ST-Programmen benutzen lassen, lohnt es sich, über das obige Muster mit einer Hilfsfunktion namens `pureST` zu abstrahieren und damit `feed` zu refaktorisieren:

```
pureST :: result -> ST result
pureST result = ST (\ stash -> (result, stash))
feedST amount animal = pure (feed amount animal)
```

Nun kann `composeST` die beiden Operationen `removeFromStashST` und `feedST` komponieren, und zwar wie in Listing 7.

Elegant sieht das noch nicht aus, aber wer sich ein bisschen mit funktionaler Programmierung auskennt, erkennt, wo der Hase lang läuft: `ST`, `pureST` und `composeST` bilden eine sogenannte Monade. Der Begriff ist in der Haskell-Standardbibliothek vordefiniert in Form der Typklasse `Monad`. Dort heißt `pureST` allerdings `return` und `composeST` hat den Namen `>>\=` (ausgesprochen „bind“). Die Monadeninstanz für `ST` sieht folgendermaßen aus (tatsächlich muss allerdings jede Monade auch ein Funktor sein):

```
instance Monad ST where
  return = pureST
  (>>=) = composeST
```

Für Monaden bietet Haskell Spezialsyntax, die durch das Schlüsselwort `do` eingeführt wird. Damit sieht `feedFromStashST` wie in Listing 8 aus.

Intern übersetzt der Haskell-Compiler das exakt in die erste Version, indem er aus jedem `<-` einen Aufruf von `>>\=` macht.

Dieser Artikel konzentriert sich jedoch nicht auf die Syntax, sondern die Grundidee: Der Typ `ST (Animal Weight)` sagt aus, dass `feedFromStashST amount animal` eine Berechnung beschreibt, die einen internen Vorrat anfasst und möglicherweise modifiziert. Der Typ beschreibt also den Effekt, den die Funktion hat. Ganz wichtig: Sie kann „keinen anderen Effekt“ haben. Haskell meint es also ernst damit, Effekte explizit und damit kontrollierbar zu machen.

Über Effekte abstrahieren

Was ist, wenn die Highway-Behörde noch andere Faktoren wie das Wetter oder die Jahreszeit in die Fütterung einfließen lässt? Schlecht wäre, wenn dann ein komplett neues Refaktorisieren anstünde. Außerdem: Was ist, wenn der Vorratsstand nicht mehr einfach nur in einer Variablen, sondern in einer Datenbank sitzt oder von einem Webservice verwaltet wird? Historisch gesehen ist die Antwort aus dem Haskell-Lager: „Nimm eine andere Mo-

nade!“ oder „Ändere die Monade!“ Dann wird aber schon wieder eine Refaktorisierung fällig, über die Java-Programmierer nur lachen können. Außerdem unterscheidet die neue Monade nicht mehr, welche Teileffekte in einem Programmteil gelten – alles läuft in derselben großen Monade. Dem Problem ist mit einer cleveren Kombination von Typklassen und sogenannten Monadentransformatoren beizukommen (durch den sogenannten „mtl style“), aber das funktioniert in der Praxis ebenso schlecht, wie es sich lächerlich kompliziert anhört.

Dabei ist die Idee nicht kompliziert: Jede Funktion soll im Typ bekannt geben, welche Effekte sie generiert. Wenn eine Funktion `f` Effekte `a` und `b` generiert und eine Funktion `g` Effekte `b` und `c`, dann erzeugt eine Funktion, die beide aufruft, die Effekte `a, b, c`. Um die einfache Idee umzusetzen, Ausdrücke mit den Effekten zu annotieren, die darin passieren, kommt nun Idris ins Spiel.

Richtungsweisend: Typen anders handhaben

Idris ist eine weitere funktionale Programmiersprache in der Haskell-Tradition, die noch hauptsächlich in der Forschung beheimatet ist. Für produktiven Einsatz ist Idris noch nicht geeignet. Aber die Sprache setzt fortschrittliche Ideen für den Umgang mit Typen um, die dort eine noch aktivere Rolle als in Haskell spielen. Um das Beispiel von oben weiterzuführen, ist erst einmal der Haskell-Code in Idris zu übersetzen (Listing 9). Die Ähnlichkeiten sind ebenso offensichtlich wie einige Unterschiede:

- Bei Typsignaturen steht ein einfacher Doppelpunkt statt des doppelten.
- Typsynonyme werden mit Gleichungen definiert.
- Jede Gleichung hat eine Typdeklaration – sogar die Typdefinitionen haben Typdeklarationen, wobei ein Typ den Typ `Type` hat.
- `Animal` hat einen Typ: `Type -> Type`. Das liegt daran, dass Animal einen Typparameter hat, also einen Typ als Eingabe akzeptiert und wieder einen Typ liefert.
- Bei den beiden Klassen `Dillo` und `Parrot` ist jeweils die komplette Signatur des Konstruktors angegeben, nicht nur die Typen der Attribute.

Programme aus Typen generieren

Idris nimmt offensichtlich Typen etwas ernster als Haskell, im Wesentlichen ist aber vieles ähnlich. Tatsächlich spektakulär an Idris ist, dass die Sprache den Entwicklungsprozess auf den Kopf stellt: Entwickler*innen sind gewohnt, ein Programm zu schreiben, das der Compiler dann auf Typkorrektheit überprüft. In Haskell inferiert der Compiler beispielsweise die Typsignaturen der meisten Funktionen automatisch. Und da sitzt die Magie in Idris: Da schreiben Entwickler*innen den Typ zuerst, und zwar immer.

Mit ein paar Tastendrücken kann Idris dazu eine mögliche Definition anbieten – also genau umgekehrt wie in Haskell. Das Idris-Buch von Edwin Brady, dem Autor der Sprache [2], listet dafür folgendes Beispiel, das aus einer Liste von Zeichenketten eine Liste ihrer Längen macht:

```
allLengths : List String -> List Nat
```

Im Beispiel ist ein weiterer Unterschied zu Haskell zu sehen: Der Listentyp ist als `List` geschrieben statt mit `[]`. Die IDE generiert auf Knopfdruck dazu ein erstes Definitionsskelett:

c't Fotografie
Mehr Mut zur
FARBE
So gelingen lebendige Architekturfotos
Warum ZORN kreativer macht
Heft-DVD
Vollversion
Gute Aussichten für Fotobegeisterte.
Sparen Sie 35% im Abo und sammeln wertvolles Know-how:
2 Ausgaben kompaktes Profiwissen für 14,60 € (Preis in DE)
Workshops und Tutorials
Tests und Vergleiche aktueller Geräte
Inklusive Geschenk nach Wahl
z. B. Kamera-Reinigungsset
Jetzt bestellen: ct-foto.de/miniabo
c't Fotografie
+49 541/80 009 120
leserservice@heise.de

```
allLengths : List String -> List Nat
allLengths list = ?allLengths_rhs
```

Der nächste Code generiert die benötigten zwei Gleichungen für Listen:

```
allLengths [] = ?allLengths_rhs_1
allLengths (first :: rest) = ?allLengths_rhs_2
```

Der doppelte Doppelpunkt steht für das Listen-Cons – gegenüber Haskell sind also der einfache und der doppelte Doppelpunkt genau vertauscht. Die ?-Namen sind Platzhalter, auch bei ihnen kann man einen Knopf drücken, der da etwas mit dem richtigen Typ hinschreibt:

```
allLengths [] = []
allLengths (first :: rest) = []
```

Das Ergebnis ist zwar typkorrekt, aber falsch. Damit es besser wird, muss der Typ besser werden, also präziser beschreiben, was die Funktion machen soll. Eine einfache Methode ist es, Information über die Länge der Listen unterzubringen, die Eingabeliste muss nämlich genauso groß sein wie die Ausgabeliste. Das geht mit dem Typ `Vect`:

```
allLengths : Vect len String -> Vect len Nat
```

Die Typvariable `len` gibt also die Länge an. Was genau die Länge ist, ist `allLengths` egal, aber sie muss bei Ein- und Ausgabe gleich sein. Wenn das nicht gilt, wird der Compiler das Programm nicht durchlassen. Die IDE kann folgende Definition stiften:

```
allLengths : Vect len String -> Vect len Nat
allLengths [] = []
allLengths (first :: rest) = 0 :: allLengths xs
```

Es wird besser, ist aber immer noch nicht richtig: `length x` statt der `0` müssen Programmierer*innen selbst ergänzen. Oft kann Idris die kniffeligen Teile einer Definition finden, der Rest ist dann zwar Handarbeit, aber nicht mehr so schwierig. Dieses Beispiel ist platzbedingt primitiv, in der Praxis ist das, was der Idris-Compiler so an Code generiert, oft wie Zauberei. Der Entwicklungsprozess wird so zu einer Unterhaltung zwischen Entwickler und Compiler.

Das Beispiel `allLengths` zeigt bereits etwas, was mit „normalen" Typen nicht geht: Im `Vect`-Typ steht die Länge – im Typ steckt somit ein Wert. Idris durchbricht die alte Zweiklassengesellschaft, in der Typen (Compile-Zeit) und Werte (Laufzeit) strikt getrennt sind. Die Konsequenz davon ist, dass auch auf der Typebene Programme laufen, die die Werte in den Typen manipulieren. Typsysteme, in denen das geht, benutzen dafür den Begriff „Dependent Types" [3].

Das Beispiel illustriert aber zwei zentrale Ideen in Idris: Die Typen dienen als Spezifikation einer Funktion – weit darüber hinaus, was unter anderem in Java möglich ist. Und wenn die Spezifikation präzise genug ist, lassen sich Teile der Implementierung automatisch generieren. In diesem Artikel geht es vor allem um den ersten Aspekt: Typen als Spezifikation, insbesondere als Spezifikation von Effekten.

Effekte in Idris

Wie sich herausstellt, sind Dependent Types das richtige Werkzeug, um Effekte abzubilden. Dafür ist es nicht nötig, sie fest in die Sprache einzubauen – Idris liefert eine Library namens Effects mit. In Listing 10 sieht man die `feedFromStash`-Funktion: Die eigentliche Definition ist fast die gleiche wie in Haskell – abgesehen davon, dass `feedST` hier `feedF` heißt: Der Code basiert also ebenfalls auf einer Monade. Der Typ allerdings verrät, dass es nicht die ST-Monade ist, sondern die `Eff`-Methode aus der Effects-Bibliothek. In ihr gibt es eine Monade für alles, die hat aber einen Typparameter, der verrät, welche Effekte diese spezielle Operation hat. In diesem Fall ist das eine Liste, in der `STASH` steht – die Operation greift also auf den Vorrat zu.

Falls die Funktion noch andere Effekte hat, werden diese in die Liste aufgenommen. Angenommen, die Funktion druckt noch den Vorratsstand aus – dann kommt in der Liste der Effekt `STDIO` dazu (Listing 11).

`Effects` erlaubt demnach detaillierte Beschreibungen an jeder Operation, um zu kennzeichnen, welche Effekte sie hat, und solche Operationen zu kombinieren. Diese Idee geht weit darüber hinaus, noch zusätzliche „Services" anzuschließen. Ein möglicher Effekt ist zum Beispiel Nichtdeterminismus. Das könnte – zugegebenermaßen etwas künstlich – aussehen wie in Listing 12.

Der `SELECT`-Effekt ist bei `Effects` mitgeliefert und erlaubt, nicht nur einen Programmverlauf zu beschreiben, sondern viele auf einmal. In diesem Fall steht als Futtermenge ein Aufruf von `select`, der sagt, dass die Futtermenge zwischen `lowAmount` und `highAmount` liegen muss. Das Programm steht nicht nur für einen sequenziellen Durchlauf, sondern für alle möglichen Futtermengen aus diesem Bereich, und das Programm lässt sich auch so starten, dass es alle Möglichkeiten des Verlaufs berechnet. Das geht zum Beispiel so:

```
the (List _) (runInit [10, ()] (feedFromStashND 5 12 a1))
```

Die `runInit`-Funktion akzeptiert die initialen Werte für den `STASH`-und den `SELECT`-Effekt: `SELECT` benötigt keinen, und bei `STASH` ist es der initiale Vorrat. Das `the (List _)` sorgt dafür, dass die Listenvariante von `SELECT` benutzt werden soll. (Es gibt noch andere.) Dann kommen alle möglichen Rückgabewerte für Entnahmemengen zwischen 5 und 12 heraus:

```
[Dillo Alive 15, Dillo Alive 16, Dillo Alive 17,
Dillo Alive 18, Dillo Alive 19, Dillo Alive 20,
Dillo Alive 20, Dillo Alive 20]
```

Die Funktion `feedF` fehlt noch und ist so definiert:

```
feedF : Weight -> Animal Weight -> Eff (Animal Weight) []
feedF amount animal = pure (feed amount animal)
```

Die Effektliste bei `Eff` ist leer, die Funktion hat also keine eigenen Effekte. Die Implementierung ruft einfach `feed` auf und liefert dann das Ergebnis mit der Monadenfunktion `pure`. Die Kontrolle über die Effekte findet komplett über die Typebene statt – das geht aber nur deshalb, weil Idris erlaubt, auf dieser Ebene Listen zu manipulieren, Dependent Types sei Dank.

Interface versus Implementierung

Die Innereien von Effects basieren auf einer Idee namens „Algebraic Effect Handlers". Zentral dafür ist, dass jeder Effekt als

Listing 12: Nichtdeterminismus als Effekt

```
feedFromStashND : Weight -> Weight -> Animal Weight -> Eff
                                        (Animal Weight) [STASH, SELECT]
feedFromStashND lowAmount highAmount animal =
    do amount <- select [lowAmount .. highAmount]
       realAmount <- removeFromStash amount
       feedF realAmount animal
```

Objekt abgebildet wird. Statt dass also, wie in Java, eine Funktion aufgerufen wird, die „etwas macht“, wird eine Beschreibung erzeugt, was auszuführen ist. In einem separaten Schritt wird dann die Beschreibung interpretiert und die dazugehörige Aktion ausgeführt. Diese Trennung ist analog zur der zwischen Interface und Implementierung.

Dieser Abschnitt gibt einen kurzen Überblick, wie das funktioniert. Das Highway-Beispiel benötigt einen Datentyp für die Effekte auf den Futtervorrat. Da es nur eine Operation auf dem Vorrat gibt (`removeFromStash`), gibt es nur eine Klasse:

```
data StashT : Effect where
  RemoveFromStash : Weight -> sig StashT Weight Stash Stash
```

`Effect` ist der Typ für Effekttypen. Der merkwürdige Resultattyp `sig StashT Weight Stash Stash` besagt Folgendes:

- Als Ergebnis kommt ein Wert vom Typ `Weight` heraus (wie viel Futter tatsächlich ausgegeben wurde).
- Die Operation braucht Zugriff auf eine Ressource vom Typ `Stash` (den Vorrat) und produziert eine neue Ressource, nämlich die Menge des restlichen Futters nach der Operation.

Etwas Boilerplate-Code wie in Listing 13 definiert dann den Typ `STASH` und die Funktion `removeFromStash` zur konkreten Verwendung in Programmen. Bis hier ist nur beschrieben, dass es die Operation `removeFromStash` gibt, nicht, wie sie implementiert ist. Das passiert durch die Implementierung einer Typklasse namens `Handler` – Idris benötigt dafür nicht das Haskell-Schlüsselwort `instance` (Listing 14).

Hier steht, wie sich der `StashT`-Effekt in einer beliebigen Monade (darum die Typvariable `monad`) interpretieren lässt; `stash` ist die Vorratsressource. Es wird die tatsächliche Futtermenge ermittelt. Resultat und neue Futtermenge werden an die übergebene Funktion `next` übermittelt. Da der Handler `next` auch mehr als einmal aufrufen kann, kann zum Beispiel der Nichtdeterminismus-Handler das weitere Programm mehrmals ausführen, mit unterschiedlichen Werten aus `select`. Andere Instanzen von Handler sind denkbar, um zum Beispiel den Vorrat in einer Datenbank abzuspeichern: Das ist also eine Form von Dependency Injection.

Listing 13: Definition eines neuen Effekts

```
STASH : EFFECT
STASH = MkEff Stash StashT

removeFromStash : Weight -> Eff Weight [STASH]
removeFromStash amount = call (RemoveFromStash amount)
```

Listing 14: Implementierung des STASH-Effekts

```
Handler StashT monad where
handle stash (RemoveFromStash amount) next =
  let realAmount = min stash amount
  in next realAmount (stash - realAmount)
```

Effektsysteme: Ein kleines Problem und eine große Lösung?

Java-Programmier*innen mögen wenig Verständnis für die funktionale Herangehensweise aufbringen: Es sind umständliche Vorbereitungen nötig, Effekte abzubilden – mit Lambdas, Typklassen, Kategorietheorie –, was in Java „einfach so“ funktioniert. Der Preis für den leichten Start in Java ist jedoch hoch. Viele Bugs entstehen durch unkontrollierte Effekte: fehlende Initialisierung, falsche Reihenfolge von Datenbankoperationen, unerwartete Exceptions.

Ein Ausweg aus der Misere ist nicht einfach, was ein Grund dafür sein dürfte, dass Effektsysteme erst seit kurzer Zeit reif genug für den Produktiveinsatz sind. Aber sie werden ein mächtiges Werkzeug in der Programmierwerkstatt der Zukunft. Die Techniken zum Erstellen kleiner Programme funktionieren dabei genauso im Großen. Sie sind dort umso nützlicher, weil der Effekttyp auf einen Blick verdeutlicht, was ein großer Batzen Code so alles anstellen kann.

Die Implementierung von Effektsystemen als Bibliotheken erfordert das Programmieren auf Typebene, und das hat noch mehr Vorteile:

- Die Typen können in höherem Maße als Dokumentation und Spezifikation von Funktionen dienen.
- Sie können zur automatischen Programmgenerierung dienen.
- Die Typen können „Beweise“ von kritischen Programmeigenschaften zum Ausdruck bringen.

Das Programmieren mit Beweisen ist besonders zukunftsträchtig und birgt leistungsfähige Mittel, um korrekte und robuste Programme zu schreiben, vor allem im Zusammenhang mit Effekten. Zum Beispiel lassen sich auf der Typebene die Zustände des Systems so codieren, dass das Typsystem sicherstellen kann, dass alle Zustandsübergänge korrekt sind.

Abgesehen davon macht das Programmieren in Idris Spaß – es ist eine Herausforderung an das Hirn, belohnt aber mit tieferem Verständnis und robusten Programmen.

In welche Richtung wird Haskell künftig steuern?

Idris gehört hoffentlich die Zukunft. Allerdings ist die Sprache noch nicht reif für den produktiven Einsatz. Haskell ist es hingegen schon lange – was machen aber Haskell-Programmierer*innen, die vom Programmieren mit Effekten profitieren wollen? Glücklicherweise (und zum Leidwesen mancher) wird Haskell stetig weiterentwickelt. Ein Bündel von Erweiterungen stellt unter dem Titel „Dependent Haskell“ Entwickler*innen ebenfalls Möglichkeiten zur Verfügung, auf der Typebene zu programmieren. Das reicht aus, um ähnliche Effektsysteme wie das in Idris zu implementieren. Dementsprechend ist in den vergangenen Jahren eine Vielzahl von Library-Projekten gestartet, die ähnliche Möglichkeiten anbieten – am weitesten fortgeschritten sind fused-effects und polysemy. Sie erreichen nicht ganz den Komfort der Idris-Version, sind aber durchaus produktionsreif. (ane@ix.de)

Quellen

[1] Graham Hutton; Programming in Haskell (2. Aufl.); Cambridge University Press 2016

[2] Edwin Brady; Type-Driven Development with Idris; Manning 2017

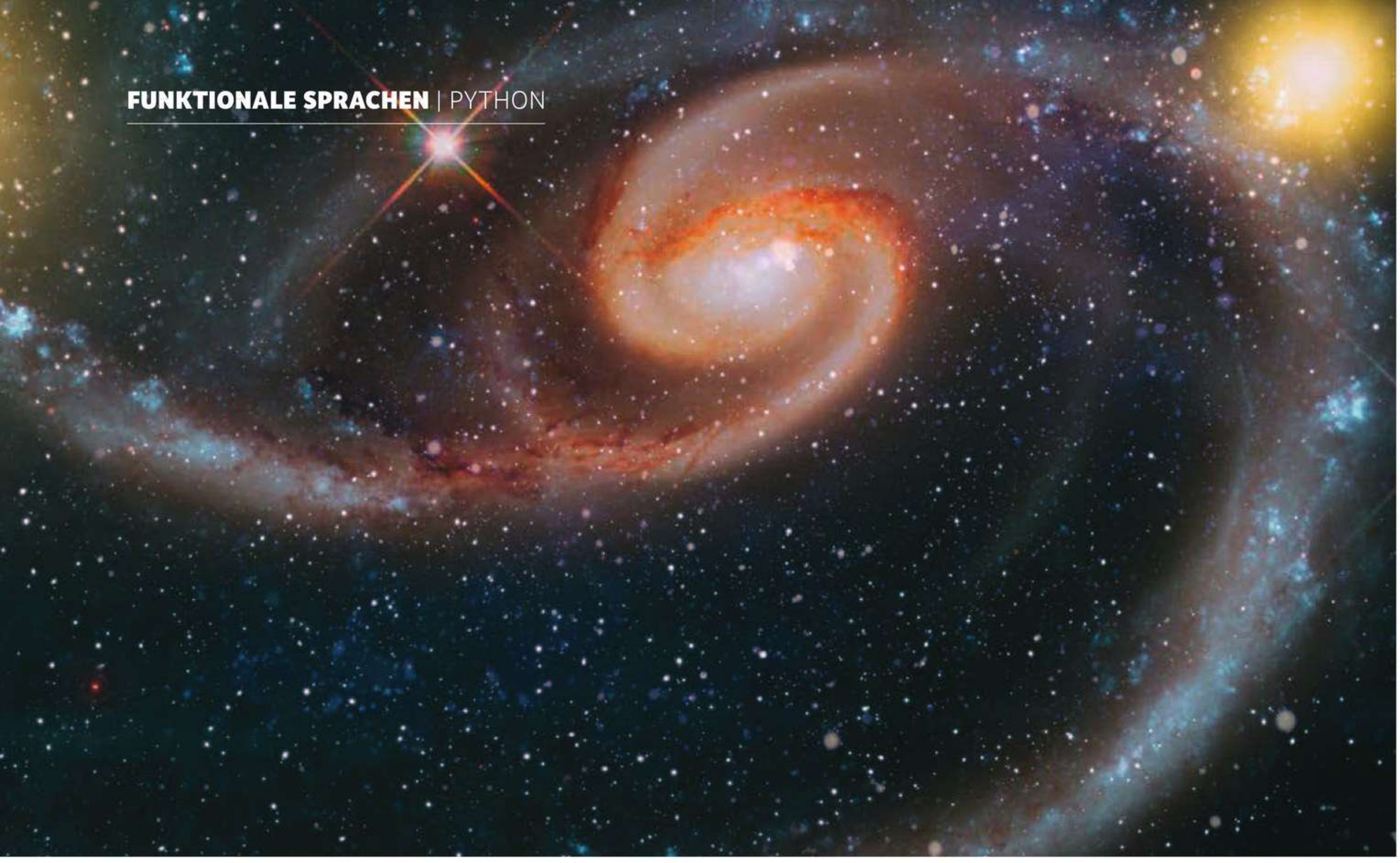
[3] Daniel P. Friedman, David Thrane Christiansen; The Little Typer; MIT Press 2018

[4] Alle Listings finden sich unter https://ix.de/zfzh



Dr. Michael Sperber

ist Geschäftsführer der Active Group. Er ist international anerkannter Experte für funktionale Programmierung.



Python: Vergangenheit und aktueller Erfolg

Geschichte einer Berühmtheit

Gerhard Völkl

Python hat sich von einer einfachen Scripting-Sprache zu einer der am häufigsten verwendeten Programmiersprachen entwickelt. In innovativen Bereichen wie Data Science oder maschinellem Lernen ist die Sprache inzwischen die erste Wahl.

Auf die Frage nach seiner Erfolgsgeschichte antwortete der Sänger Harry Belafonte einmal: „Ich habe dreißig Jahre gebraucht, um über Nacht berühmt zu werden." Dieses Zitat trifft gleichermaßen auf Python zu. Im Jahr 1989 entwickelte Guido van Rossum neben seiner Arbeit am Centrum Wiskunde & Informatica (CWI), dem nationalen niederländischen Forschungsinstitut für Mathematik und Informatik in Amsterdam, eine eigene Programmiersprache. Techniken zur Erstellung einer eigenen Sprache lernte er unter anderem durch Beiträge zu ABC, ebenso wie man einen Bytecode-Compiler und einen Parser schreibt oder wie man die unterschiedlichen Datentypen gut im Speicher unterbringen kann.

Was van Rossum an den damals häufig verwendeten Sprachen C oder Pascal gefiel, übernahm er für sein Projekt. Er hatte sich das Ziel gesteckt, die Lücke zwischen der sehr rudimentären Programmierung in Shells und dem komplexeren C zu schließen. Python sollte eine Programmiersprache für alle sein, nicht nur für Leute mit abgeschlossenem Informatikstudium.

Etwa zwei Jahre später setzten van Rossums Kollegen am CWI Python für kleinere Programme ein. Die neue Sprache stieß auf großes Interesse und zog immer weitere Kreise. Als Forschungsinstitut ist das CWI nicht kommerziell ausgerichtet. Entsprechend konnte van Rossum den kompletten Sourcecode von Python kostenlos im damals weitverbreiteten Usenet zur Verfügung stellen.

Hier liegt sicherlich einer der Gründe, warum Python die Entwickler*innenwelt im Sturm eroberte. Heute folgen große Konzerne bewusst dieser Strategie, etwa Microsoft für TypeScript, Google für Go oder Apple für Swift.

Installation

Wer selbst einmal Python ausprobieren möchte, kann alles dafür Notwendige als fertige Software für Microsoft Windows oder Apple macOS von der Python-Foundation-Seite herunterladen.

Für Linux-Systeme gibt es dort den Sourcecode. Viele Linux-Distributoren bieten eine Installation über ihre eigenen Kanäle an, etwa mit dem Befehl

```
apt-get install python
```

Egal auf welcher Plattform lässt sich Python via `python` starten und Nutzer*innen können die ersten Befehle eingeben. Da Python grundsätzlich mit einem Interpreter arbeitet, führt es Ausdrücke nach `>>>` sofort aus (Abbildung 1).

```
>>> a = 1 + 2
```

Der Ausdruck veranlasst Python, intern eine Variable `a` zu erstellen und ihr den Wert 3 zuzuweisen.

```
>>>a
3
```

Gibt man `a` ein, wertet Python diesen Ausdruck aus. Konkret überprüft es, ob eine Variable mit diesem Namen existiert und welchen Wert sie aktuell hat. Vor der ersten Verwendung einer Variablen müssen Programmierer*innen diese nicht erst deklarieren oder dem System mitteilen, von welchem Typ sie ist. Das spart zusätzlichen Verwaltungsaufwand. Python stört sich ebenfalls nicht daran, wenn die Variable einen neuen Wert von einem anderen Datentyp, etwa dem Typ `String`, bekommt.

Erstes Programm

Als Einstieg ins Thema Programme mit Python dient ein kleines Beispiel, das einem hilft, alle Dateien mit einer bestimmten Endung, etwa „.doc“, aus einem Verzeichnis zu löschen. Im ersten Schritt könnte es hilfreich sein, zusammenzustellen, was dafür alles notwendig ist. Diese Beschreibung in einfacher Sprache, manchmal Pseudocode genannt, könnte so aussehen:

```
Verzeichnis = "c:/example_path/"

Für jede Datei im Verzeichnis
 Wenn die Datei auf ".doc" endet
          Lösche Datei
```

Jetzt hier die Umsetzung in ein Python-Programm:

```
from os import listdir, remove

folder = 'c:/example_path/'

for file in listdir(folder):
    if file.endswith('.doc'):
        remove(folder + file)
```

Die erste Zeile bedeutet etwas Verwaltungsaufwand, aber der Rest sieht dem Pseudocode recht ähnlich. Dahinter steckt ein Designprinzip von Python: Möglichst alles einfach halten und so lassen, wie Programmierer*innen es von der natürlichen Sprache gewohnt sind.

iX-TRACT

- Ursprünglich als einfach erlernbare Universalsprache konzipiert, hat sich Python mittlerweile fest in den Bereichen Data Science und Machine Learning etabliert.
- Module wie pandas und NumPy erleichtern den Umgang mit großen Datenmengen. Jupyter Notebook ist ideal zur explorativen Datenanalyse.
- Seit Version 3.6 lassen sich Variablen und andere Programmobjekte in Python mit Typinformationen versehen, was es dem Compiler ermöglicht, Fehler schon bei der Entwicklung zu identifizieren.



Der Python-Interpreter bietet zwei Herangehensweisen, entweder im interaktiven Dialog mit dem Anwender oder mit dem Ausführen kompletter Programmdateien (Abb. 1).

Listing 1: folder_example_fun.py

```
from os import listdir, remove

def delete_files(folder, extension='.tmp'):
  for file in listdir(folder):
    print(file)
    if file.endswith(extension):
        remove(folder + file)
        print(f'{file} gelöscht')

delete_files('/example_path/', '.doc')
```

Am Anfang eines Python-Programms steht, welche Funktionen und Programmobjekte es neben den eingebauten Standardbefehlen braucht. Damit alles einigermaßen übersichtlich bleibt, sind Funktionen in Bibliotheken, in Python Module genannt, zusammengefasst. In diesem Fall ist es das Modul os. Darin befinden sich die Funktionen, die etwas mit dem Betriebssystem (Operation System) zu tun haben. Für das Beispielprogramm kommen die Funktionen `remove`, um Dateien zu löschen, und `listdir`, um Elemente eines Directory aufzulisten, zum Einsatz.

Im Codeausschnitt ist die Zeile nach dem Befehl `if` durch Leerzeichen eingerückt. Dadurch erkennt man, dass `remove` nur erfolgt, wenn die Bedingung im Befehl

```
if (file.endswith('.doc'))
```

zutrifft. Diese Einrückung gehört bei Python zur Sprachsyntax und verrät der Sprache, wann etwas ausgeführt werden soll. Gleiches gilt bei Schleifen, etwa der `for`-Schleife. Auch hier rücken Entwickler*innen die Befehle ein, die die Schleife abarbeitet. Im Vergleich zu anderen Programmiersprachen spart dieses Vorgehen viele geschweifte Klammern und Semikolons.

Prinzipiell sind Python-Programme Textdateien mit der Endung .py. Der Befehl

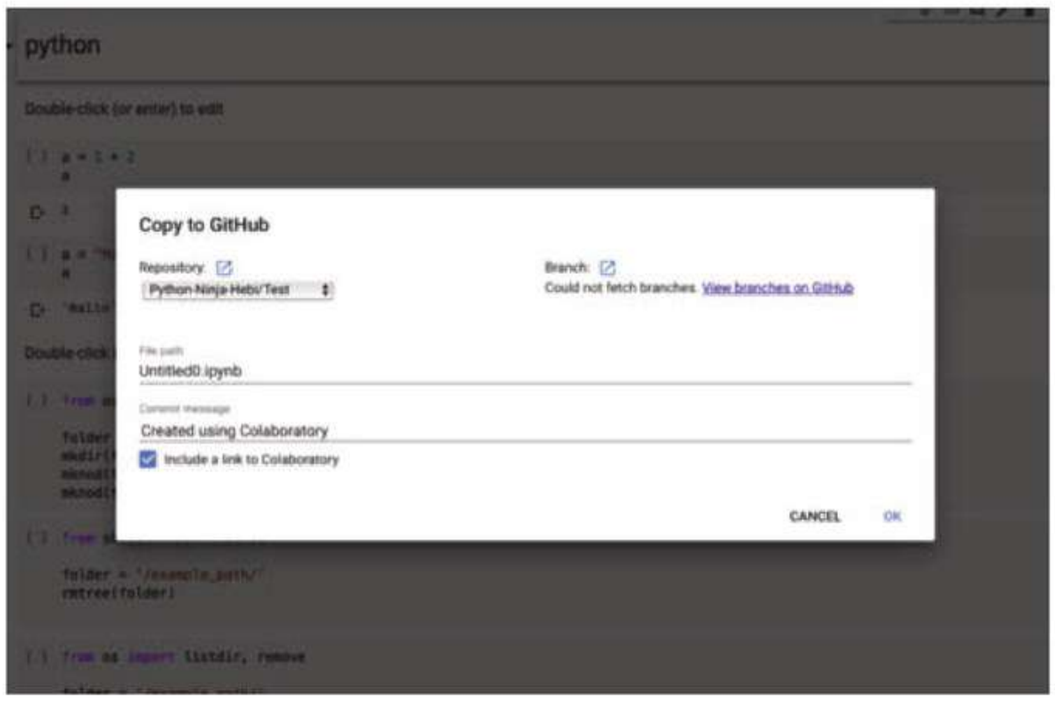
```
>python folder_example.py
```

startet den Python-Interpreter, der die Befehle in der Datei folder_example.py ausführt (Listing 1).

Grundsätzlich lassen sich Python-Programme mithilfe jedes Texteditors erstellen. IDEs wie Microsoft Visual Code helfen jedoch bei der Syntax und dabei, schneller zu programmieren. Darüber hinaus existieren komplette Entwicklungsumgebungen wie PyCharm, die ebenfalls in einer kostenlosen Community Edition zur Verfügung stehen.

Eigene Funktionen lassen sich in Python über den Befehl `def` einbinden.

```
def delete_files(folder, extension='.tmp'):
    for …
```



Mit der Software Jupyter können Anwender*innen Programmdokumente im Browser bearbeiten und ausführen (Abb. 2).

Der Parameter `extension` ist mit `'.tmp'` vorbelegt. Gibt man für diesen keinen Wert an, ist dies automatisch sein Standardwert.

Immer mehr Module

Über die Jahre ist die Anzahl der Python-Bibliotheken (Module) stetig angewachsen. Grund dafür ist sicherlich, dass Bibliotheken, die in Programmiersprachen wie C oder C++ geschrieben sind, sich relativ einfach in Python integrieren lassen. Guido van Rossum und seine Mitstreiter haben von Anfang an eine Infrastruktur aufgebaut, mit der jeder diese Module möglichst einfach auf seinem Rechner installieren kann. Dazu reicht häufig ein einzelner Befehl: `pip`.

```
>pip install numpy
```

Das Werkzeug pip schaut auf dem Server pypi.org nach, ob dort ein Modul mit dem Namen NumPy liegt, und installiert die passende Version für das aktuell verwendete Betriebssystem. Wer eigene Module erstellt, kann diese kostenlos über die Plattform pypi.org anderen zur Verfügung stellen, was wesentlich zur Verbreitung von Python beigetragen hat. Heutzutage gilt dieses Modell als ein wichtiges Kriterium für die schnelle Akzeptanz neuer Programmierumgebungen, wie sich bei `nodes` (JavaScript) und dem entsprechenden Installer für Komponenten `npm` ablesen lässt.

Dekade 2: Daten erforschen mit Python

Im Jahr 2001 saß Fernando Perez in Colorado an seiner Doktorarbeit und analysierte riesige Datenmengen aus einer Supercomputersimulation im Bereich Teilchenphysik. Zuerst setzte er C für einige Aufgaben ein, dann wieder Perl. Leider harmonierten diese Sprachen nicht mit seiner Arbeitsweise als Forscher. Wenn er neue Daten erhielt, wollte er diese möglichst interaktiv analysieren und sofort sehen, welche Werte die Ergebnisvariablen hatten oder wie sich dadurch seine Grafiken veränderten. Passte etwas nicht, wollte er schnell in den Prozess eingreifen und Variablen verändern. Da Python mit einem Interpreter arbeitet, war es die Programmiersprache für Fernando Pérez, mit der er seine Arbeitsweise umsetzen konnte.

Die von Python mitgelieferte Shell war Fernando Pérez zu wenig. Deshalb entwickelte er die Software IPython (interactive Python), später in Projekt Jupyter umbenannt (Abbildung 2). Damit verändert man eine Datei, Notebook genannt, im Webbrowser und kann diese dort gleich ausführen.

Eine Notebook-Datei besteht aus vielen verschiedenen Abschnitten. Ein Abschnitt kann enthalten:

- Python-Befehle, die ein Interpreter ausführt, der entsprechende Ergebnisse liefert;
- Text, formatiert mit Markdown und Formeln in LaTex;
- Überschriften.

Die Entwicklungsumgebung ist somit der Webbrowser. Die Ausführung der Python-Befehle läuft in einem Serverprozess, der das Ergebnis wieder an den Browser zurückliefert. Dadurch ist auch eine über verschiedene Rechner verteilte Entwicklung möglich. Die aktuelle Version von Jupyter lässt sich einfach mit dem Befehl `pip` installieren.

```
pip install jupyter
```

Datenanalyse

Um mit Python große Datenmengen aufzubereiten und zu analysieren, haben sich drei Module als sehr nützlich herauskristallisiert:

- NumPy zum schnellen Verarbeiten großer Zahlenmengen;
- Matplotlib für die grafische Darstellung;
- pandas zum Arbeiten mit Datenmengen in unterschiedlichsten Formaten.

```
pip install numpy, mathplotlib, pandas
```

Die meisten Python-Programme oder interaktiven Notebooks, die Daten analysieren, importieren zuerst die drei Module:

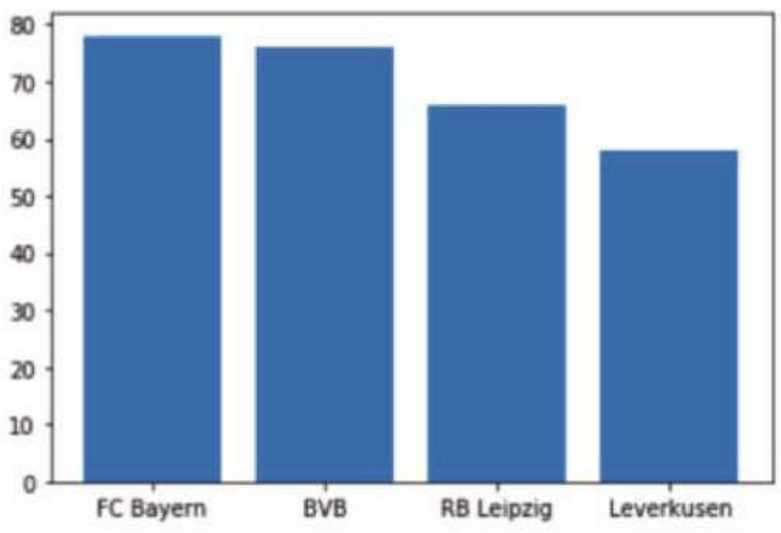
```
import numpy as np
from matplotlib import pyplot as plot
import pandas as pd
```

Da viele Python-Anwender*innen eher pragmatisch arbeiten, wollen sie nicht ständig `numpy` ausschreiben, wenn sie dieses Modul benutzen, sondern verkürzen den Namen zu `np`.

Listen bestehen in Python aus eckigen Klammern, zwischen denen die Elemente durch Kommata getrennt sind.

```
liste = [78, 76, 66, 58]
```

In Python beansprucht eine Zahl etwa 32 Byte internen Speicherplatz. Zusätzlich zum eigentlichen Zahlenwert ist, wie bei jedem anderen Programmobjekt, eine Beschreibung zum verwendeten Datentyp und zusätzlich ein Referenzzähler gespeichert. Der Referenzzähler zeigt an, ob andere Programmobjekte auf das aktuelle verweisen. Ohne dieses Beiwerk reichen beispielsweise für eine Integerzahl 4 Bytes. Bei 1000 Elementen wären dies 4000 Bytes. Bei der klassischen Herangehensweise von Python (ca. 32 Bytes je Zahl) sind es dagegen 32 000 Bytes.



Mit dem Python-Modul Matplotlib lassen sich die unterschiedlichsten Grafiken erstellen (Abb. 3).

	Team	Punkte
1	FC Bayern	78
2	BVB	76
3	RB Leipzig	66
4	Leverkusen	58

Das Objekt pd.DataFrame kann beliebig viele Spalten umfassen (Abb. 4).

Bei vielen Tausend Zahlen, etwa im Zusammenhang mit Big-Data-Anwendungen, schlägt dies natürlich zu Buche.

Abhilfe dafür schafft das Modul NumPy, das Arrays von Zahlen und anderen Elementen mit dieser kompakten Speicherung bietet.

```
y = np.array(liste)
```

Darüber hinaus arbeitet das Modul NumPy bei einer Rechenoperation gleich mit mehreren Zahlen des Arrays (Vektorrechnung) und nicht mit einer nach der anderen. Dies bringt einiges an Geschwindigkeit.

```
>>> y / 78 * 100
array([100.,  97.43589744,  84.61538462,  74.35897436]
```

Ein Array im Modul NumPy kann ebenfalls Zeichenketten enthalten. Wichtig ist allerdings, dass alle Elemente vom gleichen Typ sind.

```
x = np.array(["FC Bayern", "BVB", "RB Leipzig", "Leverkusen"])
```

Grafik

Visualisierung hilft, auch komplexere Datenanalysen schnell zu erfassen. Hier liegt das Einsatzgebiet des Moduls Matplotlib. Es dient dazu, die gängigen Typen an Geschäftsgrafiken, etwa Säulendiagramme (Abbildung 3) oder beliebige selbst erstellte Grafiken, zu erzeugen.

```
plot.bar(x,y)
plot.show()
```

Das gezeigte Beispiel mit den Fußballklubs und den Punkten, die diese am Ende der Bundesliga 2018/2019 hatten, besteht aus

Listing 2: object_example.py

```
import numpy as np
from matplotlib import pyplot as plot
import pandas as pd

import matplotlib.cm as cm

class TeamPlot:
  def __init__(self, team, punkte):
    self.d = pd.DataFrame({'Team' : x,
                           'Punkte' : y}, index=range(1,len(team)+1))

  def print(self):
    print(self.d)

  def plot(self):
    plot.title("Statistik")
    plot.barh(self.d.Team, self.d.Punkte)
    plot.show()


class TeamPlotExt(TeamPlot):
  def __init__(self, team, punkte):
    super().__init__(team,punkte)
    self.map = cm.get_cmap('jet')

  def plot(self):
      plot.title("Statistik")
      plot.barh(self.d.Team, self.d.Punkte, 7
                                      color=self.map(self.d.Punkte))
      plot.show()
```


MAGAZIN FÜR PROFESSIONELLE

***iX* SPECIAL 2020 – MODERNE PROGRAMMIERSPRACHEN**

Postfach 61 04 07, 30604 Hannover; Karl-Wiechert-Allee 10, 30625 Hannover

Redaktion: Telefon: 0511 5352-387, Fax: 0511 5352-361, E-Mail: post@ix.de
Abonnements: Telefon: 0541 80009-120, Fax: 0541 80009-122, E-Mail: leserservice@heise.de

Herausgeber: Christian Heise, Ansgar Heise
Chefredakteur: Dr. Oliver Diedrich (odi@ix.de) -616
Stellv. Chefredakteur: Markus Feilner (mfe@ix.de) -388
Konzeption und redaktionelle Leitung: Nicole Bechtel (nb) -378, Carina Schipper (csc) -384
Leitende Redakteure: Alexander Neumann (ane@ix.de) -813, Bert Ungerer (un@ix.de) -368
Redaktion: Jürgen Diercks (jd@ix.de) -379, Rainald Menge-Sonnentag (rme@ix.de), Matthias Parbel (map@ix.de) -321
Redaktionsassistenz: Carmen Lehmann (cle@ix.de) -387, Michael Mentzel (mm@ix.de) -153
Layout und Satz: Madlen Grunert, Lisa Hemmerling, Kirsten Last, Steffi Martens, Marei Stade, Matthias Timm, Ninett Wagner, Heise Medienwerk, Rostock
Chefin vom Dienst: Barbara Gückel
Korrektorat: Barbara Gückel; Marei Stade, Ricardo Ulbricht, Ninett Wagner, Heise Medienwerk, Rostock
Hergestellt und produziert mit Xpublisher: www.xpublisher.com
Xpublisher-Technik: Melanie Becker, Anna Hager, Pascal Wissner
Titel: Idee: *iX*; Titel- und Aufmachergestaltung: Lisa Hemmerling
Abbildungen © Adobe Stock: Titelmotiv: Adobe Stock; Montage: *iX*;
Bildmotive: Butch (S. 50), Denis Rozhnovsky (S. 72), dottedyeti (S. 26), flashmovie (S. 152), Inga Nielsen (S. 38), Maksym Yemelyanov (S. 98), Maximusdn (S. 6), Maximusdn (S. 30), Maximusdn (S. 34), Maximusdn (S. 110), Maximusdn (S. 118), Media Whalestock (S. 60), muratart (S. 88), pechenka_123 (S. 12), Peter Jurik (S. 84), Peter Jurik (S. 140), science photo (S. 46), sdecoret (S. 66), sdecoret (S. 126), Sergey Nivens (S. 10), Tryfonov (S. 137), tsuneomp (S. 56), zeleniy9 (S. 20), 3000ad (S. 16), ธนพล สินสร้าง (S. 78), ธนพล สินสร้าง (S. 146), (S. 104), Дмитрий Панченко (S. 132)
Verlag und Anzeigenverwaltung:
Heise Medien GmbH & Co. KG, Postfach 61 04 07, 30604 Hannover; Karl-Wiechert-Allee 10, 30625 Hannover; Telefon: 0511 5352-395, Fax: 0511 5352-129
Geschäftsführer: Ansgar Heise, Dr. Alfons Schräder
Mitglieder der Geschäftsleitung: Beate Gerold, Jörg Mühle
Verlagsleiter: Dr. Alfons Schräder
Anzeigenleitung: Michael Hanke -167, E-Mail: michael.hanke@heise.de, www.heise.de/mediadaten/ix
Leiter Vertrieb und Marketing: André Lux -299
Werbeleitung: Julia Conrades -156
Druck: Dierichs Druck + Media GmbH & Co. KG, Frankfurter Straße 168, 34121 Kassel
Sonderdruck-Service: Julia Conrades -156
Verantwortlich: Textteil: Dr. Oliver Diedrich; Anzeigenteil: Michael Hanke
***iX* Special 2020 – Moderne Programmiersprachen:** Einzelpreis: 14,90 €, Österreich 16,40 €, Schweiz 27.90 CHF, Luxemburg 17,10 €

Eine Haftung für die Richtigkeit der Veröffentlichungen kann trotz sorgfältiger Prüfung durch die Redaktion vom Herausgeber nicht übernommen werden. Die gewerbliche Nutzung abgedruckter Programme ist nur mit schriftlicher Genehmigung des Herausgebers zulässig.
Honorierte Arbeiten gehen in das Verfügungsrecht des Verlages über, Nachdruck nur mit Genehmigung des Verlages. Mit Übergabe der Manuskripte und Bilder an die Redaktion erteilt der Verfasser dem Verlag das Exklusivrecht zur Veröffentlichung. Für unverlangt eingesandte Manuskripte kann keine Haftung übernommen werden. Sämtliche Veröffentlichungen in *iX* erfolgen ohne Berücksichtigung eines eventuellen Patentschutzes. Warennamen werden ohne Gewährleistung einer freien Verwendung benutzt.

Printed in Germany
© Copyright by Heise Medien GmbH & Co. KG

Die Inserenten

Alkmene Verlags- und Mediengesellschaft mbH	Frankfurt am Main	S. 79
B1 Systems GmbH	Vohburg	S. 9
BYTEC GmbH	Friedrichshafen	S. 156
dpunkt.verlag GmbH	Heidelberg	S. 35
mitp Verlags GmbH & Co. KG	Frechen	S. 47
Thomas Krenn AG	Freyung	S. 2
Trivadis Germany GmbH	München	S. 27
WIBU-SYSTEMS AG	Karlsruhe	S. 57

Stellenanzeigen

Heise Gruppe GmbH & Co. KG	Hannover	S. 144
TU Berlin	Berlin	S. 144
WDR mediagroup GmbH	Köln	S. 145

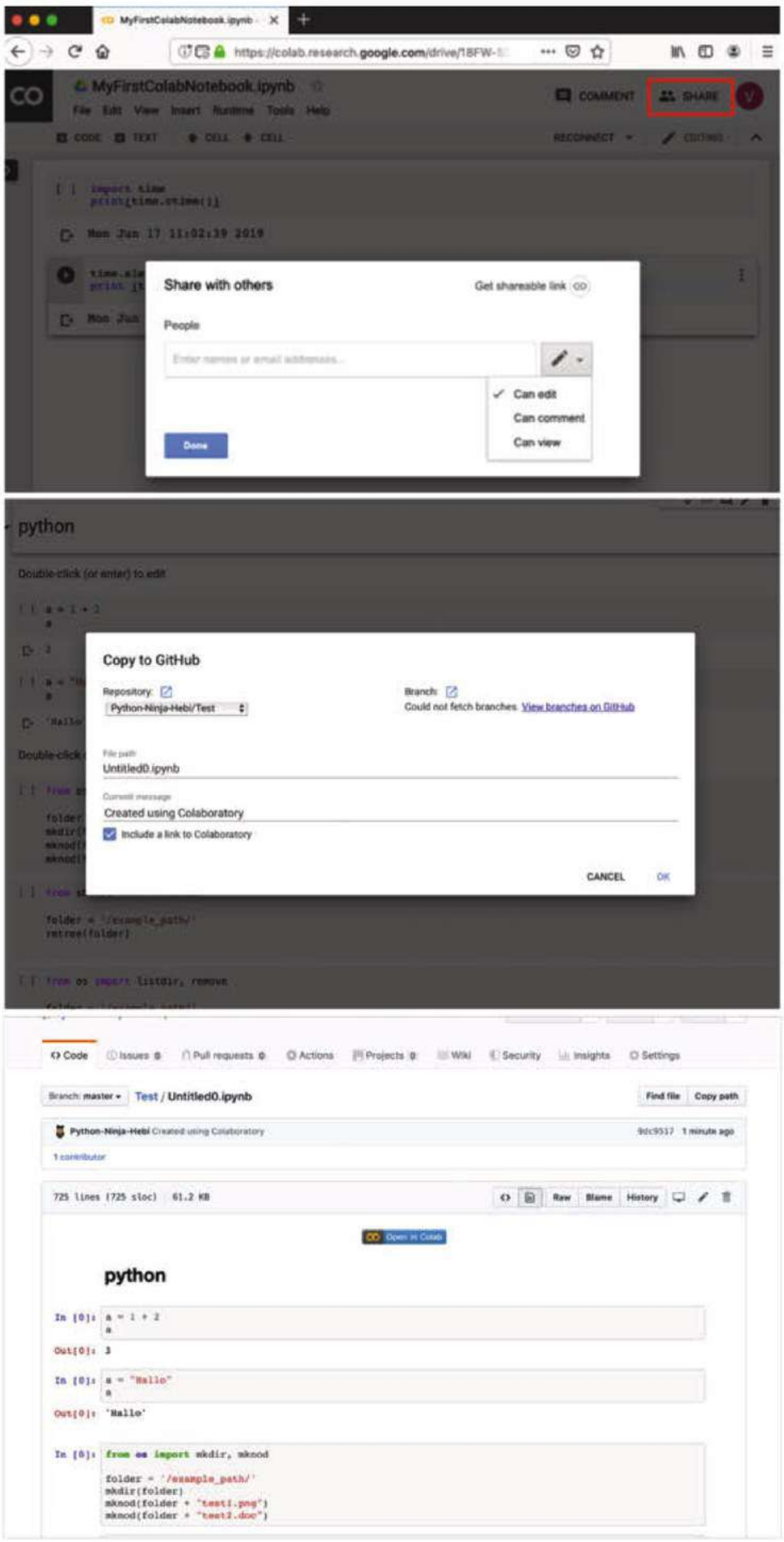
Die hier abgedruckten Seitenzahlen sind nicht verbindlich. Redaktionelle Gründe können Änderungen erforderlich machen.

zwei NumPy-Arrays. Schöner wäre es natürlich, wenn man diese in einer einzigen Tabelle zusammenfassen könnte. Genau das ermöglicht das Modul pandas.

```
d = pd.DataFrame({'Team' : x,
                  'Punkte' : y}, index=[1,2,3,4])
```

Ein Objekt vom Typ `pd.DataFrame` kann sich aus beliebig vielen Spalten zusammensetzen, hier dem Teamnamen und den entsprechenden Punkten (Abbildung 4).

Das Modul pandas kann DataFrames in Tabellenform ausdrücken, darin suchen oder sie anderweitig verarbeiten. Da pandas das dafür Arrays auf Basis von NumPy verwendet, die relativ wenig Speicherplatz in Anspruch nehmen und gleichzeitig mit mehreren Elementen arbeiten, erweist sich dieser Vorgang als sehr effizient.



Auf Plattformen wie Google Colab lässt sich Jupyter Notebook ausführen und mit GitHub teilen (Abb. 5).

Klassen

Python war von Beginn an eine prozedurale Programmiersprache. Es hat sich aber nie anderen Herangehensweisen, wie der objektorientierten Programmierung, verschlossen. Daher gibt es in Python die Möglichkeit, Klassen zu definieren (Listing 2).

Die Klasse `TeamPlot` besteht aus den Methoden `__init__`, `print` und `plot`. Die Methode `__init__` führt Python beim Erzeugen eines neuen Elements vom Typ `TeamPlot` aus.

```
tp = TeamPlot(x, y)
```

`__init__` und genauso alle anderen Methoden eines Objekts weisen als Erstes den Parameter `self` auf. Dieser ermöglicht es, auf das aktuelle Objekt zuzugreifen.

```
self.d = pd.DataFrame(…)
```

`self.d` steht für die Variable `d` des aktuellen Objekts, das im Beispiel einen neuen Datenframe als Inhalt bekommt.

Alle Methoden, die mit zwei Unterstrichen wie `__init__` beginnen, gehören zu den internen, von Python vorgegebenen Methoden. Möchte man selbst eine Methode definieren, die nur innerhalb der Klasse bekannt ist, sollte diese mit einem Unterstrich beginnen, beispielsweise die Methode `_nur_intern`.

Doch Achtung, Python stellt nicht sicher, dass diese Zugriffsregeln strikt eingehalten werden. Ein Sicherheitskonzept mit der Definition von zum Beispiel „private, protected und public“, wie in anderen Programmiersprachen üblich, gibt es nicht, denn man will eben alles so einfach wie möglich halten.

Möchten Entwickler*innen die Klasse `TeamPlot` erweitern, ziehen sie dazu beispielsweise eine abgeleitete Klasse `TeamPlotExt` heran. Darin können sie neue Methoden erstellen oder vorhandene überschreiben. Diese neue Klasse liefert im Beispiel die Säulendiagramme mit unterschiedlichen Farben. Daneben lässt sich eine Klasse in Python nicht nur von einer anderen Klasse, sondern von mehreren ableiten.

```
class TeamPlotExt(TeamPlot):
```

Das Erforschen großer Datenmengen im Blick

Durch das Projekt Jupyter und die dazugehörigen Notebook-Dateien hat Fernando Pérez die Programmiersprache Python um eine andere Herangehensweise erweitert. Forscher*innen oder Analyst*innen wie Pérez kommen aus der Data Science, der Kultur des Erforschens großer Datenmengen. Im Gegensatz dazu erstellen klassische Softwareentwickler*innen ein Programm, debuggen es und liefern es bestenfalls fehlerfrei an Anwender*innen aus, die es möglichst lange nutzen. Zu diesen Softwareentwickler*innen gehört der ursprüngliche Erfinder von Python, Guido van Rossum.

Dieser neue Bereich der Data Science hat in den letzten Jahren in Python sehr stark zugenommen. Bedingt durch die Ausbreitung des Internets entstanden in allen Ecken des täglichen Lebens Datenmassen, für die man ein Werkzeug zur Analyse braucht. Der Hype um das maschinelle Lernen hat die Zahl der Python-Nutzer*innen zusätzlich erhöht, da auch das genau in diesen Bereich des Analysierens und Forschens passt.

Durch die Trennung der Oberfläche im Browser und der Verarbeitung in einem Serverprozess lässt sich Jupyter Notebook

ohne eine Installation von Python auf dem eigenen Rechner ausführen. Beispielsweise bietet Google mit Colab die Möglichkeit, sich im Netz anzumelden und dort über seinen Browser Jupyter Notebook zu erstellen und abzuarbeiten (Abbildung 5).

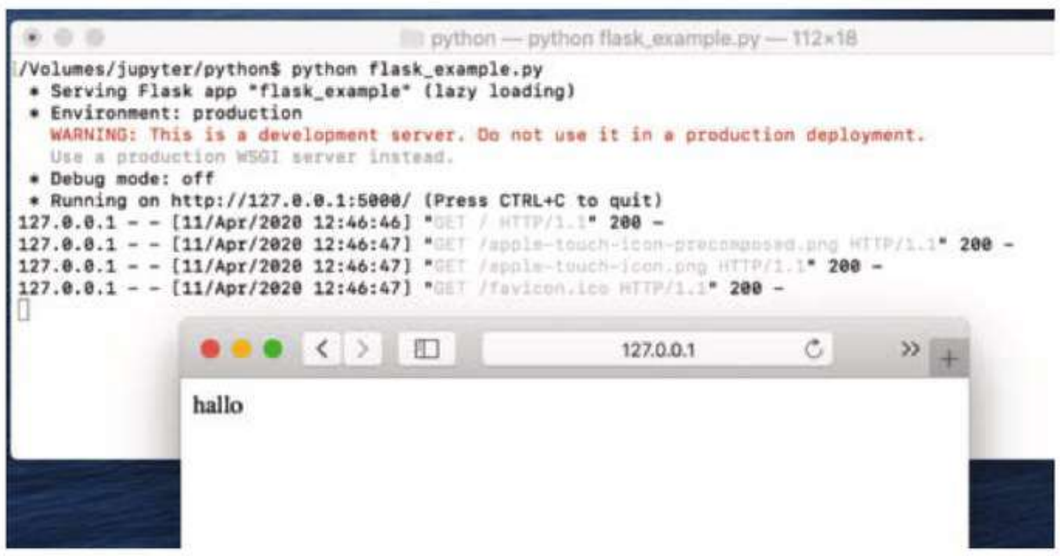
Dekade 3: Python im Web

Im Jahr 2009 veröffentlichte Armin Ronacher sein Web-Framework Flask, mit dem man mit einigen Zeilen Python-eigene Webservices erstellen kann (Abbildung 6). Zunächst war das Ganze als Aprilscherz gedacht. Armin Ronacher hatte bereits einige häufig genutzte Python-Module programmiert, wie das Modul Werkzeug für die Webkommunikation zwischen Servern und Anwendungen über WSGI (Web Server Gateway Interface, gesprochen Whiskey). Viele machten zu dieser Zeit irgendein „Framework“. Also fasste er seine Module zu einer Datei zusammen, schrieb eine umfassende Dokumentation dazu und fertig war das Web-Framework Flask. Jetzt mussten Python-Entwickler*innen keine neue Sprache mehr lernen, wenn sie im Web programmieren wollten.

Die Beispielanwendung zu Flask (Listing 3) hier ist ein einfacher Webservice. Rufen Entwickler*innen ihn auf, bekommen sie in einem Webbrowser „Hallo“ als Antwort zurück.

Auf dem Server startet man den Service wie jedes andere Python-Programm mit

```
>python flask_example.py
```



Mit dem Framework Flask lassen sich in Python eigene Webservices erstellen (Abb. 6).

Falls der Webservice auf demselben Rechner wie der Browser läuft, lässt er sich standardmäßig aufrufen über

```
http://127.0.0.1:5000.
```

Das Beispielprogramm für Flask erzeugt als Erstes ein Objekt `app` von der Klasse `Flask`, über das die komplette Kommunikation zwischen Service und dem Webclient abläuft.

```
from flask import Flask
app = Flask(__name__)
```

Da die Kommunikation über ein Objekt stattfindet, lässt sich einfacher skalieren. Braucht man beispielsweise bei einer gro-



Listing 3: flask_emaple.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hallo():
    return 'hallo'

@app.route('/<name>')
def hallo_name(name):
    return 'hallo ' + name

if __name__ == '__main__':
    app.run()
```

Listing 4: folder_example_typed.py

```
from os import listdir, remove

folder: str = 'c:/example_path/'


def delete_files(folder: str, extension: str = '.tmp') -> None:
    file: str
    for file in listdir(folder):
        print(file)
        if file.endswith(extension):
            remove(folder + file)
            print(f'{file} gelöscht')
```

ßen Webseite mehrere Tausend Verbindungen gleichzeitig, kann diese einfach mehrere Tausend `app`-Objekte erzeugen. Die Architektur der Anwendung muss man deshalb nicht grundlegend verändern.

```
@app.route("/")
def hallo():
    return "hallo"
```

Im Python-Programm definieren Entwickler*innen, welche Funktion der Webservice im Browser bei der Eingabe eines bestimmten Pfads ausführt, hier „/“. Dafür zuständig ist der Function Decorator (`@app.route`), der direkt die Ausführung der Funktion beeinflusst und immer vor der Funktionsdefinition steht.

```
if __name__ == "__main__":
    app.run()
```

Python startet den Service über die Methode `app.run.` am Ende der Dateien.

Die Idee hinter Flask lautet, eine Webseite aus möglichst kleinen Services (Microservices) zusammenzubauen. Dadurch soll alles möglichst übersichtlich bleiben und leicht zu warten sein.

Zukunft: Herausforderungen

Da die Programmiersprache Python in den derzeit expandierenden Bereichen wie Webentwicklung, Datenanalyse mit maschinellem Lernen und Scripting Verwendung findet, ist sie auf allen Hitlisten vorne mit dabei. Wird das so weitergehen oder hat sie ihren Zenit überschritten? Eine Herausforderung wird sein, ob Entwickler*innen Python in wirklich großen Projekten, bestehend aus mehreren Hunderttausend Programmzeilen, wirtschaftlich einsetzen können.

Die einfache Herangehensweise von Python hat in kleinen Projekten Vorteile, kann aber bei großen nachteilig sein, etwa wegen der nicht vorhandenen expliziten Typendeklaration. Bei großen Projekten ist es grundsätzlich besser, wenn der Compiler bereits während der Entwicklung einen möglichen Fehler erkennt, als wenn dieser nach der Auslieferung bei Anwender*innen auftritt. Dafür braucht der Compiler aber mehr Wissen über das Programm, beispielsweise konsequente Typendeklaration.

Mehr Möglichkeiten

Ab der Version 3.6 haben die Kernentwickler*innen von Python bereits die Möglichkeit geschaffen, Variablen oder andere Programmobjekte mit Typinformationen zu versehen (Type Hints).

```
folder: str = 'c:/example_path/'
```

Bei einer Variablenzuweisung kann man nach einem Doppelpunkt den gewünschten Typ angeben.

```
def foo(p: float) -> float:
    return p/2
```

Bei einer Funktionsdefinition bekommt ein Parameter ebenfalls nach einem Doppelpunkt eine Typinformation. Der Rückgabewert lässt sich nach einem Pfeil `->` mit einem Typ versehen.

Wie das Beispielprogramm zum Löschen der Dateien mit Typdeklaration aussieht, zeigt das Listing 4. Würde das Programm einen Typfehler enthalten und mit dem Python-Interpreter ausgeführt, würde überraschenderweise neben dem produzierten Laufzeitfehler keine zusätzliche Fehlermeldung angezeigt, da Python aktuell die Type Hints nicht beachtet.

Spannend wird das Ganze, wenn zusätzliche Programme wie Type Checker oder Entwicklungsumgebungen wie PyCharm daraus Nutzen ziehen. Mit diesen können Entwickler*innen vor der Auslieferung das Programm auf mögliche Fehler prüfen lassen.

Neben dem Bereich der professionellen Softwareentwicklung wird in Zukunft die Entwicklung für die unterschiedlichen Plattformen immer wichtiger werden. Angefangen bei Smartphone-Apps bis hin zum Internet of Things. Der stärkste Konkurrent ist hier JavaScript, das nicht nur in Webbrowsern läuft, sondern für das es bereits Umgebungen für Desktop-Anwendungen und Smartphones gibt.

Im Kasten „Sehenswertes Videomaterial zum Thema“ finden sich Vorträge zu Python und Jupyter Notebook. (csc@ix.de)

Sehenswertes Videomaterial zum Thema

- Stanford Seminar: Jupyter Notebooks and Academic Publication
- Armin Ronacher: „Flask for Fun and Profit“, PyBay2016
- Armin Ronacher: „Developing an Open Source Library“
- Russell Keith-Magee: Keynote der PyCon 2019
- Nico Kreiling: „PyData-Workflow mit Jupyter Lab“

Die Videos stehen unter ix.de/zqfe zur Verfügung.

Quellen

Alle Listings, Tools und Videos: ix.de/zqfe



Gerhard Völkl

ist Fachjournalist für Softwareentwicklung, Data Science und Computergrafik.

WERDEN SIE DATENEXPERTE!

Starten Sie jetzt Ihre zertifizierte Weiterbildung

Schnell und flexibel ans Ziel!

- Alle Grundlagen anschaulich vermittelt
- Lernen anhand realer Business-Cases
- Praxisübungen mit echten Datensets
- Persönliches Coaching durch Mentoren
- Flexibilität für berufsbegleitendes Lernen
- Qualifizierung mit anerkanntem Zertifikat



DATA AWARENESS (4 Wochen)

- ganzheitlicher Überblick über alle wichtigen Datentechnologien
- keine Vorkenntnisse benötigt
- in deutscher und englischer Sprache



DATA ANALYST (3 Monate)

- Daten strukturieren, analysieren und visualisieren
- in kürzester Zeit Python erlernen
- keine Vorkenntnisse nötig
- in deutscher und englischer Sprache



DATA SCIENTIST (ca. 6 Monate)

- alle Kompetenzen für den Berufsalltag als Data Scientist erlernen
- Implementierung und Optimierung von Machine-Learning-Algorithmen
- in deutscher und englischer Sprache

Fordern Sie jetzt Informationen an:

Heise Medien

heise.de/datenexperte



Raku verstehen und anwenden

Ganz natürlich

Herbert Breunung

Raku ist eine vielseitige, graduell typisierte Allzwecksprache, die mehrere Paradigmen erlaubt und im Normalfall zu Bytecode kompiliert wird. Syntax und Semantik lehnen sich an Perl an.

Die Stärken von Raku liegen in der umfassenden Implementierung von Unicode, Parallelität und Asynchronizität, dem Parsen von Text mithilfe von Grammatiken und einer weitreichenden Metaprogrammierung. Bei der Gestaltung wurde besonderer Wert auf die Umsetzung linguistischer Prinzipien gelegt, die von natürlichen Sprachen abgeleitet sind.

Details wie Wortwahl, Groß- und Kleinschreibung sowie Reihenfolge der Teilausdrücke sind absichtlich so geformt, dass Programmierer*innen sich individuell, präzise, kompakt und natürlich (wie in einfachem Englisch) ausdrücken können, ohne dass es zu Zweideutigkeiten kommt. Allerdings hielt sich Larry Wall nicht streng an das Prinzip der geringsten Überraschung (POLS), sondern brach auch mit etablierten Konventionen, wenn sie ihm unintuitiv oder unlogisch erschienen. So änderte er einen beträchtlichen Syntaxanteil regulärer Ausdrücke gegenüber Perl, um die Regeln in sich konsistent und mit dem Rest der Sprache einheitlich zu gestalten.

Wie für Perl waren Flexibilität und Effektivität bei wiederkehrenden Aufgaben wesentliche Entwurfsziele. Raku ist umfangreicher als Perl. Um das Erlernen zu erleichtern, wurden die gleichen Sprachelemente in Schreibweise und Bedeutung so oft wie möglich wiederverwendet und so viel Komplexität wie möglich verborgen. So lässt sich eine Datei ohne `handle` auslesen:

```
$inhalt = slurp 'gedicht.txt'
```

Die Grundausstattung

Das Format ist frei, eingerückt wird nach Empfinden. Allerdings sollen Variablen, Operatoren und Schlüsselwörter von Leerzeichen umgeben sein. Kommentare bis zum Zeilenende beginnen weiterhin mit #. Neu sind dagegen mehrzeilige Kommentare:

```
#`( alle Klammerarten
    können Kommentare eingrenzen )
```

Um strukturierte Inhalte mit Listen und Überschriften einzufügen, die sich nach LaTeX und HTML konvertieren lassen, gibt es die

Auszeichnungssprache POD. Sie wurde ebenfalls generalüberholt, aber ihre Schlüsselwörter beginnen immer noch mit =.

```
=begin comment
    Wer kein ` tippen kann oder will,
    nimmt POD für mehrzeilige Kommentare.
=end comment
```

Das Semikolon trennt Anweisungen, daher ist das letzte im Block optional. Geschweifte Klammern bilden Blöcke. Vor den Klammern stehen Schlüsselwörter wie `if`, `while`, `sub` oder `method`, um sie von Hashliteralen zu unterscheiden. Runde Klammern gruppieren Teilausdrücke, steuern ihre Präzedenz und umgeben Signaturen. Nach Befehlen wie `if` und `while` sind sie optional.

Stringliterale sitzen in einfachen Anführungszeichen '. Doppelte Anführungszeichen "" interpolieren Steuerungszeichen, Variablen und Codeblöcke. In spitzen Klammern sind durch Leerzeichen getrennte Wörter einzeln quotiert: `<Moe Joe Larry>` entspricht `'Moe', 'Joe', 'Larry'`.

Kommata sind Operatoren, die eine Liste von Werten erzeugen, die durch sie getrennt sind. Variablen erkennt man wie in Perl an einem einleitenden Sonderzeichen, der Sigil: `$ Skalar` enthält einzelne Werte, `@ Array` ist numerisch indizierbar und `% Hash` bezeichnet ein assoziatives Array. Die Zeichen @ und % lassen sich vollständig vermeiden, denn Arrays und Hashes sind Objekte, die man auch in einem Skalar ablegen kann. An der Verwendung ändert sich dadurch nichts. Die Sigil bleibt beim Zugriff auf Array- oder Hashelemente unverändert.

Arrayindizes schreibt man in eckigen Klammern: `@rray[2]` oder `@rray[2,3]`. Dem folgend haben Arrayliterale die Form `[1,2,3]` und `@(1,2,3)`. Hashindizes hingegen sind in geschweifte Klammern gesetzt. Um Anführungszeichen zu sparen, verwenden Entwickler*innen meist `%ash<index>` statt `%ash{'index'}` (siehe `<> alias qw`).

Analog zum Hashindex stehen Hashliterale in geschweiften Klammern: `{ ... } alias %( ... )`. Die drei Punkte deuten Listen von Paaren an, die sich wie in Perl `key => 'value'` oder `:key<value>` schreiben lassen.

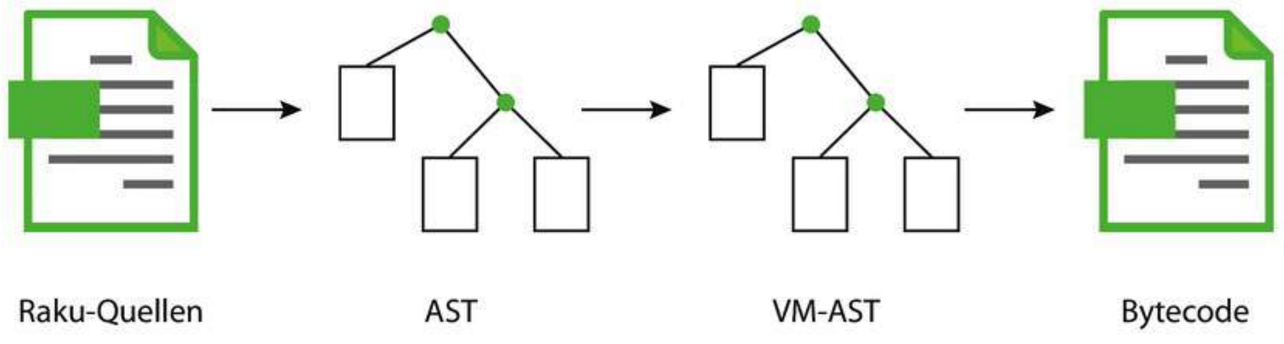
Nach der Sigil steht bei besonderen Variablen eine sekundäre Sigil (Twigil = Tweak + Sigil), die die Funktion hervorhebt. Die häufigsten Twigils sind . und ! für Objektattribute, * in globalen Variablen, wie die Argumente an das Programm `@*ARGS` oder die Standardausgabe `$*OUT`. Des Weiteren ? für Fragen an den Compiler, wie beim Dateinamen des aktuellen Programmteils `$?FILE` oder der Nummer der aktuellen Codezeile `$?LINE`.

Der Weg des Kamels

Larry Wall entwarf Raku als Perl 6, eine Weiterentwicklung von Perl 5. Er gestattete sich einmalig, sämtliche Kompatibilitäten zu brechen. Die erste Ankündigung im Juni 2000 leitete eine mehrjährige Phase der Ideenfindung ein, weswegen ein erster in Haskell geschriebener Prototyp erst 2005 erschien. Betaversionen (Perl 6.b) gab es ab Juli 2010 und die erste offizielle Vollversion Weihnachten 2015 (6.c Christmas). Aktuell ist seit November 2018 die Version 6.d Diwali.

In einem öffentlichen Wahlverfahren stimmte im Dezember 2019 die Mehrheit der Entwickler*innen und Nutzer*innen für eine Umbenennung zu Raku, einer Verkürzung von Rakudo, dem De-facto-Standardcompiler mit eigener virtueller Maschine (MoarVM), der auf der Kommandozeile noch „perl6" heißt (siehe Abbildung). Programm, Dokumentation und wesentliche Informationen sind unter raku.org erhältlich.

Damian Conway taufte Rakudo bereits Mitte der 2000er-Jahre mit Anspielung auf das Perl-Maskottchen, ein Kamel. Auf Japanisch bedeutet Rakuda-dō „der Weg des Kamels". Zudem wird Rakudo mit Himmel und Raku mit Freude übersetzt.



Rakudo ist in der einfachen Sprache NQP (Not Quite Perl) geschrieben und kompiliert Raku mit zwei Zwischenstufen zu Bytecode. Das erleichtert es, Rakudo auf andere Plattformen zu portieren. Die Ausgabe von JVM-Bytecode funktioniert bereits.

Die meisten Variablen sind lexikalisch lokal mit `my` zu deklarieren, wodurch ihre Lebensdauer auf das Ende des aktuellen Blocks begrenzt ist. Subroutinen (Funktionen) erkennt man am Schlüsselwort `sub`. Sie lassen sich wie Lambdas verwenden, unabhängig vom Weglassen des Namens. `use` fordert das Laden von Modulen (Paketen) und die Sprachversion – `use v6.c;`.

Objekte bis ganz unten

Das minimale Programm `say 'servus'` kommt ohne Klasse und `main`-Routine aus. Auch ohne sichtbare Objektorientierung lässt sich viel erreichen. Trotzdem ist sie für ein tieferes Verständnis essenziell. Denn Bestandteile der Sprache wie Literale, Variablen, Typen, Funktionen, Signaturen, Schleifen und reguläre Ausdrücke sind Objekte, die den gleichen Spielregeln unterliegen wie selbst geschriebene. Die Vererbungshierarchie dieser auch in Raku geschriebenen Klassen ist das zentrale, ordnende Prinzip der Interna.

Obwohl Klassen die Schablonen für die Erzeugung von Objekten sind, bekommt jedes Objekt automatisch auch eine Methode namens `clone`, die eine identische Kopie generiert:

```
my $clone = $object.clone; # kein tiefer Klon
```

Das ermöglicht eine prototypbasierte Objektorientierung wie in JavaScript. Objektorientierung enthält außerdem mehrfache Vererbung, autogenerierte Getter/Setter, positionale und benannte Parameter in Methodensignaturen, das Überladen von Methoden und die Delegation von Methodenaufrufen auf Methoden der Attribute.

Der autogenerierte Standardkonstruktor `Magazin.new` leitet den Inhalt des benannten Argumentes `$name` in das gleichnamige Attribut vom Typ `Str` (Listing 1; alle Listings siehe ix.de/zpcs). Anhand der Deklaration `has Str $.name` erzeugt der Compiler die

iX-TRACT

- Die von Larry Wall entwickelte Sprache Raku ist eine Weiterentwicklung von Perl 5.
- Im Jahr 2019 änderte sich der Name von Perl 6 in Raku – eine verkürzte Form des Compilers Rakudo.
- Kennzeichen der Sprache sind Metaprogrammierung und das Verwenden von Grammatiken.

Listing 1: Vererbung und Nutzung eines Attributes

```
use v6.d;

class Magazin    { has Str $.name }

class IX is Magazin {

    method new () {
      self.bless( name => 'iX');
    }

    multi method title (Int $nr)   {
      "$.name Ausgabe $nr"
    }
    multi method title (Str $name) {
      "$.name Sonderausgabe $name"
    }
}

my $mag = Magazin.new( name => 'Make');
say $mag.name;      # gibt 'Make' aus
my $ix = IX.new();
say $ix.name;       # gibt 'iX' aus
say $ix.title(5);   # 'iX Ausgabe 5'
say $ix.title('Sprachen');
# 'iX Sonderausgabe Sprachen'
```

Listing 2: Rolle implementieren und anwenden

```
role Summary {
    has @.article;
    method title ()   {...}
    method summary () {
      say self.title;
      for @.article -> $article {
        say "$article.author - $article.title"
      }
    }
}

class IX is Magazin does Summary {
..
IX.new.summary; # gibt Zusammenfassung aus
```

Listing 3: Typprüfungen während der Zuweisungen

```
my $höhe = 32;
say $höhe.WHAT;           # (Int)
$höhe = 'dreiundzwanzig'; # kein Typfehler
say $höhe.WHAT;           # (Str)
my Int $weite = '33';     # ! '33' ist Stringobjekt
```

Methode `get` des Attributs: `Magazin.name`. Mit dem Anhang `... is rw;` wären sogar Zuweisungen erlaubt (`rw` steht für read write):

```
$mag.name = "c't" # lvalue-Zuweisung
```

Da private Methoden mit ! beginnen, heißt der private Accessor `$!name`. Folglich deklariert `has Str $!name;` ein privates Attribut. Die Klasse `IX` erbt von `Magazin` das Attribut und implementiert einen eigenen Konstruktor. Darin wird das neue Objekt in Anlehnung an Perl per `bless` erzeugt und der Attributwert gesetzt. Die Methode `title` ist als `multi` implementiert, um auf verschiedene Eingaben reagieren zu können (in diesem Beispiel nur Illustration). Da das Argument `'Sprachen'` vom Typ `String` ist, ruft `$ix.title('Sprachen')` die untere Alternative auf, die `"$.name Sonderausgabe $name"` liefert.

Der wohl interessanteste Aspekt der Raku-Objektorientierung sind Rollen. Damit lassen sich Schnittstellen und abstrakte Klassen darstellen. Hauptsächlich „heilen“ sie aber überkomplexe Vererbungshierarchien und zirkuläre Abhängigkeiten. Wie Traits oder Mixins in anderen Sprachen besitzen Rollen Attribute und Methoden, sind aber nicht instanziierbar. Klassen erben von ihnen. Dabei auftretende Namenskonflikte erkennt der Compiler bereits zur Kompilierzeit und erzeugt Fehler, es sei denn, die erbende Klasse fängt sie ab. Bindet man die Rolle erst zur Laufzeit an eine Klasse oder ein Objekt, geschieht das später. Da die Vererbung von Klasse zu Klasse zur Kompilierzeit stattfindet, ist sie statisch, wie vom Schlüsselwort `is` angedeutet. Demgegenüber kündigt `does` die Erbschaft von Rollen an, da sie dynamisch sein kann, so wie das Wort Rolle vorübergehendes Verhalten ausdrückt (Listing 2).

Die Rolle `Summary` vererbt `@.article` und `summary`, verlangt jedoch dafür, dass die erbende Klasse eine Methode `title` besitzt. Das geschieht durch den Yada-Operator ..., der eine Methode als Stummel (Stub) markiert und auch optisch auf fehlenden Quellcode hinweist. Stubs sind von der Klasse zu überladen, um Fehlermeldungen zu vermeiden. Eine nur aus Stubs bestehende Rolle fungiert als Schnittstelle. In Raku lassen sich Stubs und Implementation frei kombinieren. Rollen dürfen einen oder mehrere Parameter besitzen, die erst beim Verwenden der Rolle nachzureichen sind. Ist der Parameter ein Datentyp, lässt sich damit ein abstrakter Datentyp implementieren.

Da Raku versucht, Nutzern größtmögliche Freiheiten zu bieten, gibt es Typprüfungen zur Kompilierzeit, zur Laufzeit und in mehreren Phasen dazwischen. Dieses Verhalten fasst das Schlagwort „graduelle Typisierung“ zusammen. Da Perl-Nutzer es nicht gewohnt sind, Typen zu verwenden, kommt die Sprache diesem Programmierstil entgegen. Der Datentyp `Cool` akzeptiert Strings und Zahlen aller Art und erlaubt quasi typfreies Programmieren innerhalb einer streng typisierten Umgebung.

Eine Frage des Typs

Das Integerliteral `32` ist ein Wertobjekt, wie es auch `Int.new(32)` erzeugt, denn `32.say` entspricht `say 32`. Dieses Wertobjekt wird in Zeile 1 von Listing 3 dem Containerobjekt `$höhe` zugewiesen. Der Container ist durch seine Deklaration ohne Typangabe an keinen Typ gebunden. Deshalb erzeugt nur die Zuweisung an `$weite` den Fehler „Type check failed in assignment ...“.

Werteobjekte sind unveränderlich. Das erlaubt wichtige Compileroptimierungen, die sonst aus Sprachen wie Haskell bekannt sind, wo Variablen nie ihren Inhalt ändern. Raku kombiniert das mit der Flexibilität veränderbarer Variablen und versucht, scheinbare Gegensätze aufzuheben.

Der Typ `Int` darf auf Kosten der Geschwindigkeit über die Registergröße und bis zum Speicherlimit wachsen. Für hardwarenahe Programmierung und um Schnittstellen zu kompilierten Bibliotheken per Foreign Function Interface – FFI – zu schreiben, gibt es Spezialtypen, für die strenge Größenbeschränkungen gelten. Um eigene Typen von bestehenden abzuleiten, muss man keine Klasse schreiben:

```
# mit Fingern abzählbare Zahlen
subset Finger of Int where 0 <= * <= 10;
```

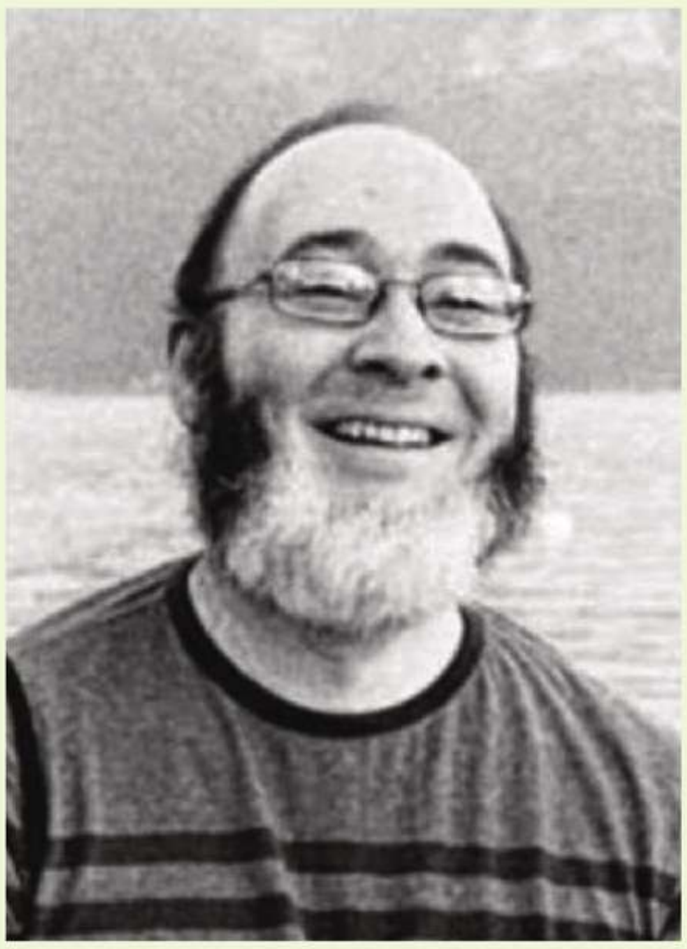
Die Typ-Klassen-Hierarchie ist reich und enthält neben Erwartbarem wie Int, Str, Real, Bool, List auch Datumsangaben (DateTime), Datenpuffer (Buf), binäre Daten (Blob), Unicode Codepoints (Uni), Sequenzen (Seq), Mengen (Set), gewichtete Mengen (Bag), logische Quantenzustände (Junction), komplexe Zahlen (Complex), Zahlenbereiche (Range) und rationale Zahlen (Rat). Letztere enthalten ganzzahlige Nenner und Zähler und erlauben Grundrechenoperationen ohne vermeidbare Rundungsfehler:

Typen sind das Herz von Raku

Jonathan Worthington, Chefentwickler des Raku-Compilers Rakudo, des Web-Frameworks Cro und der Raku-IDE Comma, erläutert im Interview mit *iX*-Autor Herbert Breunung die Besonderheiten der Programmiersprache.

Herbert Breunung: Sind Perl und Raku Sprachen mit menschlichem Antlitz?

Jonathan Worthington: Ja, nicht zuletzt wegen des großen betriebenen Aufwandes, damit alles funktioniert und Problemstellungen so kommuniziert werden, wie Leute es erwarten. Zum Beispiel sagen wir nicht einfach „Syntax Error!“, sondern versuchen herauszufinden, was gemeint war und welche Änderung das Problem beheben würde.



Eine umfangreiche Sprache zu sein ist Teil des Konzeptes?

Von Menschen gesprochene Sprachen sind auch umfangreich, nicht zuletzt da sie für vielfältige Probleme eingesetzt werden. Ich denke, ein Grund, warum Raku oft als groß empfunden wird, ist die unkenntliche Trennlinie zwischen Sprache und Standardbibliothek. Eine Gruppe von Mengenoperationen passt als Operatoren in die Sprache. Man muss sich nicht um sie kümmern, es sei denn, man braucht sie. So wie man auch nicht die Typen der Standardbibliothek auswendig lernt.

Auch ohne die erweiterbaren Bestandteile hat Raku viel Syntax, weil die Designer*innen sich davon Vorteile versprechen. Die Syntax zur asynchronen Verarbeitung von Datenströmen und die Grammatiken nehmen Programmierer*innen viel Gehirnarbeit ab. Letztlich geht der Trend zu umfangreicheren Sprachen wie C#, Java und JavaScript, die alle mit der Zeit stark anwuchsen.

Was uns zur parallelen und asynchronen Programmierung bringt, wo wir ältere und neuere Konzepte mit einigen innovativen Details sehen.

Für beides wird gesorgt, denn es lässt sich nicht leugnen, dass Programmierung heute so aussieht. In guter Perl-Tradition haben wir Ideen von überall genommen, hatten aber auch eigene. So war der Befehl `await`, der auf asynchrone Operationen warten lässt, bereits weit verbreitet. Seinen lästigen Zwilling `async` (vergleichbar mit Goroutines in Go) unterstützen wir aber auch.

Parallelität und Nebenläufigkeit sind verschiedene Dinge und haben unterschiedliche Bedürfnisse. Beispielsweise gibt es eine parallele Listenverarbeitung per `map` und `filter`, die sich automatisch über Threads verteilen lässt. Da die Laufzeitumgebung für parallele Threads entworfen wurde, können wir darin eine gute Effektivität erreichen.

Welche Aufgaben erfüllen Typen in Raku?

Es gibt verschiedene Arten von Typen. Sie sind für Multimethoden (MMD) sehr nützlich, also Gruppen von Methoden mit gleichem Namen, aber unterschiedlichen Argumenttypen. Sie bilden das Herz der Sprache. Auch Operatoren nutzen MMD, was es ermöglicht, Werte verschiedener Typen zu addieren. Und wenn das nicht reicht, erlaubt es das Metamodell, eigene Typen zu definieren.

Raku nutzt Klassen, um alle Interna zu organisieren. Und dank des Metaobjektprotokolls (MOP) lassen sich Klassen schreiben, die das Verhalten von Klassen ändern. War das inspiriert vom Common Lisp Object System (CLOS), und wie weit reichen die Möglichkeiten?

Ja, CLOS und Smalltalk waren eine große Inspiration, wie auch ein Papier über den Aufbau eines Metamodells mit Traits (Rollen). Das MOP ist die Antwort auf Fragen wie „Was bedeutet es, eine Methode aufzurufen?“ oder „Wie funktioniert Vererbung?“. Jeder Sprachbestandteil hat seine Metaklasse, aber es lassen sich auch eigene schreiben. Ein Monitor ist eine Art Klasse, deren Methoden je Instanz nur von einem Thread gleichzeitig ausgeführt werden können.

Das ändert die Bedeutung, eine Methode aufzurufen, wenn wir erst einen Lock anfordern, sie dann ausführen und zum Schluss den Lock lösen. Damit ist es erlaubt, Verhalten und Metaverhalten zu unterscheiden. Auch Red, ein Object-relational Mapping für Raku, benutzt dieses Vorgehen, um Konflikte zwischen Methodennamen des Programms und dem darzustellenden Modell zu vermeiden und dynamisch Typen zu erzeugen. Das ließe sich zwar auch mit Codegeneratoren lösen, bringt aber wieder eigene Probleme mit sich.

Mit Metaprogrammierung entsteht erst gar kein generierter Code, den jemand aus Versehen bearbeiten könnte. Ich vermute, wir kratzen da noch an der Oberfläche. Das Begeisterndste ist, Leute zu sehen, die das auf unvorhergesehene Art verwenden.

Einer von Rakus Tricks: Die Sprache borgt dem Nutzer ihren Parser, um ihn YAML einlesen zu lassen und die Sprache zu erweitern. Macht sie das zu einer flüssigen Sprache?

Tatsächlich ist die Grammatik, die die Syntax von Raku definiert und einliest, in Raku geschrieben. Reguläre Ausdrücke stoßen an ihre Grenzen, sobald sie rekursive Strukturen erfassen sollen, und werden auch sonst schnell schwer wartbar. Grammatiken lösen diese Probleme und skalieren sehr gut von einfachen Aufgaben bis zum Einlesen ganzer Programmiersprachen. Wird der gleiche Mechanismus benutzt, um Raku einzulesen und nutzerdefinierte Grammatiken umzusetzen, öffnet das die Tür, um damit auch die Sprache zu erweitern oder andere Sprachen einzubetten. Das ist noch experimentell, da der neuen Syntax auch eine Bedeutung (Semantik) gegeben werden muss. Ich arbeite gerade daran, ein Document Object Model (DOM) zu schaffen, das sich für Raku und eigene Erweiterungen verwenden lässt. Damit sind alle Zutaten für diese Funktionalität beisammen.

Die Sprache begeistert mich wegen der Probleme, die ich damit lösen kann – eher noch, als die Sprache zu erweitern. Das DOM eines regulären Ausdruckes (Regex), der die Eingabe in ein (HTML-)Form-Feld validieren soll, lässt sich zu etwas für HTML5-Musterattribute umwandeln. Und das Web-Framework kann dafür sorgen, dass bereits der Client die Validierung vornimmt. Das ist eine kleine, aber hilfreiche Sache. Darum drehen sich letztlich viele Dinge.

Listing 4: Parallele Auswertung zweier Dateien

```
use JSON::Tiny;

sub MAIN( $datei1, $datei2 ) {
    my $promise1 = start from-json( slurp( $datei1 ) );
    my $promise2 = start from-json( slurp( $datei2 ) );
    my ($inhalt1, $inhalt2) = await $promise1, $promise2;
    say $inhalt1 eqv $inhalt2 ?? 'Dateiinhalte identisch' ,
                              !! 'Dateiinhalte verschieden';
}
```

Listing 5: Asynchrone Dateiüberwachung mit zwei Supplies

```
my $rakuchange = supply {
    whenever IO::Notification.watch-path($dir) -> $file {
      emit $file.path if $file.path ~~ /<.pm .p6> $/
    }
}
react {
    whenever $rakuchange -> $path {
      say "Raku Datei $path geändert!"
    }
}
```

```
my Rat $r = 0.2;
say $r.nude; # Ausgabe: (1 5)
```

`nude` steht für NUmerator, DEnumerator (Zähler, Nenner). Die Präfixe ?, ~ und + wandeln in die wichtigsten Typen Bool, Str und Numeric (Rolle aller Zahlen-Typ-Klassen) um. Sie sind syntaktischer Zucker für Konvertierungsmethoden, die den Namen des Zieltyps tragen: `? $antwort` ruft lediglich `$antwort.Bool` auf. Aber sie sind auch mit den Infix-Operatoren + und ~ konsistent. Denn das normale Plus konvertiert Operanden zu Numeric und addiert dann. Entsprechend fügt ~ zwei Werte im Stringkontext zu einer Zeichenkette (vormals .). Java überlädt dafür das +, aber in Raku stehen Operatoren für eine Funktion mit fixem Eingabe- und Ergebnistyp, die auf möglichst viele Typen anwendbar ist. Daher empfiehlt es sich, für eigene Operatoren unbenutzte Zeichen zu verwenden.

Weil es Typen ohne eindeutigen Wert gibt, wie etwa Failure (Ausnahme), Nil (kein Wert), Junction (Superposition mehrerer Werte), gibt es zwei Basistypen. `Mu` – most undefined – ist zwar die Basisklasse aller Typen. Von größerer praktischer Bedeutung ist aber das tiefer in der Vererbungshierarchie liegende `Any`, die Basisklasse aller Typen mit eindeutigem Wert.

```
(my $nothing).say; # (Any)
```

`Any` bedeutet so viel wie uninitialisierter Rohzustand, wohingegen `Nil` aussagt: Es gibt kein Ergebnis oder dieses Listenelement ist leer. Das Literal `Int` interpretiert der Compiler als „Klasse `Int`“, deren Instanzen die Werteobjekte sind. `$ix.title(Int)` würde daher die Typprüfung der Signatur in Listing 1 überstehen, die lediglich auf Klassenzugehörigkeit prüft. Aber beim Evaluieren des in "" eingefassten Strings gäbe es unweigerlich einen Fehler. Um davor zu schützen, verlangt man per `:D` eine (definierte) Instanz, ein `:U` würde lediglich Typobjekte wie `Int` erlauben:

```
multi method title (Int:D $nr){ ..
```

Parallel und asynchron

Auch wenn Raku ältere Techniken paralleler Ausführung wie Threads, Locks und Semaphore unterstützt (Monitor und Actor als Modul), erweisen sich die neueren Techniken Promise, Supply und Channel als interessanter. Sie sind weniger fehleranfällig und einfacher zu benutzen.

Ein Promise – in anderen Sprachen Future – ist ein Mechanismus, der den Empfänger einer einmaligen Nachricht vom Sender entkoppelt. Der Empfänger muss deshalb nachfragen, ob das Ergebnis noch in Arbeit ist, bereitsteht oder nicht kommen wird. Mit einem Promise erhält man einen einfachen Alarmwecker, der in 10 Sekunden ein Wahr (`True`) als Ergebnis liefert:

```
(Promise.in(10); alias Promise.start({ sleep 10; True});)
```

Das Programm in Listing 4 erwartet zwei Dateinamen als Argumente. `slurp` liest eine Datei und das enthaltene JSON wandelt `from-json` in eine Datenstruktur um. Weil das eine Anweisung ist, benötigt `start` – kurz für `Promise.start` – hier keinen Block, um daraus einen Promise zu erzeugen. Zuletzt wartet `await` auf den Erfolg beider Versprechen und liefert beide Ergebnisse, die `eqv` auf Typ- und inhaltliche Gleichheit prüft.

Mit jedem `start` erzeugt der Scheduler im Hintergrund einen von der VM verwalteten (grünen) Arbeitsthread, der auf einem eigenen Prozessorkern laufen kann. Auftretende Ausnahmen werden automatisch an den Hauptthread weitergeleitet.

In Raku lassen sich die Versprechen mit `all`, `any` und `one` kombinieren, was einer Verknüpfung mit `and`, `or` und `xor` entspricht. So ist es möglich, mit `or` verknüpfte Anfragen parallel an vier DNS-Server zu stellen. Beim Eintreffen der ersten gültigen Antwort wird automatisch alles abgebrochen und aufgeräumt.

Supply und Channel betreiben asynchrone Datenströme zwischen Threads. Supply funktioniert wie eine Mailingliste: Jeder Teilnehmer erhält jede Nachricht, ist aber frei darin, sie später zu lesen und zu löschen, ohne andere zu beeinträchtigen. Ein Channel ist mit dem Brötchensilo einer Selbstbedienungsbäckerei vergleichbar: Mehrere Mitarbeiter befüllen es; nimmt sich ein Kunde eine Teigware, ist es für andere unerreichbar. Beide Klassen lassen sich wie Arrays verwenden. Der Befehl `whenever` ist eine Variante der `for`-Schleife, in der die Datenquelle (links des Pfeils ->) asynchron arbeitet (Listing 5).

Die Klasse `IO::Notification` gehört zum Sprachumfang und erzeugt mit `watch-path` einen Supply, der Änderungen im Verzeichnis `$dir` meldet. Auf ankommende Dateiobjekte reagiert `whenever` und befüllt mit ihnen die Schleifenvariable `$file`.

Der reguläre Ausdruck /../ prüft, ob der Dateipfad auf den Dateiendungen von Raku-Quellcode `.pm` oder `.p6` endet. Haben sich Raku-Quellen geändert, so versendet ein zweiter, in `$rakuchange` abgelegter Supply per `emit` den Dateipfad. Um ihm zu lauschen, ist ein `react`-Block zu eröffnen. Hier reagiert `whenever` weitgehend in Echtzeit und die Ausgabe erfolgt. Der `supply`-Block hingegen sieht sich nur zur Arbeit verpflichtet, wenn jemand ein Datenpaket anfragt. Das Beispiel in Listing 5 soll andeuten, wie das Raku-Web-Framework Cro Supply und Channel nutzt, um mit wenig Programmieraufwand Datenströme asynchron aufzuteilen, zu filtern und zu kombinieren.

Parallelität lässt sich sehr einfach bei der Abarbeitung von Listen durch die Hinzugabe der Methoden `race` (Reihenfolge der

Listing 6: Grammatik zum Einlesen von Potenzen in E-Notation

```
grammar Potenz {
    token TOP { <basis> <[eE]> <exponent> }
    token basis { <vorzeichen>? <zahl> ['.' <zahl> ]? }
    token exponent { <vorzeichen>? <zahl> }
    token vorzeichen { '+' | '-' }
    token zahl { \d+ }
}
my $fund = Potenz.parse('23.4E+2');
```

Listing 7: Action-Klasse zur Berechnung des Zahlenwertes einer Potenz

```
class Calc {
    # reagiert auf die Regel TOP
    method TOP ($match)  {
      $match.make $match<basis>.Numeric
                                  *10 ** $match<exponent>.Numeric
    }
}
my $fund = Potenz.parse('23.4E+2', actions => Calc.new);
say $fund.Str;  # 23.4E+2
say $fund.made; # 2340
```

Ergebnisse ist wichtig) oder `hyper` (Reihenfolge egal) erreichen. Der folgende Einzeiler ermittelt ineffektiv die Primzahlen bis zur Million, indem er Berechnungen für Pakete von 1024 Werten auf 12 Kerne verteilt. Ohne die Angabe der Optionen erlaubt es Raku, das Programm für den vorhandenen Prozessor zu optimieren:

```
say ^100000.race(:1024batch, :12degree).grep(*.is-prime).elems;
```

Beliebteste Funktion

Bei einer im Juni 2019 abgehaltenen Umfrage wählten Raku-Nutzer Grammatiken zur beliebtesten Funktion (siehe ix.de/zpcs). Diese zerlegen Text in syntaktische Bestandteile, wobei jeder Teil gemäß seiner Bedeutung zu beschriften ist: Potenz 23.4E+2 besteht aus der Basis 23.4 und dem Exponenten +2. Beide Teile lassen sich nochmals aufspalten: in Vorkommazahl 23, Nachkommazahl 4, Exponentenvorzeichen + und Exponentenbetrag 2 (Listing 6).

Ein `token` ist eine Methode, die einen regulären Ausdruck anwendet. Damit werden Leerzeichen ignoriert, was ein angenehmeres Formatieren erlaubt. `<zahl>` zeigt an, dass `token zahl` einzusetzen ist, `<[eE]>` ist eine Zeichenklasse aus e und E und `[]` gruppiert. Der Rest folgt den etablierten Regex-Regeln.

Die Grammatik prüft mit der Regel `TOP`, ob sie auf `'23.4E+2'` passt, und liefert an `$fund` ein Match-Objekt. `$fund<basis><zahl>[0]` beinhaltet `23`. `$fund<basis>` enthält bereits einen Baum mit den Teilfunden `$fund<basis><zahl>[0]`, `$fund<basis><zahl>[1]` und `$fund<basis><vorzeichen>`. Nur der skalare Stringkontext `$fund<basis>.Str` liefert den String, den `token basis` erkannte (`23.4`).

Parsen bedeutet in der Regel, jeden Teilfund zu verarbeiten, um aus dem Eingangstext Schritt für Schritt eine Datenstruktur oder ein Ergebnis zu gewinnen – im vorliegenden Fall die Größe der Zahl. Die in Listing 7 gezeigte Methode erhält das von `TOP` erzeugte Match-Objekt und berechnet aus den Werten von Basis und Exponent das Gesuchte. Ein weiteres wichtiges Themengebiet von Raku ist die Metaprogrammierung. Sie erlaubt es, die Sprache mit den Mitteln der Sprache zu erweitern – etwa einen Fakultätsoperator, der 4! berechnen kann (24):

```
sub postfix:<!>($n) { [*] 1 .. $n }
```

Den Charakter von Raku macht letztlich auch eine Vielzahl bequemer Detaillösungen aus: etwa die Methode `@array.pick(3)`, die drei zufällige Arrayelemente wählt. Oder die aus einfachen Kommentaren der Subroutine MAIN automatisch generierte Dokumentation von Kommandozeilenskripten, die immer dann gezeigt wird, wenn die Argumente an das Skript nicht zur Signatur von MAIN passen.

Raku bietet keinen Minimalismus, wie ihn Python und Go bieten, sondern enthält wie Perl ein großes Maß an Sonderzeichen und anderen syntaktischen Eigenheiten. Die Auswahl von 1500 Modulen ist ansehnlich, liegt aber weit unter der des CPAN (Comprehensive Perl Archive Network). Der Programmstart erfordert eine zusätzliche Zehntelsekunde und das Ausführungstempo liegt mittlerweile in der Größenordnung von Perl, Python und Ruby – solange man auf reguläre Ausdrücke verzichtet. Wem Ausdrucksstärke und Individualismus wichtiger sind, der findet in Raku eine vielseitige Programmiersprache, die allen aktuellen Aufgaben gewachsen ist und noch Potenzial für zukünftige besitzt. Die Seite covid.observer/de/ zeigt ein aktuelles Anwendungsbeispiel von Raku. (nb@ix.de)

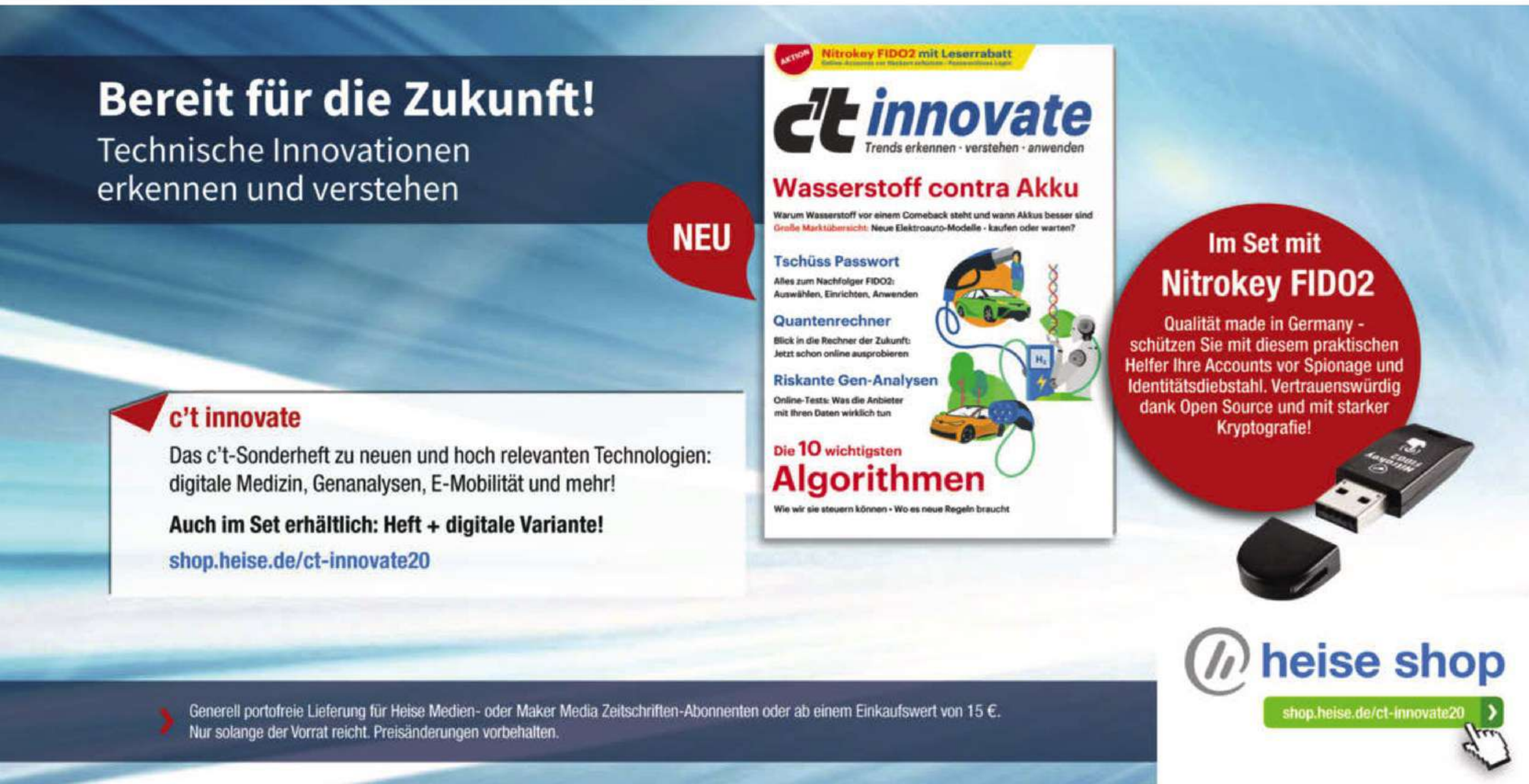
Quellen

Alle Listings zum Download und Ergebnisse der Umfrage in der Perl-Community: ix.de/zpcs



Herbert Breunung

schreibt seit 2007 über Perl und Raku, hält Vorträge, ist Autor des Raku-Moduls Math::Matrix und forscht derzeit über Compiler zur elektronischen Klangsynthese.





F# in Enterprise-Entwicklungen

Funktional abtauchen

Roman Sachse

Mit F# können Entwickler*innen Geschäftsdomänen modellieren und Geschäftsanwendungen aufbauen. Durch die vollständige Integration in das .NET Framework lassen sich Bibliotheken von C# aus Quellen wie Nuget integrieren oder Teilbereiche der eigenen Software in F# schreiben und von C# aus nutzen.

Funktionalen Programmiersprachen haftet in den Köpfen vieler Softwareentwickler*innen noch immer etwas Esoterisches an. Entwickler*innen, die sich mit solchen Sprachen beschäftigen, benützen Sätze wie: „Eine Monade ist nur ein Monoid in der Kategorie der Endofunktoren." Sie interessieren sich mehr für theoretische Konzepte und Diskussionen zu statischen Typen als dafür, funktionierende Geschäftssoftware zu schreiben. Microsoft hat bei der Einführung von F# in .NET nicht gerade dazu beigetragen, dieses Bild zu ändern. Insbesondere am Anfang wurde F# als Sprache für mathematische Anwendungen oder für das Bearbeiten großer Datenmengen gepriesen. Zusätzlich hat Microsoft die Sprache bisher recht stiefmütterlich behandelt.

Obwohl in den letzten Jahren pragmatische funktionale Sprachen wie Elm, Elixir, Clojure und auch F# stärker in den Fokus rückten, sind die klassische objektorientierte Programmierweise und Sprachen wie Java, C# oder PHP immer noch der Standardweg, Software zu schreiben. Ein Grund dafür ist, dass funktionale Konzepte auch in diesen Sprachen immer mehr Einzug halten und die Notwendigkeit wegfällt, sich grundsätzlich mit einer funktionalen Sprache auseinanderzusetzen. Dies ist sehr schade, da funktionale Denkweisen wie die grundsätzliche Unveränderlichkeit von Daten und unidirektionale Datenströme innerhalb einer Anwendung viele Probleme objektorientierter Anwendungen lösen, aber in Sprachen, die diese Konzepte nicht in ihrer DNA haben, nur sehr schwer oder umständlich umzusetzen sind.

Die Basics

Zuerst gilt es, mit einigen Buzzwords aufzuräumen. F# ist eine statisch typisierte Functional-first-Sprache mit Features und Idiomen sowohl für funktionales als auch für objektorientiertes Programmieren. Die Sprache hat ein sehr mächtiges statisches Typsystem, das in den meisten Fällen in der Lage ist, den Typ eines Ausdrucks oder Wertes ohne die Angabe von Typparametern zu inferieren. Das bedeutet, dass es häufig nicht notwendig ist, die Typen bei Benutzung anzugeben. Der Compiler kann diese aus dem Kontext ableiten.

F# verfolgt einen Functional-first-Ansatz. Functional first bedeutet, dass viele Konzepte aus der funktionalen Programmierung als Standardansatz in der Implementierung der Sprache

gewählt wurden. Hierzu gehören unter anderen die Unveränderlichkeit erzeugter Daten, die Möglichkeit, Funktionen selbst als Werte innerhalb der Sprache zu behandeln, aber auch fortgeschrittenere funktionale Konzepte wie Partial Application oder Currying. Der letzte Punkt erlaubt die Umwandlung einer Funktion mit mehreren Argumenten in eine Sequenz von Funktionen mit jeweils einem Argument. Der Rückgabewert eines solchen Aufrufs ist eine neue Funktion, die nur noch die nicht übergebenen Parameter benötigt. Trotz alledem ist es in F# möglich, die Werkzeuge objektorientierter Programmierung zu definieren und zu nutzen. Dazu zählen unter anderem Klassen, Objekte und Interfaces, aber natürlich auch Klassenvererbung und das Überschreiben von Methoden von Elternklassen. Dieser Artikel wird sich nur mit einem kleinen Teilbereich der beschriebenen Möglichkeiten beschäftigen.

Listing 1 veranschaulicht, wie sich Werte und Funktionen mithilfe von `let` zur Identifikation an Namen binden oder zuweisen lassen. Diese Zuweisungen sind, im Gegensatz zu Variablen anderer Sprachen, standardmäßig unveränderlich. Entsprechend lässt sich ein Wert, der an einen Namen gebunden ist, nicht mehr direkt ändern.

F# verfügt über eine mächtige Typinferenz. Dadurch ist es oft nicht erforderlich, den Typ wie in Zeile 1 zu deklarieren. Funktionen werden wie in Listing 1 (`als_Brutto`) ebenfalls mithilfe von `let` deklariert und Funktionsparameter werden mit einem Leerzeichen getrennt und ohne Komma an den Namen angehängt.

Weiter ist F# eine ausdrucksbasierte Sprache. Dadurch besitzen alle Strukturen, auch Kontrollstrukturen wie `if/else` oder Schleifen, immer einen Rückgabewert und lassen sich einem Namen zuweisen. In anderen Sprachen ist die Semantik für solche Kontrollstrukturen häufig die von Anweisungen, sie haben also keinen Rückgabewert. Dies unterscheidet F# stark von Sprachen, in denen es Unterscheidungen zwischen Ausdrücken und Anweisungen gibt. Der Wert von `teuer` ist der Rückgabewert des `if/else`-Ausdrucks innerhalb der Zuweisung. In Funktionen wird immer der Rückgabewert des letzten Ausdrucks automatisch zurückgegeben. Die Funktion `als_Brutto` liefert also wieder einen `decimal` zurück.

Listing 1: Werte und Funktion mit let zuweisen

```
let netto : decimal = 19.99m

let als_Brutto netto =
 preis * 1.19m

let brutto = als_Brutto netto

let teuer =
 if brutto > 200
 then "ja"
 else "nein"
```

Deep Dive

Der Auftrag in unserem Beispielprojekt lautet, eine Software für eine Tauchschule zu schreiben. Kunden können Kurse für unterschiedliche Fähigkeitslevels (Open Water Diver, Adventure Diver etc.) buchen und absolvieren. Am Ende eines erfolgreichen Kurses bekommen die Teilnehmer ein Zertifikat. Es dient als Voraussetzung für Kurse auf höheren Levels. Für jeden Level gibt es zusätzlich Spezialzertifikate (Deep Diver, Wreck Diver etc.).

Listing 2: Spezialzertifikate mit dem Typsystem von F# implementieren

```
type Level =
  | OpenWaterDiver
  | AdventureDiver
  | AdvancedOpenWaterDiver

type Speciality =
  | BoatDiver
  | NightDiver
  | DeepDiver
  | WreckDiver
  | CavernDiver
  | IceDiver
```

Listing 3: Level, Speciality und Course einbinden

```
type CourseRegistrationWorkflow =
  {
    Level : Option<Level>
    Speciality : Option<Speciality>
    Course : Option<System.Guid>
  }
```

Listing 4: Eine DU initialisieren

```
let level = AdventureDiver

// Instanziierung eines Records
let workflowResult : CourseRegistrationWorkflow =
 {
   Level = Some level
   Speciality = Some DeepDiver
   Course = Some (Guid.Parse("f9b91bea-5424-41de-a385-5f13d821bb1e"))
 }
```

Diese Informationen lassen sich mit dem Typsystem von F# implementieren (Listing 2).

Auf den ersten Blick sehen diese Definitionen wie Enums aus C# oder anderen Sprachen aus. Diese sogenannten Discriminated Unions (im Folgenden DUs) sind jedoch ungleich mächtiger als herkömmliche Enums. Jeder einzelne Fall kann eine beliebig große, unterschiedliche Menge von Daten enthalten. Über sogenanntes Pattern Matching haben Entwickler*innen jederzeit wieder Zugriff darauf. Am besten stellt man sich die Datenstruktur als eine Vererbungshierarchie mit einer abstrakten Basisklasse (`Level`) und konkreten Unterklassen (`OpenWaterDiver`, `AdventureDiver` etc.) vor.

Der Tauchschule schwebt ein Kursbestellsystem vor, das der Nutzerin in drei Schritten zu einer Kursbuchung verhilft. Im ersten Schritt soll sie auswählen, für welchen Level sie einen Kurs buchen möchte, der zweite Schritt legt das Spezialzertifikat fest und im dritten Schritt geht es um den tatsächlichen Kurs. Auch diese Anforderungen lassen sich mithilfe des F#-Typsystems einfach implementieren (Listing 3).

Als Typ für diesen Workflow dient ein F#-Record. Records sind einfache Datenhaltungscontainer, im Normalfall ohne eigenes Verhalten. Jeder ihrer Slots hat einen eigenen Namen, hier `Level`, `Speciality` und `Course`. Darüber lassen sie sich eindeutig identifizieren. Ein hilfreiches Feature von F#-Records und anderen Datentypen wie Discriminated Unions oder Tupeln ist die automatisch definierte Gleichheit. Anders als in C#-Klassen müssen die Entwickler*innen keine Vergleichsmethoden für Gleichheitsüberprüfungen einsetzen. Zwei Records sind gleich, wenn sie denselben Typ haben und die Werte in ihren Slots ebenfalls identisch sind.

iX-TRACT

- Hinter F# verbirgt sich eine statisch typisierte Functional-first-Sprache mit Features und Idiomen sowohl für funktionales als auch für objektorientiertes Programmieren.
- F#s mächtiges statisches Typsystem inferiert den Typ eines Ausdrucks oder Wertes ohne die Angabe von Typparametern.
- Werte und Funktionen lassen sich zur Identifikation an Namen binden oder diesen zuweisen. Im Gegensatz zu anderen Sprachen sind diese Variablen unveränderlich.

In Listing 4 sieht man, wie sich neue Instanzen dieser Typen erzeugen lassen. Die einzelnen Fälle einer DU sind gleichzeitig Konstruktorfunktionen für den Typ, hier `AdventureDiver` für den Typ `Level`. Da diese Fälle bis jetzt noch keine Werte verpacken, benötigt diese Funktion keine weiteren Parameter. Auf den ersten Blick wirkt die Initialisierung eines Records etwas ungewöhnlich, sie orientiert sich mit ihren geschweiften Klammern an der Definition des Typs. Der Compiler erkennt den Typ entweder anhand der (optionalen) Typdeklaration oder der Form des Records.

In der Definition des Typs `CourseRegistrationWorkflow` sind die Domänentypen `Level`, `Speciality` und `Guid` in sogenannten Options verpackt. Options drücken in F# normalerweise fehlende Werte im Programm mithilfe des Typsystems aus. Andere Sprachen wie Java oder C# sehen für diesen Fall häufig die Verwendung von `null` vor.

Heutzutage gilt die Nutzung von `null` für die Verwendung von Werten, die möglicherweise nicht vorhanden sind, als ein großes Antipattern. Tony Hoare, der Erfinder von `null`, hat `null` als „billion-dollar mistake“ bezeichnet. Der Grund dafür ist, dass `null` ein Mitglied jedes Types ist, man also `null` für jeden Typ nutzen kann. Entsprechend kann jeder Wert zu jeder Zeit `null` sein. Von Entwickler*innen verlangt das eine extrem defensive Programmierweise, in der sie häufig Prüfungen, ob Werte nicht `null` sind, in ihren Code einbauen müssen. Nur so lassen sich die berühmtberüchtigten NullPointer- oder NullReferenceExceptions vermeiden.

Bei der Arbeit mit F# existiert `null`, außer bei der Interoperabilität mit C#, nicht. Zur Kennzeichnung, dass möglicherweise Daten fehlen, dienen deshalb die oben genannten Options. Der zentrale Unterschied zu `null` ist, dass das mögliche Fehlen von Werten explizit gemacht wird. Weiter sorgt der Compiler dafür, dass man auch immer den Fall behandelt, dass ein Wert nicht vorhanden ist. Entwickler*innen könnten vermuten, dass Options besondere Sprachkonstrukte innerhalb von F# verkörpern, was jedoch nicht stimmt. Options sind einfache Discriminated Unions, die die Standardbibliothek definiert. Eine solche Definition sieht beispielsweise so aus:

```
type Option<'T> =
  | Some of 'T
  | None
```

Das Beispiel beleuchtet zwei weitere Konzepte der Sprache im Allgemeinen und von DUs im Speziellen. Einerseits können einzelne Fälle unterschiedliche Daten vorhalten. Andererseits ist es möglich, ähnlich wie in anderen Sprachen mit generischen Datentypen zu arbeiten. Das drückt der generische Typparameter `'T` aus. Hierbei spielt der Buchstabe keine Rolle, wichtig ist das öffnende einfache Anführungszeichen. Die gezeigte Typdefinition erlaubt, alle Arten von Typen im `Some`-Fall zu verpacken, wenn tatsächlich Daten vorhanden sind. Im `None`-Fall werden keine Daten verpackt.

Pattern Matching

Bleibt die Frage, wie man an die verpackten Daten kommt, um mit ihnen weiterzuarbeiten. Mithilfe von Pattern Matching lässt sich einfach herausfinden, mit welchem der möglichen Fälle einer DU der Typ konstruiert wurde. Daneben kann die Entwicklerin auf die verpackten Werte der einzelnen Fälle zugreifen. Das nächste Codebeispiel zeigt den Bruttorechner, allerdings ist es möglich, dass der Nettopreis fehlt. Grund dafür könnte beispielsweise sein, dass der Preis über eine Datenbankabfrage oder einen anderen Input eingeflossen ist. Die Bruttoberechnung soll allerdings nur stattfinden, wenn tatsächlich ein Wert vorhanden ist.

In anderen Sprachen fände dafür eine `null`-Überprüfung statt und im Falle eines `null`-Wertes gäbe man wieder `null` zurück. Wenn man die Überprüfung vergisst, wirft das System zur Laufzeit eine Exception. Da in F# `null` nicht zur Verfügung steht, machen Entwickler*innen das mögliche Nichtvorhandensein eines Wertes mit `Option<float>` explizit und berechnen den Bruttobetrag nur, wenn auch ein Wert vorliegt. Der nächste Codeausschnitt veranschaulicht, wie ein Pattern Matching über den Nettopreis stattfindet, im `Some`-Fall der Wert direkt ausgepackt und der berechnete Bruttobetrag wieder in ein `Some` verpackt zurückgegeben wird. Ansonsten gibt man wieder `None` zurück.

Da F# eine ausdrucksbasierte Sprache ist, ist das gesamte Pattern Matching ein Ausdruck. Der Rückgabewert dieses Ausdrucks ist auch der Rückgabewert der Funktion, weil er der letzte Ausdruck innerhalb der Funktion ist. Somit weist die Funktion `als_`

Listing 5: Records initialisieren

```
let step0 =
 {
   Level = None
   Speciality = None
   Course = None
 }

let step1 =
 {
   Level = Some AdventureDiver
   Speciality = None
   Course = None
 }

let step2 =
 {
   Level = Some AdventureDiver
   Speciality = Some DeepDiver
   Course = None
 }

let step3 =
 {
   Level = Some AdventureDiver
   Speciality = Some DeepDiver
   Course = Some (Guid.Parse("f9b91bea-5424-41de-a385-5f13d821bb1e"))
 }
```

Listing 6: Repräsentation illegaler Zustände

```
let sollte_nie_passieren =
 {
   Level = None
   Speciality = None
   Course = Some (Guid.Parse("f9b91bea-5424-41de-a385-5f13d821bb1e"))
 }
```

Listing 7: CourseRegistrationWorkflowStep

```
type CourseRegistrationWorkflowStep =
  | ChooseLevel
  | ChooseSpeciality of Level
  | ChooseCourse of Level * Speciality

let step1 = ChooseLevel
let step2 = ChooseSpeciality AdvancedOpenWaterDiver
let step3 = ChooseCourse (AdvancedOpenWaterDiver,DeepDiver)

let showRegistrationStep step =
   match step with
   | ChooseLevel ->
       showLevelStep

   | ChooseSpeciality level ->
       showSpecialityStep level

   | ChooseCourse (level,speciality) ->
       showCourseStep level speciality
```

`Brutto` als Typ sowohl für ihren Input als auch für ihren Output ein `Option<float>` auf.

```
let als_Brutto (netto : Option<float>) =
  match netto with
  | Some wert -> Some (wert * 0.19)
  | None      -> None
```

Illegale Zustände

Zurück zum Kursbuchungssystem. Hier gilt es, sich daran zu erinnern, dass man den Registrierungsworkflow mithilfe eines Records mit drei optionalen Feldern modelliert hat. Zu Beginn des Workflows liegen noch keine Informationen vor. Nach dem ersten Schritt ließ sich der Level füllen, nach dem zweiten Schritt die Spezialität und schließlich der Kurs auswählen. In Listing 5 sieht man mögliche Instanziierungen des Records.

Prinzipiell funktioniert dieses Modell also. Es hat allerdings ein grundsätzliches Problem: Es erlaubt die Repräsentation eigentlich illegaler Zustände der Domäne innerhalb der Software. Jedoch sollte es nicht möglich sein, einen Kurs auszuwählen, ohne vorher einen Level und eine Spezialität ausgewählt zu haben. Im aktuellen Zustand der Anwendung bedeutet das kein Hindernis (Listing 6).

Entsprechend müssten Entwickler*innen bestenfalls Tests schreiben, um sicherzustellen, dass dieses Problem nicht auftritt, oder im schlechtesten Falle darauf vertrauen, dass niemand im Team einen Fehler in der Umsetzung macht. Listing 7 zeigt, wie solch eine Lösung aussehen könnte.

Man modelliert nun den Workflow nicht mehr über einen Record, sondern mit einer Discriminated Union. Anfangs (`ChooseLevel`) stehen noch keinerlei Informationen zu Verfügung, weshalb Entwickler*innen keine Werte innerhalb des DU-Falles verpacken müssen. Weiter lässt sich der zweite Fall (`ChooseSpeciality`) nur über einen Level erstellen. Die Konstruktorfunktion benötigt einen Level, im Beispiel ein `OpenWaterDiver`, als Parameter und erstellt ansonsten keinen `CourseRegistrationWorkflowStep`. Ein Kurs lässt sich im letzten Schritt (`ChooseCourse`) ausschließlich wählen, wenn ein Level und eine Spezialität angegeben wurden. Hier sieht man, dass es möglich ist, mehrere unterschiedliche Werte innerhalb eines DU-Falles zu verpacken. Das geschieht mithilfe von Tupeln, die sich in F# mit `*` deklarieren lassen.

Die Funktion `showRegistrationStep` benutzt Pattern Matching und kann alle verpackten Werte direkt auspacken. `ChooseCourse` lässt sich nur generieren, wenn es einen Level und eine Spezialität gibt. Da in F# keine `null`-Werte existieren, können sich Entwickler*innen sicher sein, dass die Funktion `showCourseStep` alle Werte bekommt, die sie für ihre ordnungsgemäße Arbeit benötigt.

Mit F# und der Hilfe von DUs hat sich das Problem so lösen lassen, dass weder Tests noch Vertrauen notwendig sind. Alleine durch das Typsystem lässt sich gewährleisten, dass solch ein illegaler Zustand in der Software nicht auftreten kann. In objektorientierten Sprachen bestehen ebenfalls Möglichkeiten, solche illegalen Zustände unrepräsentierbar zu machen. Man könnte beispielsweise den `CourseRegistrationWorkflowStep` als abstrakte Basisklasse und die einzelnen Fälle als Subklassen implementieren.

Interessanterweise geschieht genau das, wenn der F#-Compiler diesen Code in die Intermediate Language des .NET Frameworks übersetzt. Allerdings treten hierbei mehrere Probleme auf. Einerseits ist der zu erstellende Code bedeutend länger, vier Klassen inklusive Methoden zur Gleichheitsprüfung etc. Deswegen kommt diese Implementierung häufig nicht zum Einsatz. Andererseits gibt einem der F#-Compiler während der Übersetzung eine Warnung oder optional einen Fehler aus, sollte man innerhalb eines Pattern-Matching-Ausdrucks einen oder mehrere Fälle einer Discriminated Union vergessen haben. Dadurch lassen sich deutlich fehlerunanfälligere Programme entwickeln.

Listing 8: Illegale Zustände mit DUs vermeiden

```
type Level =
  | OpenWaterDiver
  | AdventureDiver
  | AdvancedOpenWaterDiver

type OpenWaterDiverSpeciality =
  | BoatDiver
  | NightDiver

type AdventureDiverSpeciality =
  | DeepDiver
  | WreckDiver

type AdvancedOpenWaterDiverSpeciality =
  | CavernDiver
  | IceDiver

type LevelWithSpeciality =
  | OpenWaterDiver of OpenWaterDiverSpeciality
  | AdventureDiver of AdventureDiverSpeciality
  | AdvancedOpenWaterDiver of AdvancedOpenWaterDiverSpeciality


type CourseRegistrationWorkflow =
  | ChooseLevel
  | ChooseSpeciality of Level
  | ChooseStart of LevelWithSpeciality

// kompiliert nicht
let step3 = ChooseCourse (AdvancedOpenWaterDiver DeepDiver)

// kompiliert
let step3 = ChooseCourse (AdvancedOpenWaterDiver CavernDiver)
```

Illegale Zustände, die Zweite

Jetzt geht es daran, das System in den Produktiveinsatz zu bringen. Leider bekommt man relativ schnell die Rückmeldung, sich beim Abholen der Anforderungen nicht genug mit dem Businessexperten auseinandergesetzt zu haben. Es gibt nicht nur eine Abhängigkeit zwischen den gewählten Levels und Spezialitäten und dem zur Auswahl stehenden Kurs, sondern die Spezialitäten hängen auch von dem gewählten Level ab. `DeepDiver` und `WreckDiver` sind beispielsweise Spezialitäten, die nur innerhalb des Levels `AdventureDiver` zur Verfügung stehen, während `IceDiver` und `CavernDiver` lediglich beim `AdvancedOpenWaterDiver` erlaubt sind.

Das Beispiel in Listing 7

```
let step3 = ChooseCourse (AdvancedOpenWaterDiver,DeepDiver
```

führt einen illegalen Zustand innerhalb der Anwendung herbei, der sich so nicht vorhersehen lässt und den man nach Möglichkeit verhindern will. Dafür lassen sich wieder Discriminated Unions nutzen.

Nach der Präzisierung des Domänenmodells gilt es, unterschiedliche Spezialitäten für jeden Level zu wählen. Zusätzlich gibt es einen neuen Fall `LevelWithSpeciality`. Dieser ersetzt die vorherige Nutzung des Tupels `Level * Speciality` im Schritt `ChooseStart`. Diese Änderung ermöglicht dem Compiler, die Konstruktion illegaler Zustände zu verhindern und den vorher möglichen, aber in unserer Domäne illegalen dritten Schritt gar nicht erst zuzulassen.

Illegale Zustände, die Dritte

Nachdem Entwickler*innen sichergestellt haben, dass der Kursworkflow keine illegalen Resultate liefert, lässt sich der weitere Ablauf in der Anwendung skizzieren. Listing 9 beginnt mit der

Definition einer weiteren DU für das Ergebnis eines Registrierungsversuchs. Die Funktion `registerForCourse` bekommt sowohl den gewählten Kurs als auch den gewählten Trainer (beides `Guids`) und prüft, ob noch ein Platz in diesem Kurs frei ist. Im Erfolgsfall kann die Buchung bestehen bleiben und Bestätigungsmails werden versendet. Man bekommt `Accepted` zurückgegeben, ansonsten `Course_Full`. Anschließend verarbeitet die Funktion `renderRegistrationResult` das Ergebnis. Aus der Nutzersicht erscheint die zu dem Ergebnis passende Seite.

Ein grundsätzliches Problem zeigt sich, wenn man den Code auf illegale Zustände abklopft: Sowohl `Trainer` als auch `Course` sind einfache `Guids`. Sicherheiten, dass die Funktionen nicht mit vertauschten Parametern aufgerufen werden und es so zu den merkwürdigsten Fehlern zur Laufzeit kommt, fehlen. Das Problem verstärkt sich dadurch, dass `registerForCourse` und `renderRegistrationResult` die Parameter in unterschiedlicher Reihenfolge benötigen. Unit-Tests helfen hier nicht, Entwickler*innen sind also auf Code-Reviews angewiesen. Es wäre gut, wenn der Compiler helfen könnte und fehlerhaften Code von vorne herein nicht kompiliert.

Als ersten Ansatz bietet F# sogenannte Typaliasse, die es erlauben, andere Namen für bestehende Typen zu verwenden.

```
type Trainer = System.Guid
type Course = System.Guid

let registerForCourse (course : Course) (trainer : Trainer) =
  //...
```

Das Codebeispiel demonstriert die Definition zweier solcher Aliasse: `Trainer` und `Course`. Sie tragen zumindest dazu bei, die Klarheit innerhalb des Codes für andere Entwickler zu erhöhen. Allerdings sind es tatsächlich nur andere Namen für denselben Typ. Entsprechend ist der Compiler außerstande zu erkennen, wenn die Parameter im Aufruf vertauscht wurden. Als Lösung für dieses Problem kommen wieder DUs zum Einsatz, dieses Mal sogenannte Single-Case Discriminated Unions. Auf den ersten Blick sehen sie wie Typaliasse aus, sind jedoch ein eigener Typ.

Listing 10 zeigt, wie sich Typen definieren und nutzen lassen. Es sind normale DUs, die nur einen einzigen möglichen Fall haben. An dieser Stelle ist es nicht mehr realisierbar, die Parameter von `registerForCourse` zu vertauschen. Der Compiler steht dem im Weg. Ganz ohne weiteres Pattern Matching lassen sich in F# auch die Werte einer Single-Case-DU direkt in der Funktionsdeklaration auspacken und einzelne Werte direkt benutzen (siehe Funktion `registerCoursePage`). Dieser Mechanismus eröffnet eine elegante Möglichkeit, primitive Datentypen typsicher zu verpacken und illegale Zustände innerhalb der Anwendung unrepräsentierbar zu machen.

Listing 9: Eine DU für das Ergebnis einer Registrierung definieren

```
type RegistrationResult =
  | Accepted
  | Course_Full

let registerForCourse (course : Guid) (trainer : Guid) =
  if not (isFull course trainer) then
    // store in DB, send verification mail etc.
    Accepted
  else
    Course_Full

let registerCoursePage trainer course =
  match registerForCourse course with
  | Accepted ->
      renderConfirmationPage trainer course

  | Course_Full ->
      renderCourseFullPage course
```

Listing 10: Typen definieren und benutzen

```
// Single Case Discriminated Unions
type Trainer = Trainer of System.Guid
type Course = Course of System.Guid

// Verpacken der Guid mit DU-Konstruktor
let trainer = Trainer (System.Guid.NewGuid())

let registerForCourse (course : Course) (trainer : Trainer) =
  //...

// direktes Auspacken von Single-Case DUs in der Funktionsdeklaration
let registerCoursePage (Trainer trainer) (Course course) =
  // die verpackten Guids stehen hier in trainer und course zur
                                                        Verfügung
```

Ausblick

Insbesondere wer sich sowieso im .NET-Umfeld bewegt, kann F# als weiteres Werkzeug mitsamt interessanten Konzepten heranziehen, um die Produktivität zu steigern. Im Rahmen dieses Artikels ist es nicht möglich, den Sprachumfang in Gänze zu zeigen. Deshalb folgen hier noch ein paar Highlights. F# bietet die Option, Maßeinheiten wie Zentimeter, Meter, Kilometer und andere zu primitiven Datentypen hinzuzufügen und auch in Bezug zueinander zu definieren. Versehentlich falsche Einheiten miteinander zu verrechnen, kommt demnach nicht vor.

Darüber hinaus verfügt F# über die am einfachsten lesbare `async/await`-Syntax im gesamten .NET-Umfeld. C# hat sich einiges von der Sprache für `async/await` aus neueren Versionen abgeschaut.

Mit Fable (JavaScript) und Fabulous (iOS, Android, Mac, WPF über Xamarin.Forms) bringt F# voll funktionsfähige und gut dokumentierte Umgebungen mit. Gerade mithilfe der Elm-Architektur erlauben sie, funktionelle und funktionale Frontend-Anwendungen zu schreiben.

Im SAFE-Stack finden sich Bibliotheken, Werkzeuge, Vorgaben und Dokumentationen, um F# sowohl im Frontend als auch auf dem Backend gemeinsam einzusetzen. Das verhilft Entwickler*innen dazu, den häufig gehegten Traum einer einheitlichen Codebasis mit geteilter Geschäftslogik sowohl im Frontend als auch im Backend zu erfüllen.

F# lässt sich in Visual Studio, Visual Studio Code und JetBrains Rider nutzen. Damit können Entwickler*innen zwischen drei voll funktionsfähigen Entwicklungsumgebungen wählen, die es erlauben, F#-Projekte auf produktive Art und Weise zu entwickeln und zu verwalten. Schließlich gehört zur Sprache auch eine sehr aktive und hilfreiche Community, die insbesondere im .NET-Bereich Vorreiter für die Entwicklung Open-Source-basierter Anwendungen ist. (csc@ix.de)

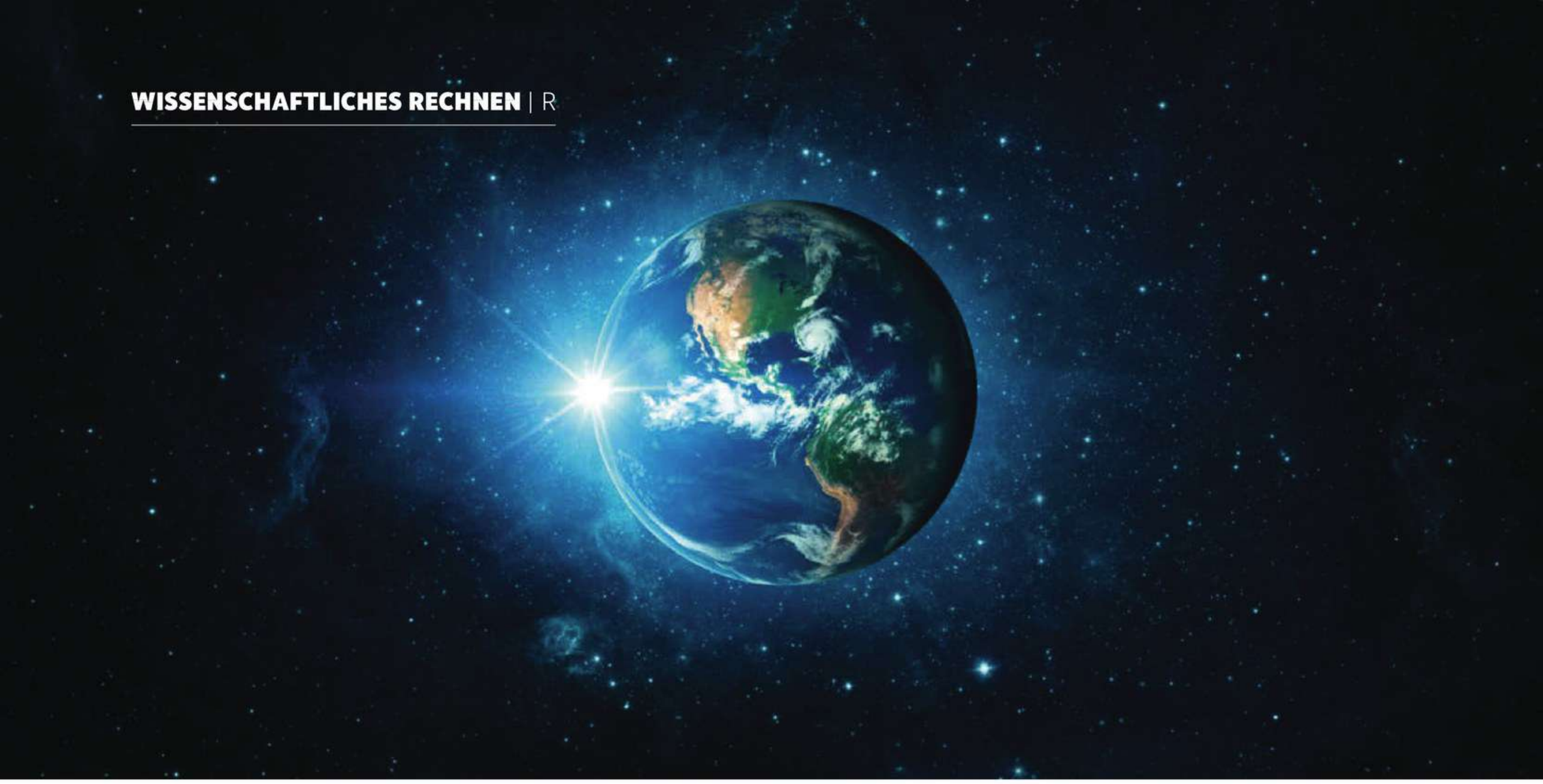
Quellen

Alle Listings unter ix.de/zc73



Roman Sachse

ist Softwareentwickler und seit mehreren Jahren aktiv in der Domain-driven-Design- und F#-Community unterwegs. Aktuell arbeitet er an Software im deutschen Gesundheitssystem und veröffentlicht Videos und Streams zu Themen wie F# und Event Sourcing.



R: Statistikumgebung für Datenanalyst*innen

Alles erreichen

Günter Faes

Von der Datenmanipulation über die Auswertung bis zur grafischen Darstellung beherrscht R alle Disziplinen der Datenanalyse – und hat sich damit fest im Unternehmenseinsatz etabliert.

Auf mehr als 25 Jahre Entwicklungshistorie blickt R zurück und hat sich durch ihre Stabilität und den Funktionsumfang als Standardsprache für statistische Datenanalyseanforderungen fest etabliert.

R wurde als funktionale, objektorientierte und freie (GNU-) Programmiersprache für statistische Berechnungen entwickelt und hat ihren Weg über die Hochschulen in die Wirtschaft gefunden. Mittlerweile ist R nicht mehr nur eine Programmiersprache, sondern eine Suite zur Datenmanipulation, Auswertung und grafischen Darstellung. Es handelt sich um ein vollständig geplantes und kohärentes System und nicht um eine spezifische und unflexible Werkzeugsammlung.

Das kann R

R liegt Stand Mai 2020 in der Version 4.0.0 vor und lässt sich auf allen gängigen Plattformen einsetzen. Über das Paketsystem stehen mehr als 15 500 Pakete und 40 Task Views parat, mit denen sich der Funktionsumfang auf ein breites Spektrum moderner Statistik erweitern lässt. Diese Pakete stehen Datenanalyst*innen ebenso wie R über CRAN (Comprehensive R Archive Network) zum Download bereit. Die Verteilung über CRAN ist an bestimmte Bedingungen geknüpft, die eine hohe Qualität der Pakete gewährleisten sollen. Das schließt unter anderem die Dokumentation sowie funktionsbeschreibende Beispiele ein. Task Views sind von Verantwortlichen gepflegte, nach Aufgaben gebündelte Pakete. Für das Gebiet der Zeitreihenanalyse etwa findet sich im Task View „TimeSeries“ eine Paketsammlung, die sich komfortabel und komplett als ctv-Paket (ctv: Infrastruktur für Task Views) in die R-Umgebung einbinden lässt.

Der ständig wachsende R-Paketbestand liefert eine immer größere Auswahl an Hilfen für eine Vielzahl von Aufgaben. Dazu gehört auch, dass die R-Umgebung einen der wesentlichen Schritte eines Datenanalyseprojektes, die „Datenvorbereitung“ (Datenbanken auslesen, Daten validieren und bereinigen etc.), vollständig unterstützt. Unter dem Task View „Database with R“ findet sich eine Paketsammlung, die Zugriff auf alle gängigen Datenbanksysteme ermöglicht.

Wer ein statistisches Verfahren oder eine Datenmanipulationsmethode in der R-Umgebung vermisst, kann ohne großen Aufwand eigene Funktionen schreiben und zu Paketen bündeln.

iX-TRACT

- Die Programmiersprache R prägt seit 25 Jahren die statistische Datenanalyse.
- RStudio und das Paketsystem bieten eine flexible Umgebung für alle Anwendungsbereiche.
- Für hohe Performance lässt sich C++-Code direkt in R-Skripte einbinden.

R ist nicht allein auf das eigene Ökosystem zur Erweiterung des Funktionsumfanges beschränkt. Bei Bedarf lassen sich Programmiersprachen wie Python, Julia oder C++ in ein R-Skript einbinden.

Noch mehr Performance

Die Standarddistribution von R umfasst einen Interpreter und eine Kommandozeilenumgebung, die sogenannte Konsole. Es steht außer Frage, dass vom Interpreter ausgewertete Funktionen und Skripte langsamer als kompilierter Code sind. Deswegen bietet R von Anfang an eine API zum Einbinden von Fortran- oder C++-Code, mit dem sich auch rechenintensive Verfahren performant durchführen lassen.

Zur Datenanalyse gehört der Umgang mit großen Datenvolumen und rechenintensiven Lernverfahren – hier kann purer R-Code zu langsam sein. Abhilfe schaffen Pakete aus dem Task View „High-Performance and Parallel Computing with R“, die kompilierten Code oder Methoden der Parallelisierung nutzen. Genügen diese Möglichkeiten nicht, kommt das Paket Rccp zum Einsatz. Rccp bindet C++-Code direkt in ein R-Skript ein. Davon machen diverse Pakete bei den integrierten Funktionen Gebrauch und tragen so zur Optimierung der R-Umgebung bei. Rccp eröffnet aber die grundsätzliche Möglichkeit, Performanceengpässe in Analyseprojekten durch Integration von eigenem C++-Code aufzulösen – dazu sind allerdings C++-Kenntnisse erforderlich. Es stehen also verschiedene Optionen zur Verfügung, um mit R performante Datenanalyseprojekte zu realisieren.

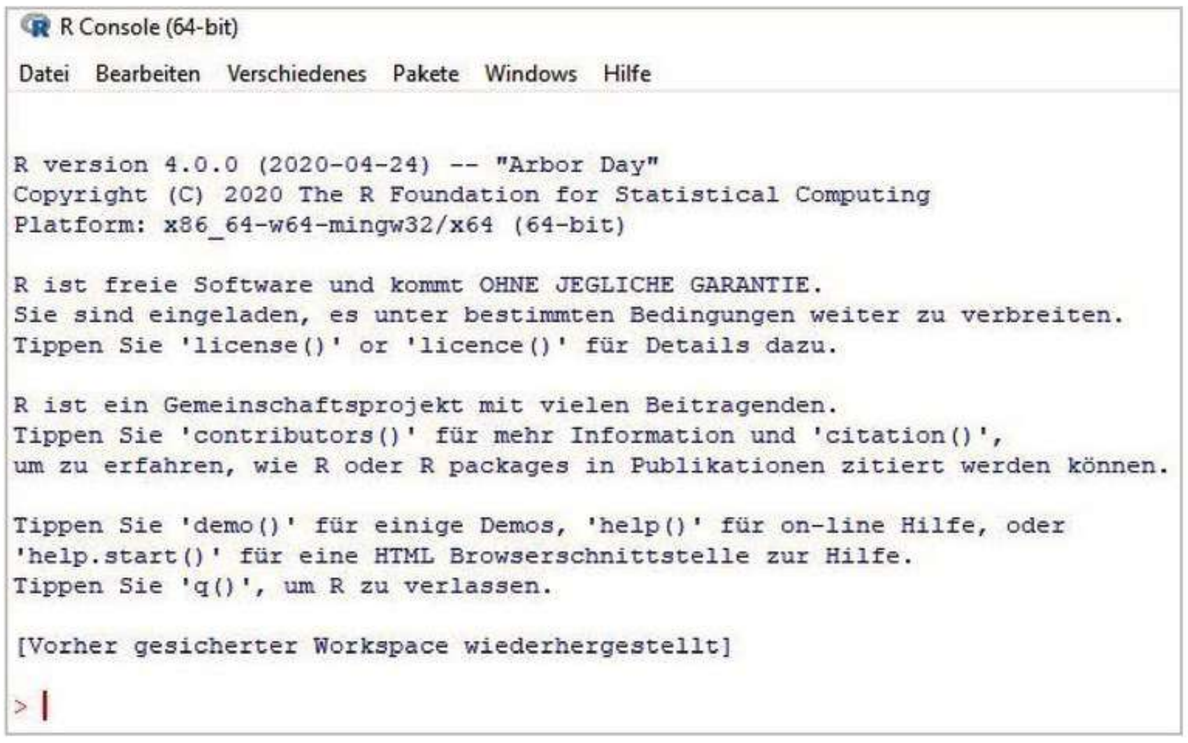
Vom Einsteiger bis zum Profi

R, über die Standarddistribution installiert, bietet als Arbeitsoberfläche die schlicht gehaltene R-Konsole an (Abbildung 1). Die R-Konsole enthält alles Grundlegende zur Datenanalyse, ist als reine Kommandozeile aber nicht mehr zeitgemäß. Um schnell datenanalytische Fragen zu klären, mag sie insbesondere in Verbindung mit dem R-Commander (Abbildung 2) dennoch hilfreich sein.

Der R-Commander stellt menügeführten Zugriff auf gängige Datenbearbeitungsmethoden und statistische Verfahren bereit, inklusive grafischer Darstellung. Für gelegentliche Analysen ist der R-Commander damit ein geeignetes Tool, da sich Anwender aufgrund der Menüführung nicht unbedingt jedes Argument oder jeden Parameter etwa für eine Testdurchführung merken müssen.

Für den alltäglichen Einsatz ist RStudio mit seiner durchdachten grafischen Benutzeroberfläche die bessere Wahl. RStudio ist eine integrierte Entwicklungsumgebung (IDE) für R, die RStudio PBC (Public Benefit Corporation) in verschiedenen Editionen für die meisten gängigen Betriebssysteme anbietet.

Ein Kernelement von RStudio ist der Skriptbereich. Zum Entwerfen neuer Skripte gibt es darin Funktionen wie etwa Autovervollständigung, Syntaxhervorhebung, Codefaltung und zeilenweise Codeausführung. Dabei geht die Kontrolle über die R-Umgebung nicht verloren. Informationen zu eingebundenen Variablen, zur Ausführungshistorie und zu Paketen sind jederzeit vorhanden und einsehbar.



Als Kommandozeilenanwendung bleibt die R-Konsole einfach und überschaubar (Abb. 1).

Die integrierte R-Konsole dient als Skriptausgabebereich und Grafiken lassen sich komfortabel in einem eigenen Bereich abbilden (Abbildung 3).

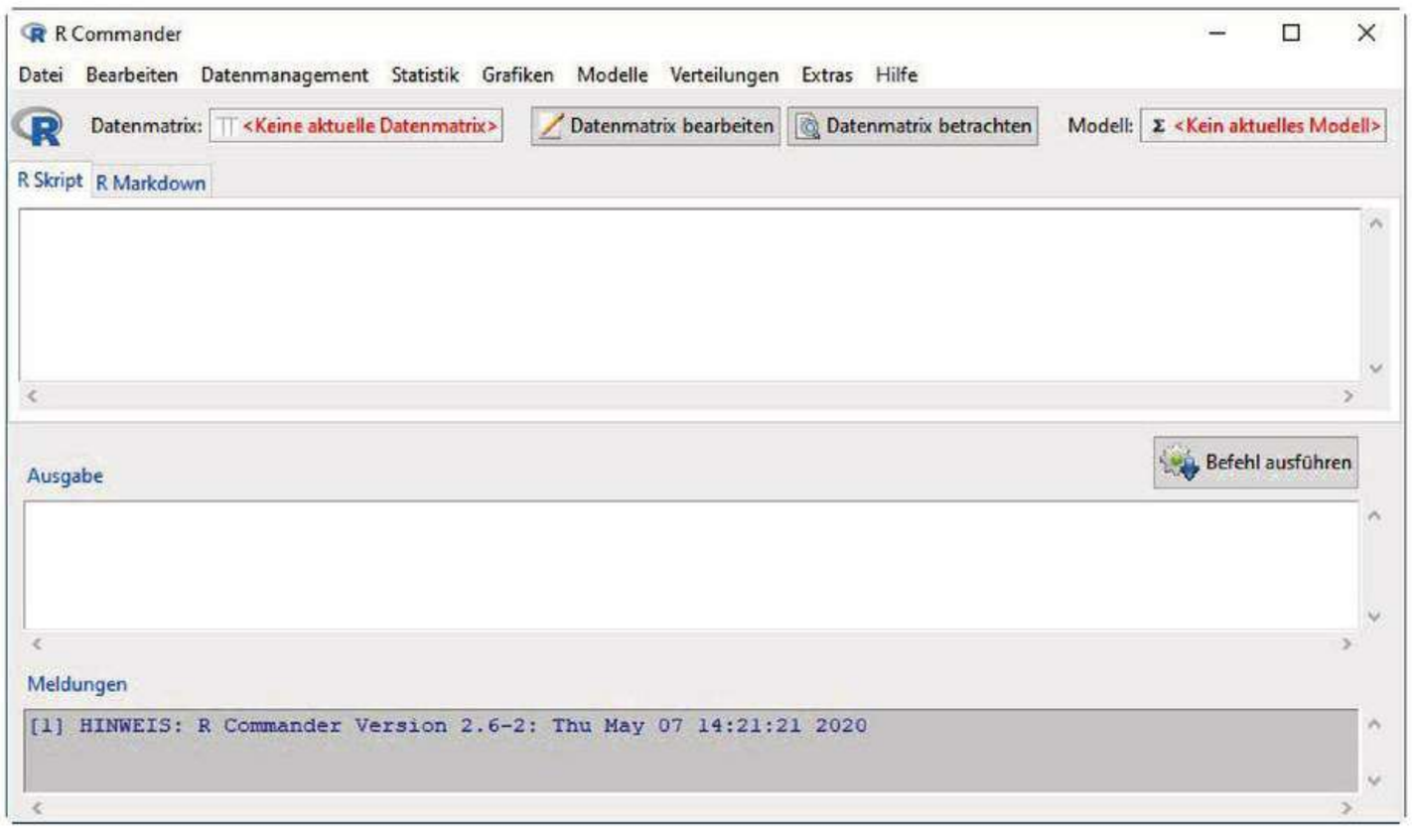
RStudio beschränkt sich nicht allein auf R, die IDE unterstützt auch das Schreiben von Python-, C++- und SQL-Skripten. Präsentationen der Datenanalyseergebnisse als interaktive Webanwendung und als umfassende Dokumentation erstellen Analyst*innen über die integrierten Shiny- und Markdown-Funktionen. Zur übersichtlichen Organisation lässt sich jedes Datenanalysevorhaben als eigenes Projekt anlegen und Git zur Versionsverwaltung einbinden.

Schon diese nicht vollständige RStudio-Übersicht macht deutlich, warum sich RStudio über die letzten Jahre zur beliebten IDE für Datenanalyst*innen entwickelt hat. Darüber hinaus steuert das RStudio-Team der R-Community eine Vielzahl von themenorientierten Paketen bei, die durch ihre breite Anwender*innenbasis die Community aktiv beeinflussen. Als Beispiele seien die Pakete tidyverse und ggplot 2 genannt.

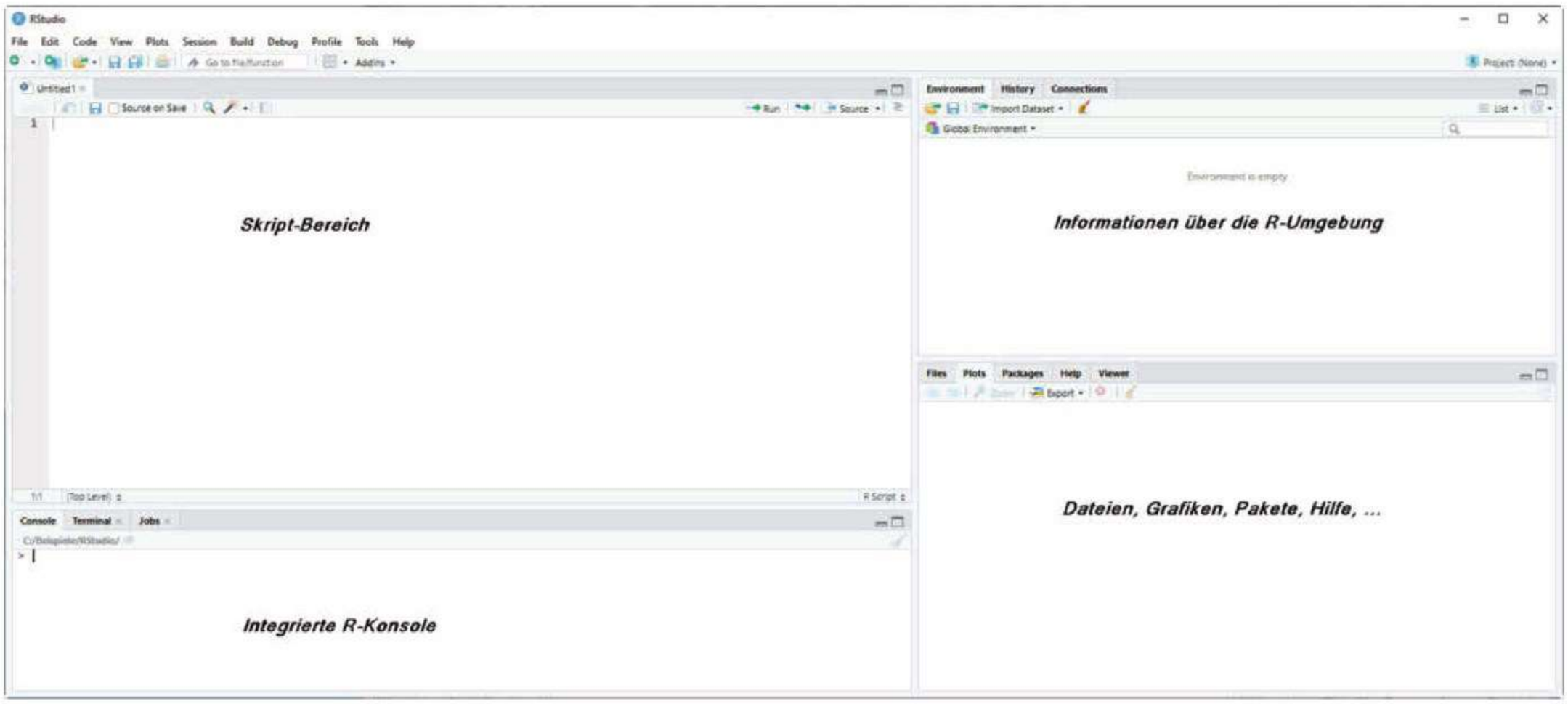
Nicht unerwähnt bleiben soll allerdings Microsofts R Application Network, das eine performanceoptimierte R-Edition unter der Bezeichnung Microsoft R Open (aktuelle Version MS R Open 3.5.3) als Open Source zur Verfügung stellt. Diese Edition bietet Multithreading-Bibliotheken zum optimierten Auslasten von Mehrkernprozessoren insbesondere bei Matrizenrechnungen. Microsoft empfiehlt RStudio als IDE für R Open.

In der Wirtschaft etabliert

Unter den verschiedenen von RStudio PBC angebotenen RStudio-Editionen kann die bisher primär betrachtete Open-Source-Ver-



Der R-Commander bietet eine Menüsteuerung zur schnellen Datenanalyse (Abb. 2).



Die IDE RStudio umfasst die gesamte R-Umgebung (Abb. 3).

sion aufgrund ihrer Eigenschaften als Wegbereiter für kommerzielle Editionen gesehen werden. Wer RStudio schon im Studium kennen und schätzen gelernt hat, möchte damit sicher auch im beruflichen Alltag weiterarbeiten. Die Marketingstrategie von RStudio PBC schließt darüber hinaus auch Anwender*innen von Python ausdrücklich mit ein – eine Programmiersprache, die nicht nur in der akademischen Ausbildung, sondern generell beim Einstieg in die Softwareentwicklung häufig zum Einsatz kommt.

Im Rahmen der individuellen Ausbildung oder für kleinere Teams ist die RStudio-Open-Source-Edition sicher als Startpunkt geeignet, enthält sie doch von RStudio Desktop über RStudio Server bis zum rudimentären Shiny Server alle wesentlichen Bestandteile. Für Anwendungen, die kommerziellen Interessen genügen müssen, reicht diese Edition hingegen nicht mehr. Allein die Darstellung von Datenanalyseergebnissen als Webapplikation über Shiny wird schnell den Einsatz des gehosteten Service Shinyapps.io erfordern – für den RStudio PBC allerdings Lizenzgebühren verlangt.

Die zumindest vorläufig kostenfreie RStudio-Cloud-Edition befindet sich noch im Betastatus. Sie könnte sich zu einer interessanten Alternative zur professionellen Enterprise-Edition entwickeln. Die Enterprise-Edition vermarktet RStudio PBC als RStudio Team in den zwei Versionen Standard und Enterprise. Beide unterscheiden sich in der Anzahl der Nutzer und der Verfügbarkeit. RStudio Team bündelt RStudio Server Pro, RStudio Package Manager und RStudio Connect (unterstützt die Ergebnisveröffentlichung im Unternehmen) und ist für Anwender über entsprechende Lizenzvereinbarungen zugänglich. Die Integration von RStudio Team gestaltet sich je nach den Unternehmensanforderungen unterschiedlich schwierig. RStudio PBC bietet über Servicepartner dazu umfassenden weltweiten Service an.

Die R-Integration in wirtschaftliche Prozesse endet nicht mit einem RStudio-Ökosystem wie RStudio Team, sondern geht deutlich tiefer. Seit Veröffentlichung von SQL Server 2016 lassen sich R-Skripte innerhalb der SQL-Server-Umgebung ausführen, die direkt auf die Datenbank zugreifen. Auf seiner Cloud-Computing-Plattform Azure bietet Microsoft weitere Dienste mit R-Unterstützung an – sie reichen von der Azure-SQL-Datenbank über Azure Machine Learning bis zur speziell auf Data-Science-Anforderungen zugeschnittenen virtuellen Maschine (DSVM). Das DSVM-Image enthält das oben erwähnte optimierte Microsoft R Open, den MS Machine Learning Server, RStudio Desktop und RStudio Server. An RStudio führt also auch hier kein Weg vorbei.

Ein gutes Beispiel für die R-Integration ist die Unternehmenssoftware SAP. Vereinfacht beschrieben löst R-Code über das SQL-Interface und die Calculation Engine R-Prozesse auf der SAP-HANA-Datenbank aus. Die R-Codeausführung mit den selektierten Daten wird an eine R-Instanz übergeben, ausgeführt und die Ergebnisse an den R-Prozess auf der HANA-DB zurückgegeben. Dort erfolgt die weitere Verarbeitung. So entsteht ein effektiver Prozess, um prädiktive Analysen beispielsweise auf Umsatzaussichten zu erstellen.

Bevorzugte Wahl

Aus technischer Sicht gibt es also gute Gründe, R in wirtschaftliche Prozesse zu integrieren. Ob auch Datenanalyst*innen R nutzen, um ihre täglichen Businessanforderungen zu bewältigen, lässt sich nicht so einfach belegen. Es gibt verschiedene Umfragen zu diesem Thema, die aber aufgrund unterschiedlicher Fragestellungen abweichende Ergebnisse liefern. Indizes zur Popularität von Programmiersprachen – darunter IEEE Language Ranking, TIOBE, RedMonk oder Stack Overflow – zeigen, dass beispielsweise Python oft einen höheren Rang als die Statistikumgebung R einnimmt. Die Frage, ob R und Python anhand solcher Rankings vergleichbar sind, soll hier aber weder gestellt noch bewertet werden.

Zweifellos schneidet Python im Allgemeinen besser ab, je mehr aber Datenanalyseprojekte mit Mitteln der Statistik bearbeitet werden, desto stärker rückt die Statistikumgebung R in den Vordergrund. Für Datenanalyst*innen empfiehlt es sich aber grundsätzlich, neben R auch Python und/oder Julia (für hochperformante Anforderungen) zu beherrschen. (map@ix.de)



Günter Faes

ist freiberuflicher Data Scientist und Autor des Buchs „Datenanalyse mit Julia". Er betreibt zur Unterstützung der Statistik- und R-Community Internetseiten und einen YouTube-Kanal.



Sentimentanalyse von Texten mit R

Die Welt verstehen

Manuel Böhm

Am Beispiel von Merkels Regierungserklärung zur Coronakrise lässt sich das automatische Analysieren von Texten gut zeigen. Hier kann die Statistiksprache R ihre Stärken ausspielen.

Das Auswerten unstrukturierter Daten, vor allem von Texten, erfreut sich auch bei den Anwendern der Statistikprogrammiersprache R wachsender Beliebtheit. Forschungs- und Anwendungsgebiete gibt es viele, etwa die Sentimentanalyse, die Empfindungen und Stimmungen messen soll. Ihr Ziel ist, die Polarität eines Textes einzuschätzen: Ist er eher positiv oder negativ gemeint? Ursprünglich wurde diese Technik von E-Commerce-Plattformen wie Amazon eingesetzt, um herauszufinden, ob ein Produkt empfehlenswert ist oder nicht. Der Fokus hat sich nun weit vom Marketing entfernt: Börsenanalysten untersuchen die Laune von Investoren, Politikwissenschaftler klassifizieren Parlamentsdebatten und Sozialwissenschaftler die Diskurse in sozialen Medien.

Speziell bei großen Textmengen bietet sich das Analysieren mit R an. Es gibt dazu zahlreiche Pakete für verschiedene Anwendungszwecke. Das Folgende zeigt Beispiele anhand dieser Bibliotheken:

```
library(dplyr) #für Data Prep
library(tidyr) #für Data Prep
library(tidytext) #für die Textanalyse
library(ggplot2) # zur Visualisierung
```

Beispielsweise wird der Befehl `%>%` aus dem Paket dplyr häufig eingesetzt, um in sogenannten Pipes mehrere Anweisungen nacheinander auszuführen. Die hier genutzten sind aus dem vollständigen Skript herauskopiert, das auf GitHub zur Verfügung steht (siehe ix.de/zkra). Untersuchungsgegenstand ist die Regierungserklärung von Angela Merkel vom 23. April 2020 zur Corona-Epidemie. Es sei darauf verwiesen, dass die folgenden Ausführungen allein dem Zweck dienen, die Textanalyse zu veranschaulichen; eine Aussage zur Qualität der Rede wird explizit nicht getroffen.

Datenaufbereitung zum Text Mining

Zu Anfang steht die gesamte Erklärung in einer einzelnen Zelle eines Dataframes namens `text`. Dataframe bezeichnet hier die typische Darstellung von Datensätzen in R mit den Beobachtungen als Zeilen und den Variablen als Spalten. Das erste Auf-

iX-TRACT

- Verschiedene optionale Pakete liefern zahlreiche zusätzliche Funktionen für die Textanalyse mit R.
- Die Analyse benutzter Wörter zeigt aufschlussreiche Einblicke in jeden Text.
- Der Vergleich unterschiedlicher Reden zeigt den grundsätzlichen Charakter der Beiträge.

bereiten erledigt die Anweisung `unnest_token()` aus dem Paket tidytext. Sie referenziert die Spalte `text` und schafft eine neue Spalte `word` mit einem Begriff pro Zeile. Zuvor wurde die Spalte `text` mithilfe der Befehle `mutate()` und `gsub()` so bearbeitet, dass Nummern und Satzzeichen mit einem Leerzeichen ersetzt und Wörter kleingeschrieben werden. Der Befehl `head()` liefert die ersten Zeilen des neuen Dataframes (siehe Listing 1).

Für eine erste Einschätzung ist diese Form hilfreich, der Befehl `nrow()` gibt 3425 Wörter aus. Problem: Es sind sogenannte Stoppwörter ohne Informationsgehalt dabei – „aber" kommt zum Beispiel 16 Mal vor. Zum Abgleich lassen sich die Stoppwörter des Pakets tidytext benutzen und per `anti_join()` Befehl herausfiltern:

```
text_words <-text_words %>%
  anti_join(get_stopwords(language ="de"))

## Joining, by = "word"
```

Das Ergebnis ist ein Dataframe mit 1676 „Nicht-Stoppwörtern", bestehend aus 1038 Einzelbegriffen, was auf einen abwechslungsreichen Text schließen lässt. Nach diesem Schritt möchte man vielleicht wissen, welche Wörter häufig vorkommen:

```
text_words %>%
  count(word, sort =TRUE) %>%
  top_n(10)

## Selecting by n

## # A tibble: 10 x 2
##    word              n
##    <chr>         <int>
##  1 europa           20
##  2 dass             17
##  3 leben            16
##  4 pandemie         16
##  5 deutschland      14
##  6 virus            14
##  7 menschen         12
##  8 müssen           12
##  9 europäischen     11
## 10 wochen           11
```

Nun zeigt sich schon die politische Natur der Rede und das Thema „Virus". Für die eigentliche Sentimentanalyse hätte man die Stoppwörter im Dataframe zwar behalten können, doch lohnt sich der gefilterte Blick auf die aussagekräftigen Begriffe für eine erste Einschätzung. Die Möglichkeiten für eine weitere Analyse sind vielfältig. In diesem Fall könnte die `stemming`-Funktion die Begriffe auf ihre Grundform minimieren, sodass „europa" und „europäisch" gemeinsam gezählt werden. Ferner gibt es Pakete für Word-Clouds, zum Berechnen der Ähnlichkeit

Listing 1: head() liefert die ersten Zeilen des neuen Dataframes

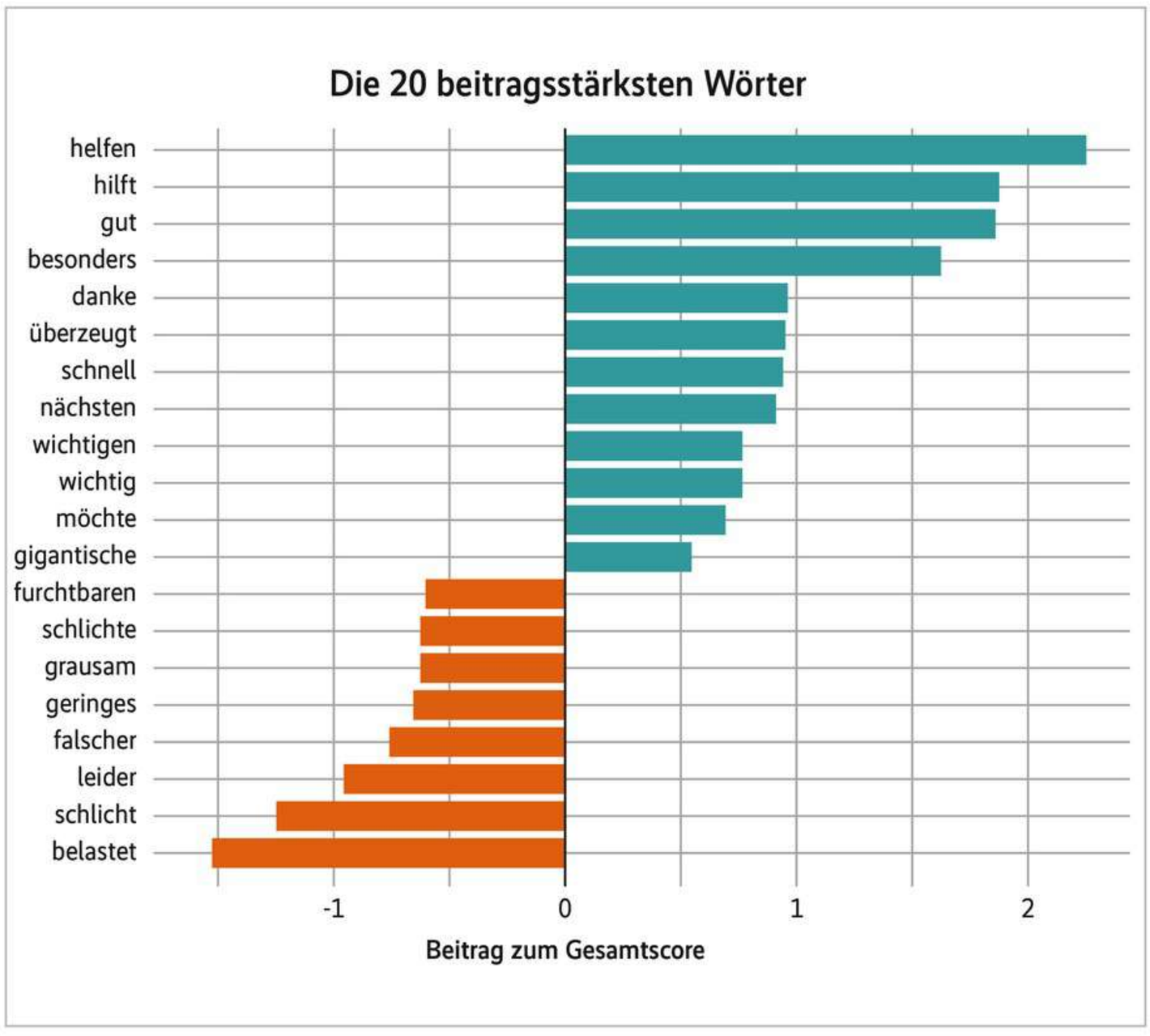
```
library(tidytext)
text_words <- text %>%
  mutate(text = gsub("[0-9]", "", text, fixed = F)) %>% # ohne Nummern
  mutate(text = gsub("[[:punct:]]", "", text, fixed = F)) %>% # ohne
                                                              Satzzeichen
  mutate(text = tolower(text)) %>% # Kleinschreibung
  unnest_tokens(word, text) # Zerlegung in einzelne Wörter
head(text_words)

##           word
## 1         sehr
## 1.1   geehrter
## 1.2       herr
## 1.3  präsident
## 1.4      liebe
## 1.5 kolleginnen
```

verschiedener Texte anhand der benutzten Wörter und zum Erkennen häufiger Wortpaare.

Verknüpfung mit einer Sentimentbibliothek

Basis der Sentimentanalyse ist eine Bibliothek, die positiv und negativ klassifizierte Signalwörter enthält. Vor allem für die englische Sprache kann man auf verschiedene in R-Paketen integrierte Libraries zugreifen. Hier kommen die SentiWS-Bibliothek und der Sentimentwortschatz der Leipziger Universität zum Einsatz (siehe ix.de/zkra). Diese Bibliothek steht unter einer CC-BY-SA-NC-Lizenz, das heißt, man darf sie im wissenschaftlichen Kontext ohne Einschränkungen verwenden. Der Analyst muss zunächst die Textdateien in R einlesen und so aufbereiten, dass ein Dataframe entsteht, in dem jedem Begriff der Score von -1 (negativ) bis +1 (positiv) sowie die Polarität zugewiesen ist. Der Befehl `head()` gibt wieder die oben genannten Einträge des so erschaffenen Dataframes `SentiWS_df` aus:



Die beitragsstärksten Wörter bieten einen aufschlussreichen Einblick in den Charakter der Rede – sowohl inhaltlich als auch zur Polarität (Abb. 1).

```
head(SentiWS_df)

##         word sentiment Polarität
## 1      Abbau   -0.0580  negative
## 2    Abbruch   -0.0048  negative
## 3  Abdankung   -0.0048  negative
## 4 Abdämpfung   -0.0048  negative
## 5     Abfall   -0.0048  negative
## 6     Abfuhr   -0.3367  negative
```

Die hier angewandte Methode basiert auf der Überprüfung dieser Signalwörter in der Rede. Für sie ist in der Bibliothek ein Wert (Klassifikation) hinterlegt, über den der gesamte Text klassifiziert wird. Zum Abgleich des Lexikons mit dem Dataframe `text-words` dient der Befehl `inner_join()` aus dem dplyr-Paket. Er gibt im neu definierten Dataframe nur noch jene Wörter aus, die sowohl im Text als auch im Sentimentwortschatz vorkommen – mit den neu zusammengefügten Spalten aus beiden Dataframes. Die herausgefilterten Zeilen enthalten neutrale Begriffe, die den Sentimentwert nicht beeinflussen. Der Dataframe hat nun folgende Form:

```
text_words <-text_words %>%
  inner_join(SentiWS_df, by="word")
head(text_words)

##               word sentiment Polarität
## 1            liebe    0.1131  positive
## 2 außergewöhnliche    0.3395  positive
## 3     historisches    0.0040  positive
## 4         schlicht   -0.6230  negative
## 5            danke    0.1910  positive
## 6      schwierigen   -0.0245  negative
```

Damit ist die Datenaufbereitung abgeschlossen und die eigentliche Auswertung kann beginnen. Das Format mit einem Begriff pro Zeile erleichtert das Ganze. Die Aufgabe besteht nun darin, zu bestimmen, ob der Text eher positiv oder negativ konnotiert ist. Eine simple Methode stellt die Anzahl positiver und negativer Signalwörter gegenüber. `count()` zählt die beiden Ausprägungen der Spalte `Polarität`:

```
count(text_words, Polarität)
## # A tibble: 2 x 2
##   Polarität     n
##   <chr>     <int>
## 1 negative     61
## 2 positive    190
```

Es entsteht das Bild eines positiven Texts mit rund dreimal mehr positiven als negativen Wörtern. Zum Bestimmen des Sentiments der gesamten Rede lässt sich schlicht die Summe der Spalte `sentiment` heranziehen. Auch dieser Wert zeigt ein positives Bild:

Listing 2: ggplot() visualisiert den Beitrag der 20 stärksten Wörter

```
library(ggplot2)
text_words %>%
  group_by(word, Polarität) %>%
  summarise(contribution = sum(sentiment)) %>%
  arrange(desc(abs(contribution))) %>%
  head(20) %>%
  ungroup() %>%
  mutate(word = reorder(word, contribution)) %>%
  ggplot(aes(x=word, y=contribution, fill = Polarität)) +
  geom_col(show.legend = FALSE) +
  labs(y = "Beitrag zum Gesamtscore",
       x = NULL) +
  coord_flip() +
  labs(title = "die 20 beitragsstärksten Wörter")
```

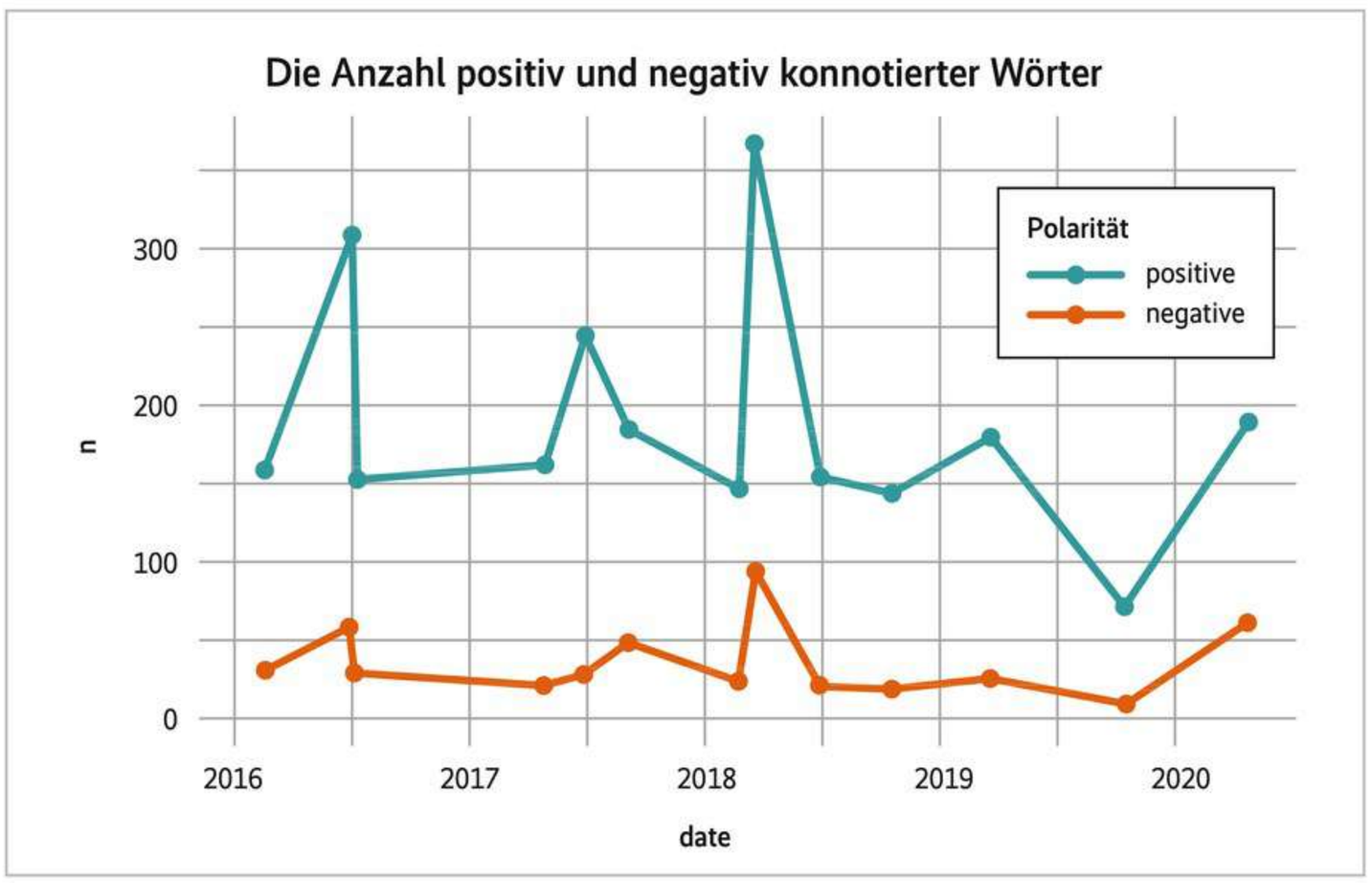
```
sum(text_words$sentiment)
## [1] 9.4853
```

Diese Logik lässt sich auch anwenden, um die 20 Begriffe mit dem größten Einfluss auf den Wert des Textes zu finden. In Listing 2 wird der Beitrag pro Begriff als Summe der hinterlegten Werte berechnet. Das Wort „helfen“ kommt beispielsweise sechsmal vor und hat einen Sentimentwert von `0.373`; es ergibt sich ein Beitrag von `6*0.373=2,238`. Die Gruppierung nach `word` ermöglicht es, die Summe pro Begriff zu berechnen, die Gruppierung nach Polarität braucht man für die Farbfunktion der Grafik (Abbildung 1). Für den `ggplot()`-Befehl werden die `aesthetics (aes)` mit x- und y-Werten angegeben und an den folgenden Befehl für ein Balkendiagramm weitergegeben; die standardmäßige horizontale Darstellung wird mit `coord_flip()` in die Vertikale gerückt.

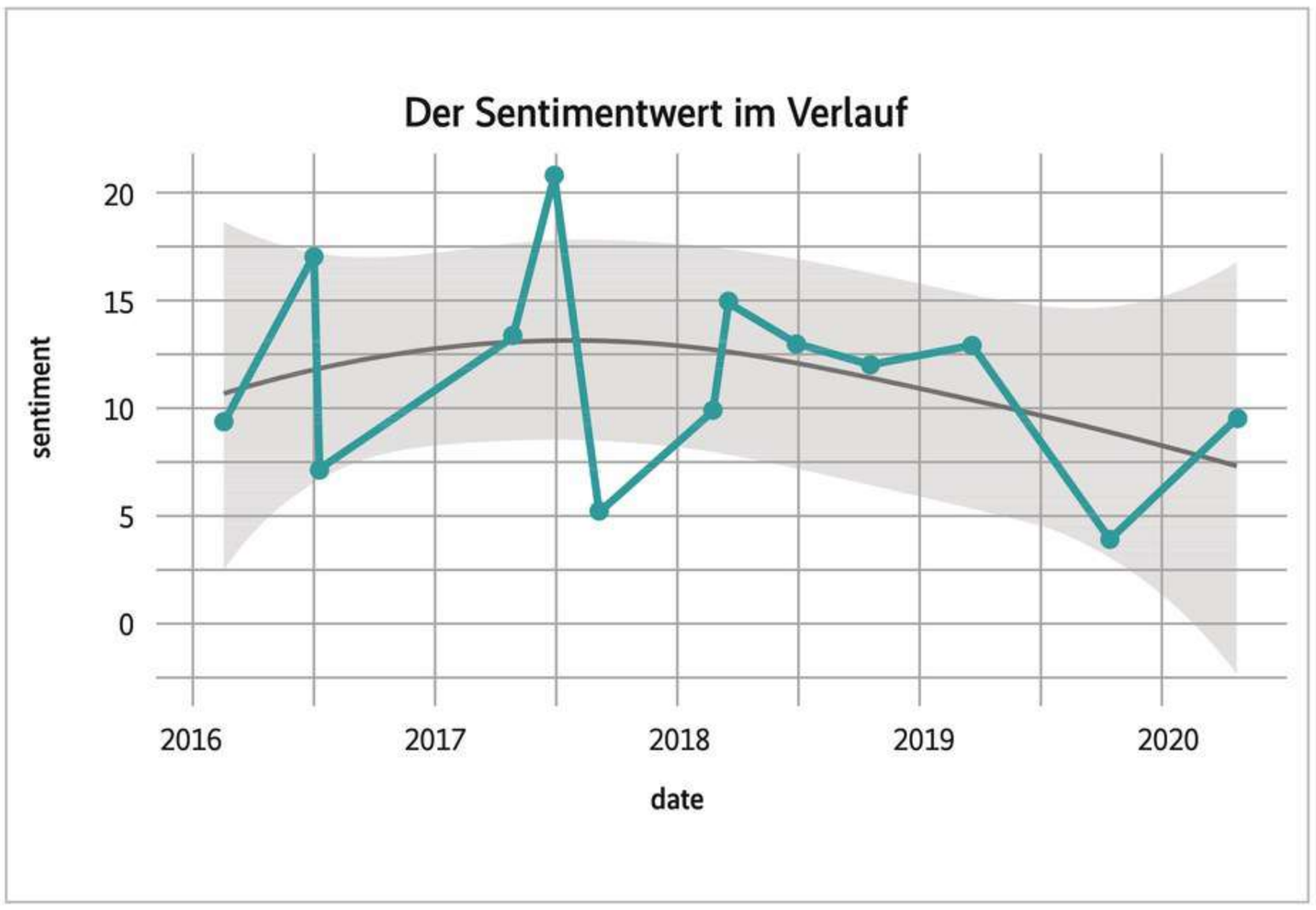
Mit richtigen Formulierungen überzeugen

Unter den hier angegebenen Top-20-Begriffen sind elf positive, die zudem einen höheren Beitrag als die negativen aufweisen. Das erklärt die positive Konnotation der Rede. So überraschend die Prägung angesichts des Themas der Rede sein mag – passend ist sie. Beispielhaft der letzte Satz: „Aber gemeinsam […] wird es uns gelingen, diese gigantische Herausforderung zu meistern: gemeinsam als Gesellschaft, gemeinsam in Europa.“ Die Formulierung soll Zuversicht vermitteln und Handlungsfähigkeit bezeugen, die Resultate der angewendeten Methoden unterstreichen das.

Um die Rede weiter einordnen zu können, lohnt es sich, sie mit ähnlichen zu vergleichen. Denn dass eine Regierungs-



Der Blick auf Reden seit 2016 zeigt, dass dort generell mehr positive als negative Wörter vorkommen (Abb. 2).



Der Sentimentwert zu Merkels Rede zur Pandemie passt in das Muster vergangener Reden (Abb. 3).

erklärung generell mehr positive als negative Wörter enthält, scheint plausibel. Doch was ist ein „normales" Niveau in diesem Zusammenhang? Bei dieser Einschätzung hilft ein Dataframe mit den Regierungsansprachen seit 2016. Er lässt sich dank der schon definierten Aufbereitung in das gleiche Datenformat bringen. Die zusätzliche Spalte `Datum` dient zum Unterscheiden der einzelnen Reden. Der Dataframe `texts_words` hat folgendes Format:

```
head(texts_words)
## # A tibble: 6 x 4
##   date       word              sentiment Polarität
##   <date>     <chr>             <dbl>     <chr>
## 1 2020-04-23 liebe             0.113     positive
## 2 2020-04-23 außergewöhnliche  0.340     positive
## 3 2020-04-23 historisches      0.004     positive
## 4 2020-04-23 schlicht         -0.623     negative
## 5 2020-04-23 danke             0.191     positive
## 6 2020-04-23 schwierigen      -0.0245    negative
```

Die Analyse kann nun wie zuvor stattfinden. Die Anzahl positiver und negativer Wörter pro Rede/Datum lässt sich wieder mit `count()` herausfinden und mit `ggplot()` visualisieren:

```
texts_words %>%
  group_by(date) %>%
  count(Polarität) %>%
  ggplot(aes(x=date, y=n, group=Polarität, color=Polarität)) +
  geom_line(size=1)+
  geom_point(size=2) +
  theme_minimal() +
  labs(title ="die Anzahl positiv und negativ konnotierter Wörter")
```

Listing 3: Die Bestandteile der Grafik werden nach ggplot() ausgeführt

```
texts_words %>%
  group_by(date) %>%
  summarise(sentiment = sum(sentiment)) %>%
  ggplot(aes(x=date, y=sentiment)) +
  geom_smooth(size=1, se=TRUE, span=1, color="grey") +
  geom_line(size=1, color='darkblue')+
  geom_point(size=2, color='darkblue') +
  theme_minimal() +
  labs(title = "der Sentimentwert im Verlauf")

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

Abbildung 2 zeigt, dass eine Regierungserklärung tatsächlich generell mehr positive als negative Begriffe enthält. Doch ist bei dieser Betrachtung die unterschiedliche Länge der Texte zu beachten. Um diesem Umstand Rechnung zu tragen, kann man das Verhältnis der Anzahl positiver zu negativen Wörtern untersuchen. Hierbei sticht ein positiver Ausreißer der Rede vom 29. Juni 2017 hervor; in dieser Ansprache zur EU und G20 prägt vor allem der positiv konnotierte Begriff „gemeinsam" den Text. Der aktuelle Beitrag von Merkel zur Coronakrise fällt als jener mit dem kleinsten Verhältnis positiver zu negativen Wörter auf, was auf eine verhältnismäßig negativ konnotierte Rede schließen lässt.

Der Verlauf des Sentimentwerts ergibt sich wieder als einfache Summe per Datum mit der Befehlskette in Listing 3. Hier zeigt sich die Stärke des Pakets ggplot2; die Funktion `geom_smooth` hinterlegt eine geglättete Linie mitsamt Konfidenzintervall (`se=TRUE`), um eine Bandbreite des gewöhnlichen Sentimentniveaus zu bekommen und Ausreißer zu bestimmen (Abbildung 3). Es fallen zwei Abweichungen auf: Die Rede vom 29. Juni 2017 (EU, G20) fällt mit positiv konnotierten Begriffen wie „gelungen" und „miteinander" auf, die vom 9. März 2017 zur Flüchtlingskrise mit negativen wie „kriminell" und „gezwungen". Auch der Vortrag vom 17. Oktober 2019 zur künftigen Ausrichtung der EU wird von Begriffen wie „leider" und „belastet" bestimmt. Die Rede zur Coronakrise bleibt im Muster. Somit lässt sie sich als eine beschreiben, die verhältnismäßig viele negativ besetzte Wörter benutzt, die allerdings nicht allzu gravierend ausfallen.

Fallstricke und Fazit

Das hier gezeigte Vorgehen gibt trotz seines einfachen Charakters einen aufschlussreichen Einblick in die einzelnen Reden. Es lässt aber auch die Fallstricke erahnen: Man darf nicht vergessen, dass ein solches maschinelles Sieben vor allem ein heuristisches Verfahren zum Klassifizieren großer Textmengen ist, das jedoch beim tieferen Verständnis deutliche Beschränkungen aufweist. Weitere Schwierigkeiten können sich aus der Mehrdeutigkeit und dem Kontext von Begriffen ergeben. Da einzelne Fehleinordnungen bei der Fülle an Produktbewertungen im Internet nur eine geringe Rolle spielen, haben sich Verfahren zum Behandeln solcher Aspekte in anderen Disziplinen herausgebildet. Christian Rauh hat beispielsweise die SentiWS-Bibliothek ergänzt, um Verneinungen in der Analyse zu berücksichtigen (siehe ix.de/zkra). (jd@ix.de)

Quellen

Weiterführende Quellen unter ix.de/zkra



Manuel Böhm

arbeitet als Analyst in der Automobilindustrie. Sein Interesse an Data Science zeigt er auch privat auf seinem Blog aufschrieb.blogspot.com.



Technische Universität Berlin

Bei der Technischen Universität Berlin ist folgende Stelle zu besetzen:

Leiterin*Leiter (d/m/w) des Servicebereiches Informationstechnik und Organisationsmanagement – Entgeltgruppe 13 TV-L Berliner Hochschulen

Teilzeitbeschäftigung ist ggf. möglich.

Zentrale Universitätsverwaltung – Abteilung II Personal und Recht / Servicebereich Informationstechnik und Organisationsmanagement

Kennziffer: ZUV-258/20 (besetzbar ab sofort / unbefristet / Bewerbungsfristende 28.06.2020)

Aufgabenbeschreibung:

- selbstständige und eigenverantwortliche Leitung des Servicebereichs mit 9 Beschäftigten
- Konzeptionierung und Durchführung von ERP-Implementierungs- und Roll-out-Projekten sowie Weiterentwicklung der ERP-Applikationen (SAP)
- Systemadministration und Anwendungsbetreuung der HR-Module im SAP-System
- Gestaltung und Begleitung übergreifender Prozesse in der Zusammenarbeit mit weiteren Bereichen, u.a. Abteilung Finanzen und der Zentraleinrichtung Campusmanagement
- IT-Koordination für die Abteilung Personal und Recht, u.a. Optimierung der Hard- und Software-Architektur, strategische Entwicklung der IT-Infrastrukturen, Verwaltung sowie Kontrolle der IT-Ausgaben
- Erstellung fachlicher Vorgaben für die Abbildung des universitären Organisationsmanagements aufgrund von Kundenanforderungen und Erkenntnissen aus der Qualitätssicherung

Erwartete Qualifikationen:

- erfolgreich abgeschlossenes wissenschaftliches Hochschulstudium (Master, Diplom oder Äquivalent) der Informatik, Wirtschaftsinformatik oder angewandten Informatik oder gleichwertige Fähigkeiten und Erfahrungen
- mehrjährige Erfahrung in der Personalführung und im IT-Bereich, insb. in der Planung und Begleitung von IT-Umsetzungsprojekten
- sehr gute Kenntnisse von Hard- und Softwareumgebungen für IT-Netzwerkstrukturen sowie über Datenbankmanagementsysteme (DBMS), speziell MSSQL
- sichere Anwendung der einschlägigen Office-Softwareprodukte und fundierte ERP-Kenntnisse (SAP-Software, insb. HR-Module)

Der ausführliche Ausschreibungstext ist im Internet abrufbar unter:
http://www.personalabteilung.tu-berlin.de/menue/jobs

Heise Gruppe

JOB GESUCHT?

Ein gutes Team braucht viele verschiedene kluge und kreative Köpfe – und gleichzeitig den Freiraum, diese Potenziale zu entfalten und einzusetzen.

Gestalten Sie mit uns die Zukunft der Medienwelt!

Jetzt informieren und bewerben unter
www.heise-gruppe.de/karriere.

Heise Medien

Online-Fachredakteure (m/w/d) Server & Netzwerke oder Software-Entwicklung



Online-Fachredakteure (m/w/d) Server & Netzwerke oder Software-Entwicklung

Um unser digitales Abo heise+ noch attraktiver zu machen, suchen wir Online-Fachredakteure (m/w/d) mit dem Schwerpunkt Software-Entwicklung oder Profi-Hardware am Standort Hannover zunächst befristet auf 2 Jahre.

Deine Aufgaben

- Als Online-Fachredakteur (m/w/d) schreibst du Ratgeber, Tests und Hintergrundartikel, primär aus deinem Fachgebiet, wie z.B. zu Themen aus der Software-Entwicklung oder zu Profi-Hardware.
- Du setzt Dich intensiv mit Themen aus deinem Fachgebiet auseinander und hältst unsere Leser mit deinen fundierten Artikeln auf dem neuesten Stand.
- Dafür stimmst Du dich eng mit den Ressorts und Redaktionen ab

Deine Talente

- Kenntnisse im Bereich Systemmanagement und Administration, professioneller Hardware (Server, Netzwerke) oder Software-Entwicklung und Programmieren bringst Du bereits mit.
- Darüber hinaus verfügst Du über textsicheres Deutsch, bist sicher im Lesen und Verstehen englischsprachiger Beiträge und hast ein Gespür für prägnante Formulierungen.

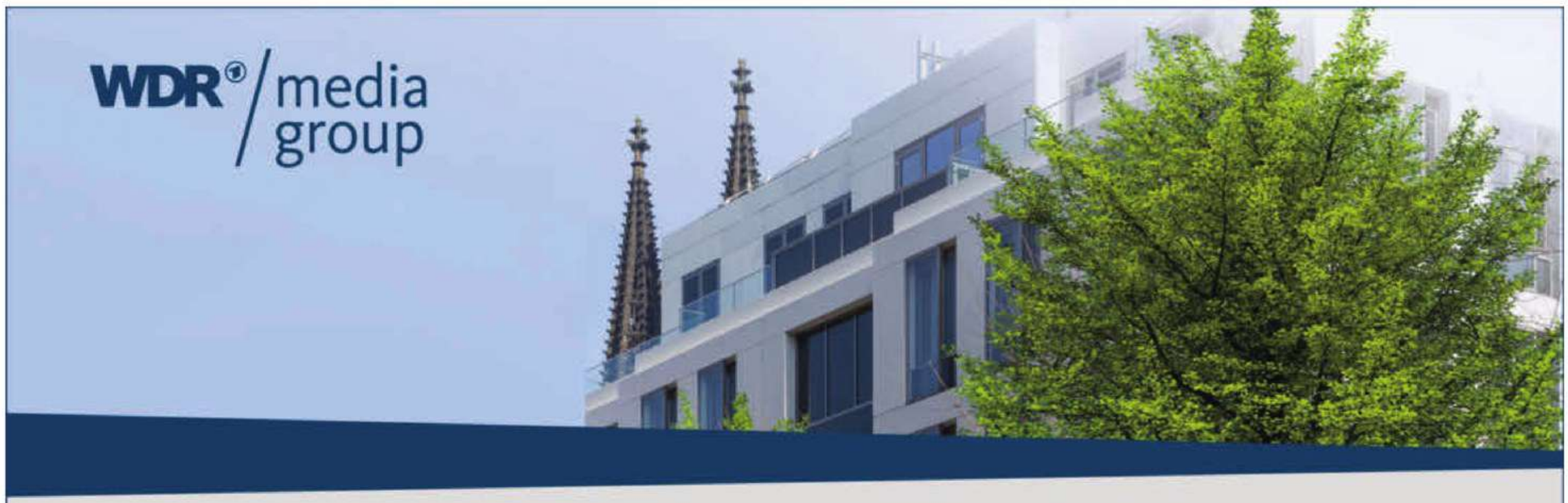
Wir bieten dir

- Die Mitarbeit beim renommierten IT-Medium heise online in einem qualifizierten und engagierten Team.
- Regelmäßige Weiterbildungsmöglichkeiten.
- Viele Benefits, wie eine tarifliche Bezahlung, flexible Arbeitszeiten, eine Kantine, ein Mitarbeiter-Fitnessprogramm und vieles mehr.

Bitte bewirb dich online:
karriere.heise-gruppe.de
Bei uns ist jede Person, unabhängig des Geschlechts, der Nationalität oder der ethnischen Herkunft, der Religion oder der Weltanschauung, einer Behinderung, des Alters sowie der sexuellen Identität willkommen.

Wir freuen uns auf Deine Bewerbung!

Dein Ansprechpartner
Dr. Volker Zota, Chefredakteur heise online
Tel.: 0511 5352-608



Die WDR mediagroup digital GmbH ist ein führender IT- und Broadcast-Service-Dienstleister. Mit ihren Leistungen richtet sie sich vor allem an den öffentlich-rechtlichen Rundfunk. Das Portfolio umfasst neben Managed Services im IT-Arbeitsplatzservice und RZ-Dienstleistungen den Betrieb von automatisierten Digitalisierungssystemen sowie den Betrieb und Support von um-fangreichen IT-Infrastrukturen und Anwendungssystemen. Zudem entwickeln Experten der WDR mediagroup digital Software-Konzeptionen, Web- und App-Anwendungen und bieten spezialisierte Broadcastdienstleistungen an.

Gehen Sie mit uns Ihren nächsten Karriereschritt und übernehmen Sie jetzt am Standort Köln zukunftsweisende und herausfordernde Aufgaben als

Abteilungsleiter (m/w/d) IT-Applikationen

Sie tragen die disziplinarische Verantwortung für die vierzigköpfige Abteilung „Anwendungen". Als Motivator (m/w/d) und Visionär (m/w/d) begeistern Sie Ihr Team für neue Herausforderungen und fördern Potenziale.

Teamleiter (m/w/d) Rechenzentrum

Als erfahrene Führungskraft und echter Teamplayer (m/w/d) übernehmen Sie zukünftig die fachliche Führung eines fünfköpfigen Teams und bilden die Schnittstelle zwischen den Kunden und Delivery-Einheiten.

Senior-Administrator (m/w/d) Linux

Bei der Administration und Betreuung der datenbankgestützten Internetapplikationen des WDR behalten Sie nicht nur die IT-Sicherheit, sondern auch stets die optimalen Ergebnisse im Blick.

Administrator (m/w/d) Softwaremanagement

Sie stellen die Funktionsfähigkeit aller Applikationen in der „private Cloud"-Umgebung und auf den Client-Betriebssystemen sicher. Eine starke Teamarbeit mit anderen Spezialisten hilft Ihnen dabei.

Datenbankadministrator (m/w/d) MS SQL

Mit Ihrem analytisch-strukturierten Arbeitsstil und Freude an Technik gewährleisten Sie gemeinsam im Team die Funktionsfähigkeit von über 400 MS-SQL-Datenbanken.

Junior-Anwendungsadministrator (m/w/d)

Ob bei der Administration von Transcoding-, Hörfunk- und Fernsehanwendungen oder von Applikationsserver-Umgebungen – Sie finden stets elegante Lösungen für komplexe Aufgabenstellungen.

Unser Angebot:

- **Motivierendes Umfeld** – durch abwechslungsreiche Projekte und viel Gestaltungsfreiraum
- **Attraktive Vergütung** – durch Haustarifvertrag, inkl. Familienzuschläge und Weihnachtsgeld
- **Work-Life-Balance** – durch 31 Tage Urlaub, inkl. Freistellung an Heiligabend und Silvester
- **Konstante Karriereentwicklung** – durch individuelle Fortbildungsmöglichkeiten
- **Gute Erreichbarkeit** – durch exzellente Anbindung zum ÖPNV, inkl. Zuschuss zum JobTicket
- **Verpflegung** – durch hauseigene Kantine und vergünstigte Restaurantbesuche (Innenstadt)

Sie finden, das klingt gut? Dann entdecken Sie, welche Vorteile wir noch für Sie bereithalten!
Informieren Sie sich zu den einzelnen Stellenangeboten auf:
https://wdr-mediagroup.com/unternehmen/karriere/offene-stellen

WDR mediagroup digital GmbH | Ludwigstraße 11 | 50667 Köln | www.wdr-mediagroup.com



Quantencomputer programmieren – ein Einstieg

Miniaturwelt

Alexandra Waldherr

Quantencomputer beschleunigen die Analyse komplexer Systeme. Sie eignen sich damit für Kryptografie im Bereich der Telekommunikation ebenso wie für das Berechnen und Simulieren von Quantensystemen.

Für das Verständnis von Quantum Computing ist es zunächst wichtig, den Quantencomputer als radikal andere Hardware und als Backend anzuerkennen. Die Branche arbeitet immer noch mit Gates, allerdings materialisieren sich hier sogenannte Pauli-Gates physisch nicht als Flip von 0 und 1 in der CPU, sondern beispielsweise als Laser- oder Mikrowellenimpulse, die Energieniveaus eingefangener Elementarteilchen (Qubits) rotieren können.

Damit stellen Quantencomputer ein derzeit noch fragiles Experiment dar. Es geht um Transformationen, Kalkulationen und Umformungen sowie um das Nachstellen und Verstehen molekularer Prozesse, aber nicht um das langfristige Speichern von Information. Auch können Qubits nicht kopiert oder eingefügt werden.

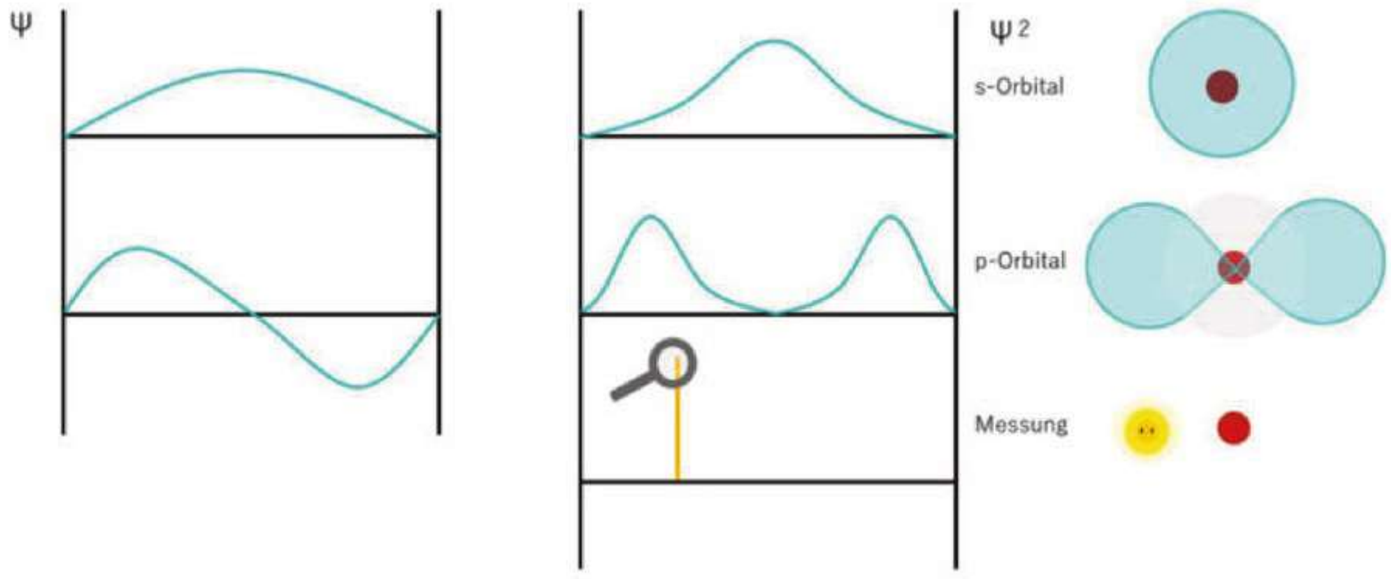
iX-TRACT

- Quantencomputer stellen eine neue Form von Hardware dar.
- Im Blickpunkt steht das Nachstellen molekularer Prozesse, nicht das langfristige Speichern von Informationen.
- Quantencomputer bieten der Forschung eine Möglichkeit, Quantensysteme zu berechnen und zu simulieren.

Quantencomputer bieten kein Copy-and-Paste

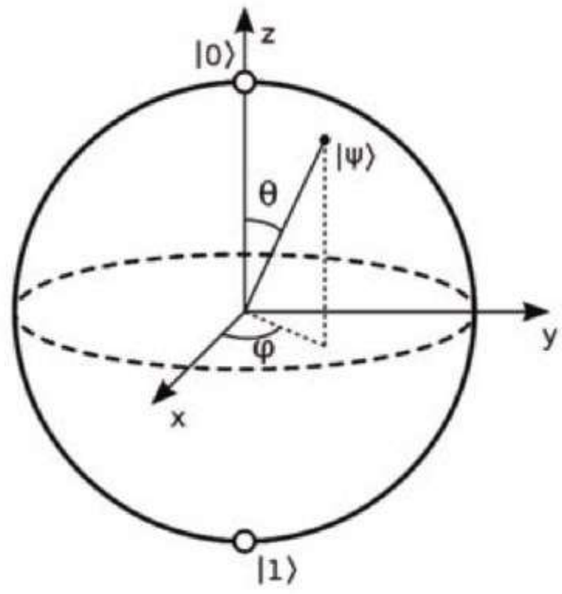
Bevor der Artikel den Qubits anhand eines anschauliches Beispiels ein Gesicht geben wird, lohnt es sich zu verstehen, warum Qubits so viel bessere Kalkulationen ermöglichen und wie das quantenmechanische Auswertungsverfahren dazu eigentlich funktioniert.

Grob gesagt gibt es ab einer gewissen Größe ein Limit, wie weit sich Energieniveaus oder andere physikalische Zustände voneinander unterscheiden können. Daher kommt auch das Wort „Quanten“, was so viel heißt wie „quantisierte Teilchen“. Das Kernproblem in dieser Miniaturwelt tritt dann zutage, wenn mehr Energie als für den Grundzustand notwendig vorhanden ist, aber zu wenig, um das nächste Level zu erreichen. Hier handelt es



Die Abbildung zeigt die niedrigsten Energiezustände pro Schale in einem Atom. Das Quadrat der Wellenfunktion beschreibt den Orbitalbereich. Bei einer Messung fällt diese komplett zusammen, das Elektron ist lokalisiert (Abb. 1).

Quelle: Wikipedia/Smite-Meister



Die abstrakte Manipulation des Zustandes der Bloch-Sphäre drückt man über θ und ϕ aus. Manche Rotationen sind besonders wichtig (Abb. 2).

sich um quantisierte Niveaus; bildlich gesprochen kann man zwischen Erdgeschoss und erstem Stock nicht einfach aus einem Aufzug aussteigen.

Ein Mensch kann sich natürlich real in einem Aufzug zwischen Stockwerken befinden, einem quantisierten Teilchen ist das verboten. In der Forschung wird es an einer der zwei Stellen auftauchen müssen. Haben Teilchen beispielsweise 20 Prozent mehr Energie, befinden sich 80 von 100 weiterhin im Grundzustand. Hat ein Teilchen aber 80 Prozent mehr Energie, wird ein Beobachter es zu 80 Prozent im metaphorischen ersten Stock wiederfinden.

Insgesamt fasst die Wissenschaft alle möglichen Positionen in einer Wellenfunktion Ψ zusammen. Aus der Physik ist bekannt, dass sich Wellenfunktionen nicht real messen lassen. Um die real messbare Wahrscheinlichkeit zu jeder dieser Positionen zu erhalten, müssen Anwender*innen diese Funktion daher zunächst quadrieren. Als anschauliches Beispiel betrachte man Elektronen in einem Atom: Aus der Schrödinger-Gleichung erhalten Anwender*innen pro Energieniveau eine Funktion und jede entspricht quadriert als Ψ^2 – oder bildlicher als Orbitale – den wahrscheinlichen Aufenthaltsorten von Elektronen (Abbildung 1). Das Elektron ist überall zu einer gewissen Wahrscheinlichkeit. Das nennt sich Superposition und birgt die extreme Macht der Quantencomputer, viele Zustände parallel abzutasten. Je größer $\Psi(x)^2$, desto wahrscheinlicher ist ein Messergebnis an Position x. Allerdings kollabiert mit der Messung sofort die Wellenfunktion und als Qubit müsste solch ein Teilchen erneut in die richtige Superposition gebracht werden. Und genau hier versteckt sich das momentane Kernproblem im Quantencomputing, nämlich die Error Rate und die Dekohärenzzeit verfügbarer Quantencomputer. Wenn der Quantencomputer nicht die endgültige Messung durchführt, sondern eine Umweltstörung zuvorkommt, stimmen die erhaltenen Wahrscheinlichkeiten nicht mehr und das System liefert einen Fehler. Derzeit liegen die Dekohärenzzeiten zum Einsatz kommender Superconducting-Schaltkreise bei einigen Nano- bis Mikrosekunden und die Systeme bräuchten mindestens 100 physische Elementarteilchen, um einen ausreichend verlässlichen Qubit darstellen zu können.

Bloch-Sphäre: Visualisierung hilft zu verstehen

Die Frage ist, wie man nun ein Qubit in Superposition bringt. Dazu müssen Anwender*innen einen Quantum Circuit mit diversen Gates erstellen, die den Quantenzustand (entspricht der Wellenfunktion Ψ) verändern. Abstrakt lässt sich das mithilfe der Bloch-Sphäre beschreiben. Auf dieser Kugel würden Nord- und Südpol den klassischen Bits 0 und 1 entsprechen, nur dass das Qubit hier alle möglichen Zustände dazwischen bis zur Messung ebenso abdecken kann. Zwei wichtige Konzepte lassen sich mithilfe der Sphäre illustrieren (Abbildung 2):

- Die Position auf der Bloch-Sphäre gibt den Quantenzustand an und wird als Zustandsvektor bezeichnet. Die zwei Komponenten `a, b` sind komplexe Zahlen, wobei a^2 die Wahrscheinlichkeit angibt, Zustand |0> zu erhalten, und b^2 die Wahrscheinlichkeit, Zustand |1> zu erhalten. Visuell über die Bloch-Kugel: Je näher am Nordpol, desto eher |0> und am Äquator 50 : 50 für |0> : |1>.

$$\Psi = a * |0\rangle + b * |1\rangle$$

Derzeit populäre Sprachen für die Quantenprogrammierung

Die erste strukturorientierte Programmiersprache für Quantencomputer, QCL (Quantum Computation Language), entwickelte Bernhard Ömer 1998 an der TU Wien. Sein Ansatz war ein programmiertes Flussschema (Quantum Circuit), das seriell beispielsweise in Mikrowellenpulse übersetzt und dann an den Qubits ausgeführt wird. Die Übersetzungsebenen waren demnach recht komplex.

Heute sind IBM, Google und Microsoft mit ihren Programmiersprachen Qiskit, Cirq und Q# führend. Alle bieten kostenlose Programme und Zugriff auf Simulationen an, vor allem Q# und Qiskit sind in der Entwicklung bereits recht weit und deren Communitys wachsen stetig. Beispielsweise gibt es eine gut gefüllte Quantum Computing Stack Exchange Platform und mehrere Qiskit-Slack-Channels.

Cirq beruht auf Googles NISQ-Ansatz. Das steht für Noisy Intermediate-Scale Quantum Technology, akzeptiert die derzeit imperfekten und recht unzuverlässigen Quantencomputer und versucht den Fokus auf die Optimierung der momentan zur Verfügung stehenden Ressourcen zu legen. Die Simulationen geben sehr genau Auskunft über die derzeitige Architektur von Googles Chips. Circ ist als Alpha frei verfügbar.

Q# ist ein Teil von Microsofts Quantum Development Kit und baut auf .NET Core auf; Entwickler*innen können die Sprache und das benötigte Tooling somit recht einfach unter Visual Studio und Visual Studio Code einrichten. Der Übersetzungsprozess von Q# an einen Simulator wird meist über C# gesteuert und auch die Syntax ähnelt der Sprache. Lokal lassen sich hier circa 30 Qubits simulieren.

Das Microsoft Quantum Development Kit ist als Preview in Microsofts Visual Studio Marketplace erhältlich (Dokumentation siehe ix.de/zawx).

Mit IBMs QExperience-Plattform können Entwickler*innen nicht nur Quantum Circuits programmieren und simulieren, sondern sie auch direkt über die Cloud auf einem echten Quantencomputer ausführen. Die hier verwendete Sprache Qiskit ist stark an Python angelehnt.

Praxisbeispiel – erste Schritte auf IBMs Quantencomputer

Ein gefürchteter Anwendungsfall für Quantum Computing ist der Shor-Algorithmus, der die weitverbreitete RSA-Kryptografie per Primfaktorenzerlegung einfach knacken könnte. Weitere Anwendungsfälle liegen unter anderem im Bereich der sogenannten Postquantenkryptografie. Hier kann man beispielsweise die fragile Superposition von Qubits nutzen, um einen Lauscher in der Leitung zu identifizieren; angelehnt an das BB48-Protokoll.

Um das Beispiel praktisch selbst nachzustellen, empfiehlt es sich, sich zunächst auf der Quantum-Experience-Seite von IBM zu registrieren. Das System stellt alles Nötige für die Entwicklung und Simulation entsprechender Experimente bereit. Rechts am Bildschirm stehen alle verfügbaren Backends gelistet: Melbourne ist der leistungsstärkste frei verfügbare Quantencomputer mit derzeit 15 Qubits.

Unter „My Account“ befindet sich ein Anmeldetoken. Man kann innerhalb der Plattform arbeiten, im hier gezeigten Beispiel kommt Qiskit per Installation über `pip install qiskit` zum Zug (Listing 1).

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
from qiskit.tools.visualization import plot_bloch_multivector

IBMQ.save_account('#longtokenhere')
IBMQ.load_account()
```

Listing 1: Verbindung zu Qiskit und QExperience

1. Circuit bauen und Gates einfügen: Das Experiment benötigt zunächst einen Quantum Circuit mit zwei Qubits und zwei klassischen Bits. Letztere sind essenziell, um die finalen Informationen verlässlich zu speichern. Ein Hadamard-Gate bringt dann das erste Qubit in Superposition zwischen |0> und |1>. Anschließend sind beide Qubits parallel zu verschränken (das heißt, beide sind im selben Zustand), um sie in Superposition von |00> und |11> zu bringen. Mit-

```
#circuit = QuantumCircuit(Nr. Qubits, Nr. klassische Bits)
circuit = QuantumCircuit(2, 2)

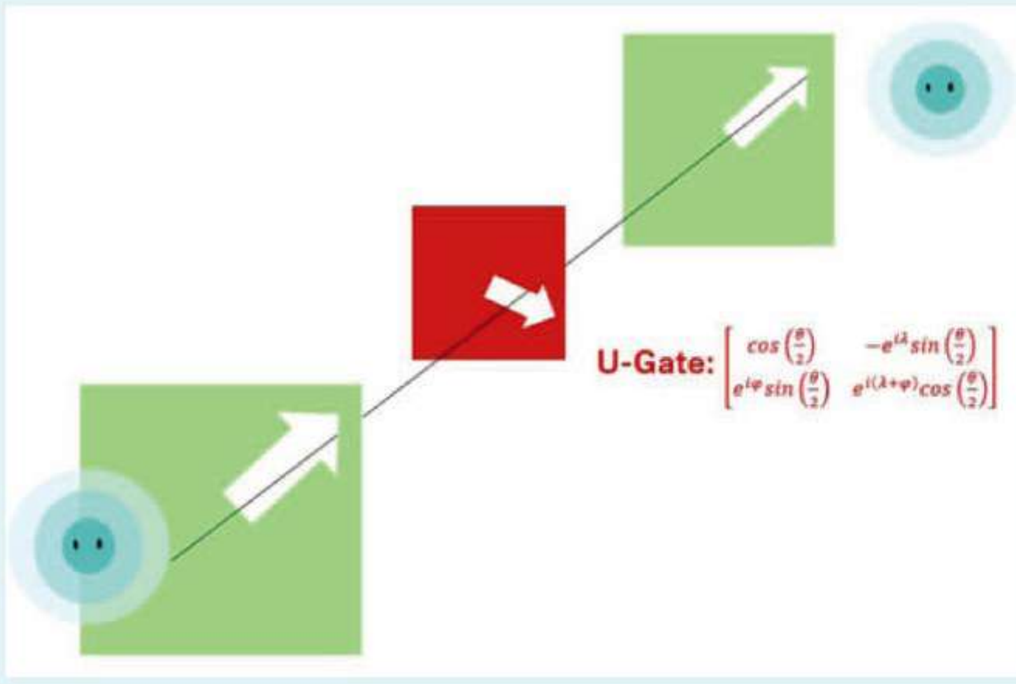

#Bell-Pair Entanglement
circuit.h(0)          #Superposition (control qubit)
circuit.cx(0,1)       #Verschränkung der zwei Qubits


#Barriere ("Übertragung")
circuit.barrier()
#circuit.u3(20, 50, 70, 1)  #-> Eve
circuit.barrier()


circuit.measure(0,0) #Alice
circuit.measure(1,1) #Bob

circuit.draw()
```

Listing 2: Code eines vereinfachten Algorithmus zum Versenden verschränkter Teilchen

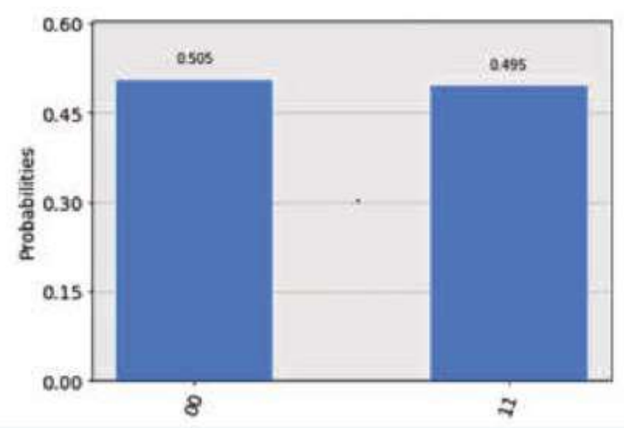


Die Bezugsbasis beim Erstellen und Messen eines Systems sollte übereinstimmen oder zumindest bekannt sein, da sonst die Polung der Bloch-Sphären nicht gleich ist (Abb. 3).

```
#refresh account and load provider
IBMQ.load_account()
provider = IBMQ.get_provider('ibm-q')

#use least occupied quantum computer
qucomp = provider.get_backend('ibmq_london')
results = execute(circuit, backend=qucomp, shots = 500).result()

plot_histogram(results.get_counts(circuit))
```

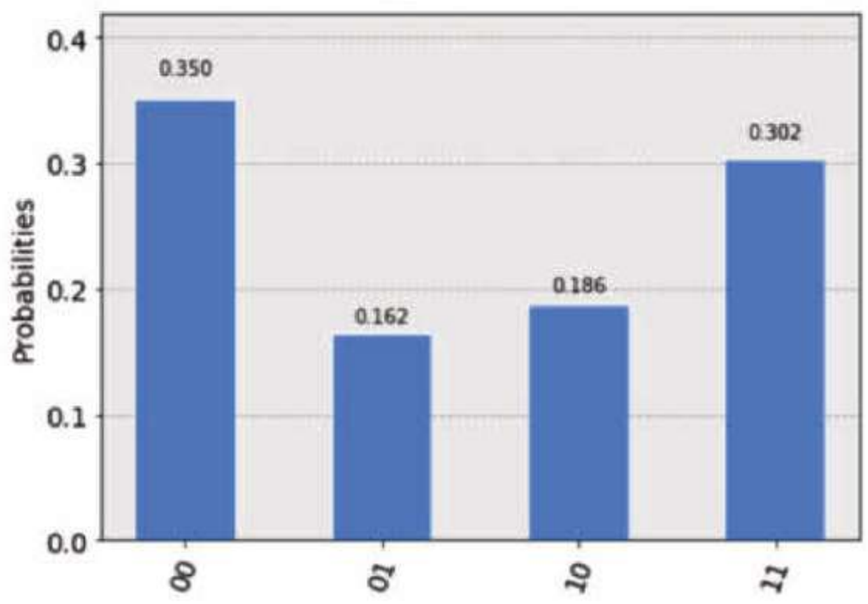


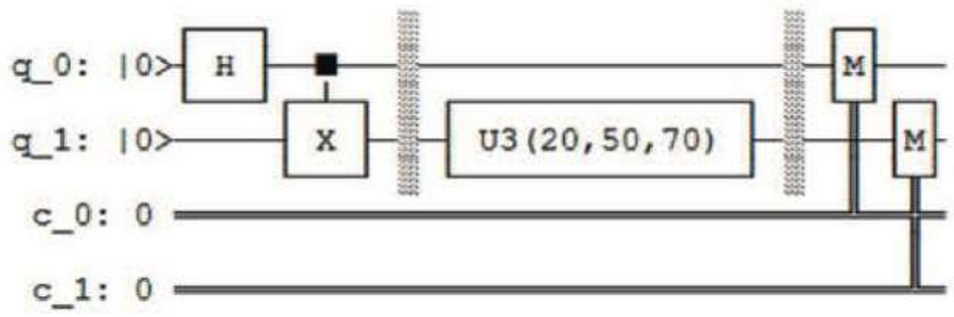
Die Ergebnisse sind nun |00> und |11> im Verhältnis 0,5 : 0,5. In den meisten Runs ist eine Störung der Hardware derzeit nicht unüblich; bei unterschiedlichen Ergebnissen mit |01> oder |10> liegt das Problem bei der äußerst großen Empfindlichkeit der Quantenhardware auf Umwelteinflüsse (Abb. 5).

```
#refresh account and load provider
IBMQ.load_account()
provider = IBMQ.get_provider('ibm-q')

#use least occupied quantum computer
qucomp = provider.get_backend('ibmq_london')
results = execute(circuit, backend=qucomp, shots = 500).result()

plot_histogram(results.get_counts(circuit))
```





Das „lauschende“ U3-Gate bewirkt eine totale Verfälschung, und sobald Alice und Bob auch nur einen Teil der Ergebnisse verglei-chen, merken sie, dass unerwartet viele |01>- und |10>-Zustände vorhanden sind und die übertragenen Daten nicht genutzt werden sollten (Abb. 6).

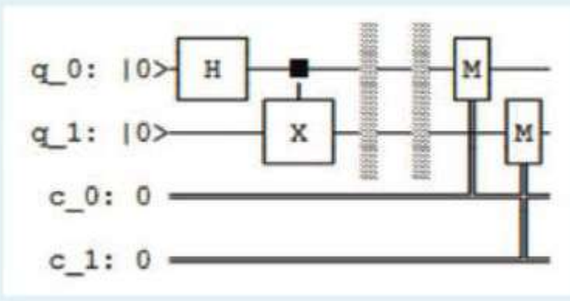
tels Barrieren muss sichergestellt sein, dass die Qubits von der Initialisierung (Senden) zur Messung (Empfangen) abgetrennt sind (Listing 2).

2. Schematische Darstellung auswerfen: Mit dem Befehl `cirquit.draw()` lässt sich der programmierte Circuit als Ablaufschema ausgeben (Abbildung 4).

3. Circuit an den Simulator oder Quantencomputer schicken: Hier können Anwender*innen jetzt auf einen von IBMs Quantencomputern zugreifen und den Algorithmus dort 500 Mal laufen lassen, um anschließend die Wahrscheinlichkeit mittels Histogramm auswerten zu können. Es liegen nur die Zustände |00> und |11> vor (Abbildung 5).

4. Spion (Eve) einschalten: Jetzt kommt der Kniff bei Quantensystemen: Solange nicht bekannt ist, in welcher Orthonormalbasis gemessen wird (vereinfacht gesagt ist die Ausrichtung beider Pole der Bloch-Sphäre nicht bekannt), können Dritte diese nur willkürlich aussuchen. Somit ist bei der Messung mit einem „neuen“ Nord- bzw. Südpol die Entfernung des Zustandsvektors zu |0> bzw. |1> komplett unterschiedlich, wodurch die Superposition falsch kollabiert. Die erhaltenen Qubits sind nicht mehr gleich und verhalten sich in verschiedenen Bezugsbasen bei Messungen von Alice und Bob unterschiedlich. Um diesen Fall nachzustellen, verschiebt man den Beginn des Kommentars zwischen den zwei `cirquit.barrier()` nur noch vor `-> Eve.cirquit.u3(20,50,70,1)` aktiviert einen Lauscher als Zwischensubstanz mit komplett randomisierter Messbasis. Die ersten drei Zahlenwerte lassen sich willkürlich ändern, das vierte Argument gibt an, dass es sich um den zweiten Qubit handelt und im übertragenen Sinne somit die Leitung zu Bob abgehört wird.

Das Circuit-Schema repräsentiert den ersten, unbeeinflussten Kommunikationsweg (Abb. 4).



Das U3-Gate rotiert zwar das Qubit, aber es läuft auf dasselbe hinaus, als wäre die Bezugsbasis des Messapparates falsch eingestellt. Wer jetzt den Circuit noch mal von Anfang an neu baut und an das gewählte Backend schickt, kann eine drastische Änderung erkennen (Abbildung 6). Achtung: Alle Befehle inklusive der Circuit-Definition anfangs sind noch einmal auszuführen! Setzt man den Circuit nicht auf 0, würden die neuen Gates einfach hinten an den bereits bestehenden Circuit angehängt.

- Mathematisch lassen sich die Komponenten in Matrixschreibweise angeben. Die Gates müssen diese Matrix manipulieren, um auf eine Veränderung des Zustandsvektors hinzuwirken. Folglich lassen sich die Gates mathematisch ebenso in Matrizenform ausdrücken, wobei die Argumente die Winkel beschreiben, um die man den Zustandsvektor rotieren möchte.

Der grobe Ablauf für jedes Programm mit Quantencomputern ist simpel:

1. Qubits und klassische Bits initialisieren (Quantum and Classical Register);
2. mit Operationen in Form der üblichen Quantum Gates einen Quantum Circuit bauen;
3. den Quantum Circuit an einen Simulator oder ein echtes Quantum-Backend schicken;
4. Ergebnisse analysieren.

Softwareentwicklung baut auf denselben Modellen auf

Die genannten Programmiersprachen verwenden die gleichen Gates und ähnliche Bezeichnungen, da auch die Softwareentwicklung auf denselben quantenmechanischen und mathematischen Modellen aufbaut. Stereotypisch ist sicherlich das Hadamard-Gate, das ein Qubit in Superposition bringt. Mithilfe des CX-Gates lassen sich Qubits verschränken. Anwender*innen können allein mit diesen zwei Gates bereits Bell-Paare „bauen", was für eine erste Auswertung von Messdaten wichtig ist.

Die fundamentalen Gates seien hier kurz aufgelistet (weitere siehe Tabelle). Leser*innen, die einen tieferen Einblick in Circuits suchen, sollten dazu einen Blick in den Quantum Algorithm Zoo von Stephen Jordan werfen, der als Forscher in Microsofts Quantum Systems Group beschäftigt ist.

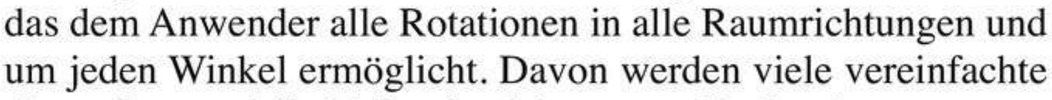
Unitary-Gates: Das elementare Einheitsgatter ist das U3-Gatter, das dem Anwender alle Rotationen in alle Raumrichtungen und um jeden Winkel ermöglicht. Davon werden viele vereinfachte Gates für spezielle Fälle abgeleitet, was die Bestimmung notwendiger Parameter erleichtert.

Matrix-U3-Gate:

$$\begin{bmatrix} cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}sin\left(\frac{\theta}{2}\right) \\ e^{i\varphi}sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\varphi)}cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

Unitary-Gates:

```
circuit.u3(theta, phi, lambda, qubit)
circuit.u2(theta, phi, qubit)
circuit.u1(lambda, qubit)
```



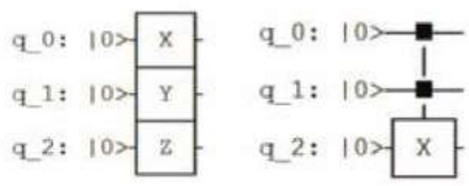
Pauli-Gates: Es gibt drei Pauli-Gates für alle Raumachsen: Sie tauschen die Zustände der x-, y- oder z-Achse, zum Beispiel zwischen |0> und |1> (X-Gate), was dem typischen NOT-Gate entspricht. Daher ist das CX-Gate (controlled NOT) beim Quantum Computing analog zu CNOT. Das CCX-Gate (controlled-controlled-NOT) ist als Toffoli-Gate bekannt.

Pauli-Gates:

```
circuit.x(qubit)
circuit.y(qubit)
circuit.z(qubit)
```

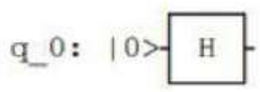
CX-Gate: `circuit.cx(ctr qubit, target qubit)`

Toffoli-Gate: `circuit.ccx(ctr1, ctr2, target qubit)`



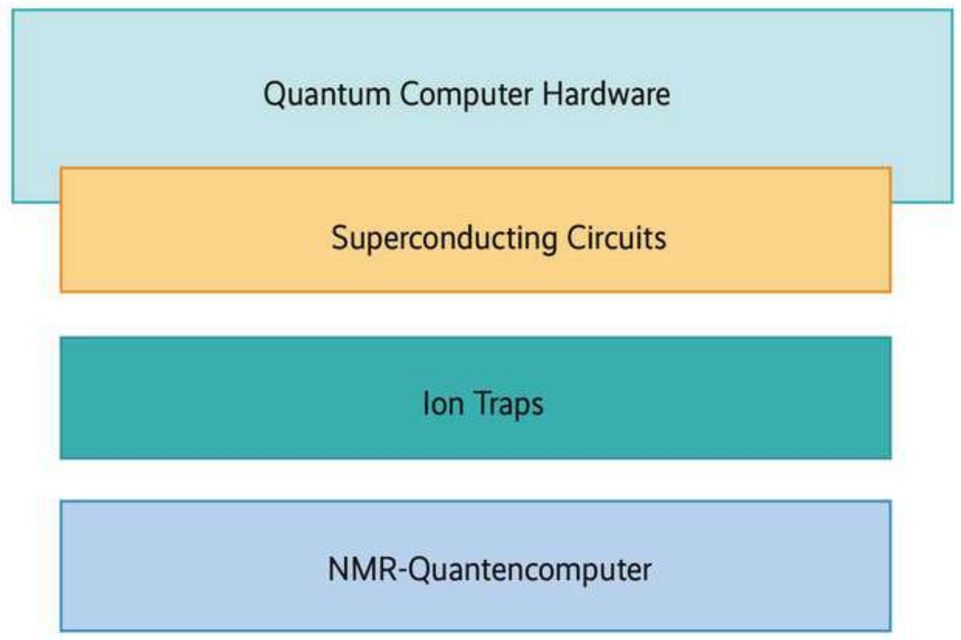
Hadamard-Gate: Das Hadamard-Gate rotiert um die x- und z-Achse und setzt das Qubit von der Anfangsposition |0> auf den „Äquator" zu Position |+>. Damit kann der Anwender eine 50:50-Superposition zwischen |0> und |1> einstellen. Um sich alle Gates vorzustellen, bietet sich in Qiskit der `statevector_simulator` an, der die Ergebnisse gut vergleichbar auf der Bloch-Sphäre visualisiert.

Hadamard-Gate: `circuit.h(qubit)`



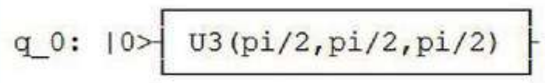
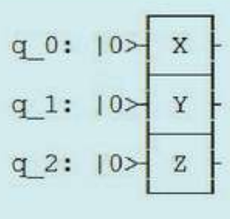
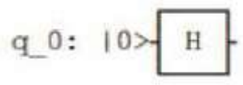
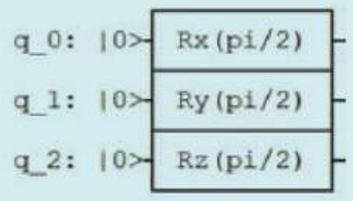
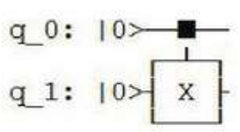
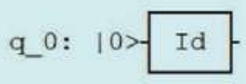
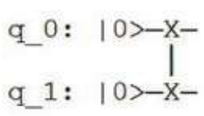
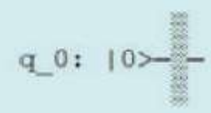
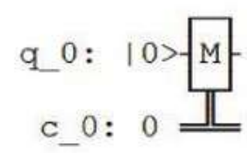
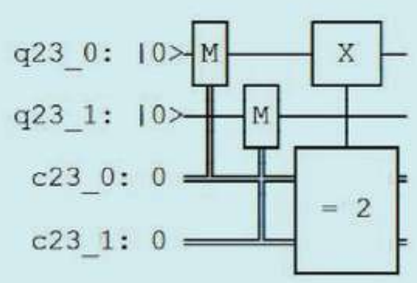
Je nach Anwendungsgebiet können die umgesetzten Systeme unterschiedliche Algorithmen lösen. Es gibt Ansätze zu topologischen und universellen Quantencomputern (vergleiche Abbildung 7). Wenn von Systemen mit über 1000 Qubits gesprochen wird, wie sie derzeit die Firma DWave produziert, handelt es sich um adiabatische „Quantencomputer", die spezifisch für spezielle Optimierungsprobleme gebaut wurden. Es stellt sich nicht die Frage, ob Quantencomputer viel leisten können, sondern wann die Technologie stabil genug ist, die Systeme groß genug sind und wann es genug Softwareentwickler*innen dafür gibt.

Der enorme Vorteil von Quantencomputern ist die vereinfachte und beschleunigte Analyse komplexer Systeme. Daher gibt es viele Überlegungen zu Anwendungen für Finanzgeschäfte oder gekoppelt mit dem Training und der Optimierung von künstli-



Derzeit verfolgt die Forschung drei Ansätze für die Entwicklung von Quantencomputern: Miniaturschwingkreise, die ebenso zwei Energieniveaus einnehmen können, sind industriell am meisten gefragt. Ion Traps manipulieren einzelne Atome, sind aber großteils noch in Entwicklung. NMR-Quantencomputer waren die ersten Modelle, skalieren derzeit aber schlechter als andere Systeme (Abb. 7).

Die wichtigsten Quantengatter (Quantum Gates), deren Befehl in Qiskit inklusive Darstellungsweise im Circuit

Operation	Circuit	Zweck
U3-Unitary-Gate	`circuit.u3(theta,phi,lambda,qubit)` 	Manipulation des Zustandsvektors in allen Raumrichtungen
Pauli-Gates	`circuit.x(qubit)` `circuit.y(qubit)` `circuit.z(qubit)` 	tauscht Zustände durch Rotation um x-, y- oder z-Achse (Bloch-Kugel)
Hadamard-Gate	`circuit.h(qubit)` 	50:50-Superposition, Anfangszustand: Antizustand
Rotationsgatter	`circuit.rx(angle, qubit)` `circuit.ry(angle, qubit)` `circuit.rz(angle, qubit)` 	50:50-Superposition, Anfangszustand: Antizustand
CNOT-Gate	`circuit.cx(ctr,qubit)` 	Pauli-X-Gate, wenn der Kontroll-Qubit sich in Zustand |1> befindet → wird verschränkt! (nicht nur CX, auch CH)
Identity-Gate	`circuit.iden(qubit)` 	entspricht Multiplikation mit der Identitätsmatrix, oftmals Kontroll-Gate (Wert bleibt gleich)
Swap-Gate	`circuit.swap(qubit1,qubit2)` 	tauscht die Zustandsvektoren zweier Qubits
Barriere	`circuit.barrier()` 	verhindert explizit, dass benachbarte Gates gemeinsam ausgeführt werden
Measurement	`circuit.cx(qubit, bit)` 	misst den Zustand des Qubits und speichert das Ergebnis
c_if-Operation	`circuit.gate(qubit).c_if(creg,value)` 	Operation, wenn die klassischen Bits zusammen `== Wert` ergeben

cher Intelligenz. Adiabatische Quantencomputer sowie der VQE-Algorithmus (Variational Quantum Eigensolver) erzielen bereits große Fortschritte.

Mehr noch als nur für Endkonsumenten steigt auch die Relevanz der Technologie für das Sicherheitsbedürfnis innerhalb von Unternehmen. Je größer und verlässlicher die Quantensysteme werden, desto eher besteht die Chance, mit dem Shor-Algorithmus die derzeitige RSA-Kryptografie auf Eis zu legen. Ebenso spannend ist die Entwicklung im Bereich der Telekommunikation. Micius, ein Satellit für ein Pilotprojekt zur Quantenkryptografie, schickt bereits kontinuierlich über unseren Köpfen verschränkte Photonen zwischen Österreich und China hin und her. Das Ziel ist ein Quantenschlüsselaustausch (QKD – Quantum Key Distribution), der in komplexerer Weise mit dem BB48-Protokoll das oben demonstrierte Schema praktisch zwischen Weltall und Erde umsetzt. Dadurch können gemeinsame geheime Schlüssel unglaublich sicher ausgetauscht werden, was speziell eben für Telekommunikation zentraler Bestandteil des Geschäftsfeldes ist. Bisher wurden für die Forschung unter anderem Bilder von Erwin Schrödinger sicher ver- und entschlüsselt.

Gleichzeitig eröffnen Quantencomputer der Forschung eine Möglichkeit, Quantensysteme zu berechnen und exakt zu simulieren. Das verspricht Antworten auf Fragen wie: Welche Moleküle können sich im Körper an Enzyme binden oder Krankheiten heilen? Wie sind wichtige Proteine in unseren Zellen gefaltet? Lassen sich Datenbanken noch schneller durchsuchen? Welche Konfiguration sollen Materialien besitzen, die die Menschheit auf den Mars bringen? Welche Ansätze können Ökosysteme stabilisieren?

Je besser die Forschung die Systeme versteht, desto innovativere Quantencomputer können entstehen (vergleiche Abbildung 4). Schon heute zeigt sich, wie interdisziplinär die Quantenmechanik eigentlich ist. Quantencomputer mögen auf den ersten Blick abstrakt und problematisch klingen, bieten aber fast auf jedem technischen, wirtschaftlichen und naturwissenschaftlichen Feld einen Anwendungsfall. (nb@ix.de)

Quellen

Weiterführende Information zum Quantum Development Kit und Circuits: ix.de/zawx



Alexandra Waldherr

steht kurz vor der Matura und hat bereits am CERN, CeMM und Freudenberg Sealing Technologies mitgewirkt. Ihre Faszination gilt der Physik und Chemie des Kleinen; ihre Diplomarbeit hat sie der umweltfreundlichen Herstellung neuartiger fluoreszierender Pigmente gewidmet.

Julia für die Datenanalyse

Flinke Sache

Günter Faes

Das Designkonzept von Julia bewirkt eine hohe Codeausführungsgeschwindigkeit, die an C/C++ heranreicht. Die Frage ist, ob dies ausreicht, um Datenanalyst*innen von der Sprache zu überzeugen.



iX-TRACT

- Die Multiparadigmensprache Julia liegt seit 2012 als Open Source vor.
- Sie verfügt über eine klare Codestruktur und die Ausführungsgeschwindigkeit des Codes ist hoch.
- Um hohe Performance mit Datenbanken zu gewährleisten, ist eine Datenbank in die Julia-Umgebung integriert.

Die Sprache Julia, ursprünglich für numerisches und wissenschaftliches Rechnen entworfen, kommt immer mehr auch beim High-Performance Computing zum Einsatz. Sie liegt derzeit in Version 1.4 vor (Stand: Mai 2020) und ist seit 2012 Open Source. Merkmale der Multiparadigmensprache sind neben einem dynamischen Typsystem Multimethoden und Koroutinen. Auf kommerzieller Seite ergänzt die Firma Julia Computing, der unter anderem die vier Hauptentwickler der Sprache angehören, begleitende Softwareprodukte.

Datenanalyst*innen nutzen gerne die Statistikumgebung R und die Datenanalysefähigkeit von Python als tägliches Werkzeug. R hat eine lange Entwicklungs- und Optimierungshistorie und bietet über das Paketsystem mehr als 15 500 Pakete und 40 Task Views (nach Aufgaben gebündelte Pakete) an, die Verantwortliche pflegen und auf einem aktuellen und qualitativ hochwertigen Stand halten. Zudem liegt mit RStudio eine integrierte Entwicklungsumgebung und grafische Benutzeroberfläche vor. Python ist mit der Python Software Foundation und der Open-Source-Distribution Anaconda ähnlich komfortabel aufgestellt. Dieser Artikel zeigt, warum Julia trotzdem ein alternatives Werkzeug für Datenanalysen sein kann.

Ein Datenanalyseprojekt besteht aus Prozessschritten, die sich in ihrem Aufwand unterscheiden (Abbildung 1). Der Prozessschritt „Daten vorbereiten“ benötigt nach dieser Abschätzung ca. 50 bis 60 Prozent des Aufwands und wird nicht selten in Projekten unterschätzt. Hierbei geht es um das Selektieren von Daten aus diversen Datenbanken, die Verarbeitung von Datenströmen und um simples Auslesen von CSV-Dateien.

R, Python und Julia bieten für diesen Prozessschritt Pakete und Module zur Bewältigung dieser Aufgaben an. Hier kann Julia ihre Stärken ausspielen. Die Sprache punktet mit einer klaren Codestruktur und einer hohen Verarbeitungsgeschwin-

Geschäftsvorgang beschreiben	Daten verstehen	Daten vorbereiten	Modellbildung	Praxisbeurteilung	Praxis-implementierung
5–15%	5–10%	50–60%	5–15%	5–10%	10–15%

Ein Datenanalyseprojekt gliedert sich in die erforderlichen Prozessschritte und die Aufwandsschätzung (Abb. 1).

Listing: Fahrgastverteilung mit dem Paket OnlineStats

```
using OnlineStats, CSV, Plots, ProgressMeter
path = "train.csv"
file = CSV.File(path; allowmissing=:none)
@show length(file)
o = CountMap(Int)
@showprogress for row in file
    fit!(o, row.passenger_count)
end
plot(o)
```

digkeit. Als Beispiel dient die Herausforderung, Datenströme oder Datenvolumen zu analysieren, die nicht in den Speicher des Computers passen. Das dafür entwickelte Paket OnlineStats (Online Algorithms for Statistics, Models, and Big Data Viz) bietet die Möglichkeit der Prozessparallelisierung an (Abbildung 2).

Preise voraussagen

Die Leistungsfähigkeit von OnlineStats veranschaulicht eine Anwendung zum Vorhersagen des Taxipreises der Plattform Kaggle. Sie bietet unter „New York City Taxi Fare Prediction“ die CSV-Datei train.csv mit 55 Millionen Zeilen und einer Dateigröße von ca. 5,5 GByte an, die Angaben zur Anzahl der Fahrgäste, der Fahrtdauer und der Strecke beinhaltet. Der klar strukturierte Code im Listing bietet eine Übersicht über die Fahrgastverteilung. Auf Basis dieser Daten kann ein Taxiunternehmen die Fahrzeugtypen für seine Flotte festlegen. Aus Sicht von Datenanalyst*innen sind in New York Taxis für zwei Personen ideal und preisgünstiger im Unterhalt.

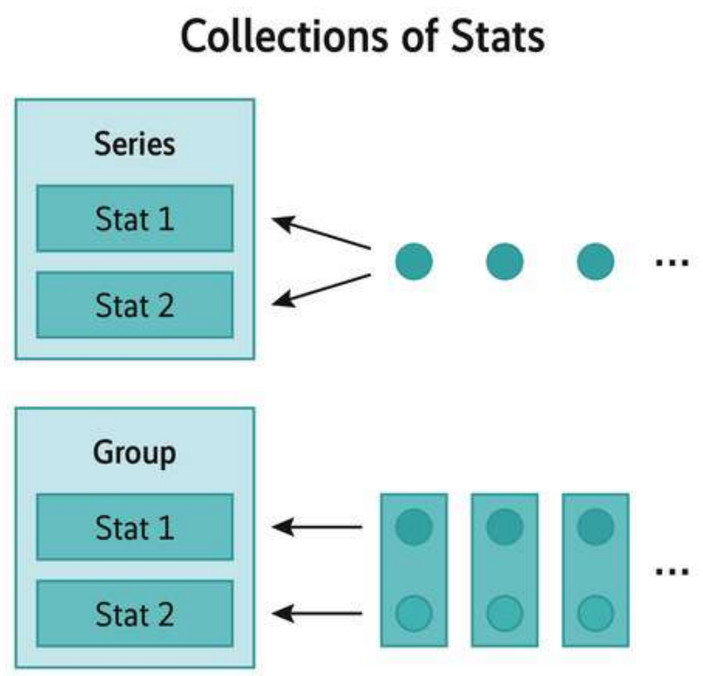
Das Zählen der Fahrgäste und die anschließende grafische Darstellung der Fahrgastverteilung ist in unter 5 Minuten erledigt (Abbildung 3). Diese Größenordnung hängt von der verwendeten Hardware und deren Auslastung ab. Das vorliegende Beispiel verwendet einen i7-6700K-Prozessor mit 32 GByte RAM, die Daten liegen auf einer Samsung SSD 970 EVO (NVMe).

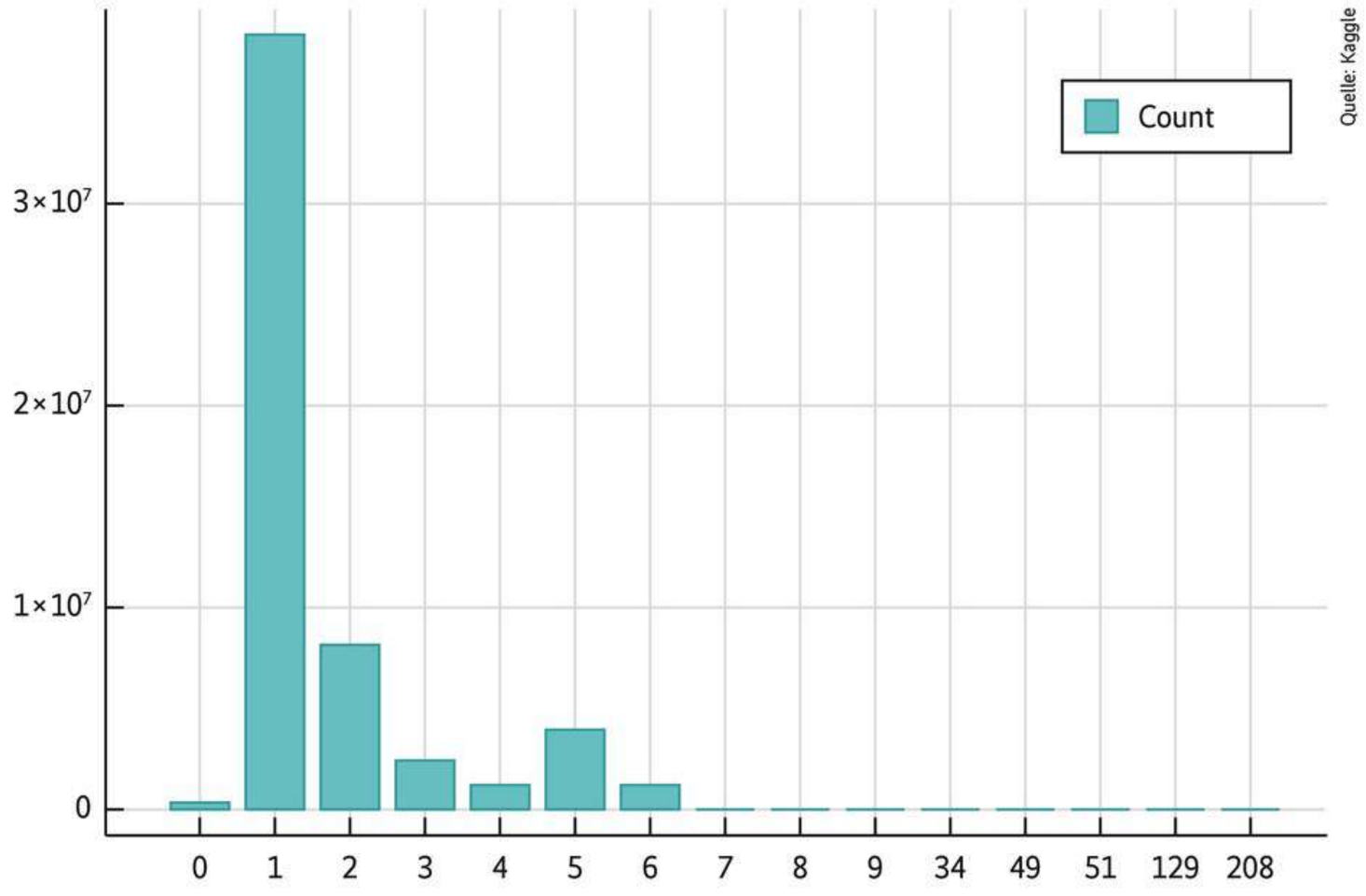
Sind die Fähigkeiten von OnlineStats nicht erforderlich, sondern ist eine Datenselektion aus diversen Datenbanksystemen nötig, gibt es auch dafür entsprechende Pakete. Die Kategorie „Database“ auf der Julia-Packages-Seite enthält eine Übersicht. Julia verfügt über ODBC- und JDBC-Treiber und integriert die Hadoop-Landschaft. Damit erfüllt sie die Erwartungen von Datenanalyst*innen.

Um auch im Umgang mit Datenbanken den Geschwindigkeitsvorteil zu nutzen, wurde JuliaDB geschaffen (Abbildung 4). JuliaDB ist eine rein analytische Datenbank und nicht als transaktionale Datenbank entworfen. Die zu analysierenden Daten werden aus den transaktionalen Datenbanken als CSV-Tabellen selektiert und in der JuliaDB abgelegt. Der Ablauf findet in der Julia-Umgebung statt, wodurch Prozesse wie Kompilierung, Parallelisierung und Analysefunktionen durch OnlineStats zum Zuge kommen. Dadurch lassen sich der Datenvorbereitungs- und der Datenanalyseschritt für zyklische Prozesse effizient verknüpfen.

Das direkte Einlesen von Dateien im CSV-Format ist eine alltägliche Datenquelle für Analyst*innen. Der Code im Listing verwendet eine CSV-Einlesefunktion des Paketes CSV. Im Zusammenspiel mit dem Paket DataFrames bietet sie die eingelesenen Daten in der Julia-Umgebung tabellarisch an. Python- und R-Anwender*innen kennen und schätzen diese Datenstruktur, da sie eine hervorragende Basis für die weiteren Analyseschritte darstellt. Der Julia-Geschwindigkeitsvorteil macht sich nur bei größeren Datenvolumen wie für Dateien in der Größenordnung des Kaggle-Beispiels bemerkbar.

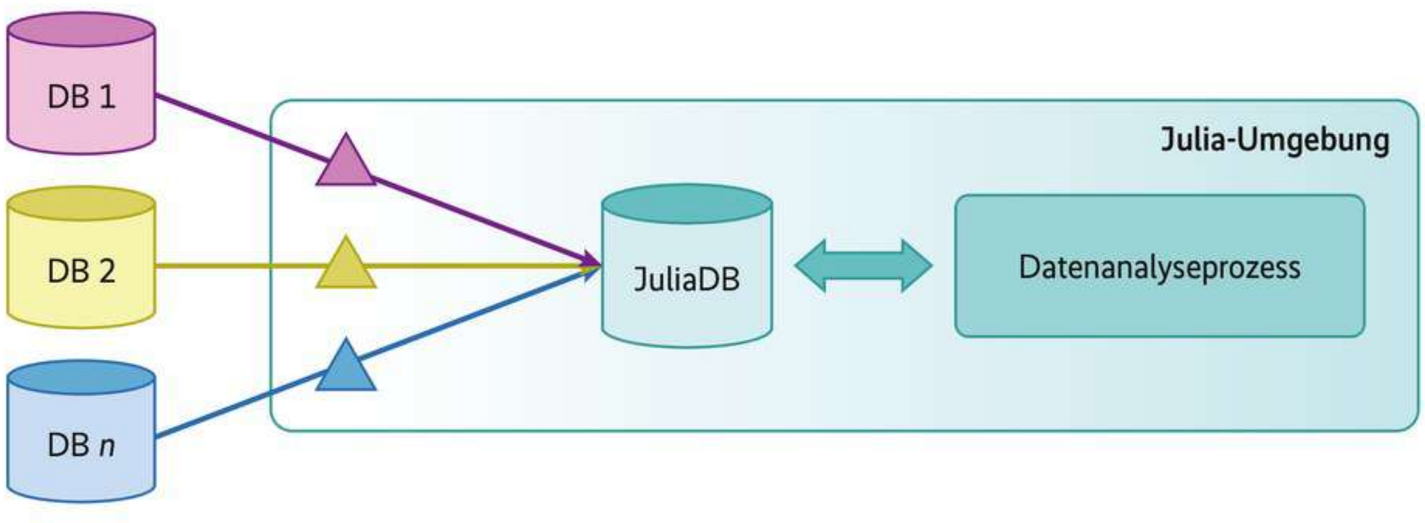
Das Paket OnlineStats unterstützt durch Parallelisierung hochperformante Analyseprozesse (Abb. 2).



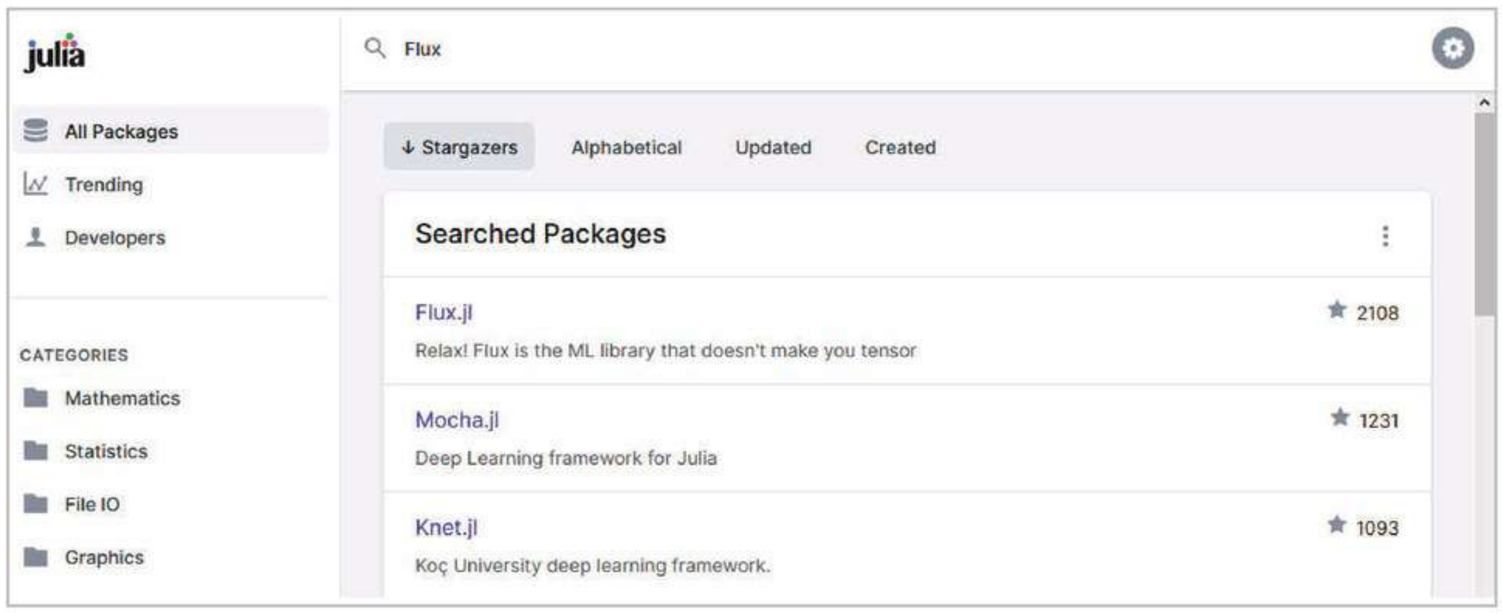


Die Fahrgastverteilung „New York City Taxi“ zeigt die Auswertung von 55 Millionen Taxifahrten (Abb. 3).

Ein weiteres Beispiel für den Geschwindigkeitsvorteil kommt aus dem Bereich „Modellbildung" des Datenanalyseprojekts. Der Aufwand dieses Schrittes ist mit 5 bis 15 Prozent im Vergleich zum Prozessschritt „Daten vorbereiten" als relativ gering eingeschätzt, was auch der aufgewendeten Arbeitszeit entsprechen mag. Bei maschinellem Lernen kommt bisweilen viel Rechenzeit hinzu. Dabei ist es aus Kostensicht unerheblich, ob der Trainingsprozess auf dem Anwendercomputer oder auf gemieteten Cloud-Ressourcen stattfindet. Um diese Kosten zu minimieren, sind zwei Punkte wichtig: Domainwissen und optimale Algorithmen in einer optimalen Umgebung.



Mit der Integration der Datenbank in die Umgebung erzielt Julia eine optimale Performance (Abb. 4).



Das Paket Flux zum maschinellen Lernen erfährt eine hohe Aufmerksamkeit (Abb. 5).

Die statistische Toolbox von Julia nutzen

Liegen Kenntnisse des zu analysierenden Aufgabengebietes (Prozessschritte „Geschäftsvorgang beschreiben" und „Daten verstehen") vor, ist das zur Datenvorbereitung und zur Modellbildung hilfreich. Mit diesem Domainwissen können Analyst*innen auf Basis der nützlichen Daten ein optimales neuronales Netz designen. Ohne dieses Wissen verlängert sich dieser Prozessschritt unnötig. Zum besseren Verständnis der Daten steht die statistische Toolbox von Julia zur Verfügung. Sie steht synonym für eine ständig wachsende und weiterentwickelte Paketsammlung rund um statistische Themen. Fehlende statistische Methoden sind R- oder Python-Funktionen zu entnehmen, die sich über die Pakete RCall und PyCall aufrufen lassen. Zugriffe auf R über RCall sind im Julia-Skript so zu implementieren, dass die dadurch entstehenden Performanceeinbußen so gering wie möglich sind. Das gilt auch für Zugriffe auf Python über PyCall.

Julia stellt optimale Algorithmen und eine optimale Umgebung zur Verfügung. Die Algorithmen stammen aus den Paketen Flux und Knet, mit deren Umgebung sich das notwendige neuronale Netz designen und die GPU einbinden lässt. Diese hochperformante Umgebung und das Domainwissen reduzieren die Machine-Learning-Kosten.

Als Open-Source-Software werden Julia und Umgebung über GitHub administriert. Zurzeit arbeiten etwa 950 Personen mit, bei der Paketentwicklung sind Tausende beteiligt. Ein verantwortliches Team managt die Weiterentwicklung, die freigegebenen Versionen sind bislang stabil.

Die Seite zu Julia Packages (vormals Julia Observer) gibt einen Überblick über die eingesammelten Pakete und bietet eine Suchfunktion. Die Anzahl der Sterne kann ein Maß für die Qualität des Paketes sein, ist aber kein ausreichendes Qualitätskriterium. Das Paket Flux zeigt mit immerhin 2108 Sternen eine hohe Aufmerksamkeit (Stand: Mai 2020, Abbildung 5).

Wohlüberlegt

Hilfreichere Qualitätskriterien sind das letzte Aktualisierungsdatum „Updated Last" und die GitHub-Funktion Insights. Flux ist als Paket zum maschinellen Lernen hervorgehoben und zeigt dadurch einen hohen Pflegegrad und eine aktuelle Dokumentation. Anhand dieser Merkmale lässt sich entscheiden, ob das Paket viel genutzt wird und man es deshalb problemlos in einem Skript oder gar in einem eigenentwickelten Paket einbinden will. Dabei ist zu bedenken, dass der Owner dieses Pakets weitreichende Entscheidungen treffen und dessen Konzept jederzeit ändern kann.

Um die User Experience auf ein hohes positives Niveau zu bringen, gibt es zwei Möglichkeiten: als Erstes einen Indikator „User Experience" einführen, der als Mindestanforderung die Stabilität und die Güte der Dokumentation über Anwender*innenfeedback beschreibt, und als Zweites ein Paketreview durch ein Julia-Gremium durchführen, das sich an ISO/IEC 25000 orientiert.

Beide Kriterien sollten auf der Julia-Packages-Seite vorhanden sein, um die Qualität eines Packages besser einzuschätzen.

Um Julia in einem Unternehmen einsetzen zu können, empfiehlt es sich, bestimmte Kriterien zu erfüllen. Neben dem betriebswirtschaftlichen Nutzen und den betrieblichen Auswirkungen sind das technische Anforderungen. Die verwendeten Pakete sollten getestet und dokumentiert sein.

Unter diesen Aspekten kann Julia ein Werkzeug zur Datenanalyse in Unternehmen sein. Bestehen besondere Anforderungen an die Systemleistungsfähigkeit, dann ist die Sprache Julia das Werkzeug der Wahl. Sie hat einen Platz neben R und Python verdient. (nb@ix.de)



Günter Faes

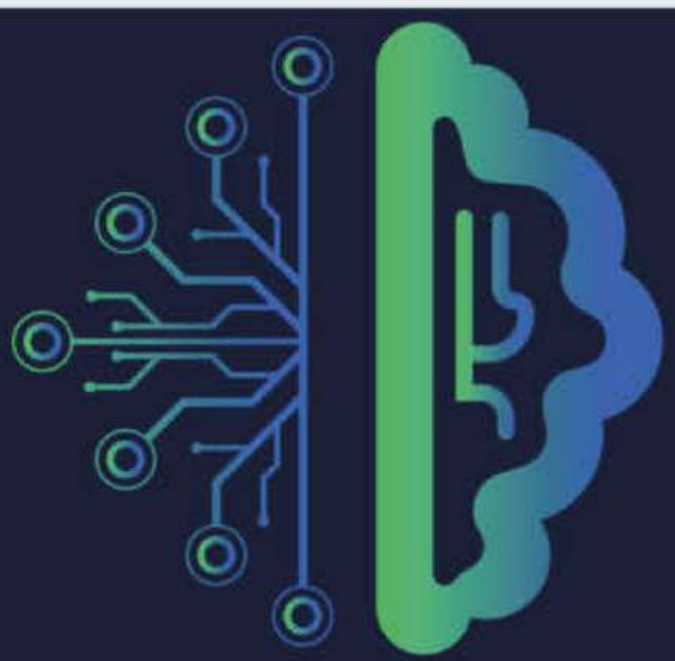
ist freiberuflicher Data Scientist und Autor des Buchs „Datenanalyse mit Julia". Er betreibt zur Unterstützung der Statistik- und R-Community Internetseiten und einen YouTube-Kanal.

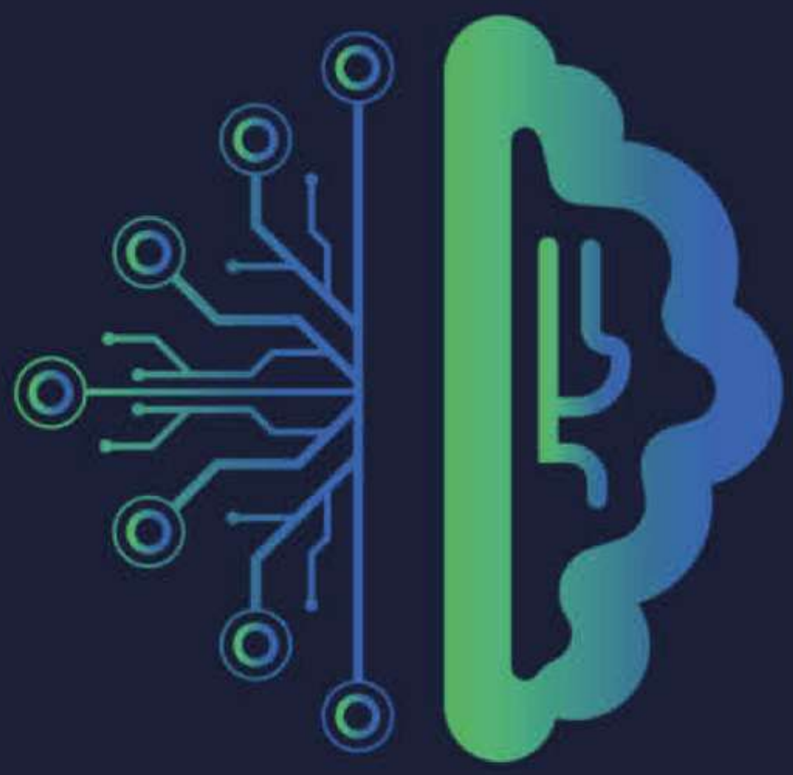
iX Workshop Workshops 2020

Künstliche Intelligenz – EINSTIEG –

Deep Learning mit Tensorflow

22. – 25. Juni 2020 | Online-Workshop





Künstliche Intelligenz – SPEZIALISIERUNG –

Bildverarbeitung mit Deep Learning (CV

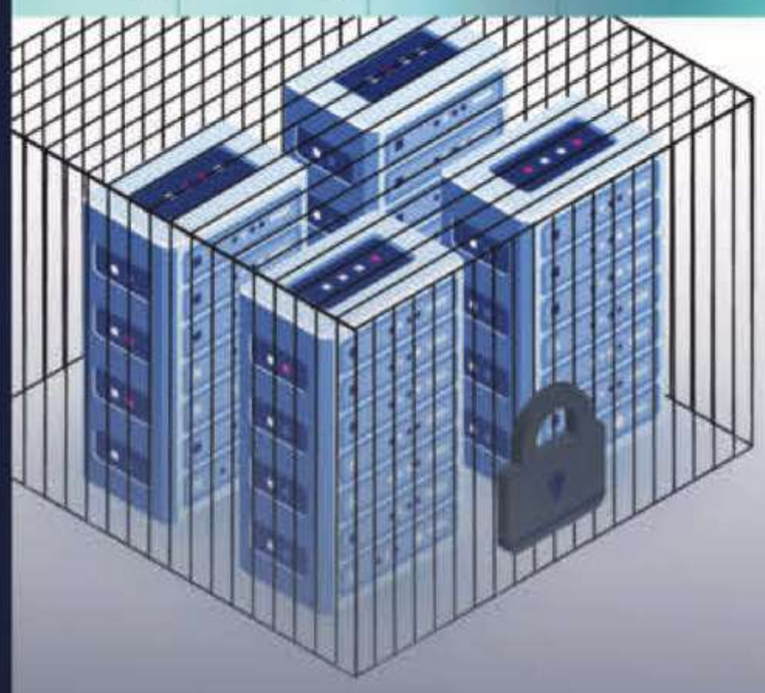
13. – 15. Juli 2020
Online-Workshop



Ihr Weg in die Cloud:

Strategien zur sicheren und effektiven Cloud-Nutzung

29. Juni - 01. Juli 2020
Online-Workshop



IT-Sicherheit - Linux Server härten:

Verschlüsselung, Zugriffskontrolle, Integritätschecks

29. Juli – 01. August 2020
in Hannover



Kubernetes für Fortgeschrittene:

Netzwerke, Service Meshes und Multicluster

31. August – 02. September 2020
in Hannover

Weitere Infos unter:
http://www.heise-events.de/workshops

Bytec ServiceNet

Always Proven To Be Good

Inspired by Johannes Vermeer The Girl With The Pearl Earring

Bytec GmbH Tel. 07541/585-0 www.bytec.de

bytec
IT einfach erfolgreich