We Are Developers!

Herbst 3/2022

Eine Themenbeilage der
Heise Medien GmbH & Co. KG

> **TYPESCRIPT**

Arbeiten mit der Compiler-API

> **CROSS-PLATTFORM**

Native Apps entwickeln mit Kotlin-Multiplattform

> **PROGRAMMIEREN**

Makros in Rust – eine Einführung

> **ARTIFICIAL IMAGINATION**

Interview : KI und Zukunft des Films

> **KI-SYSTEME**

MLOps, XAI und Model Governance

> **MACHINE LEARNING**

Kotlin und Python kombinieren für ML

Eine Community.

Der Zukunft einen Schritt voraus.

Die Zukunft ist nicht planbar, aber wir können sie mitgestalten. Unsere Mission: Wir wollen die Getränke- und Lebensmittelindustrie mit unseren smarten Lösungen so richtig auf den Kopf stellen und ungenutzte Potenziale aktivieren. Wir sind ein internationales Team aus Technologie-Enthusiast:innen, Cloud-Pionier:innen und Daten-Spezialist:innen mit den unterschiedlichsten Fähigkeiten, Talenten und Persönlichkeiten – und suchen Menschen, die genauso bunt sind wie wir. Die Chancen erkennen und Ideen umsetzen. Die sich mit dem Status quo nicht zufrieden geben, sondern immer einen Schritt weiter denken.

Angebissen?
Dann werde Teil unserer Community als:

<Scrum Master / Agile Coach>
<Senior Cloud Software Developer>
<Senior Product Owner>
<Cyber Security Manager>
</uvm...>

New-Work-Umgebung
Mobiles Arbeiten
Kreativer Freiraum

Innovative Technologien
Agiles Mindset
Internationales Setup

KRONES .digital

EDITORIAL

Artificial Imagination

Kreative KI hätte sich noch vor Kurzem keiner vorstellen können – dann kamen in rascher Folge Bildgeneratoren wie DALL·E und Midjourney heraus. Im Spätsommer löste das in Deutschland erstellte KI-System Stable Diffusion eine Revolution aus, denn es ist Open Source und macht Bildsynthese allen zugänglich. Seither tüfteln Profis und Amateure weltweit an passenden Tools, Ablegern und GUIs für Stable Diffusion. Forscher, Coder und Kreative aus dem Umfeld des Projekts sowie aus der Community erweitern laufend das Ökosystem.

Wir haben für das Herbstheft mit Glenn Marshall gesprochen, einem nordirischen Filmemacher, der seit 30 Jahren computergenerierte Kunst erstellt. Marshall ist Visual Artist – und Programmierer. Er treibt das Projekt Deforum voran, das mit Stable Diffusion an Videosynthese arbeitet. Im Interview erklärt er seine Techniken im KI-Kurzfilm „The Crow“ und wagt eine Einschätzung, was Videosynthese für die Filmbranche bedeutet. Wie das neue Genre heißt, ist noch offen: Cyberart, Neural Art, Artificial Imagination? Fest steht für ihn: Die Zukunft des Kinos ist hybrid.

Die übrigen Heftthemen sind passend zum Herbst bunt gemischt: Das Verhältnis zwischen Mensch und Maschine loten Isabel Bär und Kilian Kluge weiter aus in ihrem Beitrag zu Explainable AI und Model Governance. Hauke Brammer zeigt in einem Kotlin-ML-Tutorial, dass Machine Learning nicht nur mit Python gelingt. Ein Kotlin-Tool ist besonders gut zum Multiplattformprogrammieren geeignet, um native Apps zu erstellen: Nils Kasseckert erklärt uns KMM. Timo Zander steuert eine Handreichung zur Arbeit mit TypeScripts Compiler-API bei. Ein weiteres grundlegendes Thema sind Makros in Rust: Unser jüngster Autor Alvin Ramskogler ist davon begeistert und hat eine lesenswerte Einführung verfasst. Wie es gelingt, nicht nur anzufangen, sondern Projekte auch abzuschließen – und dabei als Team in krisengebeutelter Zeit zu bestehen, erklärt Silke Notheis im Gespräch: Stop Starting, Start Finishing.

Hoffen wir, dass die Lichter an bleiben:
Lasst euch inspirieren.

Silke Hahn

INHALT

Young Professionals schreiben für Young Professionals

Unsere Beilage zu c’t und iX basiert überwiegend auf einer Online-Artikelserie, die Young Professionals eine Bühne bietet für erste Fachartikel. Die Autoren und Autorinnen erhalten von der Heise-Developer-Redaktion Unterstützung beim Konzipieren und Schreiben.

Dein erster Fachartikel bei Heise

Die Serie ermutigt dazu, sich selbst erstmals als Autor oder Autorin zu betätigen. Sei es, um eigene Erfahrungen mitzuteilen, ein Projekt vorzustellen – oder einfach, weil du schon immer mal einen Fachartikel schreiben wolltest. Hast du eine Idee?

Schreib uns: developer@heise.de

> Arbeiten mit der Compiler-API

Timo Zander

Hinter einem Compiler steckt mehr als reines Übersetzen von A nach B. Eine Handreichung, um die Arbeit mit dem TypeScript-Compiler und seiner API zu bewältigen.

Wer nicht regelmäßig mit Low-Level-Sprachen wie C++ programmiert, hat selten Kontakt mit den inneren Mechanismen eines Compilers. Auch der Besuch einer Hochschulvorlesung zum Thema Compilerbau entfacht selten Leidenschaft für diesen Teilbereich der Informatik. Doch stark abstrahierte Sprachen wie TypeScript bieten die Chance, dieses Bild zu revidieren: Mit der API des TypeScript-Compilers tsc lassen sich dessen interne Schritte nachvollziehen und sogar eigene Sprachfeatures implementieren, ohne in die Untiefen breitenloser Leerzeichen und anderer Parsing-Gemeinheiten abzutauchen.

Der TypeScript-Compiler ist einer der wenigen, die eine öffentliche und gut dokumentierte Schnittstelle haben. Zwar lassen auch andere Compiler – wie der Java-Compiler – die Ausführung des Kompiliervorgangs per API zu. Doch viele der internen Methoden sind entweder privat und damit nicht aufrufbar oder nicht dokumentiert. Dagegen ist der Quellcode des TypeScript-Compilers ausreichend beschrieben.

Aus TypeScript werde Syntaxbaum

Der TypeScript-Compiler tsc ist im Grunde ein reiner Übersetzer: Er liest den TypeScript-Sourcecode ein und übersetzt ihn in JavaScript als Zielsprache. Die Literatur unterteilt die Arbeit jedes Compilers, abgesehen von Sonderformen wie dem hybriden Compiler, in sechs Phasen: lexikalische Analyse, syntaktische Analyse, semantische Analyse, Zwischencodeerzeugung, Programmoptimierung und Codegenerierung (siehe Abbildung 1).

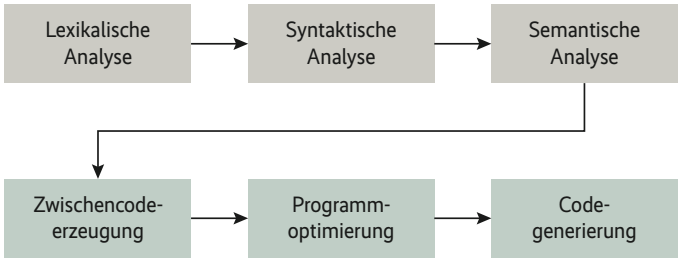
Ein Compiler liest in der lexikalischen Analyse die Eingabe buchstabenweise ein und fasst sie in Lexeme zusammen, also in sinnhafte Buchstabengruppen (etwa Variablennamen oder Operatoren). Diese Lexeme tokenisiert er. Das heißt, er charakterisiert verschiedene Lexeme mit einem Typen (unter anderem Identifier, Nummer oder abstraktes Token wie ein Zuweisungszeichen) und versieht sie mit einem optionalen Wert. Der TypeScript-Compiler implementiert Lexeme zwar nicht explizit, das sonstige Vorgehen ist aber äquivalent (zu diesem Thema und weiteren Aspekten des Artikels stehen weiterführende Hinweise unter ix.de/zr61 bereit).

Im „Hallo Welt“-Beispiel in Listing 1 wird die lexikalische Analyse den Variablennamen als Identifier erkennen. Token wie der Deklarationsoperator `const`, das Gleichheitszeichen oder der String „Hallo Welt“ werden buchstäblich erfasst und mit einem Typen versehen. So ist der String etwa ein Token vom Typ „String“ mit dem Attributwert „Hallo Welt“ (siehe Abbildung 2). Zudem ist die lexikalische Phase von Bedeutung, um überflüssige Leerzeichen oder Kommentare einzulesen und zu ignorieren. Auch die Zuordnung von Zeilennummern zu Befehlen ist Teil dieser Phase, damit der Compiler hilfreiche Fehlermeldungen ausgeben kann.

Die syntaktische Analyse wendet die spracheigene Grammatik auf die eingelesenen Token an. Nach den Regeln dieser Grammatiken entsteht in dieser Phase dann ein abstrakter Syntaxbaum (Abstract Syntax Tree, kurz: AST), der für spätere Phasen des Kompilierens grundlegend ist (siehe Abbildung 3). Der TypeScript-Compiler implementiert diese Phase in seinem Parser. In der darauffolgenden semantischen Analyse erstellt tsc aus dem Syntaxbaum Symbole:

In a Nutshell

- Was passiert, wenn der TypeScript-Compiler Quellcode in JavaScript übersetzt?
- Mit der angebotenen Compiler-API lässt sich die Arbeit des Compilers schrittweise nachvollziehen.
- Das Umsetzen eigener Projekte wie etwa eines Linters ist einfach möglich mit den API-Funktionalitäten.



>> Phasen, die ein Compiler üblicherweise durchläuft (Abb. 1)

Sie besitzen einen Namen sowie Flags, zum Beispiel `EnumMember`, `Class` oder `Function`, wodurch sie charakterisiert sind. Doch auch ihre Deklarationen, Kind-Elemente oder mögliche Exporte sind in ihnen gespeichert, wodurch Symbole umfangreiche Informationen für alle weiterführenden Kompilierschritte bieten. TypeScripts „Binder" speichert die Symbole in einer Tabelle. Hierbei erkennt das Programm Konflikte, etwa doppelt verwendete Namen, und kann sie entweder als Fehler melden oder je nach Szenario ignorieren (mehr hierzu unter ix.de/zr61). Die Aufbereitung der Symbole findet zwar typischerweise während der lexikalischen Analyse statt, aber der TypeScript-Compiler führt sie bewusst zu einem späteren Zeitpunkt durch.

Das Herzstück des TypeScript-Compilers

Daraufhin kann der TypeScript-Compiler mithilfe des Syntaxbaums und der erzeugten Symbole den zweiten Teil der semantischen Analyse durchführen: die Prüfung der Typsicherheit. Im über 45 000 Zeilen langen Type-Checker `checker.ts` implementiert er eine Vielzahl von TypeScript-Features. Er vergleicht Typen, prüft Interface- und Klassenhierarchien, garantiert die korrekte Verwendung von Klassen- und Typsymbolen und vieles mehr. Auch das Generieren hilfreicher Fehlermeldungen wie „Variable X ist kein Teil dieser Klasse, meintest du vielleicht Y?" ist ein Bestandteil davon (Listing 2).

Der Umfang des rund 200 Seiten starken TypeScript-Handbuchs deutet bereits auf die Komplexität der Sprache hin (mehr dazu unter ix.de/zr61). Auch die Methode im vorherigen Listing, die überprüft, ob zwei Typen miteinander kompatibel sind, unterstreicht diesen Eindruck: Die Implementierung umfasst über 2000 Zeilen.

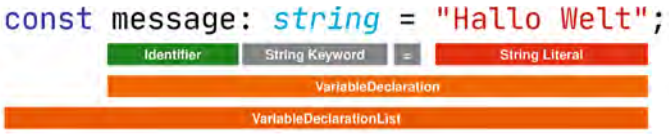
Nach der lexikalischen, syntaktischen und semantischen Analyse erzeugen typische Compiler je nach Implementierung meist Zwischencode, der näher an der Ziel- beziehungsweise Maschinensprache liegt als der Ausgangsquellcode. Mithilfe dieses Codes führen sie dann die Programmoptimierung durch. Hierzu zählen Auswertungen

Was ist ein hybrider Compiler?

Während „normale" Compiler den Quellcode meist in Maschinensprache übersetzen, verfügen hybride Versionen über eine Zwischensprache. Sie wird mithilfe eines Interpreters zur Laufzeit ausgewertet. Java ist das prominenteste Beispiel für diesen Typ: Javas Virtuelle Maschine (JVM) interpretiert zur Laufzeit den Bytecode, den der Compiler aus dem Java-Code erzeugt hat.

Listing 1: Einfacher „Hallo Welt"-Ausdruck in TypeScript

```
const message: string = "Hallo Welt";
```



>> Ein simpler Ausdruck wird durch den Compiler in verschiedene Token unterteilt (Abb. 2).

>> Beispiel eines abstrakten Syntaxbaums von TypeScript (Abb. 3)

von Ausdrücken zur Compilezeit (etwa einfache arithmetische Operationen) oder das Entfernen überflüssiger Variablen. Nach dieser Optimierung generiert der Compiler den finalen Code und gibt ihn aus. Auch hier sticht der TypeScript-Compiler durch eine Besonderheit heraus: Das Optimieren von Quellcode ist ein erklärtes Nichtziel von TypeScript und entfällt.

Auch Zwischencode erzeugt tsc nicht, sondern gibt fertigen JavaScript-Code mithilfe seines Emitters direkt aus. Der Emitter kümmert sich unter Beachtung der Konfiguration des Compilers um die korrekte Ausgabe der fertigen JavaScript-Dateien, sodass sie sowohl den richtigen Inhalt haben als auch an der korrekten Position gespeichert werden. Das Generieren von Source Maps ist Teil davon (siehe Kasten „Source Maps verbinden JavaScript und TypeScript“).

Der Compiler ist vollständig in TypeScript geschrieben: Das liegt am Compiler-Bootstrapping, das Microsoft bei der Entwicklung von TypeScript genutzt hat. Was zunächst nach einem Henne-Ei-Problem klingt, ist das Standardvorgehen im Compilerbau und somit ein wichtiger Meilenstein in dessen Reifegrad.

Einfache Idee, knifflige Umsetzung: TypeScripts Designphilosophie

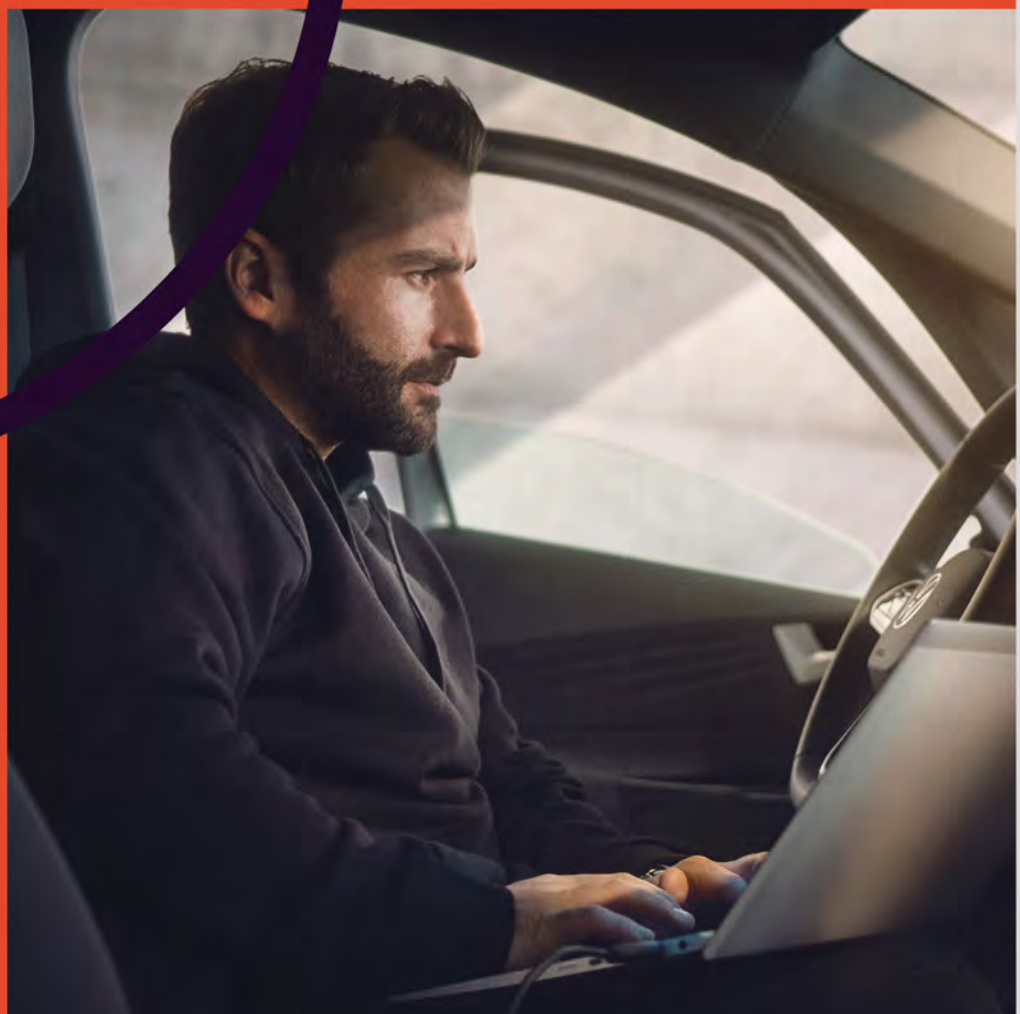
Das Konzept hinter TypeScript ist simpel: Man versehe JavaScript mit Typsicherheit und füge einige Überprüfungen hinzu. Trotzdem ist der TypeScript-Compiler alles andere als einfach gestrickt und umfasst Zehntausende Zeilen Code. Die Komplexität hat einen Ursprung: JavaScript lässt sich nicht einfach durch eine andere Sprache ersetzen. Der ECMAScript-Standard für JavaScript hat vor allem offenbart, dass Browserhersteller träge in der Adaption sind und Standards sich zudem als nicht so einheitlich entpuppen, wie sie scheinen. Daher sind TypeScripts Designziele eine wichtige Grundlage, um zu verstehen, dass tsc nicht arbiträr komplex, sondern nachvollziehbar entworfen ist.

Als Superset zu JavaScript möchte TypeScript die Sprache nicht fundamental ändern, sondern ergänzen. Im Gegensatz zu ähnlichen Projekten wie CoffeeScript ist jedes funktionale JavaScript-Programm auch korrekter TypeScript-Code. Daher soll der Compiler vor allem Fehler reduzieren und die Entwicklung angenehmer gestalten, indem er Typsicherheit und neue Sprachkonstrukte bietet. Der Pragmatismus sorgt dafür, dass formal prüfbare Korrektheit nicht zu den Zielen der Sprache zählt – auch wenn das Internet längst zeigen konnte, dass das Typsystem Turing-vollständig ist und somit theoretisch in der Lage, jedwede Berechnung auszuführen. Aus berechnungstheoretischer Perspektive ist es also ebenso mächtig wie Java, Ruby, C++ und moderne Programmiersprachen.

Wer mit der TypeScript-Entwicklung beginnt, stößt rasch auf die Einschränkung, dass TypeScript keine Typinformationen zur Laufzeit bietet und stattdessen Type Guards zu implementieren sind. Auch das resultiert daraus, dass TypeScripts Arbeit nach dem Kompilieren endet und dessen Output von der JavaScript-Engine als reines ECMAScript in-

Listing 2: TypeScript-Methode, um die Kompatibilität zweier Typen zu prüfen

```
/**
 * Checks if 'source' is related to 'target' (e.g.: is a assignable to).
 * @param source The left-hand-side of the relation.
 * @param target The right-hand-side of the relation.
 * @param relation The relation considered. One of 'identityRelation', 'subtypeRelation', 'assignableRelation', or 'comparableRelation'.
 * Used as both to determine which checks are performed and as a cache of previously computed results.
 * @param errorNode The suggested node upon which all errors will be reported, if defined. This may or may not be the actual node used.
 * @param headMessage If the error chain should be prepended by a head message, then headMessage will be used.
 * @param containingMessageChain A chain of errors to prepend any new errors found.
 * @param errorOutputContainer Return the diagnostic. Do not log if 'skipLogging' is truthy.
 */
function checkTypeRelatedTo(
    source: Type,
    target: Type,
    relation: ESMap<string, RelationComparisonResult>,
    errorNode: Node | undefined,
    headMessage?: DiagnosticMessage,
    containingMessageChain?: () => DiagnosticMessageChain | undefined,
    errorOutputContainer?: { errors?: Diagnostic[], skipLogging?: boolean },
): boolean;
```



You can't make everybody happy. Except for millions of drivers.

Develop a game-changing software architecture for all Volkswagen Group brands, markets and segments. Join us!

Apply now and join our team!

CARIAD

A VOLKSWAGEN GROUP COMPANY

terpretiert wird. Alternative JavaScript-Laufzeitumgebungen wie Deno könnten das in Zukunft ändern (wie genau, dazu mehr unter ix.de/zr61).

Vom TypeScript-Enum zur JavaScript-Variablen

Mit der Interoperabilität zu JavaScript im Hinterkopf lohnt es sich, konkrete Funktionen des Compilers zu betrachten, um diese Denkart in der Praxis zu sehen. Viele Sprachbestandteile – wie etwa Typdeklarationen – werden im Kompilat ausgelassen. Schließlich haben sie während des Kompiliervorgangs, in der semantischen Analyse, ihren Dienst getan und das Überprüfen der Typsicherheit ermöglicht. Eine dynamische Laufzeitüberprüfung erfolgt nicht. Andere Sprachbestandteile wie Enums bedürfen einer Übersetzung.

Um diese spracheigenen Konstrukte zu übertragen, geht der TypeScript-Compiler stets ähnlich vor: Zunächst liest er den TypeScript-Code ein, parst und tokenisiert ihn und wandelt ihn in den AST um. Mit der Methode `parseExpected` erkennt tsc währenddessen auch Syntax- oder Semantikfehler und verwandelt sie in Fehlermeldungen, wie Listing 3 demonstriert.

In der Ausgabe ist das Enum dann ein sofort ausgeführter Funktionsausdruck (Immediately Invoked Function Expression, IIFE). Der Vorteil ist, dass sich beim Minifizieren des ausgegebenen JavaScript-Codes mehr Buchstaben einsparen lassen als beim Umwandeln des Enum in eine reine Variable (Listing 4).

Der schlanke Code macht sich zunutze, dass JavaScript beim Zuweisen einer Variablen ihren Wert zurückgibt. So evaluiert der Ausdruck `Colors[Colors["Yellow"] = 2]` zu 2, sodass das JavaScript-Objekt beim Indexaufruf den korrek-

Source Maps verbinden JavaScript und TypeScript

Browser führen TypeScript nicht direkt aus, sondern den kompilierten JavaScript-Quellcode. Für Entwicklungszwecke ist das unkomfortabel, da so etwa das Debuggen mit Breakpoints oder das Zurückverfolgen von Laufzeitfehlern zu Codezeilen nicht möglich wäre.
Source Maps schaffen hier Abhilfe und erzeugen eine Art Übersetzungshandbuch, mit dem der Debugger und ähnliche Tools den Ursprung des Codes zurückverfolgen können. Das Konzept stammt aus der JavaScript-Welt. Dort dienen Source Maps dem Zweck, optimierten und minifizierten JavaScript-Code mit seinem menschenlesbaren Ausgangszustand zu verknüpfen.

Listing 3: Mit dieser Methode parst der Compiler das Enum-Sprachkonstrukt

```
function parseEnumDeclaration(pos: number, hasJSDoc: boolean, decorators: NodeArray<Decorator> | undefined, modifiers: NodeArray<Modifier> | undefined): ↵
                                                                                                                                        EnumDeclaration {
    parseExpected(SyntaxKind.EnumKeyword);
    const name = parseIdentifier();
    let members;
    if (parseExpected(SyntaxKind.OpenBraceToken)) {
        members = doOutsideOfYieldAndAwaitContext(() => parseDelimitedList(ParsingContext.EnumMembers, parseEnumMember));
        parseExpected(SyntaxKind.CloseBraceToken);
    }
    else {
        members = createMissingList<EnumMember>();
    }
    const node = factory.createEnumDeclaration(decorators, modifiers, name, members);
    return withJSDoc(finishNode(node, pos), hasJSDoc);
}
```

Listing 4: Ein TypeScript-Enum nach der Umwandlung in puren JavaScript-Code

```
var Colors;
(function (Colors) {
    Colors[Colors["Red"] = 0] = "Red";
    Colors[Colors["Green"] = 1] = "Green";
    Colors[Colors["Yellow"] = 2] = "Yellow";
})(Colors || (Colors = {}));
```

Listing 5: Beispielhafte Auswahl von ES2020-Funktionen, für die der Compiler Abwärtskompatibilität bietet

```
interface User {
    name?: string;
}

const Max: User = {};
console.log(Max?.name)
```

Listing 6: Die Ausgabe moderner ES2020-Funktionen durch den Compiler variiert je nach Zielsprachniveau

```
// Target < ES 2020
console.log(Max === null || Max === void 0 ? void 0 : Max.name);

// Target >= ES 2020
console.log(Max?.name);
```

Listing 7: Die Umwandlung von TypeScript zu JavaScript erfordert nur eine Zeile Code

```
import * as ts from "typescript";

const inputSourceCode = /* read input file here */ "";
const result = ts.transpileModule(inputSourceCode, { compilerOptions: {
module: ts.ModuleKind.CommonJS }});

console.log(result.outputText);
console.log(JSON.stringify(result.diagnostics));
```

Listing 8: Eine beispielhafte TypeScript-Klasse als Compiler-Eingabe

```
class Person {
  private type: string | null = null;
  protected age: number = 23;

  constructor(public name: string, public userName: string, private ↵
                                                        email: string) {
    this.name = name;
    this.userName = userName;
    this.email = email;
  }

  public printAge = () => {
    console.log(this.age);
    this.setType(this.age < 18 = 'jung' : "alt");
  }

  private setType = (type: string) => {
    this.type = type;
    console.log(this.type);
  }
}

const person = new Person('Franz', 'fmueller', 'example@email.com');
person.printAge(); // Prints: 23
```

Listing 9: Ausgabe einer TypeScript-Klasse in gewöhnlichem JavaScript

```
var Person = /** @class */ (function () {
    function Person(name, userName, email) {
        var _this = this;
        this.name = name;
        this.userName = userName;
        this.email = email;
        this.type = null;
        this.age = 23;
        this.printAge = function () {
            console.log(_this.age);
            _this.setType(_this.age < 18, 'jung', "alt");
        };
        this.setType = function (type) {
            _this.type = type;
            console.log(_this.type);
        };
        this.name = name;
        this.userName = userName;
        this.email = email;
    }
    return Person;
}());
var person = new Person('Franz', 'fmueller', 'example@email.com');
person.printAge(); // Prints: 23
```

ten Wert zurückgibt. Im Klartext bedeutet das: `Colors["Yellow"]` ergibt 2, `Colors[2]` ergibt „Yellow“.

Angepasst an jede ECMAScript-Version

Die jährlichen ECMAScript-Neuerungen erleichtern die Arbeit des TypeScript-Compilers. TypeScript implementiert diese Features oft weit vor ihrer offiziellen Einführung durch die Browserhersteller. Daher enthält tsc für jedes Sprachniveau einen Transformer, der in Listing 5 nicht vorhandene Sprachfeatures in abwärtskompatibles JavaScript umwandelt.

Das Null-sichere Verketten von Eigenschaften war beispielsweise eine der Neuerungen von ECMAScript 2020 (ES2020). Ist der TypeScript-Compiler allerdings auf eine ältere Sprachversion konfiguriert, übersetzt er die Funktion nicht nativ, sondern bildet sie in JavaScript nach (Listing 6).

Im weitesten Sinne ist dieses Übersetzen neuerer Features in älteres Standard-JavaScript das Äquivalent zur Zwischencodeerzeugung normaler Compiler. Der TypeScript-Compiler erstellt also stets JavaScript in der modernsten Sprachversion (ES.Next) und wandelt es in das entsprechende Ziellevel um.

Die bereitgestellte API des TypeScript-Compilers kommt primär für interne Zwecke zum Einsatz und wird daher selten als Feature genannt. Lediglich ein GitHub-Wiki-Artikel führt grob in ihre Funktionsweise ein und demonstriert anhand anschaulicher Beispiele ihren Nutzen (der Link zum Artikel findet sich unter ix.de/zr61). Die Schnittstelle ermöglicht zahlreiche praktische Einsatzszenarien.

So ist es etwa einfach, mit wenigen Befehlen ein TypeScript-Programm mithilfe der API zu kompilieren und als JavaScript-Datei auszugeben. Alle Compilereinstellungen, die

Listing 10: Eingabe für den DIY-Linter mit einigen stilistischen Fehlern

```
// Expliziter Return-Type erforderlich (void)
function test() {
  return;
}

function test2(): number {
  return;
}

// ...

// Der any-Type soll nicht explizit deklariert werden
// Variablen sollen nach camelCase-Notation benannt sein
let TestVaRiaBlE: any;

// ...

// Variablen namens "Type" müssen Enums sein, keine Strings
const objectType: string = 'User';
```

```
C:\Program Files\nodejs\node.exe .\out\ex01_linter_01.js
ex01_input.ts (2,1): A function must declare an explicit return type
```

>> Der Linter gibt die Fehlermeldung samt Codeposition aus (Abb. 4)

für gewöhnlich die tsconfig.json konfiguriert, finden hierbei Beachtung, wie Listing 7 verdeutlicht.

Die Schnittstelle ist einfach gestaltet, sodass das Übersetzen von TypeScript zu JavaScript in einer Codezeile möglich ist (Transpiling). Es ist dabei spannend, sich die Kompilate für verschiedene TypeScript-Features anzuschauen, um auch die Sprache und ihre technischen Limitationen zu verstehen. In der Variable `diagnostics` liefert der Compiler Fehler- und Warnmeldungen aus dem Kompiliervorgang (Listing 8).

Listing 9 zeigt die Ausgabe einer TypeScript-Klasse in gewöhnlichem JavaScript.

Der Kreativität sind technisch kaum Grenzen gesetzt: Codegenerierung, Umwandlung von XML oder JSON in TypeScript-Code und zurück oder doch das reine Generieren einer Codedokumentation aus dem Quellcode – das alles ist möglich. Während viele Ideen eher Spielerei sind, bergen einige Einsatzzwecke tieferen praktischen Nutzen.

Listing 11: Struktur des mit tsc-API selbstgebauten Linters

```
import { readFileSync } from "fs";
import * as ts from "typescript";

export function customLinter(sourceFile: ts.SourceFile) {
  lintNode(sourceFile);

  function lintNode(node: ts.Node) {
    // Hier können Regeln implementiert werden

    ts.forEachChild(node, lintNode);
  }

  function report(node: ts.Node, message: string) {
    const { line, character } = sourceFile.getLineAnd↵
                                          CharacterOfPosition(node.getStart());
    console.log(`${sourceFile.fileName} (${line + 1},${character ↵
                                                    + 1}): ${message}`);
  }
}

// Datei einlesen
const fileName = "ex01_input.ts";
const sourceFile = ts.createSourceFile(
  fileName,
  readFileSync(fileName).toString(),
  ts.ScriptTarget.ES2015,
  /*setParentNodes */ true
);

customLinter(sourceFile);
```

Do-It-Yourself-Linter mithilfe der Compiler-API

Einen eigenen Linter zu erstellen, ist solch ein nützlicher Einsatzzweck, denn Linter zählen zu den Standardwerkzeugen aller Entwicklerinnen und Entwickler. Der ESLint-Regelsatz für TypeScript ist der Standard-Linter in der JavaScript-Welt. Konkret zeigt Listing 10, wie sich einige beliebte `eslint-typescript`-Regeln nachbauen lassen, demonstriert aber auch das Prüfen auf projekt- und anwendungsspezifische Regeln.

```
ex01_input.ts (2,1): A function must declare an explicit return type
ex01_input.ts (14,5): Variables with explicit any types are not allowed
ex01_input.ts (14,5): Variables must be named in camelCase. TestVaRiaBlE is invalid.
```

>> Die Ausgabe des Linters zeigt mehrere Fehler an (Abb. 5).

Listing 12: Linter-Methode, die auf explizite Rückgabetypen prüft

```
function lintNode(node: ts.Node) {
    switch (node.kind) {
      // Funktionen brauchen expliziten return-Type
      case ts.SyntaxKind.FunctionDeclaration:
        const fnStatement = node as ts.FunctionDeclaration;

        if(fnStatement.type === undefined) {
          report(fnStatement, 'A function must declare an explicit return
type');
        }
        break;

      case ts.SyntaxKind.VariableDeclaration:
        const varNode = node as ts.VariableDeclaration;
        const varType = checker.typeToString(checker.getTypeAtLocation(var-
Node.type));
        break;
    }

    ts.forEachChild(node, lintNode);
  }
```

Listing 13: Struktur des Linters nach Hinzunahme von Type-Checking

```
export function customLinter(sourceFile: ts.SourceFile, ↵
                                                  checker:ts.TypeChecker) {
    // ...
}

const program = ts.createProgram([fileName], {
  target: ts.ScriptTarget.ES5,
  module: ts.ModuleKind.CommonJS
});

customLinter(program.getSourceFile(fileName), program.getTypeChecker());
```

Jetzt informieren und bewerben unter **www.jobs.schwarz**

Deine Entwicklung durch unsere Vielfalt

Du begeisterst dich für die Entwicklung und hast Lust den digitalen Pulsschlag der Schwarz Gruppe weiter hochzutreiben. Unsere eigene Cloud, Apps und die vielfältigen Unternehmensprogramme wollen wir modern und für die Zukunft gerüstet haben. Dein Beitrag ist maßgeblich für ein positives Nutzererlebnis.

Hier kannst du dich austoben

STACKIT Cloud: Wir bauen unsere eigene Cloud (Entwicklung, Infrastruktur Cloud-Services) und du entwickelst und stellst die entsprechenden Tools für unsere Services bereit (z. B. Bosh, Kubernetes, Built Pipelines, Ansible Scripte etc.).

Java/Go Development: Du baust zukunftssichere „Betriebssystem-Services", welche unsere internen Kunden auf unserer internen Cloud STACKIT sowie VMware oder den Public Clouds unterstützen. Zusammen mit den Architekten löst du technische Fragestellungen, z. B. durch Produktevaluationen oder der Durchführung von Proof of Concepts.

Web Frontend Development: In internationalen Teams entwickelst du Web-Anwendungen für die interne Digitalisierung, z. B. Lidl Webshop und andere interne Webseiten, interne Apps (Web-Anwendung für interne Prozesse), umfangreiche ERP-System-Projekte.

C# .NET Development: Du gestaltest die (Neu-)Entwicklung unseres Warenwirtschaftssystems in Richtung einer Cloud-Native Applikationsarchitektur von der ersten Idee bis hin zum internationalen Roll Out. Du implementierst Softwarelösungen mit C# und JS-Frameworks (On Premises und Cloud-Umgebungen).

Das sind wir

- Du arbeitest mit deinen Kollegen an richtungsweisenden Projekten, welche unser gesamtes Unternehmen in die digitale Zukunft bringen
- Obwohl wir ca. 5.000 IT-Begeisterte sind, arbeitest du in schlanken und agilen Teams mit flachen Hierarchien (und nein, das behaupten wir nicht einfach, sondern leben es tatsächlich)
- Um am Geist der Zeit zu bleiben, programmierst du mit „state of the art"-Technologien
- Wir werden immer internationaler, daher kann es gut sein, dass du auf immer mehr englischsprachige Kollegen triffst

Das bieten wir

- Mobiles Arbeiten
- Weiterbildung und Karrieremöglichkeiten
- Gesundheit und Sport
- Mobilität: Jobticket, E-Bikes, Mitfahr-App
- Work-Life-Balance / Angebote zur Kinderbetreuung
- Mitarbeiterrabatte / Corporate Benefits

Listing 14: Weitere Linter-Methode zum Überprüfen auf Namenskonvention und Variablentyp

```
<function lintNode(node: ts.Node) {
    switch (node.kind) {
      // ...
      case ts.SyntaxKind.VariableDeclaration:
        const varStatement = node as ts.VariableDeclaration;

        // Variablentyp darf - wenn er explizit ist - nicht any sein
        const varType = checker.typeToString(checker.getTypeAtLocation(varStatement.type));
        if(varStatement.type !== undefined && varType === 'any') {
          report(varStatement, 'Variables with explicit any types are not allowed')
        }

        // Variablennamen müssen im camelCase formatiert sein
        const variableName = checker.getSymbolAtLocation(varStatement.name).getName();
        if(/^[a-z]+((\d)|([A-Z0-9][a-z0-9]+))*([A-Z])?$/.test(variableName) === false) {
          report(varStatement, `Variables must be named in camelCase. ${variableName} is invalid.`)
        }

        break;
    }

    ts.forEachChild(node, lintNode);
}
```

Um die Datei mit der TypeScript-Compiler-API verarbeiten zu können, muss sie zuerst zu einem `ts.SourceFile`-Objekt werden. Dazu dient die `createSourceFile`-Methode des TypeScript-Parsers, die die Rohdatei tokenisiert. Dadurch lässt sich – wie in Listing 11 – ein Linter implementieren.

Die rekursive Implementierung ist nötig, da TypeScript-Programme beliebig tief verschachtelt sein können. In der gezeigten `lintNode`-Methode lässt sich die eigene Logik implementieren. Ausgehend von der zuvor gezeigten Eingabedatei lassen sich die Rückgabetypen von Methoden einfach überprüfen: Die Deklaration von Funktionen ist für den TypeScript-Compiler lediglich eine Unterform der `SignatureDeclarationBase` und geht mit einem Typ einher. Dieser wird bei Funktionsdeklarationen als deren Rückgabetyp interpretiert. Ist er `undefined` (also nicht explizit angegeben), so kommt es wie in Listing 12 zur Warnung, dass das nicht gestattet ist.

Beim Ausführen des selbst gebauten Linters gibt dieser eine Fehlermeldung aus. Die Methode `test` kritisiert er als Regelverstoß, ignoriert allerdings `test2` (siehe Abbildung 4). Das `SyntaxKind`-Enum von TypeScript hilft dabei, die verschiedenen Sprachkomponenten zu unterscheiden und die eigenen Regeln umzusetzen. Die bisher gezeigte Implementierung nutzt ausschließlich Methoden des TypeScript-Parsers. Für komplexere Regeln, die unter Umständen auch Typinformationen benötigen, gilt es, den eingelesenen Quellcode als TypeScript-Programm zu erstellen. Dabei durchläuft der TypeScript-Compiler automatisch alle Kompilierschritte bis auf die Ausgabe, sodass er über die Typen der Variablen und Elemente verfügen kann. Die `createProgram`-Methode akzeptiert alle Compilerkonfigurationen. Für das Codebeispiel in Listing 13 genügen allerdings Sprachlevel und JavaScript-Modultyp.

Listing 15: Beispiel für ein einfaches TypeScript-Enum

```
enum ObjectType {
    User,
    File,
    Folder
}
```

Listing 16: Ergänzung des Linters, um die Nutzung von Enums zu gewährleisten

```
function lintNode(node: ts.Node) {
    switch (node.kind) {
      // ...
      case ts.SyntaxKind.VariableDeclaration:
        const varStatement = node as ts.VariableDeclaration;

        // ...

        // Enums müssen genutzt werden für Type-Variablen
        if(/type/i.test(variableName) && varType.toLocaleLowerCase() ===
 'string') {
          report(varStatement, `Use Enums to represent types, not strings`);
        }

        break;
    }

    ts.forEachChild(node, lintNode);
}
```

Mithilfe des Type-Checkers lassen sich die Typinformationen von Variablendeklarationen genauer untersuchen. An dieser Stelle hätte das Programm zwar auch die durch den Parser eingelesenen Token buchstäblich auf die Zeichenkette „any“ untersuchen können, allerdings bietet der Type-Checker weitere nützliche Funktionen: So liest er den Variablennamen sicher aus, damit der Linter ihn auf seine Namenskonvention überprüfen kann. Das geschieht mit einem regulären Ausdruck in reinem JavaScript. Der verwendete reguläre Ausdruck ist eine Vereinfachung und nicht dafür geeignet, jede erdenkliche Zeichenkette in CamelCase zu erkennen (Listing 14).

Die Ausgabe des Linters bemängelt die in der Eingabe gemachten Fehler zuverlässig (siehe Abbildung 5).

Die dritte Regel ist eher stilistischer als technischer Natur: Variablen, die „Type“ im Namen tragen, dürfen keine Strings sein, sondern nur Enums. Sie eignen sich besonders gut für das Unterscheiden verschiedener Objekttypen im Code (Listing 15).

Das Umsetzen dieser Regeln in den bestehenden Linter erfordert das Prüfen von Variablen auf den Bestandteil „Type“ in ihrem Namen sowie auf ihren Typ. Der Variablentyp wird in Kleinbuchstaben (Lower Case) umgewandelt, da sowohl `string` als auch `String` Typen in TypeScript sind. Nach dem Implementieren dieser Regel sind in der finalen Ausgabe des Linters alle Fehler zu sehen, die in der oben gezeigten Eingabedatei gemacht wurden (siehe Abbildung 6 und Listing 16).

```
ex01_input.ts (2,1): A function must declare an explicit return type
ex01_input.ts (14,5): Variables with explicit any types are not allowed
ex01_input.ts (14,5): Variables must be named in camelCase. TestVaRiaBlE is invalid.
ex01_input.ts (19,7): Use Enums to represent types, not strings
```

>> Finale Ausgabe des Linters mit allen definierten Regeln (Abb. 6)

Compilerbau ist kein Selbstzweck

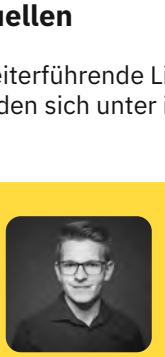
Compiler sind eine höchst spannende Angelegenheit. Dennoch haben die meisten Entwickler nicht mehr Kontakt mit ihnen als über die reine Nutzung. Ein Blick in das Innere ihrer Maschinerie – vom Einlesen der Datei über die lexikalische Analyse bis hin zur finalen Codegenerierung – kann den Spaß an diesem sonst eher theoretischen Thema wiederbeleben. Der TypeScript-Compiler ist besonders anschaulich, um Farbe in die graue Theorie zu bringen: Schließlich ist das Kompilat menschenlesbar und kein Maschinencode. Zudem ist TypeScript soweit in aller Munde, dass nahezu jeder schon einmal mit dem JavaScript-Superset in Berührung gekommen ist. Auch kann die Sprache als Open-Source-Projekt nur profitieren, wenn Interessierte die Einstiegshürde bewältigen und aktive Beitragende werden – auch, um die Abhängigkeit von und die Dominanz durch Microsoft zu vermindern.

Compiler-API senkt Einstiegshürde

Während die Idee, JavaScript typsicher zu machen, zunächst einfach klingt, ist die praktische Umsetzung komplex. Das spiegelt sich auch im Code des Compilers wider. Dank der Compiler-API wird die Einstiegshürde, sich mit ihr zu beschäftigen, aber deutlich gesenkt. Ein praktisches Projekt wie ein eigener Linter wird so mithilfe der Schnittstelle schnell zur Realität und hilft dabei, die internen Schritte des Compilers nachzuvollziehen. Denn wer beim Debuggen einmal tiefer in den Compiler-Quellcode absteigen musste, lernt mehr als in jedem Textbuch. Das gezeigte Beispiel ist dabei nur eine von unzähligen Möglichkeiten, die eigene Kreativität in Code umzusetzen. (sih)

Quellen

Weiterführende Links zu TypeScripts Compiler-API finden sich unter ix.de/zr61.



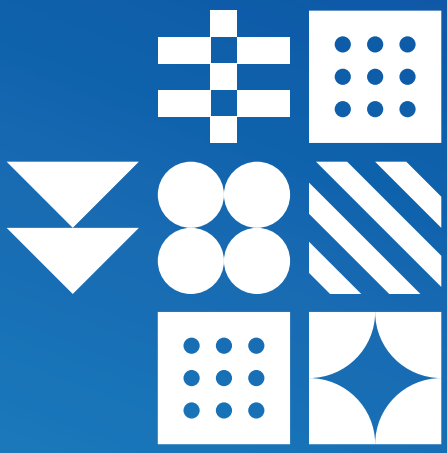
Timo Zander

hat Angewandte Mathematik und Informatik studiert. Er interessiert sich für Open Source, das JavaScript-Universum und aufstrebende Technologien.

> Multiplattformprojekte umsetzen mit KMM

Nils Kasseckert

Mit Kotlin Multiplatform Mobile lassen sich native Anwendungen für verschiedene Plattformen entwickeln, wobei die Businesslogik für die Cross-Plattform-Apps stets erhalten bleibt.



Im Alltag stellt sich oft die Aufgabe, eine neue Anwendung zu entwickeln, die auf verschiedenen Plattformen zur Verfügung stehen muss. Dabei besteht die Wahl zwischen einem Cross-Plattform-Framework und der nativen Entwicklung auf allen betreffenden Plattformen. Gegen die plattformübergreifende Entwicklung sprechen Gründe wie eine nicht optimale User Experience (UX) und eine schlechtere Akkulaufzeit. Die native Entwicklung des Produkts auf jeder einzelnen Zielplattform verursacht allerdings hohe Kosten und ist hinsichtlich der späteren Wartbarkeit nicht praktikabel. Eine Lösung für dieses Dilemma bietet das Software Development Kit (SDK) Kotlin Multiplatform Mobile (KMM). Es ermöglicht das native Erstellen von Apps unter der Nutzung eines gemeinsamen Businesscodes, der in Kotlin geschrieben ist. Die Businesslogik lässt sich anschließend als Bibliothek in Apps, ins Web und auf dem PC einbinden.

Hinter KMM steht – wie auch hinter der Programmiersprache Kotlin selbst – das tschechische Softwareunternehmen JetBrains. Die Alpha-Version erschien im August 2020 und lässt sich als Android-Studio-Plug-in installieren. Im Unterschied zu anderen Cross-Plattform-Frameworks schreibt KMM nur die Businesslogik als geteilten Code in Kotlin und wandelt ihn anschließend nativ in die Programmiersprachen für die jeweilige Plattform um. Auch findet anders als bei anderen Cross-Plattform-Frameworks, bei denen das User Interface (UI) meist nur als Webansicht eingebunden ist, das Erstellen des UI nativ statt, was eine bessere Nutzererfahrung bedingt. Der Businesscode lässt sich als Abhängigkeit in das jeweilige plattformspezifische Projekt integrieren, es ist jedoch auch ohne diese Abhängigkeit funktionsfähig.

KMM soll dabei helfen, den Aufwand für das Erstellen der Businesslogik zu minimieren und dennoch eine native Erfahrung zu schaffen. Mit Kotlin Multiplatform Mobile lässt sich nativer Code für Android, iOS, watchOS, macOS, Windows sowie Linux erzeugen. In der App-Entwicklung ist dieses Vorgehen besonders attraktiv, denn Teams programmieren Android-Apps seit geraumer Zeit in Kotlin. Eine Umgewöhnung auf eine neue Sprache ist somit nicht nötig. Die starke Ähnlichkeit zwischen der von Apple entwickelten Programmiersprache Swift und Kotlin vereinfacht iOS-Entwicklern und -Entwicklerinnen die Umstellung.

Wie in der klassischen Android-Entwicklung ist für das Umsetzen eines Multiplattform-Projekts Android Studio notwendig. Zusätzlich ist das KMM-Plug-in in Android Studio zu installieren. Für einen praxisnahen Einstieg in die Multiplattform-Entwicklung gilt es im Folgenden eine kleine Beispiel-App zu entwickeln, die den Gerätenamen ausgibt. Technisch gesehen geschieht das mittels plattformspezifischer APIs, während die Businesslogik auf beiden Plattformen identisch ist. Um die iOS-App umzusetzen und zu kompilieren, ist ein Mac-Computer nötig – für die Android-App besteht diese Beschränkung nicht. Grundkenntnisse in der iOS- und Android-App-Entwicklung seien im Folgenden vorausgesetzt.

In a Nutshell

- Anwendungen nativ für jedes Produkt separat zu entwickeln, verursacht hohe Kosten und Probleme bei der Wartung.
- Das Software Development Kit Kotlin Multiplatform Mobile (KMM) bietet eine Lösung für das Dilemma: Entwicklerinnen und Entwickler können damit native Apps mit geteiltem Businesscode erstellen.
- KMM ermöglicht das Wiederverwenden der in Kotlin geschriebenen Businesslogik, die sich unter anderem in Apps, im Web und auf dem PC einsetzen lässt.

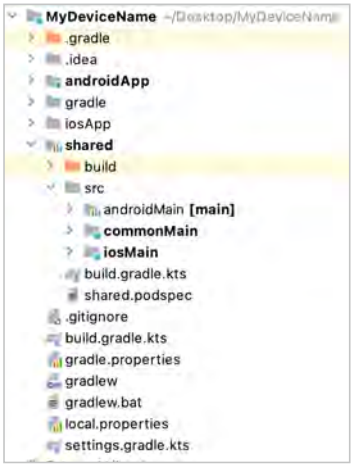
Erstellen eines neuen Multiplattform-Projekts

Zum Erstellen eines neuen Multiplattform-Projekts führt der Weg über den Button New Project in Android Studio. Dort ist als Template zunächst Kotlin Multiplatform App auszuwählen. Im Anschluss besteht die Möglichkeit, verschiedene Parameter wie die Android-Version oder den Projektnamen zu konfigurieren. Für einen leichteren Einstieg ist im GitHub-Repository von AppSupporter (weiterführende Links hierzu und weitere Ressourcen finden sich unter ix.de/zfym) ein bereits fertig konfiguriertes Projekt zu finden. Auf dem Branch solution steht die fertige App zur Ansicht bereit.

Nach erfolgreichem Öffnen des Projekts in Android Studio mit anschließender Gradle-Synchronisierung ist die Projektansicht von Android auf Project umzustellen, sonst wären einige Teile des Projekts nur schwer oder gar nicht auffindbar. Abbildung 1 stellt den initialen Projektaufbau dar.

Multiplattform-Bibliotheken: Für Netzwerkanfragen punktet Ktor

Das Projekt besteht aus den Ordnern androidApp, iosApp und shared. Zusätzlich sind in der Basisversion einige Gradle-Dateien zu finden. Im androidApp-Ordner ist die gesamte Android-App enthalten. Die Programmierung findet in der Sprache Kotlin statt. Für den Ordner iosApp gilt das allerdings nicht: Er enthält die gesamte in Swift verfasste iOS-App. Die geteilte Businesslogik befindet sich im shared-Ordner, der sich in die Unterordner androidMain, commonMain sowie iosMain aufteilt. Die Programmiersprache für diesen Ordner ist ebenfalls Kotlin. Im Unterordner commonMain sind alle Klassen enthalten, die als plattformunabhängig gelten. Typische Anwendungsfälle sind Algorithmen, Netzwerkanfragen oder zum großen Teil auch die Datenbanklogik.



>> Initialer Projektaufbau einer Kotlin Multiplatform Mobile App (Abb. 1).

Da für Netzwerkanfragen plattformspezifische APIs notwendig sind, bedarf es hierfür spezieller Multiplattform-Bibliotheken. Durchgesetzt haben sich dabei die quelloffene Kotlin-Bibliothek Ktor (für die Netzwerkanfragen) sowie SQLDelight für die plattformunabhängige Speicherung von Daten in einer Datenbank. Auf GitHub gibt es eine Liste be-

Listing 1: shared-Abhängigkeit in der Android-Build-Gradle-Datei

```
implementation(project(":shared"))
```

reits verfügbarer Multiplattform-Bibliotheken für weitere Anwendungsfälle in Kotlin (die GitHub-Links sowie weiterführende Ressourcen finden sich unter ix.de/zfym). Zudem ist es möglich, eigene Libraries zu erstellen.

Die Gradle-Dateien im Projekt sinnvoll einsetzen

Die im Projekt vorhandenen Gradle-Dateien sind elementar. Daher muss man zunächst auch sie verstehen. Sowohl auf Root-Ebene als auch im androidApp- und shared-Ordner befindet sich eine Gradle-Datei namens build.gradle.kts. Auf Root-Ebene werden allgemeine, projektweit geltende Einstellungen wie die URLs der Repositorys oder die Gradle-Version festgelegt. Die Datei Build-Gradle im Android-Ordner enthält den von Android gewohnten Aufbau. Im Unterschied zu Android-Apps, die das KMM-SDK nicht nutzen, ist eine weitere Abhängigkeit enthalten (Listing 1). Dadurch lässt sich jeglicher im shared-Ordner erstellte Code in das Android-Projekt einbinden und steht somit auch dort zur Verfügung.

Wichtiger ist die im shared-Ordner enthaltene Gradle-Datei. Listing 2 zeigt einen Ausschnitt daraus, die komplette Datei ist im Projekt unter dem Pfad shared/build.gradle.kts zu finden. Der Plug-in-Block bindet zuerst das Multiplattform- und das cocoapods-Plug-in ein. CocoaPods ist der standardmäßig verwendete Abhängigkeitsmanager unter iOS, da Gradle dort nicht zur Verfügung steht. Fast jedes Open-Source-Projekt in der iOS-Welt steht als CocoaPod-Abhängigkeit zur Verfügung. Dessen Schöpfer Eloy Durán hatte CocoaPods 2011 erstmals veröffentlicht. Inzwischen entwickelt es ein großes Entwicklerteam weiter. JetBrains nutzt in seinen SDKs ebenfalls CocoaPods. Auch in der Standardkonfiguration eines KMM-Projekts ist der shared-Code nach der Kompilierung als CocoaPods-Abhängigkeit in das iOS-Projekt eingebunden. Es besteht zwar die Möglichkeit, den shared-Code direkt als Framework einzubinden, was wegen einiger gravierender Nachteile wie dem Fehlen der Möglichkeit, andere Abhängigkeiten einzubinden, hier jedoch keine weitere Rolle spielt.

Nach dem einleitenden Plug-in-Block folgt in Listing 2 der Kotlin-Abschnitt. Darin findet die eigentliche Konfiguration des Multiplattform-Projekts statt. Neben allgemeinen Einstellungen zu CocoaPods lassen sich hier die zur Verfügung stehenden Plattformen (im Beispiel sind es Android und iOS) konfigurieren und unter `sourceSets` die Abhängigkeiten definieren. Soll eine Abhängigkeit für den gesamten shared-Code gelten, ist sie unter `commonMain` in den `dependencies`-Block zu schreiben. Abhängigkeiten, die nur im shared-Code auf iOS oder Android benötigt werden, sind im entsprechenden `androidMain`- beziehungsweise im `iosMain`-Dependency-Block zu definieren. Gleiches gilt für Testabhängigkeiten. Derzeit

Listing 2: Ausschnitt aus der Shared-Build-Gradle-Datei

```
plugins {
    kotlin("multiplatform")
    kotlin("native.cocoapods")
    id("com.android.library")
}

version = "1.0"

kotlin {
    android()
    iosX64()
    ...

    cocoapods {
            ...
        ios.deploymentTarget = "14.1"
        podfile = project.file("../iosApp/Podfile")
        framework {
            baseName = "shared"
        }
    }

    sourceSets {
        val commonMain by getting
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val androidMain by getting
        ...
        val iosSimulatorArm64Main by getting
        val iosMain by creating {
            dependsOn(commonMain)
            iosX64Main.dependsOn(this)
            ...
        }
}
...
```

Listing 3: Inhalt der Expect-Klasse Plattform

```
expect class Platform() {
    val platform: String
}
```

Listing 4: iOS-Implementierung der getDeviceName-Funktion

```
actual fun getDeviceName(): String {
    return UIDevice.currentDevice.name
}
```

Listing 5: Android-Implementierung der getDeviceName-Funktion

```
actual fun getDeviceName(): String {
    return android.os.Build.MODEL
}
```

Listing 6: Implementierung der geteilten Businesslogik

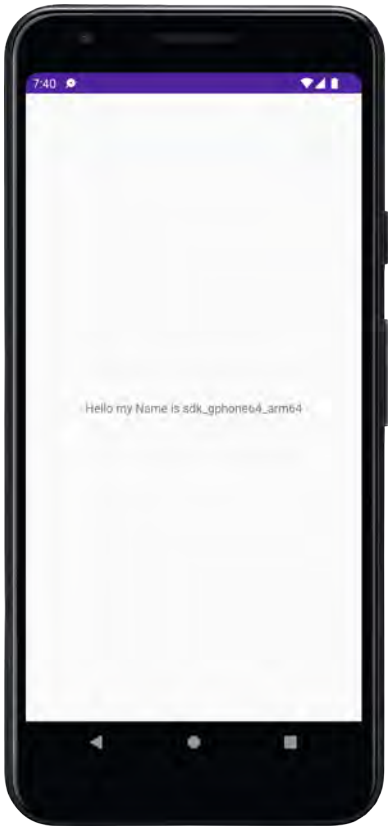
```
fun helloDevice(): String {
    val platform = Platform()

    return "Hello my Name is ${platform.getDeviceName()}"
}
```

Listing 7: Android-Aufruf der geteilten Businesslogik

```
val greeting = Greeting()
tv.text = greeting.helloDevice()
```

ist es nicht möglich, die dort definierten Abhängigkeiten auch im nativen Code zu verwenden. Um sie dennoch nutzen zu können, sind sie auf der jeweiligen Plattform erneut zu definieren. In der shared-Gradle-Datei definierte CocoaPods haben hingegen sowohl im geteilten als auch im nativen iOS-Teil Gültigkeit. Derzeit sind ausschließlich Objective-C-Abhängigkeiten im Kotlin-Code für iOS verwendbar. Eine Lösung für diese Beschränkung ist laut Roadmap und dazugehörigen YouTrack-Issues bereits in Arbeit (Links siehe ix.de/zfym). Auffällig ist die umständliche Definition der iOS-App: Während für Android nur eine Zeile notwendig ist, benötigt iOS drei. Das ist der Architekturvielfalt (ARM, x64, Simulator) der Apple-Plattform geschuldet.



>> Die fertige Android-App startet (Abb. 2).

>> So sieht die fertige iOS-App aus (Abb. 3).

Mit dem Programmieren loslegen

Zum Start der eigentlichen Programmierung ist der commonMain-Ordner zu öffnen und man muss zur Datei Plattform.kt navigieren, die sich im Unterordner kotlin/de.heise/ befindet. Die IDE generiert während der Projekterstellung standardmäßig die Kotlin-Dateien Plattform.kt und Greeting.kt. Sie dienen für dieses Beispiel als Basis. Listing 3 zeigt den Inhalt der Datei Plattform.kt. Darin ist eine Expect-Klasse definiert, die eine Variable des Typs String enthält. Expect-Klassen sind ein Kernkonzept der Multiplattform-Entwicklung. Sie ermöglichen den Zugriff auf native, plattformspezifische APIs. Grundsätzlich ähneln Expect-Klassen Interfaces. Sie kommen ausschließlich im `commonMain`-Code vor und enthalten nur Methodenrümpfe oder Variablen-Deklarationen. Im Gegensatz zu einem Interface lassen sich Expect-Klassen initialisieren und wie normale Kotlin-Klassen verwenden. Die eigentliche Implementierung der Klasse findet plattformspezifisch statt. So existiert sowohl in androidMain als auch in iosMain eine entsprechende Implementierung der Plattform-Expect-Klasse. Die konkrete Implementierung wird mit `actual` anstelle von `expect` definiert.

Die Auswahl der korrekten Klasse übernimmt der Compiler zur Build-Zeit in Abhängigkeit zur gerade verwendeten Plattform. Beim Kompilieren der iOS-Version verwendet iOS automatisch die zu iOS passende Actual-Klasse. Das gleiche Prinzip gilt für alle anderen Plattformen.

Zuerst wird die Platform-Expect-Klasse Platform.kt in `commonMain` um eine neue Methode erweitert. Sie soll den Gerätenamen als String ausgeben. Hierfür ist nach Zeile vier folgender Code einzufügen: `fun getDeviceName(): String`. Eine Methode darf auch Parameter enthalten. Für das einfache Beispiel ist das allerdings entbehrlich. Im Anschluss daran ist die Methode jedes Mal in die jeweilige Plattform zu implementieren.

In Kotlin programmiert, aber Zugriff auf native APIs

Die Dateien müssen immer unter demselben Pfad wie `commonMain` gespeichert sein, sonst findet der Compiler die jeweilige Implementierung nicht. Da die Kotlin-Klasse Plattform.kt in `commonMain` unter kotlin/de.heise/ definiert ist, muss sie es auch unter androidMain und iosMain sein. In der Plattform.kt unter iosMain ist im nächsten Schritt der in Listing 4 aufgezeigte Code unterhalb der Zeile sechs zu schreiben.

Obwohl die Programmierung in Kotlin stattfindet, ist der Zugriff auf native APIs möglich. Listing 4 greift insbesondere auf die plattformspezifische `UIDevice`-Klasse zu. Zu Beginn dürfte diese Vorgehensweise gewöhnungsbedürftig sein, zudem fehlt eine Kotlin-spezifische Dokumentation für Spezialfälle. Das soll sich jedoch laut YouTrack-Issues auf Dauer ändern. Dasselbe Vorgehen ist nun für die Plattform.kt unter androidMain anzuwenden. Hier muss man den unter Zeile vier in Listing 5 dargestellten Code einfügen. Die plattformspezifische Implementierung ist damit abgeschlossen.

Abschließend ist für den shared-Code die Businesslogik anzupassen. Als Erstes gilt es hierfür die automatisch generierte Greeting.kt im `commonMain`-Teil zu öffnen. Unter der bereits vorhandenen Funktion `greeting()` lässt sich die in Listing 6 dargestellte Funktion `helloDevice()` einfügen. Die Funktion erzeugt zuerst eine neue Instanz der Plattform.kt und ruft anschließend auf dieser Instanz die `getDeviceName()`-Methode auf. Der Rückgabewert der `getDeviceName`-Funktion ist ein String, der eine Begrüßung sowie das Ergebnis der `getDeviceName()`-Methode enthält.

Businesslogik einbinden und die fertige Android-App in Betrieb nehmen

Der nächste Schritt bindet die erstellte und geteilte Businesslogik in die jeweiligen nativen Projekte ein. Hierfür ist zuerst der Ordner androidApp zu öffnen. Die Android-App hat einen einfachen Aufbau. Der Unterordner src/main/java/de.heise.android/ enthält die Datei MainActivity.kt. Sie definiert die Hauptklasse, die die App beim Starten aufruft. Das zugehörige Layout der MainActivity.kt befindet sich in src/res/activity_main.xml. Der Aufbau des Layouts ist leicht verständlich. Die XML-Datei besteht aus einem Textfeld, das später den Gerätenamen anzeigen soll. Das Projekt generiert sowohl die `MainActivity`-Klasse als auch das Layout beim Neuerstellen standardmäßig. Um die zuvor im shared-Code erstellte Logik einzubinden, muss man nun die Kotlin-Klasse MainActivity.kt öffnen und Zeile 18 durch den in Listing 7 dargestellten Code ersetzen. Die Implementierung der Android-App ist hiermit abgeschlossen. Über die in Android Studio zur Verfügung stehenden Möglichkeiten lässt sich nun die App im Simulator oder auf dem Gerät starten. Abbildung 2 stellt das Endergebnis auf der Android-Plattform dar.

iOS-Applikationen erstellen

Für die iOS-Variante ist das Vorgehen ganz ähnlich. Für die folgenden Schritte ist ein Mac notwendig. Das ist nicht KMM, sondern der restriktiven Firmenpolitik von Apple geschuldet, die das Erstellen von iOS-Apps nur auf Apple-Geräten erlaubt. Das iOS-Projekt lässt sich nicht über Android Studio, sondern nur mittels Xcode öffnen. Xcode ist eine von Apple kostenlos zur Verfügung gestellte Entwicklungsumgebung zum Erstellen von Applikationen für Apple-Endgeräte. Es ist aus dem App Store installierbar. Nach dem erfolgreichen Einrichten von Xcode ist das Projekt im Ordner iosApp auffindbar. Dort ist die Datei iosApp.xcworkspace zu öffnen. Würde man statt der Xcworkspace- die Xcodeproj-Datei öffnen, so ließen sich die CocoaPods nicht korrekt laden.

Um Funktionen wie die Autovervollständigung zu nutzen, empfiehlt es sich eingangs das Projekt zu kompilieren. Der Trigger zum Kompilieren befindet sich in Xcode in der Menüleiste im Menüpunkt Product unter der Aktion Build. Das ist auch notwendig, wenn sich der shared-Code geändert hat und in der Architektur von Kotlin Multiplatform Mobile begründet. Der in Kotlin erzeugte Code wird nativ für die jeweiligen Plattformen zur Verfügung gestellt. Das geschieht jedoch nicht „on the fly“, sondern durch einen aktiven Trigger – in diesem Falle das Auslösen eines Builds. Es ist auch möglich, das Generieren direkt über Gradle anzustoßen. Aus Gründen der Komplexität spielt das an dieser Stelle allerdings keine Rolle.

Haben Entwickler das Erzeugen des shared-Codes vergessen, äußert sich das in zahlreichen Syntaxfehlern. Ein erneutes Erstellen des Projekts behebt die Probleme. Das iOS-Projekt wird standardmäßig in SwiftUI geschrieben und die Haupt-View lässt sich unter iOSApp/ContentView.swift finden. Die Anpassungen in der iOS-Version sind marginal: Die vorhandene Zeile `let greet = Greeting().greeting()` ist durch `let greet = Greeting().helloDevice()` zu ersetzen. Abschließend lässt sich die iOS-App entweder auf dem Gerät oder in dem in Xcode zur Verfügung stehenden Simulator starten (siehe Abbildung 3).

Ist die Entwicklung der iOS- und Android-App in verschiedene Personengruppen getrennt und es gilt nur eine Kleinigkeit wie die Backend-URL zu ändern, so benötigen Android-Entwickler keine Kenntnisse über den iOS-Code. Die Änderung des Kotlin-Codes genügt. Im Anschluss muss man die App nur noch kompilieren. Ein weiterer elementarer Vorteil ist die einfache Wiederverwendbarkeit des Businesscodes. Soll zu einem späteren Zeitpunkt neben einer iOS- oder Android-App auch noch eine Web-App entstehen, lässt sich aus dem Kotlin-Code nativer JavaScript-Code erzeugen. In solch ei-

nem Fall ist die build.gradle-Datei nur um die Konfiguration des KotlinJS-Blocks zu erweitern. Mithilfe von KotlinJS lässt sich neben JavaScript- auch TypeScript-Code erzeugen. Letzteres befindet sich jedoch noch in der Beta-Phase. Den erzeugten Code bindet man über ein lokales npm-Paket ein. Das hat den Vorteil, dass sich die Web-App mit jeglichem Webframework entwickeln lässt.

Ein neuer Weg für das Erstellen von Web-Apps ist der Einsatz von Jetpack Compose im Browser. Hierdurch ergibt sich eine noch größere Wiederverwendbarkeit, da Jetpack Compose auch bei der Android-Entwicklung zum Einsatz kommen kann. Die Umwandlung von Kotlin- in nativen JavaScript-Code läuft jedoch nicht immer problemlos ab. Besonders die Umwandlung von Kotlin-Coroutines in das JavaScript-Äquivalent bedarf einiger Tricks im nativen Teil des shared-Codes. Auch ist die erzeugte JavaScript-Bundle-Größe noch nicht optimal. JetBrains arbeitet derzeit aktiv an all diesen Punkten. Interessierte können sich auf der öffentlich verfügbaren Roadmap über den Entwicklungsstand informieren. Durch den Open-Source-Ansatz von Kotlin besteht zudem die Möglichkeit, direkt über das JetBrains-Projektmangement-Tool YouTrack Probleme zu melden und Hilfe zu erhalten.

Kotlin Multiplatform Mobile recycelt den Businesscode und spart Zeit

Das Ziel dieses Beitrags besteht darin, die Funktionsweise von Kotlin Multiplatform Mobile (KMM) zu erläutern. Dafür können Entwicklerinnen und Entwickler in mehreren Schritten der Anleitung zum Bau eines Multiplattformprojekts folgen, je nach technischen Voraussetzungen und Vorliebe entweder für iOS oder für Android. Für beide Zielplattformen hat der Artikel ausführlich den Weg zur ersten Multiplattform-App erläutert. Die Businesslogik bleibt dabei stets die gleiche, während die Oberfläche nativ erstellt wird. Teams können mit KMM den Businesscode recyceln und dennoch eine native Nutzererfahrung schaffen. Nicht umsonst spricht man von Multiplattform- anstelle von Cross-Plattform-Entwicklung. KMM lässt sich als Cross-Plattform-Entwicklung ohne Cross-Plattform zusammenfassen. Der Produktivitäts- und Zeitgewinn der hier erstellten App gegenüber einer nativen App ist marginal, in größeren Anwendungen ist durch den Einsatz der Technik jedoch eine große Zeitersparnis umsetzbar. In eigenen Projekten konnte der Autor eine Zeitersparnis von bis zu 60 Prozent gegenüber der nativen Entwicklung erreichen. Zudem erleichtert der Weg über die Multiplattform-Programmierung die spätere Wartbarkeit, denn alle Änderungen sind nur einmal notwendig und direkt auf allen verwendeten Plattformen verfügbar. (sih)

Quellen

Die Links zu Kotlin Multiplatform Mobile finden sich unter ix.de/zfym.

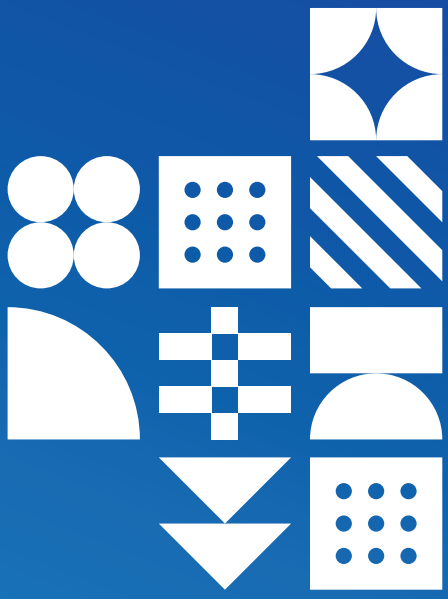
Nils Kasseckert

arbeitet als Senior App Entwickler bei einem führenden Hersteller mechatronischer, mechanischer, und elektronischer Schließsysteme. Seit 2013 betreibt er zusätzlich sein Nebengewerbe „AppSupporter“ sowie seine Firma codecreators. Er interessiert sich für Softwarearchitektur, Embedded Systems sowie alle neuen Technologien.

> Makros in Rust: eine Einführung

Alvin Ramskogler und Rainer Stropek

In Rust sind Makros ein mächtiges Werkzeug, das nicht mit den einfachen Textersetzungen in C/C++-Makros zu vergleichen ist. Ein Plädoyer für Rust-Makros.



Wer erste Schritte in Rust macht und dabei das klassische „Hello World"-Programm generiert, sieht als erste und einzige Zeile der `main`-Funktion `println!("Hello, world!")`. Auffallend ist das Ausrufezeichen nach dem `println`-Befehl. Es kennzeichnet in Rust ein Makro, um genau zu sein: ein Function-like Declarative Macro. Es gibt noch weitere Formen von Makros in Rust. Dieser Beitrag konzentriert sich aber auf die funktionsähnlichen, deklarativen Makros, die man durch das Ausrufezeichen nach dem Makronamen erkennt.

Die Tatsache, dass das „Hello World"-Programm in Rust bereits einen Makroaufruf enthält, ist charakteristisch für Rust. Makros sind in dieser Programmiersprache allgegenwärtig. Man findet sie in der Rust-Standardbibliothek genauso wie in praktisch allen Anwendungsframeworks. Makros nehmen Entwickler und Entwicklerinnen durch das Generieren von Code eine Menge Tipparbeit ab. Gut eingesetzt verbessern sie die Entwicklungsproduktivität und führen zu leichter lesbarem und wartbarem Code.

Rust-Makros funktionieren anders

Die gute Nachricht in Rust ist, dass Makros fundamental anders funktionieren als in C. Der Rust-Compiler führt bei Makros keine einfache Textersetzung durch. Makros arbeiten in Rust auf der Ebene des Abstract Syntax Trees (AST). Klammerfehler wie in C gehören daher der Vergangenheit an. Die Funktionsweise von Rust-Makros geht auch weit über die von Makros in C hinaus: Die Anwendungsbereiche von Makros reichen in Rust von einfachen Hilfskonstrukten, die ein wenig Tipparbeit ersparen und den Code besser strukturieren, bis hin zu domänenspezifischen Sprachkonstrukten (Domain-specific Languages, kurz DSL), die sich durch Makros nahtlos in den Rust-Code einbauen lassen.

Listing 3 zeigt ein erstes, kleines Beispiel, um das Prinzip deklarativer Makros zu veranschaulichen. In der Rust-Dokumentation werden solche Makros oft als „Macros by Example" bezeichnet, da man als Makro-Entwickler oder -Entwicklerin ein Beispiel angibt, wie das Makro aufzulösen ist.

Die Makrodefinition beginnt mit `macro_rules` und der Festlegung des Makronamens say_hi_to. Der Identifier `\$name` deklariert eine Metavariable, die sich im Makro verwenden lässt. In unserem Fall wird die Metavariable `\$name` als Argument beim Aufruf von `println!` einem weiteren Makro übergeben.

Rust-Makros sind eine Transformationsregel auf Ebene des Syntaxbaums

Das Besondere an Rust-Makros ist der Fragment Specifier `expr`, der nach dem Makronamen folgt. Im Beispiel legen die Autoren fest, dass Rust als Wert für die Variable \$name eine Expression akzeptieren darf. Sie legen also den Typ des Makro-

In a Nutshell

- In Rust sind Makros weitaus weniger fehleranfällig als in C/C++, da es sich um die Transformation eines Abstract Syntax Tree handelt.
- Rust-Makros haben das Potenzial, Code zu vereinfachen und die Entwicklungsgeschwindigkeit zu steigern, da manuelles Duplizieren von Code entfällt, und Entwickler können mit Makros sogar eigene Domain-specific Languages (DSL) in den Rust-Code integrieren.
- Alles in allem sind Makros in Rust ein unverzichtbares Werkzeug und ein charakteristisches Merkmal der Sprache.

Ein Blick in die Vergangenheit – Makros in C

Entwicklerinnen und Entwickler, die Programmiererfahrung in C oder C++ haben, sind oft nicht erfreut, wenn sie hören, dass Rust intensiv auf Makros setzt. Der Grund dafür ist, dass Makros in C zwar in gewissen Fällen praktisch sein können, jedoch fehleranfällig sind. Bedenken hinsichtlich Makros sind bei Rust jedoch unbegründet. Um das zu verdeutlichen, folgt ein kurzer Blick in die Vergangenheit der Makros in C.Makros sind in C Präprozessordirektiven. Bevor der Compiler den Code übersetzt, werden Makros mit einfacher Suchen-Ersetzen-Logik expandiert.

Im Codebeispiel in Listing 1 wird als Erstes die Konstante PI als Makro definiert (gemeint ist hier die Zahl π). In der zweiten Zeile folgt ein Makro, das bereits etwas komplexer ist. Es verlangt einen Parameter, um mithilfe der zuvor definierten Konstante PI die Fläche eines Kreises zu berechnen. Die `main`-Methode enthält den Aufruf des Makros. Der C-Präprozessor wird die Makros vor dem Kompilieren durch simple Textersetzung auflösen.

Auf den ersten Blick ist kein Problem im obigen Codebeispiel ersichtlich. Was passiert aber, wenn man den Aufruf des Makros `CIRCLE_AREA` verändert auf `double area = CIRCLE_AREA(radius + 2)`? Da der C-Compiler die Makros durch Textersetzung auflöst, wird daraus `double area = PI * radius + 2 * radius + 2`. Jetzt wird klar, dass dadurch die Formel für die Berechnung der Kreisfläche nicht mehr richtig ist.

C-Entwickler und -Entwicklerinnen lösen das Problem, indem sie Klammern setzen. Im vorliegenden Beispiel ließe sich das Makro ändern zu: `#define CIRCLE_AREA(r) PI * (r) * (r)`. Die Klammern werden bei der Auflösung des Makros berücksichtigt, das Ergebnis nach dem Ausführen des Präprozessors ist `double area = PI * (radius + 2) * (radius + 2)` – und das ist richtig. Listing 2 enthält ein weiteres Beispiel zum Veranschaulichen des Problems.

Führt man das C-Programm aus Listing 2 aus, ist man möglicherweise überrascht, dass das Ergebnis 5 lautet, obwohl man 6 erwartet hätte. Das liegt erneut an den Klammern. Die Formel `SUM(1, 2) * 2` wird expandiert zu `(1) + (2) * 2`. Lösen lässt sich das Problem erneut durch das Hinzufügen von Klammern: `#define SUM(a, b) ((a) + (b))` führt zum richtigen Ergebnis.

Spätestens hier wird klar, dass sich in C-Makros subtile Fehlerquellen verstecken können. Ein Makro ist stets für einen bestimmten Zweck bestimmt und erfüllt diese Aufgabe auch. Wenn das Makro jedoch in einem anderen Kontext von jemandem verwendet wird, der sich über den genauen Aufbau des Makros keine Gedanken gemacht hat, kann es leicht zu falschen Ergebnissen führen. Das ist der Grund, warum Makros in C nicht den besten Ruf haben.

Listing 1: Erste, einfache C-Makros

```
#define PI 3.14159265358979323846
#define CIRCLE_AREA(r) PI * r * r

int main(void) {
    int radius = 5;
    int area = CIRCLE_AREA(radius);
    printf("Radius: %d\nArea: %d\n", radius, area);
    return 0;
}
```

Listing 2: C-Makro führt zu fehlerhafter Berechnung

```
#define SUM(a, b) (a) + (b)

int main(void) {
    printf("%d\n", SUM(1, 2) * 2); // Result is 5 instead of 6
    return 0;
}
```

parameters auf Basis von Syntaxelementen aus dem Rust-AST fest. Hier wird deutlich, dass Rust-Makros keine Textersetzung sind, sondern eine Transformationsregel auf AST-Ebene. Der Rust-Compiler kann durch diese semantische Information die Makros besser auflösen als es bei den Präprozessordirektiven der C-Makros möglich war. Klammerfehler wie in den zuvor dargestellten C-Beispielen gibt es bei Rust-Makros nicht. Entsprechend liefert Listing 4, das eine Rust-Übersetzung des zuvor gezeigten C-Makros SUM ist, das richtige Ergebnis, ohne dass dafür Klammern einzufügen wären.

Listing 3: Erstes, einfaches Rust-Makro

```
macro_rules! say_hi_to {
    ($name:expr) => {
        println!("Hi, {}!", $name);
    }
}

say_hi_to!("Rust");
```

Listing 4: sum-Makro in Rust

```
macro_rules! sum {
    ($a:expr, $b:expr) => {
        $a + $b
    };
}

println!("{}", sum!(1, 2) * 2); // Result is 6
println!("{}", sum!(2 * 2, 3 * 3) * 2); // Result is 26
```

Listing 5: Literale in Rust-Makros

```
macro_rules! sum {
    // +---------------+-- Note literals "rechne" and "plus"
    // |               |
    // v               v
    (rechne $a:literal plus $b:literal) => {
        $a + $b
    };
}

//                 +--------+-- Note literals here
//                 |        |
//                 v        v
println!("{}", sum!(rechne 1 plus 2) * 2); // Result is 6

let _x = 42;
//                   +-- Does not work as eval requires literal,
//                   |   not an identifier or an expression.
//                         v
// println!("{}", sum! { rechne _x plus 2 } * 2); // Result is 6
```

Beim oben dargestellten Beispiel nimmt das sum-Makro ähnlich wie eine Funktion zwei Parameter entgegen. Die Aufrufsignatur eines Rust-Makros kann jedoch auch Literale enthalten. Dadurch lassen sich domänenspezifische Sprachkonstrukte in Rust umsetzen. Listing 5 definiert eine Variante des sum-Makros mit einer ganz besonderen Syntax.

Der Rust-Lexer muss den Makroaufruf erfolgreich analysieren können, der Aufruf muss aber nicht den Regeln des Rust-Parsers entsprechen. Die Syntax ergibt sich durch die Makroregeln, und der Rust Compiler prüft sie zur Übersetzungszeit. Syntaxfehler beim Aufruf des Makros führen also nicht zu Laufzeitfehlern.

Wer neugierig geworden ist und sehen möchte, wie weit man mit Domain-specific Languages mit Rust-Makros gehen kann, kann einen Blick auf Experimente wie macro-lisp werfen, wo eine Lisp-ähnliche DSL mit Hilfe von Makros umgesetzt wird (weiterführende Ressourcen zum Thema finden sich unter ix.de/zcn2).

Listing 6: Repetitions in Rust-Makros

```
macro_rules! vector {
    ($($item:expr),*) => {{
        let mut vector = Vec::new();
        $(
            vector.push($item);
        )*
        vector
    }}
}

let my_vec = vector![];
let my_vec = vector![1];
let my_vec = vector![1, 2, 3];
```

Auf Repeat: Wiederholungen durch optionale Wiederholungsoperatoren

Makros in Rust sind nicht auf eine fixe Anzahl an Argumenten beschränkt. Ähnlich wie bei Regular Expressions ist es möglich, Wiederholungsoperatoren anzugeben. Das lässt sich anhand einer vereinfachten Version des vec!-Makros aus der Rust-Standardbibliothek schön illustrieren (Listing 6: „Repetitions in Rust-Makros“).

Dieses `vector`-Makro nimmt nicht nur eine Expression als Parameter an, sondern akzeptiert wegen des Wiederho-

>> Expandieren von Makros im Rust Playground (Abb. 1)

Listing 7: Optionale Fragmente in Rust-Makros

```
macro_rules! print_all {
    ( $( $expr:expr ),+ $(,)? ) => {
        $(
            println!("{}", $expr);
        )*
    }
}

//           +-- Empty parameter list does not
//           |   work because of `+`
//           v
// print_all!();
print_all!(1, 2, 3);
print_all!(1, 2, 3,); // Note trailing comma
```

Listing 8: Aufrufvarianten von Rust-Makros

```
macro_rules! print_named {
    (a: $x:expr) => {
        println!("a = {}", $x)
    };
    (b: $x:expr, c: $y:expr) => {
        println!("b = {}, c = {}", $x, $y)
    };
}

print_named!(a: 123);
print_named!(b: 456, c: 789);
```

Listing 9: Variable Parameteranzahl in Rust-Makros

```
macro_rules! print_named {
    ($($name:ident: $val:expr),* $(,)?) => {
        $(
            println!("{} = {}", stringify!($name), $val);
        )*
    };
}

print_named!(
    foo: 123,
    bar: 456,
);
```

lungsoperators * beliebig viele (oder auch keine) Parameter. Im Body des Makros kommt ebenfalls der *-Operator zum Einsatz, um die Zeile `vector.push($item);` für jeden übergebenen Parameter zu wiederholen. Würde man im Body den * weglassen, käme es zu einem entsprechenden Compilerfehler.

Wer neugierig ist, wie der Code nach dem Expandieren der Rust-Makros aussieht, kann die Auflösung der Makros mit dem Befehl `cargo expand` durchführen. Der online verfügbare Rust Playground bietet diese Funktion ebenfalls (siehe Abbildung 1). Neben dem Wiederholungsoperator *, der 0..n Elemente

Listing 10: Einschränkung auf einen Identifier als Makro-Parameter

```
macro_rules! variable_scoping {
    ($number:ident) => {
        println!("{}", $number);
    };
}

let number = 1;
variable_scoping!(number); // Prints 1 on the screen

//                  +-- Does not work because macro
//                  |   expects identifier
//                  v
// variable_scoping!(1 + number);
```

Listing 11: Keine Forward Declaration bei Rust-Makros

```
fn main() {
    println!("{}", sum(1, 2)); // Note: Forward declaration works

    //   +-- Does not work because macro must be defined
    //   |   before it can be used.
    //               v
    // println!("{}", _sum!(1, 2));
}

macro_rules! _sum {
    ($a:literal, $b:literal) => {
        $a + $b
    };
}

fn sum(a: i32, b: i32) -> i32 {
    a + b
}
```

erlaubt, gibt es auch den Operator `+`, der mindestens ein Element verlangt. Mit dem `?`-Operator lassen sich optionale Parameter implementieren. Listing 7 zeigt die Anwendung von `+` und `?`. Es implementiert ein Makro, das die Ergebnisse aller übergebenen Expressions am Bildschirm ausgibt. Das Makro lässt sich mit oder ohne Trailing Comma aufrufen.

Aufmerksame Leserinnen und Leser werden bei den Rust-Beispielen bemerkt haben, dass der Makroaufruf mit unterschiedlichen Klammern (`[]`, `()`, `{}`) erfolgt. Das ist kein Tippfehler – Rust erlaubt alle diese Aufrufvarianten.

Aufrufvarianten

Wiederholungsoperatoren für Makro-Parameter sind nützlich, reichen aber oft nicht aus. In vielen Fällen müssen sich Makros fundamental anders verhalten, wenn eine unterschiedliche Anzahl an Parametern übergeben wird. Listing 8 zeigt, wie sich für ein Makro unterschiedliche Aufrufvarianten definieren lassen.

Eine praktische Anwendung dieser Funktion findet man beispielsweise in der Rust-Standardbibliothek beim `println`-Makro. Die oben gezeigte Implementierung des `print_named`-

Listing 12: Exportieren von Rust-Makros

```
// my_macros.rs
#[macro_export]
macro_rules! sum {
    ($a:expr, $b:expr) => {
        $a + $b
    };
}

// main.rs
mod my_macros;

fn main() {
    println!("{}", sum!(1, 2));
}
```

Makros ist zwar ein schönes Beispiel für mehrere Aufrufvarianten, hat aber einen entscheidenden Nachteil: Sie funktioniert nur mit einem oder zwei Parametern, nicht mit mehr. Darüber hinaus haben die Parameter fixe Namen (`a` im ersten Fall, `b` und `c` im zweiten). Eine alternative Implementierung wäre schön, bei der sich beliebig viele Parameter übergeben ließen. Wer das bezweckt, kann auf die zuvor erwähnten Wiederholungsoperatoren zurückgreifen. Listing 9 demonstriert diesen Ansatz. Zu beachten ist in diesem Beispiel, dass die Variable `\$name` keine Expression ist. Das Makro verlangt an dieser Stelle ausdrücklich einen Identifier entsprechend dem abstrakten Syntaxbaum von Rust.

Scoping von Variablen

Um Fehler zu vermeiden, verhindert Rust in Makros den Zugriff auf Variablen, die nicht zuvor im Scope des Makros de-

Listing 13: Geplante, neue Syntax für Rust-Makros

```
// my_macros.rs ========================================
// Note new macro syntax including exporting
// with regular `pub` modifier. No `#[macro_export]`
// required anymore.
pub macro sum ($a:expr, $b:expr) {
    $a + $b
}

// main.rs =============================================
#![feature(decl_macro)]

mod my_macros;
use my_macros::sum;

fn main() {
    // Use imported macro `sum`
    println!("{}", sum!(1, 2));

    // Note that forward declaration works with
    // new macro syntax.
    println!("{}", sub!(1, 2));
}

macro sub($a:expr, $b:expr) {
    $a - $b
}
```

EIN TEIL IT STECKT IN JEDEM MOTORRAD

JOIN OUR WINNING TEAM!

MATTIGHOFEN · MUNDERFING · LINZ · WELS · HAGENBERG

Kein passender Job für dich dabei?
Dann bewirb dich einfach initiativ!

JOBS.KTM.COM*

WEITERE SPANNENDE IT JOBS FINDEST DU AUF JOBS.AVOCODO.COM*

KTM

*KTM & Avocodo sind Teil der PIERER Mobility AG.

```
main.rs 2, U
src > main.rs > ...
1   macro_rules! variable_scoping {
2       () => {
3           println!("{}", number);    cannot find value `number` in this scope not found in this scope
4       };
5   }
6
    ▶ Run | Debug
7   fn main() {
8       let number: i32 = 1;
9       variable_scoping!();
10  }
11
```

>> Variable Scoping: Fehler wegen Zugriffs auf eine Variable außerhalb des Scope des Makro (Abb. 2)

klariert wurden. Das Programm in Listing 10 würde entsprechend nicht funktionieren, da es versucht, im Makro auf `number` zuzugreifen, das `number` jedoch außerhalb des Makros liegt (siehe Abbildung 2). Korrigieren lässt sich das Problem durch die explizite Übergabe des Identifiers. Listing 10 zeigt, wie das geht.

Nachteile von Makros in Rust gibt es auch

Ohne Zweifel sind Makros in Rust ein mächtiges Werkzeug. Sie haben jedoch nicht nur Vorteile. Die Unterstützung beim Debugging ist verbesserungswürdig. Zwar gibt es keine Probleme im Code, der ein Makro aufruft. Wer jedoch den Code innerhalb des Makros debuggen will, stößt schnell an Grenzen. Bei einfachen Makros verhält sich der Debugger wie erwartet. Werden die Makros jedoch komplexer, muss man häufig ohne Debugger auskommen.

Keine Forward Declaration

Makros in Rust unterstützen keine Forward Declaration. Aus diesem Grund muss man sie zunächst zu definieren, bevor man sie verwenden kann. Listing 11 zeigt das Problem. Die Funktion `sum` lässt sich vor der Definition in `main` aufrufen. Das sieht bei Makros anders aus. Um ein Makro zu verwenden, muss es vor dem Aufruf definiert sein, in diesem Fall also vor `main`.

Exportieren von Makros aus Modulen

Das Exportieren von Makros aus Rust-Modulen funktioniert anders als das Exportieren anderer Sprachkonstrukte (wie Strukturen oder Funktionen). Makros sind mithilfe des `macro_export`-Makros zu exportieren. Der sonst übliche `pub`-Modifier funktioniert bei Makros nicht. Listing 12 zeigt, wie sich das Makro `sum` aus einem Modul exportieren lässt.

Makros 2.0: Pläne für deklarative Makros

Seit Längerem bestehen Pläne für eine Überarbeitung des Systems für deklarative Makros in Rust. Beim Umschalten auf die Nightly-Version von Rust lassen sich die neuen Funktionen zum Teil bereits testen. Die neue Syntax kommt ohne `#[macro_export]` aus und löst bestehende Probleme wie die mangelnde Unterstützung von Forward Declarations.

Darüber hinaus ähnelt die neue Makro-Syntax wesentlich stärker der von Funktionen (Function-like Macros). Das Codebeispiel in Listing 13 zeigt die Änderungen exemplarisch. Es ist mit der Nightly-Version von Rust kompilierbar.

Makros in Rust: unverzichtbares Werkzeug

Makros in Rust sind nicht mit den Makros zu vergleichen, die man aus C kennt. In Rust sind Makros weitaus we-

niger fehleranfällig, da es sich nicht um eine textuelle Ersetzung handelt, sondern um die Transformation eines Abstract Syntax Tree. Rust-Makros haben das Potenzial, Code zu vereinfachen und die Entwicklungsgeschwindigkeit zu steigern, da manuelles Duplizieren von Code entfällt. Wie der Artikel zeigt, kann man so weit gehen, eigene DSLs in den Rust-Code zu integrieren. Diesbezüglich ist aber Vorsicht angesagt: Komplexe Makros sind nicht einfach zu entwickeln und das Debugging kann schwerfallen. Darüber hinaus besteht die Gefahr, dass der Code für Außenstehende, die mit der domänenspezifischen Sprache nicht vertraut sind, schwerer lesbar wird. Schließlich sieht der Rust-Code durch die eingebettete DSL nicht mehr aus wie allgemein übliches Rust.

Alles in allem sind Makros in Rust ein unverzichtbares Werkzeug und ein charakteristisches Merkmal der Sprache. Wer sich mit der Programmiersprache Rust beschäftigt, sollte sich auf jeden Fall Grundkenntnisse über Makros aneignen, um die vielen Makros in der Rust-Standardbibliothek und in den Rust Crates lesen zu können und im Fall des Falles den eigenen Code durch Makros kompakter zu machen. (sih)

Quellen

Links zum Rust Playground und zu den im Artikel erwähnten GitHub-Repositories sowie Verweise auf verwandte Artikel in der Rust-Kolumne Ferris Talk finden sich unter ix.de/zcn2.

Alvin Ramskogler

ist begeisterter Hobbyprogrammierer und studiert seit 2020 Informatik an der Johannes-Kepler-Universität Linz. Trotz seines jungen Alters von erst 15 Jahren hat Ramskogler bereits mehrjährige Erfahrung mit verschiedenen Programmiersprachen. Im Moment liegt der Fokus seiner Programmierarbeit auf Rust und TypeScript. Inhaltlich beschäftigt sich Alvin in der Zeit, die ihm neben Schule und Universität für das Programmieren bleibt, aktuell verstärkt dem Thema Compilerbau. Kennengelernt haben sich Ramskogler und Stropek im CoderDojo Linz, einem Programmierclub für Kinder und Jugendliche, in dem Stropek als Mentor mitarbeitet.

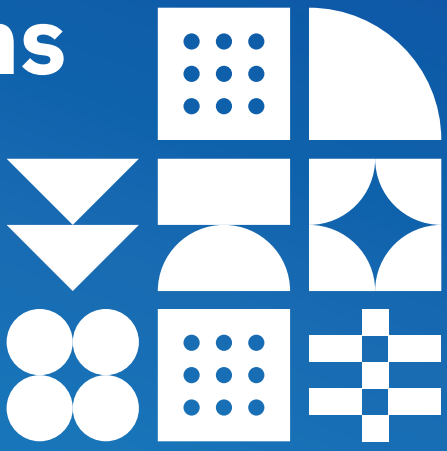
Rainer Stropek

ist IT-Unternehmer, Softwareentwickler, Trainer, Autor und Vortragender im Microsoft-Umfeld. Er ist seit 2010 MVP für Microsoft Azure und entwickelt mit seinem Team die Zeiterfassung für Dienstleistungsprofis time cockpit (timecockpit.com).

> Artificial Imagination: Die Zukunft des Films

Silke Hahn

Der Nordire Glenn Marshall bringt Deep Learning ins Kino: Sein KI-Kurzfilm „The Crow“ gewann in Cannes und Linz Preise. Im Interview teilt er Gedanken über eine neue Art von Kunst mit bildgebender KI – und gibt Einblick in seine Technik.



Der mit KI-Technik gestaltete Kurzfilm „The Crow“ führt eine neue Kategorie computergenerierter Bildkunst ein und gilt bereits als Pionierwerk eines neuen Genres, für das noch ein Name fehlt. CyberArt? Artificial Imagination? Beim Kurzfilmfestival in Cannes gewann der Animationsfilm im August den Kurzfilmpreis (Best Short Short), und in Österreich wurde er bei der International Competition for CyberArts mit dem Ehrenpreis des Prix Ars Electronica 2022 ausgezeichnet.

Mensch, Film und KI: Artificial Imagination

Wir haben mit dem Menschen hinter dem Werk gesprochen, dem nordirischen Visual Artist und Coder Glenn Marshall, und uns bei ihm über Technik und die Hintergründe seines Schaffens erkundigt, das er seit seinem ersten Prix Ars Electronica 2008 „Artificial Imagination“ nennt – ein Teilnehmer des gleichnamigen Festivals in Linz hatte ihn auf die Idee gebracht. Damals wurde Marshall ausgezeichnet für das offizielle Musikvideo zu Peter Gabriels Song „The Nest That Sailed The Sky“.

Der nun ausgezeichnete Film „The Crow“ ist eine knapp dreiminütige Animation, in der sich mittels Künstlicher Intelligenz eine Tänzerin in eine Krähe verwandelt. Das Ergebnis ist „ein eindringliches und fesselndes Werk, das die Krähe bei ihrem kurzen Tanz in einer apokalyptischen Landschaft begleitet“, bis zu ihrem „unausweichlichen Untergang“, wie es in der Filmbeschreibung heißt. Was dystopisch klingt, ist ein sinnliches Erlebnis. Bevor die Tänzerin sich durch technische Verfremdung in eine Krähe zu verwandeln begann, war sie Teil des Kurzfilms „Painted“ von Duncan McDowall und Dorotea Saykaly. Sie tanzt zu einem Stück des Komponisten Erik Satie (Gnossienne n° 3). Im Netz kursieren Ausschnitte daraus sowie verstreute Hinweise zum „Making-of“, der vollständige Film ist auf YouTube verfügbar.

Mit einem Pytii Colab Notebook und Code fand eine Verwandlung des Filmmaterials statt, soviel ließ sich bereits den Shortnotes entnehmen. Aus dem Film mit einer Tänzerin aus Fleisch und Blut wurde mit KI-Einsatz (Text-zu-Video) eine

In a Nutshell

- Text-zu-Bild: Das neue Genre KI-basierten Filmschaffens trägt noch keinen Namen: Artificial Imagination oder Cyberart umschreiben, worum es hier geht – aber auch Neural Art trifft den Kern, da neuronale Netzwerke an der Bild- und Filmsynthese mitwirken.
- Glenn Marshall ist Teil des Deforum-Projekts, das mit Stable Diffusion Pionierarbeit für die KI-gestützte Text-zu-Video-Produktion leistet. Durch technisches Wissen haben Filmmacher einen Vorsprung bei der Produktion von Text-to-Image-Werken: Wer programmieren kann, ist im Vorteil.
- In dem Kurzfilm „The Crow“ steht die Mensch-Maschine-Beziehung symbolisch im Mittelpunkt. Auch wenn man wie Marshall die ganze Kreativität in die Maschine steckt, bleibt gültig: All art is human.
- Im Interview gibt der Programmierer und Kunstschaffende Einblicke in sein Werk und die Entwicklung dieser rasch voranschreitenden Technik, an deren Ende wohl eine neue hybride Filmbranche stehen wird.

IN AGILEN WORKSTREAMS DIE CLOUD-LÖSUNGEN VON MORGEN ENTWICKELN.

DARUM SIND WIR BEI DATEV.

Gemeinsam sichere Cloud-Lösungen und innovative Apps realisieren: Als Cloud-Entwicklerin oder -Entwickler erwarten dich bei DATEV vielfältige Aufgaben in einer agilen Innovations-Kultur. Informiere dich über freie Stellen und spannende Projekte bei einem der führenden IT-Dienstleister in Europa.

Valeria und Dominik, Cloud-Entwicklerin und -Entwickler bei DATEV

DATEV.DE/KARRIERE

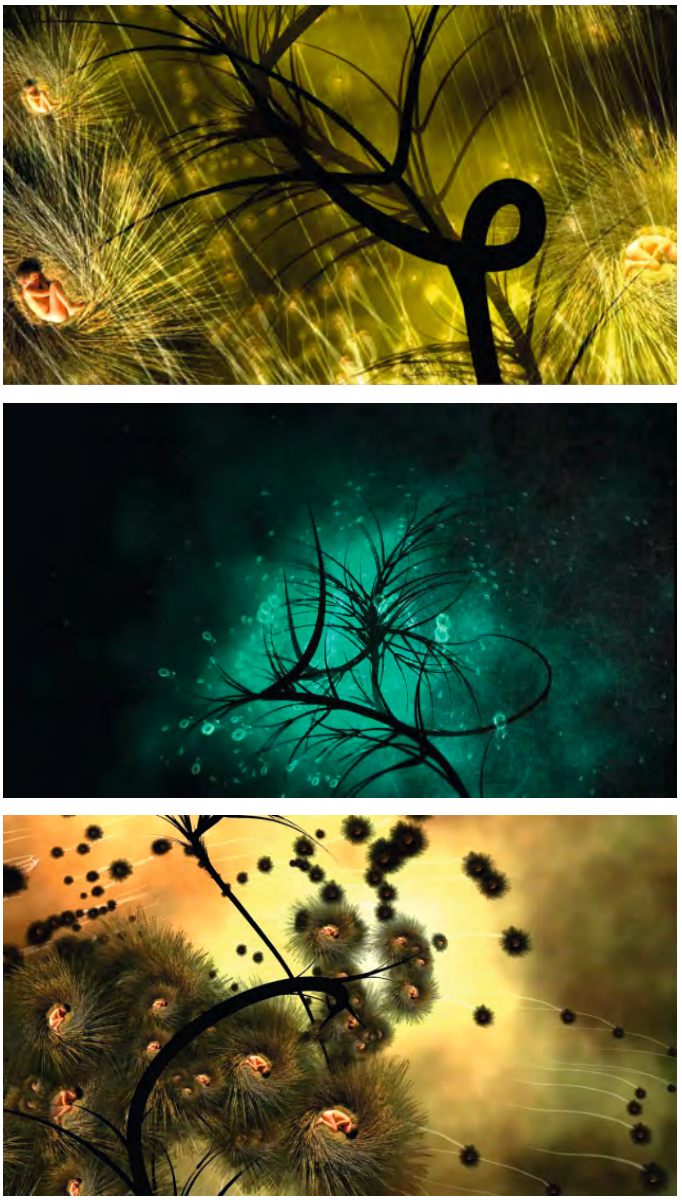
DATEV

Zukunft gestalten.
Gemeinsam.

gemäldehafte Animation mit Krähenfrau. Aber das ist noch nicht alles. Auf unter drei Minuten setzt „The Crow“ nicht nur ein visuelles und technisches Statement, sondern kommentiert die Mensch-Technik-Beziehung. Der Film werde zu einer Art Kampf zwischen dem Menschen und der KI, erklärte sein Regisseur – mit all der suggestiven Symbolik, die dieser aufgeladenen Auseinandersetzung innewohnt.

Glenn Marshall: „Kreativität in die Maschine gesteckt“

heise Developer: *Glenn, was ist denn dein Hintergrund – beruflich, wissenschaftlich, künstlerisch?*
Glenn Marshall: Das ist ein bisschen unkonventionell. Als Kind in der Grundschule hatte ich durchgehend Einsen in Kunst – ich konnte alles malen und zeichnen. Als ich dann aber auf die weiterführende Schule kam, hat mich das völlig erstickt, und ich bin in Kunst durchgefallen… dem einzigen Fach, das ich liebte und in dem ich wirklich gut war. Aber da hatte ich schon einen Computer daheim. Also habe ich meine ganze Kreativität in die Maschine gesteckt.
heise: *Was hat dich zu „The Crow“ inspiriert?*
Marshall: Den Kurzfilm „Painted“ und die ursprüngliche Liveaktion darin auf YouTube zu sehen … Mir war klar, dass meine Techniken perfekt dafür geeignet waren. Der Film wurde zur Grundlage von „The Crow“.
heise: *Oh, das klingt anders als bei den früheren Filmen, in denen du Text in zufällige Bilder mutieren lässt. Erklärst du uns kurz dein Vorgehen?*
Marshall: Die grundlegende Technik besteht darin, dass jedes Videobild in einen KI-Prozess eingespeist wird, der versucht, das Bild entsprechend einer CLIP-geführten Text-Bild-Aufforderung zu verändern.
heise: *Zum Bearbeiten hattest du also CLIP im Einsatz, Contrastive Language-Image Pre-Training von OpenAI. Was war dein Ausgangspunkt?*
Marshall: Ich hatte mich intensiv mit der Idee des KI-Stil-Transfers befasst und dabei Videomaterial als Quelle verwendet. Jeden Tag suchte ich also nach etwas auf YouTube – oder auf Stockvideo-Seiten – und versuchte, ein interessantes Video zu machen, indem ich es abstrahierte oder mit meinen Techniken in etwas anderes verwandelte. Wichtig in diesem Zusammenhang ist mein Film „Entwined“, den ich unmittelbar vor „The Crow“ gemacht habe. Für mich gehören die beiden zusammen. Nachdem ich „Entwined“ gedreht hatte, wurde mir klar, was für eine kraftvolle Quelle des Inputs Tanz ist. Damit perfektionierte ich meine Technik und war auf einem guten Weg. Also begann ich, mich im Internet nach Tanzfilmen umzusehen. Dabei bin ich auf „Painted“ gestoßen.
heise: *Zum Umwandeln von „Painted“ in deinen Film „The Crow“ brauchtest du noch einen Textprompt. Teilst du ihn mit uns?*
Marshall: Ein Bild einer Krähe in einer trostlosen Landschaft. Den habe ich natürlich auf Englisch eingegeben ('A painting of a crow in a desolate landscape'). Mit Prompts hatte ich bereits Erfahrungen gesammelt und hatte eine gewisse Vorstellung, was funktionieren würde. Das Video vollzog die Verwandlung dann von selbst… „The Crow“ hat eine Magie, die meine früheren Videos nicht hatten. Wahrscheinlich liegt das



Quelle: Glenn Marshall

>> Szenen aus „The Nest that Sailed the Sky“, dem offiziellen Musikvideo zu diesem Song für Peter Gabriel von Glenn Marshall

daran, dass die Tänzerin im zugrundeliegenden Video bereits die Bewegungen einer Krähe nachahmt und ein krähenähnliches Gewand trägt.
Deshalb funktioniert der Film so gut – die KI versucht, jedes Bild des Originals wie ein Gemälde mit Krähe darin aussehen zu lassen. Auf halbem Weg komme ich ihr entgegen, und der Film wird zu einer Auseinandersetzung zwischen der Tänzerin als Mensch und der KI – das hat suggestive Symbolik.
heise: *Inwiefern unterscheidet sich das vom Erstellen statischer Bilder?*
Marshall: Ich habe ja eine bewegte, animierte Szene erzeugt. Das ist ein bisschen anders als bei Einzelbildern. Man kann die Handlung, die Kamerabewegung und mehr beschreiben.

This ad will not influence the future of retail technology with every line of code. You will.

Bei ALDI SÜD gestalten über 2.400 IT-Kolleg:innen Tag für Tag die IT-Zukunft des Einzelhandels mit. In unseren Teams der Nationalen und Internationalen IT hast du die einmalige Gelegenheit, innovative Technologien für den weltweiten Einsatz voranzutreiben – in einem stabilen und zugleich lebendigen Arbeitsumfeld.

Die Facts.
ALDI SÜD als Arbeitgeber:

Moderner Tech-Stack: SAP, Adobe, Salesforce, M365 u.v.m.

Facettenreiches Team: Über 2.400 diverse Kolleg:innen in der IT

Global Player: IT-Projekte in 11 Ländern auf 4 Kontinenten

Future Work: Arbeit bis zu 100% remote möglich

Super Aussichten.
Und haufenweise Extras:

- Mobiles Arbeiten innerhalb Deutschlands inkl. Equipment
- Internationale Projekte
- Attraktive Vergütung
- Zukunftsorientiertes Training & Development
- Gesundheitsangebote

Deine Chance – bewirb dich jetzt!

it-jobs.aldi-sued.de

Unabhängig von den Texten und Bildern unserer Recruiting-Materialien möchten wir ausdrücklich betonen, dass bei ALDI SÜD alle Menschen gleichermaßen willkommen sind.

heise: *Manche sprechen von dir als Komponist (Composer). Komponierst du auch Musik? Oder bezieht sich das auf deine Filmkomposition mit KI-Handwerk und Technik?*
Marshall: Tatsächlich komponiere ich für manche meiner Filme auch Musik – allerdings fand ich die Bezeichnung als Komponist doch etwas komisch. Das ergibt keinen Sinn und verwirrt nur.
heise: *Wie bezeichnest du dich selbst? Composer, Developer, Machine Learner... Coder, oder vielleicht Freelance-Künstler? Falls es schon einen Namen für deine Tätigkeit gibt.*
Marshall: Wir versuchen alle noch herauszufinden, was wir sind. Zurzeit bin ich Vollzeit-KI-Künstler und Filmemacher. Aus dem Vorjahr habe ich bezahlte Projekte, die in die Richtung gehen. Andererseits bin ich der Coder-Typ, im Gegensatz zu dem „lässigen" Typ, den wir jetzt angesichts von Midjourney, DALL·E und so weiter auch haben. Es ist die Programmierer- und Entwickler-Community, die allen anderen immer einen Schritt voraus ist.
heise: *Ich sehe das Augenzwinkern... also passt KI-Künstler oder KI-Filmemacher am besten für dich?*
Marshall: Die andere Option heißt Neural Art und Neural Artist, aber „neuronale Kunst" klingt vielleicht anmaßend.
heise: *Was treibt dich an bei dem, was du tust?*
Marshall: Es hat etwas, in die neueste KI-Technologie einzutauchen und Teil der Programmierer- und Entwicklergemeinschaft zu sein, die Kunst und Animationen erschafft, die noch nie zuvor ein menschliches Auge gesehen hat. Das ist es, was mich anregt und inspiriert. Aber mit diesen Werkzeugen muss man immer noch etwas handwerklich erschaffen. Es reicht

Glenn Marshall

Quelle: Glenn Marshall

>> Porträt des Visual Artist und Programmierers Glenn Marshall

Seine berufliche Laufbahn im Bereich der Computeranimation erstreckt sich über mehr als 20 Jahre, in denen er experimentelle CGI-, generative und KI-Technologien einsetzt, um eine philosophische Vision der digitalen Kunst von morgen zu verfolgen. Er wurde mit dem Major Individual Award des Arts Council of Northern Ireland (ACNI) ausgezeichnet (2015), der höchsten Anerkennung, die an führende Künstler des Landes vergeben wird. Außerdem ist er zweimaliger Gewinner des Prix Ars Electronica, erhielt für „The Crow" den Kurzfilmpreis 2022 in Cannes. Er hat mit Peter Gabriel und Tangerine Dream an Musikvideos und Konzertvisualisierungen zusammengearbeitet. Seinen Kurzfilm „The Crow", Verknüpfungen zu einigen seiner Musikvideos, zu seinem YouTube- und Twitterkanal sowie zu weiteren Quellen über Glenn Marshall und sein Schaffen finden Neugierige unter ix.de/ztu7.

Von der Vision zur Realität

Entwickeln Sie **Zukunft**

Mit modernen Technologien zur IT-Lösung.

Als international agierende Unternehmensgruppe mit weltweit mehr als 9.000 Mitarbeitenden bieten wir ausgezeichnete Karrierechancen in der Softwareentwicklung und IT-Beratung. Wir unterstützen Sie kontinuierlich beim Ausbau Ihrer Qualifikationen. Denn unser gemeinsamer Erfolg ist die Basis Ihres persönlichen Fortschritts. Überzeugen Sie sich selbst. Steigen Sie ein bei msg und zeigen Sie uns, was Sie können!

- Flexible Arbeitszeiten
- Möglichkeit für Homeoffice & mobiles Arbeiten
- Umfangreiche Weiterbildungs- & Gesundheitsangebote

karriere.msg.group

value – inspired by people

.msg

nicht, jeden kurzen, schrulligen Test wie Werbemüll hochzuladen, um der Welt zu zeigen, wie cool diese Technologie ist. Was 99 Prozent der Leute aber zu machen scheinen.

Marshall: Ich bin entschlossen, den Weg zu weisen … und zu zeigen, dass es bloß ein Werkzeug ist, um ein vollendetes Kunstwerk zu erstellen – wie beispielsweise „The Crow".

heise: *Du bist ja schon seit geraumer Zeit als Pionier und Wegbereiter in der bildenden Kunst unterwegs. In den letzten zwanzig Jahren hast du oft zu innovativen Techniken gegriffen, zuletzt zu Deep Learning und Text-zu-Bild-Synthese. Gibt es eine Liste deiner Kunstwerke, um die Entwicklung nachzuvollziehen?*

Marshall: Mein YouTube-Kanal ist eine Art Dokumentation der Anfänge und Fortschritte der KI … beginnend mit Prozessen der Text-zu-Bild-Synthese im Januar 2021 bis zum heutigen Tag. Einer meiner ersten Filme „A Bleak Midwinter" ist ein früher Versuch, Poesie mithilfe von KI zu visualisieren. Bei meinem jüngsten Werk „Everything is in its Right Place" – einem Radiohead-Video – kommt Stable Diffusion zum Einsatz. Ich denke, es ist eines der besten Dinge, die ich je gemacht habe.

Marshall: Ihr solltet euch Stable Diffusion mal ansehen – das ist ein Open-Source-KI-Modell, das im August veröffentlicht wurde und dabei ist, die KI-Kunstszene zu revolutionieren.

heise: *Danke für den Tipp! Deine Einschätzung teile ich. Wir hatten eine Meldung zum Public Release von Stable Diffusion und bleiben am Ball …*

Marshall: Die Geschwindigkeit und die Qualität der Ergebnisse sind verblüffend. Wenn ich heute ein Remake von „The Crow" machen würde, würde es wahrscheinlich doppelt so gut ausschauen.

heise: *Wann genau hast du „The Crow" gemacht?*

Marshall: Im Dezember 2021. So schnell ist die Entwicklung der KI!

Stable Diffusion und die Medienrevolution

Der frei verfügbare Text-zu-Bildgenerator Stable Diffusion erregt seit August 2022 Aufsehen. KI-gestützte Bildgenerierung hört jedoch nicht bei statischen Bildern auf, sondern berührt die Produktion von Bewegtbild und Filmen. Neue Tools und Techniken für das Filmemachen zeichnen sich ab: Mit Text-zu-Video-Produktionen ist verstärkt zu rechnen, unter anderem das Team hinter Stable Diffusion arbeitet Tweets zufolge an der Weiterentwicklung des eigenen Modells für einen solchen Einsatz.

Quelle: Glenn Marshall

>> Standbilder aus dem Musikvideo „Everything in its Right Place" von Radiohead. Die Bilder hat Marshall mit Stable Diffusion und Deforum erstellt.

Der frühere KI-Chef von Tesla Andrej Karpathy führte ein mit Stable Diffusion erstelltes Video vor, für das er den Code auf GitHub bereitstellt. Auch Glenn Marshall setzt sich gemeinsam mit anderen für die Weiterentwicklung in Richtung Text-zu-Video ein: Hinter dem Projekt Deforum, an dem er beteiligt ist, steht eine Community aus Entwicklern, Enthusiasten und Kunstschaffenden, die sich mit der KI-Bildsynthese befassen.

Discord

Spekulationen über die Zukunft des Kinos im Zeitalter KI-generierter Bilder gehen mit dieser technischen Entwicklung einher, die in der Tat vieles, was wir an Bildkunst und künstlerischen Schaffensprozessen kennen und gewohnt sind, verändern dürfte. Eine Frage wird sein, wer künftig Filme schafft und wie die heutigen Filmschaffenden arbeiten werden. Erste Werke sind bereits greifbar und zeigen, dass neben eigener Vorstellungskraft weiterhin eine große Portion Know-how und auch Programmierkunst dazugehören, um zum jetzigen Zeitpunkt mit KI-Technik Filmkunst zu machen: Kunst kommt nicht auf Knopfdruck, sondern vom Können.

Quelle: Glenn Marshall

>> Glenn Marshall hat mit Deforum und Stable Diffusion ein Musikvideo erstellt zum Radiohead-Song „Everything in its Right Place".

heise: *Wahnsinn. Wie stellst du dir die Zukunft des Filmschaffens und Kinos vor?*
Marshall: Man muss sich nur „The Mandalorian“ ansehen – ohne zuviel zu verraten, ich mag nicht spoilern: Dort werden einige berühmte ältere Charaktere aus der ursprünglichen Trilogie mit Deepfake-Technik wieder „zum Leben erweckt“. Ein echter Schauspieler kam zum Einsatz, aber nur als Referenz für die KI, um ihm das komplette Gesicht und die Stimme einer mit KI-Technik trainierten Version des berühmten Charakters einzupflanzen … Ich würde gern einen weiteren Beatles-Film sehen, in dem alle vier überzeugend wieder zum Leben erweckt sind.
heise: *Ich denke auch an die lebenden Menschen, Regisseure und Schauspieler: Was bedeutet die technische Evolution für die Filmbranche?*
Marshall: Wir steuern möglicherweise auf eine neue hybride Filmform zu, irgendwo zwischen Live-Handlung, Animation und Deepfake. Viele Schauspieler könnten am Ende nur noch Puppen sein. Regisseure können jetzt ihre eigenen Konzeptzeichnungen und Matte Paintings für ihre Sets und Spezialeffekte entwerfen. Das gibt dem Regisseur eine persönlichere und direktere Vision. Theoretisch könnte man einen ganzen Spielfilm mit KI erzeugen. Bisher gibt es noch keine professionellen oder künstlerisch nutzbaren Beispiele, aber das wird nicht mehr lang dauern.
heise: *Gibt es ein Best- und ein Worst-Case-Szenario? Wie können Künstler und Entwickler sich auf die Zukunft vorbereiten?*
Marshall: Um ehrlich zu sein, mache ich mir über Fragen zur Zukunft der KI keine Gedanken. Ich bin im Moment und habe Spaß. Aber wenn du den gegenwärtigen Moment betrachtest, ist die Zukunft sowieso immer da.
heise: *Stimmt. TS Eliot, The Eternal Present … Woran arbeitest du gegenwärtig?*
Marshall: Ich plane eine Fortsetzung von „The Crow“, aber ich werde die dafür benötigte Live-Handlung selbst drehen. So habe ich die vollständige kreative Kontrolle über das ganze Stück und wie die KI es interpretiert – so viele Möglichkeiten …

Das Gespräch führte Silke Hahn (sih@ix.de), Redakteurin bei iX und heise Developer. (sih)

Quellen

Links zu den Videos und zum Schaffen des Visual Artist Glen Marshall findet sich unter ix.de/ztu7.

> KI-Systeme: MLOps und Model Governance

Isabel Bär und Kilian Kluge

Compliance und Vertrauen: Mit den richtigen Tools und Prozessen lassen sich KI-Systeme wirksam kontrollieren und im Einklang mit rechtlichen Vorgaben betreiben.

Moderne KI-Systeme haben den Ruf, Black Boxes zu sein, deren Funktionsweise Anwenderinnen und Anwendern und Entwicklerinnen und Entwicklern verborgen bleibt. Auf dem Weg zur künftigen Nutzung von KI stehen Unternehmen daher vor Herausforderungen, die sich bisher in der klassischen Softwareentwicklung nicht gestellt hatten. Zum einen müssen Teams aus Daten angelernte Modelle schnell und effizient in den Produktivbetrieb bringen und anschließend fortlaufend überwachen und aktualisieren. Dabei helfen die Vorgehensmodelle und technischen Komponenten der Praxis der Machine Learning Operations (MLOps).

Explainable AI macht nachvollziehbar, wie KI-Systeme Entscheidungen treffen

Zum anderen muss sichergestellt sein, dass die KI-Systeme alle relevanten rechtlichen Auflagen erfüllen und keine unternehmerischen Fehlentscheidungen oder Reputationsschäden verursachen. Dazu braucht es Model Governance, aber auch Methoden, die die Entscheidungen von KI-Systemen für alle Stakeholder nachvollziehbar machen. Hier bietet Explainable AI (Erklärbare KI, kurz: XAI) eine Vielfalt von Ansätzen, aus den komplexen mathematischen Strukturen der eingesetzten Modelle für Menschen verständliche Erklärungen zu extrahieren (vertiefende Hinweise hierzu finden sich unter ix.de/zxaa).

Ein nachhaltiger Erfolg von KI-Software kann sich nur für Unternehmen einstellen, die ihre KI-Systeme auf diesen drei Grundpfeilern – MLOps, Model Governance und Explainable AI – aufbauen (siehe Abbildung 1). Um deren Zusammenspiel greifbar zu machen, zeigt dieser Artikel anhand von Beispielen aus der Praxis, wie die Integration der drei Elemente dazu beiträgt, solide KI-Anwendungen zu bauen.

Anwendungsfall: Automatisiertes Vorfiltern von Bewerbungen

In unserem Anwendungbeispiel möchte ein Unternehmen seinen Bewerbungsprozess effizienter gestalten und plant den Einsatz einer KI-basierten, automatischen Vorfilterung von Bewerbungen: Das System soll erkennen, welche Bewerbungen vielversprechend sind und in ein Bewerbungsgespräch münden sollten.

In Zeiten, in denen Personalabteilungen nach einem flüchtigen Blick in die Unterlagen entscheiden, ob sie einer Bewerbung mehr Aufmerksamkeit widmen möchten, erscheint die automatisierte Vorfilterung als lohnender Prozess. Doch auf dem Weg zur Umsetzung eines solchen Systems lauern zahlreiche Fallstricke und Risiken (siehe Abbildung 2 und ix.de/zxaa).

In a Nutshell

- Durch die EU-Gesetzgebung ist künftig eine verschärfte Regulatorik zu automatisierten Entscheidungssystemen zu erwarten.
- Auch im Interesse unternehmerisch guter Entscheidungen sollten Unternehmen von Beginn an daran arbeiten, ihre KI-Systeme und MLOps-Infrastruktur entsprechend aufzubauen und abzusichern.
- KI-Systeme können kontrollierbar und transparent sein: Eine technisch sauber umgesetzte MLOps-Infrastruktur hilft, um rechtliche Compliance zu gewährleisten, und stellt auch die Erklärbarkeit von KI (XAI) sicher.

Klassische Softwareentwicklung funktioniert anders als Machine Learning

Sowohl Machine-Learning-basierte als auch herkömmliche Software sind Abfolgen von Anweisungen, nach denen ein Computer Daten verarbeitet. Der wesentliche Unterschied liegt in der Art und Weise, wie die Software jeweils entsteht. Traditionellerweise wird die gewünschte Funktionsweise Schritt für Schritt entwickelt.

Mit Methoden wie Unit- und Integrationstests lassen sich dabei die Funktion einzelner Komponenten und ihr ordnungsgemäßes Zusammenspiel sicherstellen. Das Ineinandergreifen der einzelnen Bausteine ist also – bis hinab auf eine jeweils zweckmäßig gewählte Abstraktionsebene – geplant und sichtbar.

Machine Learning kommt in der Regel dann zum Einsatz, wenn die klassische Vorgehensweise nicht geeignet ist, weil zwar Eingabe und gewünschte Ausgabe definiert sind, der Rechenweg dazwischen jedoch unbekannt ist. Sind genügend Beispiele von Eingaben und zugehörigen Ausgaben verfügbar, lässt sich auf den Daten ein ML-Modell trainieren, das den Zusammenhang zwischen Eingabe und Ausgabe abbildet – sofern die innere Struktur des Modells und seine Komplexität dazu geeignet sind. Während die innere Struktur eines Modells in der Regel strikt von seinen Entwicklerinnen und Entwicklernvorgegeben ist, ermittelt ein Trainingsalgorithmus die passenden Parameter.

Notwendigkeit von Model Governance und Explainable AI

Diese Eigenheiten sind die Wurzel zahlreicher Herausforderungen für Entwicklerinnen und Entwickler und Betreiberinnen und Betreiber solcher KI-Systeme (mehr dazu unter ix.de/zxaa). So meldet die Personalabteilung unseres Beispielunternehmens bereits im Vorfeld Zweifel an, ob sie sich auf die Auswahl der KI wird verlassen können und ob die automatisierte Entscheidungsfindung ähnlich dem bisherigen rein manuellen Auswahlprozess anhand sinnvoller Kriterien erfolgt. Gleichzeitig stellt sich das Management die Frage, wie Risiken bei der Vorfilterung der Bewerbungen begegnet und die Einhaltung rechtlicher Vorgaben sichergestellt werden kann (siehe Abbildung 2).

Rechtliche und regulatorische Anforderungen an KI-Systeme

Aus der rechtlichen Perspektive richtet sich der Blick zunächst auf gesetzliche Vorschriften. Im Personalbereich unterliegen Unternehmen mindestens dem Allgemeinen Gleichbehandlungsgesetz (AGG) und der Datenschutzgrundverordnung (DSGVO).

Zukünftig werden zudem viele KI-Anwendungen den Vorgaben der KI-Verordnung der EU genügen müssen, die derzeit in Brüssel Form annimmt und bestehende Regularien

Checkliste Model Governance

Der Einsatz von Machine Learning bringt Verantwortung und Verpflichtungen mit sich. Um diesen Anforderungen nachzukommen, benötigt ein Unternehmen Prozesse, durch die es

- die Zugriffe auf ML-Modelle kontrolliert
- Richtlinien/gesetzliche Vorgaben umsetzt
- die Interaktionen mit den ML-Modellen und deren Ergebnisse verfolgt
- festhält, auf welcher Grundlage ein Modell erzeugt wurde.

Model Governance bezeichnet diese Prozesse in ihrer Gesamtheit.

Checkliste:

- Vollständige Modelldokumentation oder Berichte. Dazu gehört auch das Reporting der Metriken durch geeignete Visualisierungstechniken und Dashboards
- Versionierung aller Modelle zur Herstellung von Transparenz nach außen (Erklär- und Reproduzierbarkeit)
- Vollständige Datendokumentation zur Gewährleistung hoher Datenqualität und Einhaltung des Datenschutzes
- Management von ML-Metadaten
- Validierung von ML-Modellen (Audits)
- Laufendes Überwachen und Protokollieren von Modellmetriken

ergänzen wird. Diese Verordnung teilt KI-Systeme in vier unterschiedliche Risikokategorien ein („unzulässig“, „hoch“, „begrenzt“, „minimal“). Die Risikokategorie definiert dabei die Art und den Umfang der Anforderungen, die an das jeweilige KI-System zu stellen sind. „KI-Systeme, die in den Bereichen Beschäftigung, Personalmanagement und Zugang zur Selbstständigkeit eingesetzt werden, insbesondere für die Einstellung und Auswahl von Personen“, gelten nach der EU-Verordnung per Definition als Hochrisiko-KI-Systeme und sind mit einer umfangreichen Liste an Vorgaben in Einklang zu bringen

Vor diesem Hintergrund ergeben sich vier Kernanforderungen an KI-Systeme:

- KI-Entscheidungen müssen replizierbar sein.
- Die ordnungsgemäße Funktion der eingesetzten Modelle muss überprüfbar sein.
- Auf eine Verschlechterung der Performance muss schnell reagiert werden.
- Die fachliche Korrektheit der Entscheidungsfindung muss sich überprüfen lassen.

Im Folgenden zeigen wir, wie sich die vier zunächst abstrakten Anforderungen mithilfe von MLOps bewältigen lassen. Ähnlich wie DevOps in der Softwareentwicklung bezeichnet MLOps sowohl Vorgehensmodelle als auch die zugehörigen Tools für Entwicklung, Deployment und Betrieb von ML-basierten Systemen, die ursprünglich entwickelt wurden, um die Zeitspanne vom Entwicklungsbeginn bis zum Produktiveinsatz (time-to-market) zu verkürzen und den zuverlässigen Betrieb von KI-Systemen sicherzustellen (hierzu finden sich weitere Informationen unter ix.de/zxaa). Abbildung 3 zeigt schematisch die wesentlichen Komponenten einer MLOps-Infrastruktur.

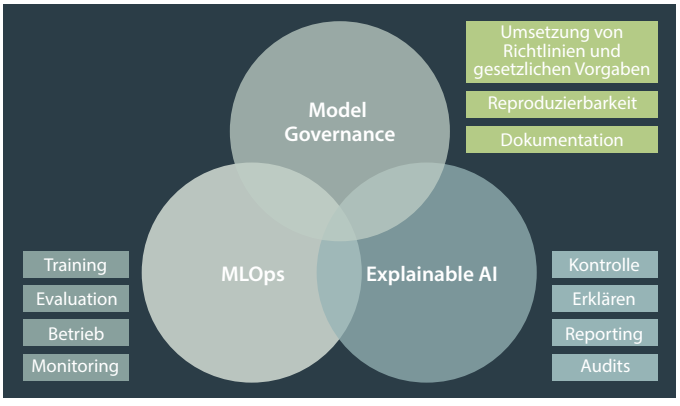
KI-Entscheidungen müssen replizierbar sein

Reproduzierbarkeit bezeichnet in der Wissenschaft die Fähigkeit, ein bestimmtes Experiment mit gleichem Ergebnis zu wiederholen. Reproduzierbarkeit ist auch für KI-Systeme relevant: So muss es möglich sein, jedes Modellergebnis replizieren zu können – beispielsweise, wenn Betroffene einer bestimmten algorithmischen Entscheidung widersprechen.

Der Grundstein für Reproduzierbarkeit wird schon zu Beginn der Entwicklung gelegt durch das Entwickeln einer robusten Trainingsprozedur. Sie umfasst in der Regel eine Data-Engineering-Komponente (Laden, Validierung, Transformation), das Training und eine abschließende Evaluation. Wie bei klassischer Softwareentwicklung sollte sämtlicher Code und sollten alle Konfigurationen einschließlich der Infrastruktur versionskontrolliert sein. Um Reproduzier- und Wiederverwendbarkeit zu gewährleisten, müssen Entwickler und Entwicklerinnen darüber hinaus für jeden Durchlauf der Trainingsprozedur Daten, Modelle und Parameter protokollieren (Experiment Tracking). Hier sollten die Verantwortlichen prüfen, ob die dokumentierten Informationen ausreichen und spezifisch genug sind, um die Erzeugung des Modells vollständig replizieren zu können.

Repositories helfen, Daten und Features wiederzufinden

Der Aufbau von Daten- und Feature-Repositories erleichtert die Nutzung, Auffindbarkeit, Wiederverwendbarkeit und



Quelle: Kilian Kluge

>> Erst das Zusammenspiel von Model Governance, MLOps und Explainable AI ermöglicht den zuverlässigen und gewinnbringenden Einsatz von KI-Systemen (Abb. 1).

Anzeige

FRITZ! und die „Friends of Debug“

FRITZ!-Produkte funktionieren reibungslos untereinander und mit den Geräten der Anwender – und das millionenfach in ganz Europa. Schon in der Basisentwicklung von FRITZ!OS, der Software aller FRITZ!-Produkte, legt das Berliner Unternehmen viel Wert auf Prävention und Analyse. Dafür wertet das Team „Friends of Debug“ bei AVM unter anderem sogenannte Crash Reports von FRITZ!-Geräten aus. Jens Krieg, Teamleiter in der Abteilung Infrastructure und Quality Management bei AVM, beschreibt, wie dieser wichtige Schritt in Zukunft dank Kotlin verbessert werden soll.

Jens Krieg, Teamleiter in der Abteilung Infrastructure und Quality Management bei AVM

Jens, worum geht es genau bei der Auswertung der Crash Reports?
Es geht um die automatisierte Erkennung von Fehlerbildern durch die Bestimmung von Mustern. Das Identifizieren von bestimmten Anomalien in den Reports geschieht auf Basis von Entscheidungsbäumen. Zusätzlich erfolgt die Wissenserkennung mittels Cluster-Analyse, um auf Fehlerquellen zu schließen und diese zu beheben. Wir wollen das mit Kotlin umsetzen, da wir mit der modernen Umgebung komplexe Sachverhalte sehr kompakt lösen können. Außerdem ist Kotlin vollständig interoperabel zu Java – wir können also unsere auf Java-basierende Datenverarbeitungsplattform weiternutzen. Gleichzeitig haben wir die Vorteile einer modernen Programmiersprache, wie z. B. das elegante Null-Pointer-Exception-Handling. Der Multi-Plattform-Gedanke von Kotlin ist ebenfalls wichtig, da wir dieselbe Sprache für die Realisierung von Back- und Frontend nutzen können.

Soweit die Theorie – wie sieht es in der Praxis aus?
Wir starten gerade richtig durch, um neue Stabilitäts- und Fehleranalysen des FRITZ!OS mittels Cluster-Analyse und Kotlin zu lösen. Deshalb suchen wir jemanden, der uns mit seinem tiefgreifenden Wissen in Kotlin unterstützen kann. Wer hier dazukommt, hat die tolle Chance einen wichtigen Teil im Entwicklungsprozess komplett von Anfang an mit aufzubauen.

Du kümmerst dich aber nicht nur ums Debuggen oder?
Ich leite noch ein zweites Team namens „Testsysteme“. FRITZ!-Produkte arbeiten auch deshalb so gut zusammen, weil sie ein gemeinsames FRITZ!OS teilen. Wir haben die Aufgabe, eine Testumgebung für die gesamte FRITZ!OS-Entwicklung aufzubauen. In dieser Testumgebung können unsere Entwicklerinnen und Entwickler ihre Änderungen am Code jederzeit überprüfen, womit Fehler frühzeitig erkannt werden können. Das erleichtert den Arbeitsprozess ganz wesentlich.

Im Team „Friends of Debug“ führen kreative Lösungswege zu mehr Qualität

Intern nennt ihr euch „Friends of Debug“. Was hat es damit auf sich?
„Friends of Debug“ ist unser Team im Qualitätsmanagement und der Name zeigt schon, wo die Reise hingeht. Wir pflegen alle einen lockeren, respektvollen Umgang miteinander und haben, was ich persönlich sehr angenehm finde, ein total gemischtes Team. Da treffen Uni-Absolventen auf erfahrene Kollegen. Dadurch entsteht ein Austausch, der überaus wertvoll ist.

Und was für einen neuen „Friend“ sucht ihr?
Wir suchen sowohl für „Friends of Debug“ als auch für die Entwicklung des Testsystems jemanden, der den Sachen auf den Grund gehen will. Der sich nicht zufrieden gibt, wenn etwas funktioniert, sondern verstehen will, warum das so ist. Deshalb ermutige ich die Leute in meinen Teams auch sich in ihren eigenen, privaten Projekten auszuleben. Das fördert die Kreativität bei der Lösungssuche, was bei der Arbeit hilft und auch die Qualität steigert.

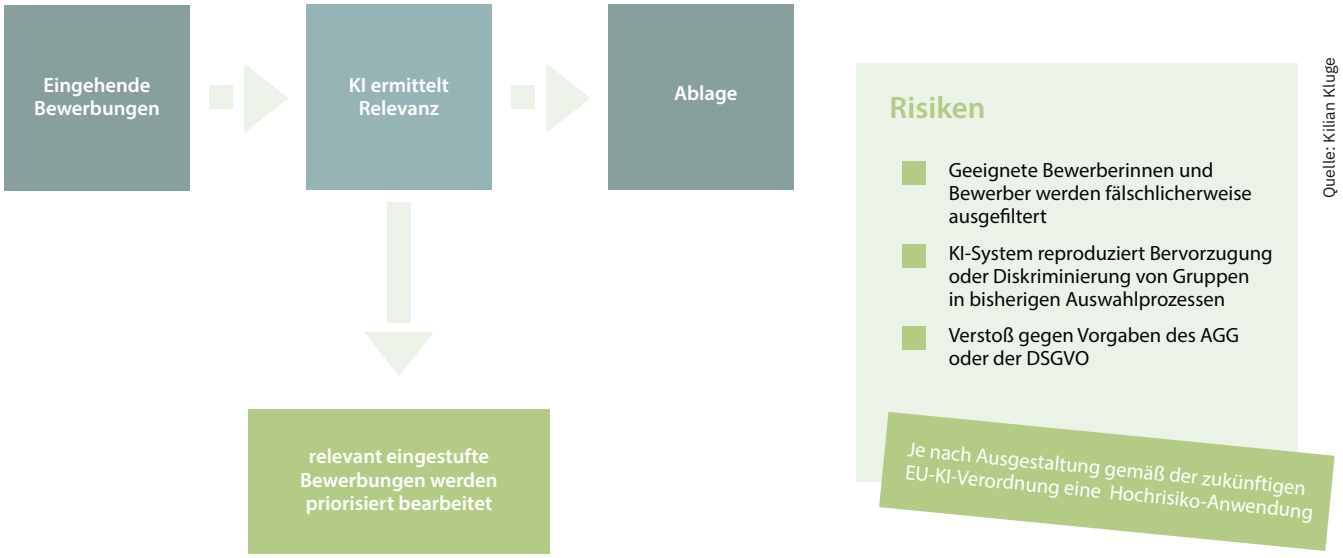
Hier geht es zu den Stellenausschreibungen:

Fullstack Entwickler / Kotlin (w/m/d)
jobs.avm.de/bist-du-fritz-genug/fullstack-entwickler-kotlin-wmd

Softwareentwickler Testsysteme (w/m/d)
jobs.avm.de/bist-du-fritz-genug/softwareentwickler-testsysteme-wmd

Über AVM:
AVM ist Europas führender Hersteller von Produkten für das digitale Zuhause. Mit rund 880 Mitarbeitenden und der bekanntesten Marke für WLAN-Router bringt AVM Millionen von Menschen ins Internet. Du hast Lust in einem offenen und motivierten Team an zukunftsweisenden Technologien mitzuwirken? Dann bewirb dich unter **jobs.avm.de**!

AVM GmbH / Recruiting Team
Alt-Moabit 95 • 10559 Berlin
Telefon +49 30 39976-600
E-Mail: work@avm.de



>> Schon die automatisierte Priorisierung von Bewerbungen durch ein KI-System birgt zahlreiche unternehmerische Risiken und unterliegt rechtlichen Vorgaben (Abb. 2).

Versionierung von Datenbeständen. Zur dauerhaften Speicherung und Versionierung von Artefakten wie Datensplits, Modellen und Metriken stehen spezialisierte MLOps-Plattformen und Tools wie Model Registries zur Verfügung. Bei der Speicherung von Modellen mit den zugehörigen Metadaten sollte sichergestellt sein, dass sie nicht nur wiederherstellbar, sondern auch manipulationssicher abgelegt werden. Je nach den Anforderungen des Anwendungsfalls kann es sich als notwendig erweisen, Revisionssicherheit zu garantieren. Um einzelne KI-Entscheidungen replizieren und auditieren zu können, ist es zudem nötig, im Betrieb die Eingabedaten so zu protokollieren, dass sie später erneut einspielbar sind (Serving Logs).

KI-Verordnung: Anforderungen

Anforderungen an Hochrisiko-KI-Systeme in dem Vorschlag der EU-Kommission zur KI-Verordnung vom 21. April 2021:

- Einrichtung eines Risikomanagementsystems (Artikel 9)
- Einhaltung und Dokumentation von Datenqualitätsstandards und Data-Governance-Prozessen (Artikel 10)
- Technische Dokumentation (Artikel 11)
- Umfassende Protokollierung des Systems zwecks Rückverfolgbarkeit (Artikel 12)
- Transparenz und Dokumentation des Systems gegenüber seinen Nutzern (Artikel 13)
- Ermöglichung einer wirksamen Aufsicht und Kontrolle durch Menschen (Artikel 14)
- Einhaltung hoher Standards im Hinblick auf Genauigkeit, Robustheit und Cybersicherheit (Artikel 15)

Die ordnungsgemäße Funktion der eingesetzten Modelle muss überprüfbar sein

Um zu prüfen, ob ein KI-System einwandfrei funktioniert und um Leistungsverluste zu diagnostizieren, müssen Entwicklerteams die Performance überwachen. Sie lässt sich je nach Anwendungsfall durch verschiedene Metriken quantitativ erfassen – im hier dargestellten Beispiel etwa die Rate korrekt als „relevant“ und „irrelevant“ eingestufter Bewerbungen (Accuracy). Die Definition geeigneter Metriken ist Bestandteil der Entwicklung. Die Metriken kommen bei der Evaluation zum Einsatz und ermöglichen den Vergleich verschiedener Modellversionen. Evaluationsergebnisse sollten wie die übrigen Daten protokolliert werden, um im weiteren Verlauf als Referenzwerte zu dienen.

Zudem ist es ratsam, Key Performance Indicators (KPI) festzulegen, die den Nutzen einer KI-Anwendung aus Perspektive des Anwendungsfalls quantifizieren. So könnte im Falle der automatisierten Vorfilterung von Bewerbungen das Reduzieren der Zeitspanne vom Eingang einer relevanten Bearbeitung bis zum Erstkontakt ein passender Indikator sein. Im Betrieb lässt sich die Performance anhand von Metriken und KPIs des jeweils produktiven Modells evaluieren (Continuous Evaluation). Das setzt häufig voraus, dass wie beim Training des Modells die Ground-Truth-Labels bekannt sind, also die Kenntnis darüber, ob eine Bewerbung, die das Modell als relevant klassifiziert, es auch tatsächlich ist. In unserem Beispiel liegt diese Information für als relevant eingestufte Bewerbungen erst dann vor, wenn eine Personalerin oder ein Personaler die Bewerbung zu Gesicht bekommen hat. Für die als irrelevant eingestuften Bewerbungen findet das jedoch nicht statt, hier wären bei Bedarf aussortierte Bewerbungen stichprobenartig zu begutachten.

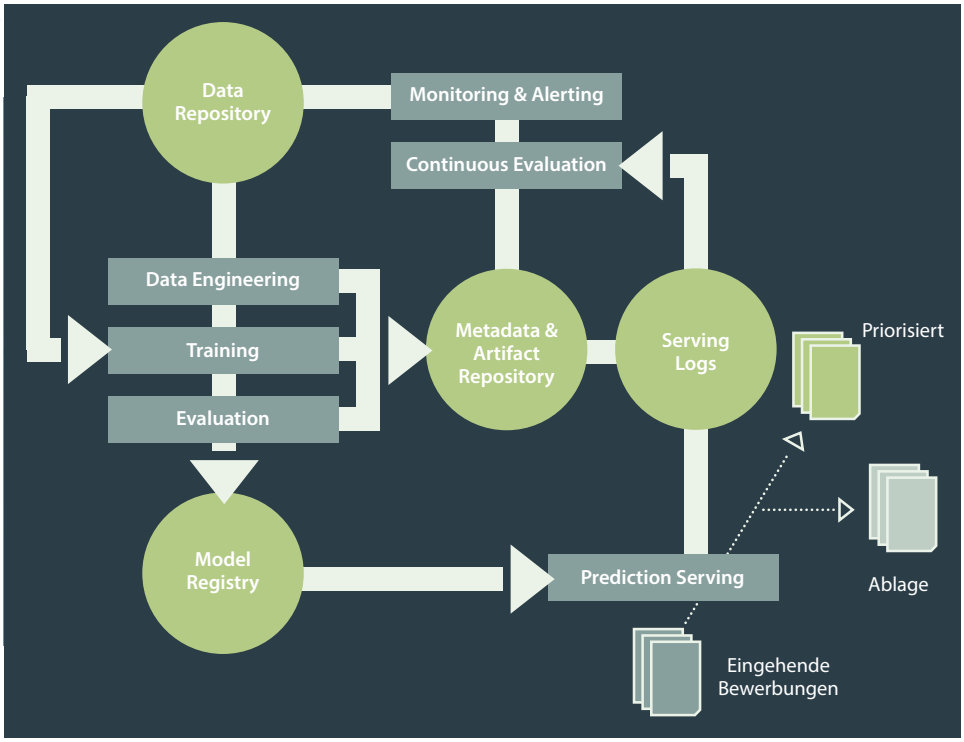
Zudem ist zu prüfen, ob sich die Eingabedaten, hier also die neu eintreffenden Bewerbungen, von denen unterschei-

den, die für den Aufbau des Modells verwendet worden sind. Bei einer zu starken Abweichung zwischen den Trainings- und Produktivdaten (Distribution Shift und Concept Drift) kann es zu verschlechterter Modellperformance kommen, weil die auf den alten Daten erlernte Modelllogik für die neuen Daten nicht mehr gültig ist. Um diese Abweichungen erkennen zu können, ist ein Daten-Repository zentral, in dem etwa Schemata und statistische Beschreibungen der Trainingsdaten vorgehalten werden.

Veränderung der Modell-Performance

Wird eine Veränderung der Modell-Performance oder eine Veränderung der Eingabedaten erkannt, ist in der Regel die Erzeugung einer neuen Modellversion angezeigt. Dazu führen Entwicklerinnen erneut die komplette Trainingsprozedur durch, beginnend mit Auswahl und

Quelle: Kilian Kluge



>> Schematische Darstellung einer MLOps-Infrastruktur für das Anwendungsbeispiel (Abb. 3)

Quelle: Kilian Kluge

>> Es gibt zahlreiche Methoden zur Erklärung von KI-Systemen. Im Beispiel könnten je nach Zielgruppe und Fragestellung die dargestellten Ansätze zum Einsatz kommen (Abb. 4).

Aufbereitung von Trainingsdaten. Hierbei kann auf die bei der Entwicklung des Modells implementierten Komponenten zurückgegriffen werden, was die exakte Reproduktion der Abläufe sicherstellt. Erzielt das neu erzeugte Modell – gemessen an den definierten Metriken – bessere Ergebnisse als die vorherige Version, wird es an dessen Stelle in Betrieb genommen.

In Fällen, wo ein Training auf jüngeren Daten keine ausreichende Verbesserung bewirkt, ist eine vertiefte Analyse notwendig. Je häufiger ein solches Retraining nötig ist, desto mehr lohnt sich die Investition in die Automatisierung von Training, Evaluation und Deployment. In Anwendungsfällen wie unserem Beispiel, in dem vergleichsweise wenige KI-Entscheidungen getroffen werden und sich die eingehenden Daten nur langsam verändern, genügt es meistens, den Prozess manuell zu steuern.

Prüfen, ob die Entscheidungsfindung fachlich korrekt war

Anders als bei menschlichen Kollegen lässt sich von einem KI-System nicht ohne weiteres in Erfahrung bringen, warum eine Bewerbung es in die engere Auswahl geschafft hat. Auch fällt es schwer, einzuschätzen, ob die Vorauswahl vernünftig getroffen wurde.

In der Folge bleibt Anwendern und Endnutzenden oft nur die Wahl zwischen blindem Vertrauen oder pauschaler Ablehnung. Hier werden die vielfältigen Methoden der Explainable AI (XAI) relevant, die es ermöglichen, Erklärungen für das Gesamtverhalten und für einzelne Entscheidungen von KI-Systemen zu erzeugen.

Explainable AI: Was bedeutet hier „erklärbar"?

Die vom US-amerikanischen National Institute of Standards and Technology (NIST) im Herbst 2021 publizierten „Four Principles of Explainable AI" geben eine praxistaugliche Antwort auf diese Frage. Die vier Prinzipien fassen den aktuellen Stand in Forschung und Praxis zusammen und lassen sich als Leitfaden für die Entwicklung „erklärbarer" KI nutzen. Eine Erklärung ist demnach unterstützende Evidenz beziehungsweise eine Begründung zu einer spezifischen Ausgabe eines KI-Systems oder für seine Funktionsweise (Prinzip „Erklärung"). Diese noch recht abstrakte und akademische Formulierung macht zunächst vor allem deutlich, dass Erklärungen je nach Anwendungsszenario ganz unterschiedliche Formen annehmen können. Dabei sind laut NIST zwei Eigenschaften entscheidend: Erklärungen müssen für ihre Zielgruppe verständlich sein (Prinzip „Meaningful") und sie müssen die Gründe für eine Entscheidung beziehungsweise für die Abläufe innerhalb eines KI-Systems korrekt widerspiegeln (Prinzip „Explanation Accuracy").

Hier ist entscheidend, dass Letzteres allein nicht ausreicht: Eine technisch akkurate und zutreffende Erklärung bleibt wertlos, wenn ihre Empfängerinnen und Empfänger sie nicht verstehen und daher nicht die richtigen Rückschlüsse ziehen können. Schließlich nehmen die Fachleute des NIST noch eine weitere Eigenschaft von KI-Systemen auf, die auf den ersten Blick wenig mit Erklärbarkeit im engeren Sinne zu tun hat, jedoch im hier diskutierten Kontext risikobehafteter KI-Anwendungen zentral ist: Ein KI-System darf nur unter den Bedingungen arbeiten, für die es entwickelt wurde, und es darf nur dann eine Ausgabe nebst zugehöriger Erklärung produzieren, wenn es für sie ein ausreichendes Konfidenzniveau erreicht (Prinzip „Knowledge Limits“).

Was sind die Zielgruppen für Erklärungen?

Offensichtlich sind Erklärungen zielgruppenspezifisch zu gestalten. Dazu ist es hilfreich, mögliche Empfängerinnen und Empfänger anhand von zwei Merkmalen zu unterscheiden: Zum einen anhand ihrer technischen Kompetenz und ihres Verständnisses der eingesetzten KI-Methodik und zum anderen anhand ihres Fachwissens bezüglich des Anwendungsfalls. So sind die für Personalentscheidungen Zuständigen aus dem Beispiel bestens mit den Prozessen und Vorschriften des Personalbereichs vertraut, nennenswertes technisches Wissen können KI-Entwicklerinnen und -Entwickler jedoch nicht voraussetzen. Umgekehrt dürfte es den IT-Fachkräften gehen, die mit dem Betrieb des Bewerbungssystems betraut sind.

Die anderen Extrempunkte des gedachten Spektrums besetzen Laien und die KI-Entwicklungsteams. Erstere sind in unserem Beispiel die Bewerberinnen und Bewerber: Sie haben weder Kenntnis des Personalwesens noch kennen sie sich mit der eingesetzten Technologie aus. Diejenigen, die das Bewerbungssystem entwickeln, bringen dagegen ein solides Fachverständnis und hohe KI-Kompetenz mit. Zwei weitere, oft relevante Zielgruppen von Erklärungen sind diejenigen, die die von dem KI-System getroffenen Entscheidungen verantworten (etwa die Geschäftsführung oder eine Behördenleitung) sowie Mitarbeitende von Prüfinstituten und Aufsichtsbehörden. Schon dieser kurze Überblick macht deutlich, dass jede dieser Gruppen einen eigenen Informationsbedarf und unterschiedliche Anforderungen an Form, Granularität und Frequenz von Erklärungen hat – die eine, universell einsetzbare Methode gibt es daher nicht.

Wie lassen sich Erklärungen erzeugen?

Die Zahl und Vielfalt von Methoden zum Erzeugen von Erklärungen für KI-Entscheidungen ist ähnlich groß wie die der Modelltypen und -varianten im Machine Learning. Wie auch dort gilt es, jeweils passend zum Anwendungsfall eine Methode auszuwählen, die sowohl technisch als auch fachlich geeignet ist. Soll beispielsweise ermittelt werden, welche Merkmale der Eingabedaten eine KI-Entscheidung auf welche Weise beeinflusst haben, bieten sich sogenannte Feature-Importance-Methoden an. Sie sind für eine Vielzahl von Datentypen verfügbar und lassen sich sowohl modellagnostisch (also ohne Zugriff auf die inneren Komponenten eines KI-Systems) als auch unter Ausnutzung modellspezifischer Eigenschaften errechnen.

Mit Anchor-Methoden zum Kern einer KI-Entscheidung vordringen

Steht dagegen die Frage im Raum, welche Teile der Eingabedaten die Ausgabe eines KI-Systems abschließend

bestimmen, eignen sich Anchor-Methoden, die die wesentlichen Gründe einer KI-Entscheidung ermitteln. Ganz gleich, welche weiteren Faktoren möglicherweise noch hinzukommen oder entfallen: Der als Anchor erkannte Teil der Eingabedaten ist bereits ausreichend, um die KI-Entscheidung unumstößlich festzulegen. Eine Analogie aus dem Arbeitsalltag einer Personalabteilung ist beispielsweise eine Stelle, für die ein Studienabschluss oder eine bestimmte Staatsangehörigkeit zwingende Voraussetzung sind. Eine Bewerberin oder ein Bewerber mag noch so viele sonstige Qualifikationen und relevante Erfahrungen mitbringen, ohne die formalen Voraussetzungen bleibt es stets bei einem „Nein". Weitere populäre Methoden sind kontrafaktische Erklärungen, die eine Entscheidung mithilfe eines passend gewählten Gegenbeispiels verständlich machen, sowie Methoden, die die von einem Modell angewandten Entscheidungsregeln extrahieren und in verständlicher Form aufbereiten. Abbildung 4 zeigt beispielhaft, wie diese unterschiedlichen Methoden im Beispiel zur Anwendung kommen könnten.

KI-Systeme kontrollierbar und transparent machen – das geht

Die in diesem Artikel vorgestellten Konzepte und Ansätze zeigen, dass KI-Systeme keineswegs unkontrollierbar und intransparent sein müssen. Nicht nur aufgrund rechtlicher Bedenken, sondern auch im Interesse unternehmerisch guter Entscheidungen sollten Unternehmen von Beginn an bedarfsorientiert und schrittweise daran arbeiten, ihre KI-Systeme und MLOps-Infrastruktur entsprechend aufzubauen und abzusichern.

Eine technisch sauber umgesetzte MLOps-Infrastruktur hilft, um rechtliche Compliance zu gewährleisten und stellt auch die Erklärbarkeit von KI (XAI) sicher. Dann wird auch die künftig vonseiten des EU-Rechts zu erwartende verschärfte Regulatorik zu automatisierten Entscheidungssystemen nicht zum unüberwindbaren Hindernis für den breiten Einsatz von KI. (sih)

Quellen

Weiterführende Links mit Ressourcen zur KI-Sicherheit und zur Model Governance finden sich unter ix.de/zxaa.

Isabel Bär

studiert Data Engineering am Hasso-Plattner-Institut und arbeitet als Werkstudentin bei INNOQ. Sie beschäftigt sich mit Fragen rund um den langfristig erfolgreichen Einsatz von Künstlicher Intelligenz (KI), wozu insbesondere MLOps und das Implementieren von Model Governance gehören.

Kilian Kluge

arbeitet als Co-Gründer von Inlinity daran, mit Explainable AI Anwendungsbereiche für KI-Systeme zu erschließen, in denen bislang regulatorische oder unternehmerische Risiken einem Einsatz entgegenstehen. Zuvor war er mehrere Jahre als IT-Berater und Entwickler in der deutschen Finanzbranche tätig und hat an der Universität Ulm zu nutzerzentrierter KI promoviert.

Wie Big Data, Data Science & Machine Learning die Automobilbranche revolutionieren

Wir produzieren eine enorme Menge an Daten, deren Potenzial in vielen Bereichen bisher nicht ausreichend genutzt wird – so auch in der Automobilbranche. Durch die Analyse und Verarbeitung solcher Daten wird es ermöglicht Prozesse zu optimieren und den effizienteren Einsatz von Ressourcen zu erhöhen.

Im Unternehmenskontext bedeutet das, dass Data Science Probleme lösen kann, die bisher nur durch aufwendige, manuelle Arbeit gelöst werden konnten, aber auch, dass maschinelle Zusammenhänge erkannt werden können, die ein Mensch eventuell gar nicht erkennen kann. Data Science birgt viele Potentiale in Bezug auf Effizienzsteigerung, Prozessoptimierung und Kundenorientierung. Dennoch haben Algorithmen und traditionelle Computersysteme Grenzen und brauchen (aus aktueller Sicht) menschliche Bedienung, um effektiv genutzt zu werden. Bei Bosch in Wien beschäftigt sich die Expertin Alina Godun damit.

Die Welt besteht aus Daten

„Ich finde es faszinierend, wie gut selbst unstrukturierte Daten die Welt beschreiben und wie Data Science bei dem Verständnis der Daten und damit auch der Welt unterstützen kann", so Godun. „Die Welt scheint auf den ersten Blick sehr chaotisch und unvorhersehbar. Beginnt man jedoch diese Daten zu analysieren, erkennt man, dass hinter diesen großen Datenmengen Regelmäßigkeiten stecken. Ich spreche in diesem Zusammenhang gerne von versteckten Mustern."

Bosch beschäftigt sich sehr intensiv mit dieser Datenwissenschaft, wobei unterschiedliche Teilbereiche wie Machine Learning und statistische Datenanalyse angewendet werden. „Ein Beispiel ist mein aktuelles Projekt", so die Data Science Expertin. „Wir verwenden Signale von unterschiedlichen Sensoren und Aktuatoren, die durch Autotestfahrten bzw. Simulationen gesammelt wurden. Solche Signaldaten zu sammeln, aufzubereiten und dann in die Datenbank zu bringen, ist eine nicht zu unterschätzende Aufgabe – das ist das Data Engineering. Die Daten verwenden wir dann später, um unterschiedlichste Aufgaben zu lösen: von der Früherkennung und Prävention von Fehlern in der Software bis zu komplexen Vorhersagen zum Verhalten von Komponenten im realen Umfeld. Diese Analysen helfen wiederum, die existierenden Produkte und Lösungen zu optimieren und neue Produkte zu entwickeln, die früher gar nicht lösbar waren."

Neuronale Netze in der Mobilität

Eine der Aufgaben von Alina ist es, die in Testautos gesammelten Daten mithilfe von explorativer Datenanalyse zu bewerten. Hierbei werden verschiedene Signale zusammengeführt und visualisiert, was es ermöglicht, Zusammenhänge und wechselseitige Einflüsse zu verstehen. Visualisierungen haben den Vorteil, dass unerwartete oder abweichende Signaldaten gut sichtbar sind, und sie ermöglichen, latente Probleme in der Software zu entdecken. Die Normsignale wiederum fließen in ein neuronales Netz als Trainingsdaten und dienen dazu, das neuronale Netz empfindlicher auf Abweichungen zu machen. Dabei simuliert das KI-Modell das „ideale" Verhalten einer Software und gleicht das mit dem realen Verhalten der Software anhand der Daten ab – Abweichungen deuten auf Fehler hin. Dieses Vorgehen hilft dabei, die Qualität der Softwarelösungen enorm zu verbessern und den Kundennutzen zu maximieren.

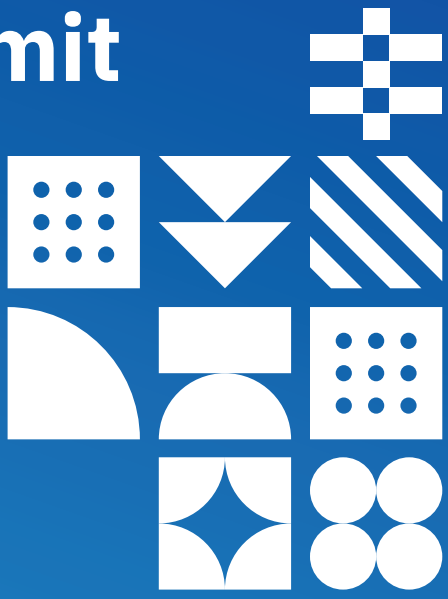
Mehr Infos zu unseren Jobs auf www.bosch.at/karriere und auf Social Media (Facebook/ LinkedIn: Bosch Österreich oder auf Instagram: @BoschOesterreich)

FACTBOX:
Bosch beschäftigt in Österreich rund 3 000 Mitarbeiter*innen. An den Standorten Wien, Linz und Hallein betreibt Bosch internationale Entwicklungs-Kompetenzzentren der Mobilitätstechnik – einschließlich Internet of Things-Lösungen im Bereich der vernetzten Mobilität. Techniktalente finden bei Bosch in Österreich ein inspirierendes Arbeitsumfeld, um Zukunftsthemen aktiv voranzutreiben.

> Perfekter Mix: ML mit Kotlin und Python

Hauke Brammer

Im Machine Learning dominiert Python, doch auch Kotlin ist für einige ML-Bereiche besonders geeignet. Gemischte Architekturen nutzen die Stärken beider Sprachen.



Im Machine Learning ist Python als Programmiersprache absolut dominant – dennoch bietet es sich an, auch hier und da über den Tellerrand zu schauen. In den letzten Jahren hat sich rund um Kotlin ein interessantes und lebhaftes Ökosystem im Bereich Data Science und Machine Learning (ML) entwickelt. Zeit für eine Bestandsaufnahme der Möglichkeiten zur Verwendung von Kotlin für drei Bereiche des Machine-Learning-Lifecycles: die Datenerkundung in Jupyter, Modell-Architekturen und das Model Serving.

In a Nutshell

- Durch ein großes Ökosystem und flexible Einsatzmöglichkeiten hat sich Python eine dominante Stellung im Machine Learning erarbeitet.
- Für den erfolgreichen Produktivbetrieb eines Machine-Learning-Systems werden nicht nur das Modell selbst, sondern viele weitere Komponenten benötigt. Hier besteht durch die Verwendung modularer Architekturen die Möglichkeit, verschiedene Programmiersprachen einzusetzen.
- Kotlin wird durch Typsicherheit, Null-Safety und bessere Performance zur optimalen Ergänzung von Python. Gerade für langlebigere und stabilere Teile des Codes ist Kotlin ideal, während der experimentelle Teil in Python ausgeführt werden kann.

Bevor es ans Erkunden der Möglichkeiten und Einsatzgebiete für Kotlin im Machine Learning geht, gilt es, einen kurzen Exkurs über Machine Learning im Allgemeinen und speziell im Business Context zu unternehmen. Die Ausgangsfrage dabei ist: „Was wollen wir mit Machine Learning erreichen?“ Das Ziel ist, ein Modell mit Daten zu füttern, damit es bestimmte Vorhersagen über die Welt treffen kann. In einer Hinsicht unterscheidet sich Machine Learning im Geschäftskontext von Machine Learning zu Forschungszwecken: Ein ML-Modell allein ist nutzlos. Das ist zunächst eine überzeichnete Aussage, die aber dennoch einen realen und etwas beunruhigenden Kern hat: Etwa 90 Prozent der ML-Modelle schaffen es nicht in die Produktion. Für kommerzielle Zwecke sind die meisten ML-Projekte offensichtlich wenig brauchbar (die Quelle hierzu und weiterführende Informationen finden sich unter ix.de/zvgb).

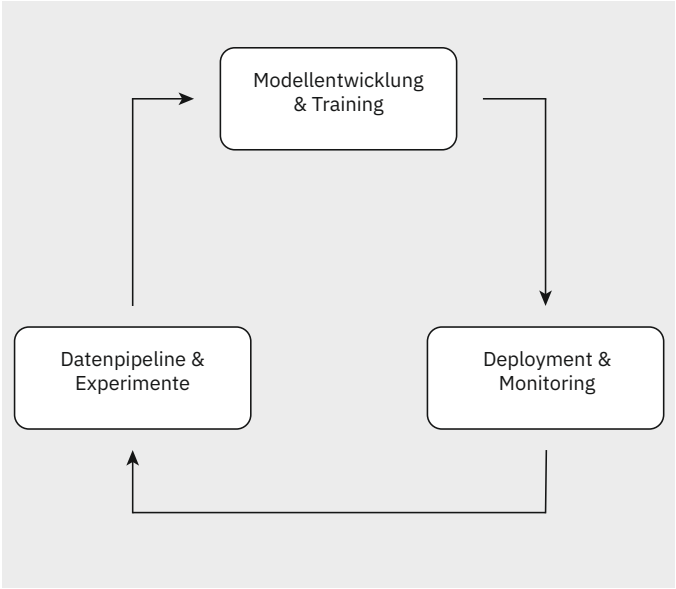
Zwar werden einige Data-Scientists spannende Erkenntnisse gewonnen haben und auch die Marketing-Abteilung freut sich, da sie ja auch „etwas mit KI“ machen. Messbarer Mehrwert entsteht jedoch nur durch ML-Modelle, die sich in konkreten Anwendungen zur Verfügung stellen lassen. Nur in einer produktiven Umgebung ist ein Modell nützlich. Das bedeutet aber, dass wir eigentlich nicht nur Modelle, sondern ganze „Machine-Learning-Systeme“ bauen wollen (siehe Abbildung 2). Also nicht nur ein einzelnes Artefakt, sondern eine ganze Reihe von Komponenten, die es ermöglichen, zu experimentieren, zusammenzuarbeiten und kontinuierlich neue Versionen der Modelle zu erstellen und diese dann in produktive Umgebungen zu heben. Dort angekommen müssen Unternehmen sich Gedanken über den Betrieb und die Überwachung des Modells und vor allem über ein potenzielles Nachfolgemodell machen.

Mehr als nur ein Modell

Viele unterschiedliche und komplexe Teile müssen für den erfolgreichen Produktivbetrieb ineinandergreifen, und im Kern steht das Modell selbst. Aber damit ein Team es ausliefern kann, muss es das Modell in einen Webservice oder in eine App einbetten (Abbildung 2, 1). Anschließend gehen von den Nutzern oder von den Umsystemen Daten ein und darauf aufbauend lassen sich Vorhersagen ausliefern (Abbildung 2, 2). Die Qualität eines Machine-Learning-Modells verschlechtert sich aufgrund sich verändernder äußerer Bedingungen und Input-Daten mit der Zeit. Daher genügt es nicht, nur ein einziges Modell zu trainieren, das für lange Zeit in Verwendung bleibt, sondern es gilt, eine Infrastruktur zu schaffen, um kontinuierlich neue Modelle zu trainieren.

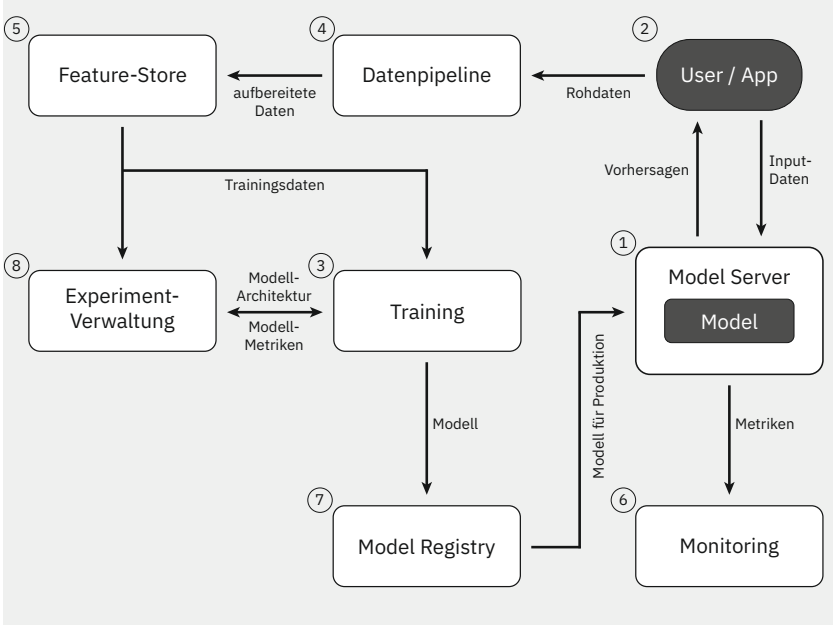
Gerade das Entwickeln großer oder tiefer Deep-Learning-Modelle findet in der Regel nicht auf dem Laptop statt, sondern in einer Cloud-Umgebung oder auf einem GPU-Cluster im Rechenzentrum, wofür eine kontinuierliche Trainings-Pipeline nötig ist (Abbildung 2, 3). Die Daten für das Training stammen aus den Produktivsystemen – allerdings sind sie erst aufzubereiten, zu filtern und zu labeln. Dafür ist eine Data- oder ETL-Pipeline nötig (Abbildung 2, 4). Wer in größeren Teams oder über mehrere Teams hinweg arbeitet, sollte die aufbereiteten Daten den anderen Teammitgliedern ebenfalls zur Verfügung stellen. Dafür ist bei kleinen Datenmengen noch ein Versionskontrollsystem ausreichend, bei größeren Projekten kommt ein Feature-Store zum Einsatz

Quelle: Hauke Brammer



>> Machine Learning: Lifecycle (Abb. 1)

(Abbildung 2, 5). Wer kontinuierlich und mit immer neuen Daten weiter trainiert, kann auch laufend neue Modelle in Produktion bringen. Diese sind allerdings zu erfassen und zu versionieren, um einen einfachen Wechsel zwischen den Modellversionen zu ermöglichen. Würde man etwa erst in



Quelle: Hauke Brammer

>> Machine-Learning-System: Für den erfolgreichen Produktivbetrieb müssen komplexte Teile ineinandergreifen, im Kern steht das Modell (Abb. 2).

Produktion im Monitoringsystem (Abbildung 2, 6) feststellen, dass ein neues Modell schlechte Ergebnisse liefert, wäre das fatal. Kein Team kann es sich erlauben, darauf zu warten, bis die Vorgängerversion des Modells noch einmal neu trainiert wurde, während das Produktionsmodell falsche Daten liefert.

Richtig versioniert

Je nach Größe stehen für die Versionierung geeignete Versionskontrollsysteme wie Git oder DVC bereit, auch eine zentrale Model Registry ist eine Option (Abbildung 2, 7). Zur kontinuierlichen Weiterentwicklung von ML-Modellen gehört es, Experimente durchzuführen: Versuche, mit einer spezifischen Modell- und Algorithmus-Konfiguration und einem Datensatz bestimmte Ergebnisse zu erzielen. Nur wenige dieser Experimente werden es in eine produktive Umgebung schaffen. Dennoch ist die Versionierung notwendig, um sich einen Überblick darüber zu verschaffen, welche Experimente man bereits durchgeführt hat und was die Ergebnisse waren, um sie in Zukunft reproduzieren und weiterverwenden zu können.

Dafür ist ein Tool erforderlich, um die Experimente zu tracken und zu verwalten (Abbildung 2, 8). Es verhindert, dass die Data Scientists die gleichen Experimente mehrfach durchführen oder Ergebnisse verloren gehen. Für erfolgreiche und produktive Machine-Learning-Projekte ist ein Satz an Tools nötig, den man sich im Verlauf eines Projekts langsam aufbaut. Gerade beim Zusammenspiel von Kotlin und Machine Learning ist wichtig zu bedenken, dass es am Ende nicht nur darum geht, mit TensorFlow ein Deep-Learning-Modell zusammenzustecken: Es geht darum, wie sich verlässliche, reproduzierbare und skalierbare Machine-Learning-Systeme bauen und betreiben lassen.

Listing 1: Docker-Image für Jupyter mit Kotlin-Kernel

```
FROM jupyter/base-notebook

USER root

# Install OpenJDK-8
RUN apt-get update && \
    apt-get install -y openjdk-8-jre && \
    apt-get clean;


# Setup JAVA_HOME for docker commandline
ENV JAVA_HOME /usr/lib/jvm/java-8-openjdk-amd64/
RUN export JAVA_HOME

# Switch back to unprivileged user
USER $NB_UID

ENV JUPYTER_ENABLE_LAB=yes

# Install kotlin kernel
RUN conda install -c jetbrains kotlin-jupyter-kernel=0.11.0.95
```

Mit Kotlin in Jupyter arbeiten

Am Anfang jedes Machine-Learning-Projekts steht eine umfangreiche Datenanalyse. Anschließend beginnt das Design erster Modell-Architekturen und Experimente lassen sich durchführen. Für dieses iterative Vorgehen sind Jupyter-Notebooks ideal: In ihnen können Teams Daten, Notizen, Code und Visualisierungen in einem gemeinsamen Dokument darstellen und der Code lässt sich direkt und interaktiv ausführen. Das Ausführen von Python-Code ist Teil der Standard-Installation von Jupyter. Dank dessen modularer Architektur ist durch das Installieren zusätzlicher Kernel aber auch das

Listing 2: Datenanalyse mit Kotlin dataframe

```
// Welche Passagiere heissen 'Jones'?
df.filter{name.contains("Jones")}


// Wieviele Passagiere sind wo zugestiegen?
df.groupBy { embarked }
    .aggregate {
        count() into "count"
    }

// Fehlende (Null) Daten fuer den Ticket-Preis und das Alter
// auffuellen und dann sortieren
df.fillNulls{ fare and age }
.perCol{ mean() }
.sortBy { fare.desc() }
```

Verwenden anderer Sprachen in den Jupyter-Notebooks möglich. So gibt es neben Kernels für Julia, Go, Rust und Fortran auch einen für Kotlin. Mit dem Befehl `conda install -c jetbrains kotlin-jupyter-kernel` lässt der Kotlin-Kernel sich in einer laufenden Jupyter-Installation installieren.

Eine alternative Möglichkeit ist das Erstellen eines Jupyter-Docker-Images mit Kotlin-Kernel, wie Listing 1 zeigt.

Nun lässt sich Kotlin im Jupyter-Notebook verwenden und mit dem `%use`-Keyword steht eine Reihe vorinstallierter Libraries bereit (die Liste findet sich unter ix.de/zvgb). Neue Libraries lassen sich mit dem Befehl `@file:DependsOn(LIBRARY)` herunterladen und installieren.

Arbeiten mit Tabellen und Dataframe

Die Library `dataframe` gehört zu den bereits vorinstallierten Libraries. Sie ermöglicht das Arbeiten auf strukturierten Daten, wie es aus der Tabellenkalkulationen und von Datenbanken her vertraut ist. Daten wie in diesem Beispiel die Passagierdaten der Titanic lassen sich mit einem einfachen `df = DataFrame.read("titanic.csv")` aus einer CSV-Datei laden. Andere Dateiformate wie Excel und JSON lassen sich ebenso einlesen, und es ist auch möglich, hierarchische Strukturen abzubilden, also Zellen und Spalten ineinander zu verschachteln. Während die Daten in einer CSV-Datei zunächst untypisiert sind, weil sie als reine Textdaten vorliegen, werden sie beim Laden in einen Dataframe on-the-fly typisiert.

Listing 3: Plot-Grafiken mit Lets-Plot

```
// Scatter Plot
letsPlot(data) {
    x = "size";
    y = "price"
} +
geomPoint(size=5) +
geomSmooth() +
ggsize(500, 250)


// Dichte Plot mit Kategorien
letsPlot(moreData) +
    geom_density(alpha=.3) {
        x="rating";
        fill="category"
    }

// Linienplot mit Fehlerbalken
letsPlot(study_data) {
    x="learning";
    color="group"
} +
geomErrorBar(width=.1) {
    ymin="points_min";
    ymax="points_max"
} +
geomLine {y="points"} +
geomPoint {y="points"}
```

Dieses automatisch generierte Schema lässt sich mit `df.schema()` abfragen. Den Dataframe können Teams bereits dazu verwenden, um ihre Daten zu erkunden. Dazu bietet die `dataframe`-Library eine mächtige, aber dennoch einfach verständliche API, die in Teilen an bekannte Befehle aus Datenbanken angelehnt ist. Der Befehl `df.select{name and age}.take(10)` fragt die Spalten „name“ und „age“ ab und gibt dabei mit `.take(10)` nur die ersten zehn Zeilen aus. Jede Operation erzeugt einen neuen Dataframe und gibt ihn zurück. Der Dataframe selbst ist hingegen unveränderlich (immutable), um ein unbeabsichtigtes Verändern der Daten zu verhindern. Listing 2 führt weitere Operationen auf dem Dataframe auf.

`dataframe` ist ein nützliches Werkzeug zum Erkunden, Analysieren und Säubern von Daten in Jupyter-Notebooks oder auch in regulärem Kotlin-Code.

Schöne Grafiken mit Lets-Plot

Viele Zusammenhänge in Daten lassen sich in einer tabellarischen Darstellung gut erkennen. Für manche Trends und Muster gibt es jedoch schnellere und einfachere Wege der Datenvisualisierung wie `Lets-Plot`. Diese Plotting-Library für Kotlin verwendet einen Schichten-Ansatz, um Plot-Grafiken zu definieren. Mit jeder hinzugefügten Schicht wird ein anderer Aspekt der Grafik definiert: die verwendeten Daten, die Auswahl der Art des Plots oder die Farben und Formen im Plot. Um beispielsweise für die Darstellung des Zusammenhangs zwischen Wohnfläche und Preisen von Häusern einen Punkte-Plot zu erstellen, lässt sich folgender Befehl verwenden: `letsPlot(data) {x = „size“; y = „price“} + geomPoint(size=5) + geomSmooth() + ggsize(500, 250)`.

Dabei gilt es, zunächst die Daten festzulegen und die verwendeten x- und y-Achsen zu definieren. Mit `geomPoint` legen wir den Typ des Plots als Punkte-Plot fest, der die Punkte mit einer Größe von 5 Pixel plottet. Um einen geglätteten Mittelwert mit Abweichungen einzuzeichnen, ist noch die `geomSmooth`-Schicht hinzuzufügen. Als Letztes ist die Größe des Plots mit `ggsize(500,250)` festzulegen. `Lets-Plot` bietet eine Vielzahl an Plot-Typen und Konfigurationsoptionen. Listing 3 führt einige Beispiele auf. Die Integration mit der `dataframe`-Library ermöglicht eine einfache Verwendung in Data-Exploration-Workflows oder in anderen Kotlin-Projekten.

Als zweiten Aspekt nach dem Erkunden von Daten geht es nun um das Erstellen von Modellarchitekturen und das Training von Modellen mit Kotlin. Die Voraussetzung dafür ist KotlinDL in der aktuellen Version 0.4.0. Die Codebasis ist Open Source und auf GitHub verfügbar (unter ix.de/zvgb findet sich der Link). KotlinDL beschreibt sich selbst, als High-Level-Deep-Learning-API, ist in Kotlin geschrieben und stark von der Python-Library Keras inspiriert. Unter der Haube verwendet es die TensorFlow Java-API und die ONNX-Runtime-API für Java. Dadurch ist auch ein einfaches Training auf Grafikkarten möglich, allerdings derzeit nur auf NVIDIA-Modellen. KotlinDL bietet einfache APIs für das Training von Deep-Learning-Modellen von Grund auf. Auch der Import bereits trainierter KotlinDL-, Keras- und ONNX-Modelle für die Inferenz ist möglich. Importierte und vortrainierte Modelle lassen sich mit Transfer-Learning an eigene Aufgaben anpassen.

Listing 4: KotlinDL: Ein einfaches Klassifizierungsmodell

```
import org.jetbrains.kotlinx.dl.api.core.layer.reshaping.Flatten
import org.jetbrains.kotlinx.dl.api.core.optimizer.Adam
import java.io.File

// Laden der Trainings- und Testdaten
val (train, test) = fashionMnist()

// Model Architektur
val net = Sequential.of(
    Input(28L,28L,1L),
    Flatten(),
    Dense(300),
    Dense(100),
    Dense(10)
)

with(net){ // this: net
    // Einstellungen fuer Training festlegen
    compile(
        optimizer = Adam(),
        loss = Losses.SOFT_MAX_CROSS_ENTROPY_WITH_LOGITS,
        metric = Metrics.ACCURACY
    )

    printSummary()

    // Training des Modells
    fit(
        dataset = train,
        epochs = 5,
        batchSize = 100
    )

    // Evaluierung des Modells mit den Testdatensatz
    val accuracy = evaluate(dataset = test, batchSize = 100)
        .metrics[Metrics.ACCURACY]

    println("Accuracy: $accuracy")

}
```

Modelltraining mit Kotlin

Das folgende Beispiel trainiert mit KotlinDL ein einfaches Modell, das Bilder in Kategorien einteilt. Listing 4 stellt den fertigen Code dar. KotlinDL liefert auch einige Datensätze mit, so auch den `Fashion MNIST`-Datensatz von Zalando. Er enthält 70.000 Bilder von Kleidungsstücken aus zehn Kategorien. Über die Hilfsfunktion `val (train, test) = fashionMnist()` lässt er sich importieren und direkt in 60.000 Trainingsbeispiele sowie 10.000 Testbeispiele aufteilen. Als Nächstes gilt es, die Architektur des neuronalen Netzes festzulegen. Hier bietet sich sequenzielle Architektur an, bei der die Daten alle Schichten der Reihe nach durchlaufen. Die Input-Daten haben eine Bildgröße von 28 mal 28 Pixeln: Das erfordert einen Input-Layer entsprechender Größe, festzulegen mit `Input(28L, 28L, 1L)`. Die zweidimensionalen Bilddaten lassen sich mit einem `Flatten()`-Layer in einen eindimensionalen Vektor der Länge 784 umwandeln, damit die folgenden Layer die Daten verarbeiten können.

Es folgen zwei vollverknüpfte `Dense`-Layer, in denen die eigentliche Lernleistung des Modells stattfindet. Am Ende steht noch ein Output-Layer (ebenfalls `Dense`), der zehn Outputs

hat, was der Zahl der zu unterscheidenden Kategorien entspricht. Damit ist die einfache Modellarchitektur fertig. Im nächsten Schritt gilt es, die Einstellungen zum Training des Modells festzulegen. Als Optimierungsalgorithmus steht der `Adam`-Optimizer bereit und die `loss function` ist auf `SOFT_MAX_CROSS_ENTROPY_WITH_LOGITS` festzulegen. Die Metrik, die der Algorithmus optimieren soll, lautet `metric = Metrics.ACCURACY`. Insgesamt sind das gewöhnliche Werte, mit denen in der ersten Iteration eines Klassifizierungsmodells nichts schiefgehen kann. Beim Training eines echten Machine-Learning-Modells wären jedoch genau auf die Aufgabe abgestimmte Parameter auszuwählen.

Aufruf der fit-Funktion

Das eigentliche Training findet mit dem Aufruf der `fit`-Funktion statt. Sie nimmt den Trainingsdatensatz zusammen mit der Anzahl der Iterationen über den Trainingsdatensatz (`epochs`) entgegen. Mit `batchSize` lässt sich festlegen, wie viele Beispiele auf einmal verwendbar sind, um die Parameter des Modells zu aktualisieren. Nach dem Training des Modells können Entwicklerinnen und Entwickler mit dem Testdatensatz und der `evaluate`-Funktion die Qualität der Modellvorhersage bestimmen. Wie das Beispiel verdeutlich, lässt sich ein Machine-Learning-Modell bereits mit wenigen Zeilen Kotlin-Code trainieren und evaluieren. Dabei ist eine Vorhersagegenauigkeit von mehr als 85 Prozent erreichbar. In einem realen Projekt wäre das jedoch nur der erste Schritt in einer ganzen Reihe von Experimenten, um die optimale Vorhersagequalität eines Modells zu bestimmen.

Model Serving mit dem Framework Ktor

Das trainierte Modell muss man in einen Webservice oder eine App verpacken, damit Nutzer damit interagieren können und es auf der Basis von Daten Vorhersagen treffen kann. Mit KotlinDL und dem Framework Ktor lässt sich in wenigen Zeilen ein ML-Modell-Server erstellen (hierzu mehr unter ix.de/zvgb). Ktor ist ein Kotlin-Framework auf Basis von Kotlin-Coroutines, das es ermöglicht, schlanke Webapplikationen und APIs zu bauen. Im Beispielprojekt (Listing 5) geht es darum, ein Objekterkennungsmodell über eine REST API zur Verfügung zu stellen.

Als Grundlage dient ein bereits fertig trainiertes Modell, das mithilfe von KotlinDL aus dem ONNXModelHub (1) zu laden ist. Einen Ktor-Server zu starten, ist kein Hexenwerk: Alles, was man dafür benötigt, ist etwas Code (2) aus Listing 6. In diesem Server-Grundgerüst lassen sich verschiedene REST-API-Endpunkte im Bereich `routing` festlegen. Hier gilt es, eine `Route` mit der `post`-Funktion zu definieren (3).

Listing 5: Model Serving mit Ktor und KotlinDL

```
package modelserver

import io.ktor.http.*
import io.ktor.http.content.*
import io.ktor.serialization.jackson.*
import io.ktor.server.application.*
import io.ktor.server.routing.*
import io.ktor.server.engine.*
import io.ktor.server.netty.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import org.jetbrains.kotlinx.dl.api.inference.loaders.ONNXModelHub
import org.jetbrains.kotlinx.dl.api.inference.onnx.ONNXModels
import java.io.File
import java.util.*

fun main() {
  val modelHub =
    ONNXModelHub(cacheDirectory = File("cache/pretrainedModels"))
  val model =
    ONNXModels.ObjectDetection.SSD.pretrainedModel(modelHub) //(1)

  embeddedServer(Netty, port = 8089) { // (2)
    install(ContentNegotiation) {
      jackson()
    }
    routing {
      post("/detect") { // (3)
        // Bilddatei empfangen und speichern (4)
        val multipartData = call.receiveMultipart()
        lateinit var imageFile: File
        val newFileName = UUID.randomUUID()
        multipartData.forEachPart { part ->
          when (part) {
            is PartData.FileItem -> {
              val fileName = part.originalFileName as String
              val fileBytes = part.streamProvider().readBytes()
              call.application.environment.log.info(
                "Received file: \"$fileName\"")
              imageFile=File("uploads/$newFileName")
              imageFile.writeBytes(fileBytes)
            }
            else -> TODO()
          }
        }
        // Inferenz: Objekte erkennen (5)
        val detectedObjects =
        model.detectObjects(imageFile = imageFile, topK = 3)
        call.application.environment.log.info(
          "Found: $detectedObjects")
        // Erkannte Objekte zurueckgeben // (6)
        call.respond(detectedObjects)
      }
    }
  }.start(wait = true)
}
```

Die Funktion kann Bilddaten entgegennehmen, sie verarbeiten und zur Inferenz an das Modell übergeben (4). Der Versand der Bilddaten erfolgt über Multipart-Formdata. Dazu iteriert der Code über die gesamten Multipart-Daten und überprüft, ob das entsprechende Feld ein `FileItem` enthält. Ist das der Fall, ist der Dateiname auszulesen und für ein späteres Debugging zu loggen.

Die eigentlichen Daten sind mit `streamProvider().readBytes()` auszulesen und in einer neuen Datei zu speichern, die eine eindeutige ID benötigt. Im Beispiel ist dafür `UUID.randomUUID()` als ID zuzuweisen. Sind die Bilddaten aus der Anfrage eingelesen, kann das Modell eine Objekterkennung durchführen (5): Mit `model.detectObjects(imageFile = imageFile, topK = 3)` erhält es die gespeicherte Bilddatei zur Erkennung. Dabei legen die Zuständigen mit `topK = 3` fest, dass sie nur an den drei wahrscheinlichsten Ergebnissen interessiert sind. Die erkannten Objekte lassen sich mit `call.respond(detectedObjects)` zurückgeben (6). Auf diese Weise entsteht in nur wenigen Zeilen Code mit Ktor und KotlinDL ein einfacher Modellserver. Die Grundlage kann jedes fertig trainierte Keras- oder ONNX-Modell sein. Für den produktiven Betrieb fehlen noch einige Schritte wie das Erfassen essenzieller Metriken, die Dauer der Inferenz und die Vorhersagequalität. Auch würde das Modell in einem produktionsreifen Modellserver nicht dynamisch vom Modellhub geladen, sondern aus einem statischen Ordner gelesen, in dem es seit dem Build-Prozess liegt.

Listing 6: Den Ktor-Server starten

```
fun main() {
    embeddedServer(Netty, port = 8089) {
        //...
    }.start(wait=true)
}
```

Pythons Dominanz im Machine Learning

Zurzeit ist Python die dominierende Sprache im Machine Learning. Die Frage, warum Entwickler und ML-Ingenieurinnen sich mit anderen Programmiersprachen beschäftigen sollten und wofür sich das lohnt, steht im Raum. Dabei ist es hilfreich, die Frage einmal umzudrehen: Warum ist Python so beliebt für Machine Learning? Zum einen ist Python eine großartige Programmiersprache und hat seit ihrem Erscheinen im Jahr 1991 eine rasante Entwicklung durchgemacht. Vor allem ist rund um Python ein ausgereiftes Ökosystem entstanden – von Bibliotheken für wissenschaftliche Anwendungen über Vektor- und Matrix-Mathematik bis hin zum Data Processing.

Da Machine Learning neben dem tatsächlichen Einsatz für Geschäftsanwendungen zunächst auch ein umfangreiches Forschungsfeld ist, sind viele der Bibliotheken in einer Sprache geschrieben, die den Entwicklern dieser Bibliotheken aus dem wissenschaftlichen Umfeld vertraut war. So kam es zu einem sich selbst verstärkenden Zyklus. Es gibt jedoch auch noch einen anderen Grund für die Beliebtheit von Python im Machine Learning und in der Data Science:

Python ist eine großartige Skriptsprache. Python-Code sieht fast aus wie Pseudo-Code, aber er funktioniert. Bei der Diskussion rund um das Für und Wider verschiedener Programmiersprachen darf aber nicht aus dem Blick geraten, dass viele Data Scientists keine Softwareingenieure sind. Im Job eines Data Scientists geht es nicht darum, eine Low-Level-Programmiersprache zu lernen, um optimalen, auf Performance getrimmten Code zu schreiben. Wenn es darum geht, ein kleines Skript zu bauen, um Daten zu analysieren, ist der Anspruch ein anderer. Ein Data Scientist muss tiefes Wissen in Statistik haben und Algorithmen entwickeln können: Dafür ist Python ideal. Es ist einfach zu nutzen und lässt sich zudem einsetzen, um auf High-Performance-Bibliotheken zuzugreifen, die unter anderem in C geschrieben sind wie NumPy und TensorFlow.

> **Python ist stark typisiert, aber auch dynamisch. Im Gegensatz dazu ist Kotlin statisch typisiert.**

Gründe, sich mit Kotlin für ML zu befassen

Es gibt jedoch andere Gründe, sich mit Kotlin für ML zu beschäftigen. Das Deployment in die Produktion ist nicht der Abschluss eines Machine-Learning-Projekts, sondern lediglich eine weitere Phase. Die meiste Zeit verbringen Machine Learning Engineers mit der Maintenance und dem Erweitern von Komponenten. Aber je mehr Code entsteht, desto unübersichtlicher wird das Ganze. Wenn die Projekte größer werden, kann ein starkes Typsystem helfen, den Überblick zu behalten. Python ist stark typisiert, aber auch dynamisch. Das hilft bei flexiblen ad-hoc-Analysen. Mit Python lassen sich ohne großen Aufwand Datenstrukturen zusammenbauen, die sogar noch während der Laufzeit definierbar sind. Im Gegensatz dazu ist Kotlin statisch typisiert. Typen werden zur Compilezeit an Typen gebunden. Das bedeutet, dass viele Typfehler, die in Python zu einem Runtime Error führen könnten, in Kotlin bereits zur Compilezeit auffallen. Das hilft insbesondere Teams, in denen viele gemeinsam an großen Projekten arbeiten.

Ein weiteres Problem in Bezug auf die Wartung und Skalierbarkeit von Projekten ist die Reproduzierbarkeit von Builds und eine vernünftige Verwaltung der Dependencies. In Python sind Builds und Dependencies stark von der Laufzeitumgebung des Codes beeinflusst. Das macht es schwierig, eine Reproduzierbarkeit herzustellen, wenn man nicht das ganze Environment in Docker verpacken will. Mit Kotlin ist es etwas einfacher, stabile Dependency Trees zu bauen. Was Kotlin Python außerdem voraus hat: Bewegt man sich in Python außerhalb der gekapselten C-Libraries, wird Python ausgesprochen langsam. Im Vergleich hat Kotlin deutlich die Nase vorn.

Kotlin und Python: Machine Learner können Stärken geschickt kombinieren

Wie lässt sich Kotlin ganz konkret für das Machine Learning nutzen? Kotlin-Liebhaber wie der Verfasser dieses Artikels würden in einer idealen Welt Kotlin für jeden Schritt in ihren ML-Projekten einsetzen. Die Realität ist differenzierter: Im Bereich Daten-Pipelines und Datenaufbereitung lässt sich Kotlin hervorragend einsetzen. Null-Safety, Typsicherheit und eine Einbindung des unendlichen Java-Universums machen Kotlin hier zu einer praxisnahen und empfehlenswerten Wahl. Im Bereich des Modell-Buildings und der Experimente mit Modellarchitekturen ist es zurzeit ratsam, weiterhin auf Python zu setzen. Hier gibt es zwar mit KotlinDL gute Ansätze, aber Python ist in diesem Umfeld flexibler, und die vorhandenen Bibliotheken sind deutlich umfangreicher. Zudem weisen sie einen höheren Reifegrad auf.

ML-System aus mehreren Komponenten bauen

Im Bereich des Model Serving, also im Produktivbetrieb von Machine-Learning-Modellen, gilt zurzeit eine klare Empfehlung für Python. Die vom Verfasser mit Python gebauten und trainierten Modelle exportiert er als ONNX-Modelle, die er ohne Probleme mit KotlinDL und Ktor als API oder Webservice zur Verfügung stellen kann. Zum jetzigen Zeitpunkt läuft es darauf hinaus, ein ML-System aus mehreren Komponenten zusammenzubauen, die sowohl mit Python als auch mit Kotlin gebaut werden. Durch die Vielzahl an Komponenten, die für erfolgreiche Machine-Learning-Projekte nötig sind, und durch die Wahl modularer Architekturen ist es möglich, die Stärken beider Programmiersprachen an den jeweils passenden Stellen einzusetzen. So lässt sich Kotlin dort einsetzen, wo tendenziell langlebigerer und stabilerer Code nötig ist. Flexible und experimentelle Ansätze lassen sich in Python gut abbilden. (sih)

Quellen

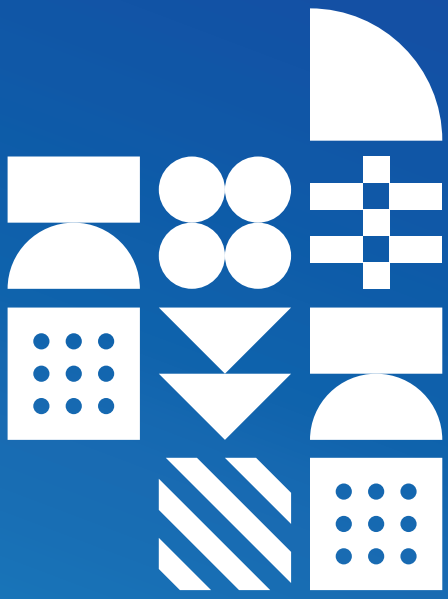
Ressourcen wie GitHub-Links und eine Liste der verwendeten Libraries finden sich unter ix.de/zvgb.

Hauke Brammer

ist Senior Software Engineer bei der DeepUp GmbH. Er arbeitet im Bereich von MLOps und ML-Engineering und hält dazu regelmäßig Vorträge. Seine Leidenschaft ist es, das Beste aus dem Software Engineering in die Welt des Machine Learning zu bringen.

> Stop starting, start finishing

Silke Hahn



Silke Notheis ist Organisationsentwicklerin beim Energieversorger EnBW. Im Interview gibt sie Tipps für das Bestehen als Team in krisengebeutelter Zeit.

Die Softwareentwicklerin Silke Notheis arbeitet bei EnBW, einem großen Energieversorger in Baden-Württemberg. Dort ist sie als Agile Coach mit dem Schwerpunkt Organisationsentwicklung tätig. Im Vorfeld der Konferenz Team Up!, bei der sie einen Vortrag über Teamentwicklung hielt, hatte *heise Developer* mit ihr über ihre Arbeit gesprochen – mit Blick auf die aktuelle Energiekrise auch über den Druck von außen, dem die Belegschaft von Versorgungsunternehmen ausgesetzt ist. Wie ihre Teams damit umgehen und was Organisationen in den aktuell schwierigen Zeiten beherzigen sollten – dazu gibt sie im Interview Denkanstöße.

heise Developer: *Silke, du bist von Haus aus Softwareentwicklerin. Was ist deine Lieblingsprogrammiersprache und wieso?*

Silke Notheis: Meine Lieblingsprogrammiersprache oder auch Programmierumgebung? Basic, da ich darin während der Grundschulzeit meine ersten Schritte auf einem C64 gegangen bin. Symbian OS, da ich zu Beginn meines Arbeitslebens beim Programmieren eines Navigationssystems auf Handys damit gekämpft habe. Und Perl, da ich in dieser Sprache am längsten tätig war. Es gab sogar Wettbewerbe, in denen versucht wurde, kurzen Programmcode zu schreiben, der möglichst viel verschleiert – was man normalerweise zu vermeiden sucht.

heise: *Spannend. Inwiefern verschleiert?*

Notheis: Man versucht, den Code möglichst kurz zu halten und die Funktionsweise des Codes möglichst schwer erkennbar zu gestalten. Es ist dann für jemanden, der das liest, wie Rätsel raten, um zu erkennen, was der Programmcode tut. Normalerweise versucht man Code möglichst klar zu schreiben, sodass ein Entwickler, der den Code übernimmt, oder auch man selbst nach einigen Monaten, schnell versteht, was er bewirkt, und man sich nicht erst lange einlesen muss.

heise: *Wie hast du denn zur IT gefunden?*

Notheis: Programmieren habe ich mir selbst erarbeitet. Daher sage ich immer, dass ich in vielen Programmiersprachen hacken kann und leider keine wirklich gut beherrsche. Vielleicht war das auch ein Grund, irgendwann nicht mehr selbst zu programmieren, sondern das Experten zu überlassen. Aufgefallen ist mir das vor allem in einem Scrum-Team, in dem ich vor 15 Jahren war. Die haben „an den Nanosekunden" geschraubt und das in C++ – da war ich lange nicht fit genug, um mithalten zu können, und bin dann in Richtung Requirements Engineering, Testing und Datenverarbeitung innerhalb des Teams abgebogen.

heise: *Bist du noch aktiv in der Softwareentwicklung tätig?*

Notheis: Aktuell bin ich nicht mehr selbst als Entwicklerin tätig, aber als Scrum Master und Coach in mehreren Teams sehr nah dran. Lustigerweise komme ich hier gerade wieder mit meinem Thema der Diplomarbeit in Berührung – Web Services. Nur dass sie nicht Apache Tomcat verwenden wie ich damals.

heise: *Was hat dorthin geführt, in die Organisationsentwicklung?*

Notheis: Das passierte stufenweise. Ich war schon immer gut vernetzt in meinen Unternehmen und habe durch meine Neugierde oft Stellen in – nicht nur meinen – Arbeitsabläufen gesehen, die sich verbessern oder vereinfachen lassen. Dadurch bin ich zu einer kleinen Runde von „ Veränderern mit Weitsicht" gekommen, die mit dem Management meines letzten Unternehmens in den Kontakt getreten sind. Wir haben größere Schritte angestoßen, die die PTV Group damals dringend benötigt hat. PTV ist ein Softwareunternehmen im Bereich Verkehrswesen und Logistik aus Karlsruhe. Dadurch ist die Transformation ins Laufen gekommen und irgendwann konnten die vielen Anforderungen nicht mehr nebenher bewältigt werden. Zwei Agile-Coach-Stellen wurden geschaffen, ich habe mich auf die interne Stelle beworben.

800 Augenpaare haben zwei Jahre lang verfolgt, was meine Kollegin und ich in die Wege geleitet haben, und so kam ich zur Organisationsentwicklung.
heise: *Wie fühlt man sich, wenn 800 Augenpaare auf einem ruhen?*
Notheis: Da das Unternehmen damals wie eine große Familie war, haben wir zwei den Druck natürlich gespürt, aber er war meistens freundschaftlich und sehr kollegial. Zudem war unser damaliger Chef großartig. Wir sind beide mit unseren Aufgaben gewachsen und durften verschiedene Erfahrungen machen – auch einiges ausprobieren, was heute zu meinem Erfahrungsschatz und Methodenkoffer beiträgt.
heise: *Jetzt arbeitest du bei einem großen Energieversorger in Baden-Württemberg. Inwiefern berührt die aktuelle Energiekrise deine Arbeit und die deiner Teams?*
Notheis: Sie berührt uns direkt, da Gesetzesänderungen wie Preisanpassungen, Änderungen der Mehrwertsteuer und einiges mehr bei den Teams landen, mit denen ich arbeite, und wir diese in unglaublich kurzer Zeit umsetzen müssen. Was vor zwei Jahren noch bis zu einem halben Jahr Planungs- und Umsetzungszeit bedeutete, sind jetzt wenige Wochen, und die Änderungen kommen häufiger.
heise: *Wie macht sich das bemerkbar, Stichwort Teamdynamik?*
Notheis: Mit den Teams macht das sehr viel, da Änderungen schneller passieren und auch mal „eine Rolle rückwärts" gemacht werden muss, wenn eine geplante Gesetzesänderung doch nicht kommt. Dadurch ist zum Beispiel auch der Anspruch da, die notwendigen Änderungen möglichst dynamisch zu halten. Hier profitieren die Business-Analysten und Softwareentwickler davon, dass sie schon länger zusammen in crossfunktionalen Teams arbeiten. Sie brauchen einander, um in kurzer Zeit das beste Ergebnis erreichen zu können.

Stop Starting, Start Finishing

heise: *Hast du einen konkreten Rat, wie Teams mit diesen Belastungen umgehen können?*
Notheis: Nicht allein damit zu sein und gemeinsam im Team nach Lösungen zu suchen ist ein wichtiger Grundstein, damit die Menschen die Mehrfachbelastung und auch Überstunden bewältigen können, die aufgrund der Kürze der Umsetzungszeit immer wieder vorkommen.

Zudem ist es wichtig, eine klare Priorität zu haben. Das ist oftmals nicht leicht zu erkennen, da viele verschiedene Punkte wichtig sind. Ich stelle dann den Product Ownern gerne die Frage: „Wenn du nur ein Feature haben kannst von den vielen, was wäre es?" Das bekommt Prio eins und darauf baue ich auf. Hier zählt dann der bekannte Satz „Stop starting, start finishing" – also nicht zwanzig Dinge gleichzeitig zu tun, sondern ein Ergebnis nach dem anderen zu bearbeiten und vor allem abzuschließen. 80 Prozent nutzbare Lösung ist besser als 30 Prozent bei x Funktionen.
heise: *Schöner Rat. Was macht für dich Scrum aus, was ist beim Scrum Master die wesentliche Aufgabe für das Team?*
Notheis: Scrum ist ein Framework, eine Arbeitsweise für ein Team, mit dem es sich schrittweise an nutzbare Lösungen für Kundenbedürfnisse herantasten kann. Für mich ein Werkzeug von vielen. Kanban, Scrumban und Lean-Methodiken sowie XP-Elemente ergänzen es zum Beispiel.
heise: *Scrumban?!*
Notheis: Das ist keine offiziell festgeschriebene Arbeitsweise, sondern beschreibt eine Mischung aus Scrum und Kanban und ist in der Realität sehr oft anzutreffen, oftmals auch unbewusst. Ich habe immer die Stacy- und Cynefin-Matrix-Modelle im Hinterkopf, wenn ich schaue, welche Arbeitsweise zum Team jeweils passt. Vereinfacht gesagt empfehle ich Kanban eher dann, wenn Arbeitsprozesse transparent gemacht und verbessert werden sollen, die regelmäßig in ähnlicher Form auftreten und wenn die Lösung zur Umsetzung bereits bekannt ist, aber nicht jeder sofort selbst umsetzen kann – das ist dann ein kompliziertes Umfeld. Sobald es komplex wird, kennt man die Lösung noch nicht und arbeitet schrittweise darauf hin, holt Feedback zu den Zwischenschritten ein und ändert auch mal die Richtung komplett oder muss Teile wegwerfen. Dann kann man gut Scrum einsetzen.

Für Schrauben den Schraubendreher

Notheis: Jedes Werkzeug sollte zum Umfeld passend eingesetzt werden. Man kann mit einem Hammer eine Schraube in die Wand bekommen – vielleicht wäre es mit einem Schraubendreher aber leichter gefallen. Oftmals erlebe ich, dass Scrum als bekannteste Arbeitsweise überall eingesetzt wird, ohne zu schauen, ob es für das Team, das Projekt, das Produkt aktuell passend ist. An dieser Stelle beginne ich immer, wenn ich zu einem Team komme. Ich schaue, welche Anforderungen auf deren Tisch landen, wie sie aktuell arbeiten und erarbeite dann zusammen mit dem Team eine passendere Methode. Dabei frage ich viel, höre zu und empfehle erst mal nur wenig. Zuhören, beobachten, den Blick von außen ins Team mitbringen – das Team dabei unterstützen, seine Arbeitsweise zu verbessern, Konflikte und Herausforderungen zu lösen und in eine gute Zusammenarbeit zu kommen: Das sind wichtige Eigenschaften eines Scrum Masters.
heise: *Wie schaut bei dir ein typischer Tag oder Monat aus?*
Notheis: Einen typischen Ablauf gibt es bei mir nicht, da ich mit den unterschiedlichsten Menschen und Teams arbeite und jeden Tag Überraschungen passieren. Daher plane ich nicht weit im Voraus, habe allerdings eine sorgsam geführte To-do-Liste mit Prioritäten. Auf diese werfe ich jeden Morgen einen Blick und schaue, was mich am Tag an Meetings erwartet und welchen einen Punkt von der To-do-Liste ich heute schaffen kann. Diesen fange ich so früh wie nur möglich an und meist schaffe ich durch diesen Trick mehr als nur einen Punkt. Wenn ich Tage habe, an denen ich fast nur in Meetings bin, die ich dann oftmals moderiere, dann ist mein einziges To-do für diesen Tag, dass am Ende des Tages meine To-do-Liste aktuell ist und ich nichts vergessen habe aufzuschreiben. Was ich mir fest einplane, das sind Vorbereitungen für Workshops, Vorträge oder auch Konzeptarbeit. Hier reserviere ich größere Blöcke für mich zu passenden Zeiten, an denen ich fit bin. Was bei mir immer vor geht, das sind persönliche Gespräche, wenn jemand mich als Coach braucht sowie Termine und To-dos für mein Scrum Team.

LEGO Serious Play und Wege aus dem Tal der Tränen

heise: *Was war denn das bemerkenswerteste Erlebnis in einem Scrum Team?*
Notheis: Davon gibt es eine ganze Menge. Mein Motto ist „spielend lernen“ und daher bringe ich gerne kurze Spiele mit, mit denen man Methoden ausprobieren und auf die Realität übertragen kann. Einmal gab es da ein Team, das ein wirklich schwieriges Spiel in der Hälfte der Zeit gelöst hat. Dieses Team hat mich genauso überrascht wie ein weiteres, das LEGO Serious Play ausprobieren wollte und nach anderthalb Stunden mit einer Teamvision aus dem Raum ging. Es gab auch schwierige Situationen, wenn aus einer einfachen Anfrage nach einem Workshop ein großer Konflikt im Team sichtbar wird, der jahrelang nicht berücksichtigt worden war.

Ich konnte den Teams bisher immer dabei helfen, den ersten Schritt aus dem Tal der Tränen zu finden. Während dieser Situationen bin ich immer sehr ruhig, biete dem Team den psychologisch sicheren Rahmen an, bin bei mir und dem Team und erkläre ihnen auch, was gerade passiert. Ich führe sie zurück auf die Arbeitsebene und zeige ihnen den ersten kleinen positiven Schritt. Nach solchen Erlebnissen fällt es mir schwer, das Dankeschön des Teams anzunehmen, da ich ihren Schmerz, die Traurigkeit und die Wut miterlebt habe. Und ich werde nie vergessen, wie es ist, den Tod eines Teammitglieds begleiten zu müssen oder dürfen. Diese Verbundenheit im Team ist heute noch spürbar, wenn wir uns treffen.
heise: *Gibt es Stolperfallen für Agile Coaches bei der Zusammenarbeit mit IT-Teams?*
Notheis: Es besser wissen zu wollen. Als Coach haben wir schon vieles erlebt und haben unsere Kochrezepte, die wir gerne anwenden.
heise: *Woran liegt es, wenn Teams Fristen nicht einhalten? Hast du Hinweise für ein gutes Zeitmanagement?*
Notheis: Die erste Frage, die ich mir hier stelle, ist: Woher kommen die Fristen für die Teams? Setzten sie sich die selbst, wie zum Beispiel den Inhalt eines Sprints, dann kann man damit arbeiten, in Retrospektiven immer wieder auf die Ursachen zu schauen und gemeinsam zu lernen, wie das Team es schafft, zuverlässigere Zeitpunkte anzugeben. Eine Schätzung ist dabei immer noch kein Versprechen. Kommen die Fristen von außen, kommt mir sofort das Projektmanagement-Dreieck in den Sinn. Im klassischen Projektmanagement wird der Scope festgesetzt: Das sind Funktionen, die für den Kunden umgesetzt werden sollen. Die Zeit und die Ressourcen – gemeint sind Menschen und Arbeitsmittel – werden darauf dann angepasst. Im Agilen ist es andersherum, ich habe ein festes Team und einen festen Zeitraum wie den Sprint im Scrum-Framework und schaue dann, was an Mehrwert für den Kunden hergestellt werden kann. Mein persönliches Zeitmanagement habe ich eben schon beschrieben. Bei Teams setze ich viel auf klaren Fokus, nicht zu viel gleichzeitig tun und enge transparente Zusammenarbeit.
heise: *Womit können Teams Fairness und Respekt untereinander stärken?*
Notheis: Fairness und Respekt sind für mich Werte und Teil der Kultur. Eine Kultur kann man meiner Erfahrung nach nur durch Verhalten verändern, von innen heraus und nicht von außen. Es beginnt bei einem selbst. Verändern kann ich jemanden nicht, nur mit meinem Verhalten vorausgehen.
heise: *Dinge, die wir von außen nicht ändern können – das bringt mich auf ein anderes Thema. Wir Frauen sind in der IT-Branche unterrepräsentiert. Wäre deine Karriere wohl anders verlaufen als Mann?*
Notheis: An der einen oder anderen Stelle mit Sicherheit. Nur mache ich mir darüber wenige Gedanken, da ich nun mal eine Frau bin und dies auch seine Vorteile hat. Mir haben zwei Workshops einer Coach-Kollegin vor 15 Jahren

sehr gut getan. Dort haben wir uns gemeinsam angeschaut, wo denn die Unterschiede in männlicher und weiblicher Kommunikation, Führung und Machtverhalten liegen. Dies zu verstehen, hat mir an einigen Stellen weitergeholfen.

Sich nicht die Butter vom Brot nehmen lassen

heise: *Interessant. Was rätst du Frauen, die sich für Arbeit in deinem Bereich interessieren?*
Notheis: Nie die Neugierde verlieren, mit offenen Augen durch die Welt gehen, sich nicht die Butter von Brot nehmen lassen und gerne auch mal etwas forsch sein. Das Bauchgefühl berücksichtigen und auch mal weibliche Gefühle oder weibliche Verhaltensmuster zulassen gehört ebenfalls dazu, sonst wären wir nicht authentisch.
heise: *Hast du einen Rat für junge Menschen, besonders Mädchen, heute vor der Berufswahl?*
Notheis: Einfach mal anfangen, etwas ausprobieren. Das machen, worauf sie am meisten Lust haben oder wo sich eine gute Gelegenheit bietet. Das Arbeitsleben verändert sich heute schnell und man kann den vorgegebenen Weg verlassen verlassen, um eine neue Richtung einzuschlagen.
heise: *Würdest du deinen Werdegang noch mal so durchlaufen wollen?*
Notheis: Ja, würde ich. Die größte Freude in meinem Beruf ist, wenn Menschen Anregungen und Ideen von mir eigenständig umsetzen und weiterentwickeln. Ein Kollege sagt dazu: "Das ist, als wäre für dich Weihnachten und Ostern gleichzeitig".
heise: *Liebe Silke Notheis, danke für die Einblicke!*

Das Interview führte Silke Hahn (sih@ix.de), Redakteurin von iX und heise Developer. (sih)

Quellen

Die Vollversion des Interviews sowie Unterlagen zu Scrum, "Scrumban" und Kanban finden sich unter ix.de/z72c.

Impressum We Are Developers!

Redaktion: iX | heise Developer
Telefon: 0511 5232-387, **E-Mail:** post@ix.de

Herausgeber: Ansgar Heise

Chefredakteur: Dr. Oliver Diedrich

Konzeption und redaktionelle Leitung: Silke Hahn (sih@ix.de) -367

Autoren dieser Ausgabe:
Isabel Bär, Hauke Brammer, Silke Hahn, Nils Kasseckert, Kilian Kluge, Glenn Marshall, Silke Notheis, Alvin Ramskogler, Rainer Stropek, Timo Zander

DTP-Produktion:
Lisa Hemmerling, Heise Medienwerk, Rostock

Korrektorat:
Cathrin Kapell, Martina Lübke, Marei Stade, Heise Medienwerk, Rostock

Titelbild:
Franziska Rex, Lisa Hemmerling, Heise Medienwerk

Verlag:
Heise Medien GmbH & Co. KG,
Postfach 61 04 07, 30604 Hannover; Karl-Wiechert-Allee 10, 30625 Hannover;
Telefon: 0511 5352-0, Telefax: 0511 5352-129

Geschäftsführer:
Ansgar Heise, Beate Gerold

Mitglieder der Geschäftsleitung:
Jörg Mühle, Falko Ossmann

Anzeigenleitung (verantwortlich für den Anzeigenteil):
Michael Hanke (-167), E-Mail: michael.hanke@heise.de,
www.heise.de/mediadaten/ix

Leiter Vetrieb und Marketing:
André Lux

Druck:
Dierichs Druck + Media GmbH & Co. KG,
Frankfurter Straße 168, 34121 Kassel

Inserenten

Die hier abgedruckten Seitenzahlen sind nicht verbindlich. Redaktionelle Gründe können Änderungen erforderlich machen.

Foto Daniel Schaffer | Lumen GmbH

Neue IT-Einheit stärkt technologische Vorreiterrolle der REWE Group im Lebensmitteleinzelhandel

Die immer engere Zusammenarbeit der beiden starken Technologie-Bereiche der REWE Group machte es in den vergangenen Monaten schnell deutlich: Als gemeinsame IT-Einheit sind die REWE digital und die REWE Systems für alle Bereiche der REWE Group am schlagkräftigsten.

In der Vergangenheit haben sich zwei Organisationen etabliert, die als Ziel stets das perfekte Einkaufserlebnis vor Augen haben. Die REWE digital wurde gegründet, um den Online-Handel voranzutreiben. Die REWE Systems hat sich dem stationären Handel verschrieben. Ab sofort kommen das Know-how und der Innovationsgeist von rund 2.200 Mitarbeitenden unter einem organisatorischen Dach als "REWE digital GmbH" zusammen. Die neue REWE digital ist somit das Zuhause für alle, die sich IT auf die Fahne geschrieben haben. Hier gehörst Du hin, wenn Du Future Thinker, IT-Spezialist:in, Software Developer, UXler:in, SAP-Expert:in, System Admin, Techniker:in, irgendetwas dazwischen oder etwas ganz anderes bist. Hauptsache, Du kommst aus der digitalen Welt. Denn wir arbeiten an IT-Projekten für die ganze REWE Group und sind gleichzeitig der Partner in Tech. Wir digitalisieren das tägliche Handelsgeschäft und bringen neue Zukunftsthemen auf den Plan. Dein Potenzial ist unsere Basis für Innovation.

Also lass uns gemeinsam im Home of IT Ideen entwickeln, diese weiterspinnen und sie bis zum fertigen Produkt perfektionieren.

REWE DIGITAL

rewe-digital.com


Capgemini
ALLEINE VORANKOMMEN?
ODER GEMEINSAM GROSSE ZIELE ERREICHEN?
Werde jetzt Teil unseres starken Netzwerks aus IT-Expert*innen! Gestalte die digitale Welt von morgen und treibe Themen wie Cloud, Data sowie Transformation mit Schwerpunkt SAP oder auch Salesforce voran. Wir bieten dir ein inspirierendes Team, flexible Karrieremöglichkeiten sowie die Freiheit, mit deiner Arbeit für dich und andere Perspektiven zu schaffen.
Erfahre jetzt mehr über uns und deine Einstiegsmöglichkeiten:
www.capgemini.de/karriere
GET THE FUTURE YOU WANT