bidt Bayerisches Forschungsinstitut für Digitale Transformation
EIN INSTITUT DER BAYERISCHEN AKADEMIE DER WISSENSCHAFTEN

Longread

Was ist Software?

Prof. Dr. Alexander Pretschner

März 2021

Was ist Software eigentlich? Software ist in aller Munde, Software Engineering gilt als die Schlüsseldisziplin schlechthin, Software „eats the world“. In diesem Beitrag erklären wir, was Software eigentlich ist, dass Software viel mehr ist als Computerprogramme, warum das Erstellen von Software nicht so einfach ist und was Software mit KI und Maschinenlernen zu tun hat.

Unsere Sicht auf Software besteht aus vier Teilen: Im ersten Teil erklären wir über die Analogie des Kuchenbackens, was ein Programm ist und wie Programme im Computer ausgeführt werden. Im zweiten Teil übertragen wir diese Ideen auf echte Programme. Im dritten Teil zeigen wir, wie Programme und Maschinenlernen zusammenhängen. Schließlich zeigen wir im vierten Teil, dass ein einzelnes Programm immer nur ein kleines Stück Funktionalität ist und dass der Bau großer Systeme neben dem Programmieren deswegen noch ganz andere Probleme birgt, die das Software Engineering adressiert.

Der Text ist entstanden in enger Kooperation von fortiss, bidt, dem Lehrstuhl für Software und Systems Engineering und dem Venture Lab Software Engineering/AI der TU München. Zielgruppe sind interessierte Laien.

Teil I: Was sind Programme?

Kuchenrezepte sind Algorithmen

Kuchenbacken hat so viel mit Software zu tun, dass wir kaum mehr benötigen, um die zentralen Ideen von Software zu erklären. Kuchenrezepte beschreiben, welche Zutaten in welcher Reihenfolge wie verarbeitet werden, bis am Ende der Kuchen fertig ist: Eier, Zucker und Butter verrühren, Mehl und Backpulver hinzufügen, den Teig kneten und dann bei 160 Grad für 30 Minuten backen. Es gibt die Zutaten, die einzelnen Zubereitungsschritte und am Ende einen Kuchen. Außerdem braucht es Hilfsmittel, die vielleicht gar nicht im Rezept stehen: Mixer, Rührschüssel und Backofen. Es gibt außerdem die Autorin und den Autor des Rezepts. Es gibt die einzelnen Schritte im Rezept, nämlich vermengen, rühren, warten und backen. Und es gibt Sie, die Bäckerinnen und Bäcker, die diese einzelnen Schritte hintereinander ausführen und sicherstellen, dass jedem Schritt das Ergebnis des vorherigen Schritts zur Verfügung gestellt wird. Damit haben wir die wesentlichen Konzepte schon zusammen.

In der Softwarewelt sind die Zutaten die sogenannten Eingaben. Der Kuchen ist die Ausgabe. Rezepte findet man in Backbüchern, die in der Informatik Bibliotheken heißen. Das Rezept beschreibt einen sogenannten Algorithmus. Algorithmen sind schrittweise Vorschriften, wie Eingaben in Ausgaben überführt werden. Wenn Algorithmen so aufgeschrieben werden, dass sie von einem Computer ausgeführt werden können, nennt man sie Programme. Programme sind Software. Autorin oder Autor des Rezepts ist die Programmiererin oder der Programmierer. Die Küchengeräte sind die sogenannte Hardware. Und Sie als Bäckerinnen und Bäcker sind ein zusätzliches, besonders wichtiges Stück Hardware: Sie sind der sogenannte Prozessor, den man oft auch CPU (central processing unit) nennt. Der Prozessor führt die einzelnen Schritte des Programms aus, so wie Sie die einzelnen Schritte des Rezepts befolgen.

Von Ihrer Küche zur Bäckerei

Das war es schon! Zumindest in der Analogie haben Sie jetzt verstanden, was ein Programm tut und wie es auf der Hardware abläuft. Zur Übung wollen wir diese Grundidee jetzt einmal übertragen, bevor wir uns echten Programmen widmen: Stellen wir uns eine vollautomatische Bäckerei vor, die unterschiedliche Kuchen backen

kann. Bevor Sie weiterlesen, überlegen Sie bitte kurz selbst, was Eingaben, Ausgaben und Hardware sind. Gleich vorab: Programm und Prozessor zu identifizieren ist hier etwas schwieriger. Haben Sie's?

Die Eingaben sind die Auswahl, etwa die Nummer, eines spezifischen Rezepts und die entsprechenden Zutaten. Ausgaben sind die fertigen Kuchen. Die Hardware besteht hier neben (großen!) Backöfen, Rührgeräten und Mixern aus Eieraufschlagmaschinen, Förderbändern, Verpackungsmaschinen und so weiter. Die werden übrigens alle ihrerseits individuell von speziellen Programmen gesteuert, die auf Prozessoren in diesen Maschinen ablaufen.

Jetzt wird es ein bisschen komplizierter: Das Zusammenspiel der einzelnen Hardwarekomponenten muss je nach gebackenem Rezept unterschiedlich koordiniert werden. Das erledigen in diesem Fall nicht mehr Sie als Bäckerin oder Bäcker, denn hier wird ja der ganze Backvorgang automatisiert. Stattdessen erledigt das jetzt ein Steuerungscomputer (der Prozessor), auf dem wieder ein spezielles Programm abläuft, nämlich ein Steuerungsprogramm. Dieses Programm steuert in Abhängigkeit des zu backenden Rezepts schrittweise Eieraufschlagmaschine, Rührgerät, Förderbänder und so weiter in der richtigen Reihenfolge an. So stellt es sicher, dass die einzelnen Stationen der Bäckerei beziehungsweise des Backvorgangs das jeweils richtige Zwischenprodukt zur richtigen Zeit geliefert bekommen. Die Algorithmen, die ursprünglich nur die Rezepte waren, werden jetzt also in Programmform um Steuerungsbefehle für die verschiedenen Maschinen, also unsere Bäckereihardware, erweitert.

Software

Die Idee der Umwandlung von Ein- in Ausgaben unter Verwendung von Hard- und Software ist sehr mächtig, weil damit sehr viele Abhängigkeiten beschrieben werden können. Ich finde es interessant, dass sich auch viele Mechanismen unseres Körpers so beschreiben lassen. Für die Regulierung des Blutzuckers etwa schüttet, stark vereinfacht gesagt, die Bauchspeicheldrüse in Abhängigkeit des aktuellen Blutzuckers kontinuierlich mehr oder weniger Insulin aus: Wenn aufgrund einer Mahlzeit der Blutzucker steigt, wird mehr Insulin ausgeschüttet; dadurch verringert sich der Blutzuckerspiegel; was wiederum eine Verringerung der Insulinausschüttung zur Folge hat. Wenn zu wenig Zucker im Blut ist, wird Glukagon ausgeschüttet, was die Leber zur Freigabe von Zucker anregt. Das System „Blutzuckermanagement" hat also eine Schleife, weil die Ausgabe (Insulin, Glukagon) direkt die Eingabe (Blutzucker) beeinflusst. Ganz ähnlich ist es im Herzen. Noch stärker vereinfachend, führt ein erhöhter Sauerstoffbedarf der Muskeln durch körperliche Betätigung zu einer Erhöhung der Herzfrequenz; dadurch erhöht sich die Sauerstoffmenge, die den Muskeln pro Zeiteinheit zur Verfügung gestellt wird; irgendwann ist hinreichend viel Sauerstoff verfügbar, und die Herzfrequenz wird nicht weiter erhöht. Die Mechanismen, die diese sogenannten Regelkreise verwalten, kann man durchaus als Programme verstehen, wenngleich nicht im Sinn einer Folge von Programmschritten.

Das Phänomen einer Schleife von Aus- zu Eingaben finden wir in vielen Maschinen. Klimaanlagen messen die Zimmertemperatur; öffnen bei zu hoher Temperatur Ventile, um Kühlflüssigkeit fließen zu lassen; dadurch verringert sich die Zimmertemperatur; das führt nach einer gewissen Zeit dazu, dass die Ventile weniger Kühlflüssigkeit transportieren; wodurch dann durch die Außentemperatur die Zimmertemperatur wieder steigt; und so weiter. Die Funktionalität, die diese Regelungsfunktion übernimmt, wird durch Programme implementiert.

Was ist nun Software? Software sind Programme, die auf einem oder mehreren Computern ausgeführt werden. Programme wie Excel oder Ihr Mailprogramm sind Software, die auf einem Computer im Sinn von Desktop, Laptop, Tablet oder Smartphone ausgeführt wird. Programme wie die Google-Suche, Facebook oder Instagram

laufen zu einem Teil auf Ihrem Laptop oder Ihrem Smartphone, um Ihnen Interaktionen zu ermöglichen. Gleichzeitig wird deren Funktionalität, also die tatsächliche Durchführung einer Suche oder das Zur-Verfügung-Stellen von Suchergebnissen, Bildern oder Nachrichten durch ein Zusammenspiel von mehreren, oft Tausenden von Computern ermöglicht. Die Existenz dieser Computer bleibt uns dabei meist vollständig verborgen.

Die Funktionalität Ihres Autos wird in großen Teilen ebenfalls durch Programme zur Verfügung gestellt. In einem Auto finden sich häufig mehr als 100 Computer, auf denen ganz unterschiedliche Programme ablaufen. Diese steuern, wie beschleunigt und gebremst wird, wie der Scheibenwischer funktioniert oder wie Sie Ihren Sitz bewegen. Software steuert eigentlich alle Geräte in unserer unmittelbaren Umgebung. Weil man diese Computer, auf denen die entsprechenden Programme ausgeführt werden, nicht sieht und sie üblicherweise in Maschinen verbaut sind, heißen solche Systeme eingebettete Systeme. Und weil diese Programme unmittelbar auf Reize aus der physikalischen Umwelt reagieren und diese physikalische Umwelt umgekehrt auch beeinflussen können (Temperatur, Abstand und so weiter), gleichzeitig aber als Software gewissermaßen virtuell im Cyberspace existieren, nennt man sie auch cyberphysikalische Systeme. Solche Systeme können beliebig groß und komplex sein, etwa die Raumstation ISS, aber auch sehr klein, etwa eine Lichtsteuerung im Wohnzimmer, die einfach nur kontinuierlich auf die äußeren Lichtverhältnisse reagiert. Ingenieurinnen und Ingenieure arbeiten heute daran, immer zahlreichere Systeme auf ganz unterschiedlichen Größenskalen zu bauen und im Sinn der Kommunikation zwischen Systemen miteinander zu vernetzen. Die Idee der Vernetzung vornehmlich kleinerer Systeme kennen Sie unter dem Schlagwort Internet der Dinge, das uns etwa die Automatisierung und Vernetzung von Licht, Herd, Heizung, Energieversorgung und Bewässerung im Smart Home und Smart Garden ermöglicht. Dass das Sicherheits- und Privatheitsprobleme birgt, ist ganz klar, führt aber an dieser Stelle zu weit.

Heutzutage betrachten Ingenieurinnen und Ingenieure in interdisziplinären Projekten in der Regel übrigens nicht mehr allein die genannten technischen Systeme, sondern sogenannte soziotechnische Systeme, in denen dem Zusammenspiel verschiedener technischer Systeme auch menschliche Akteure hinzugefügt werden. Ein schönes Beispiel ist ein hybrider Verkehr, in dem sowohl von Menschen gesteuerte als auch autonome Fahrzeuge gleichzeitig existieren und aufeinander reagieren müssen. Und weil Menschen unerwartete oder „unlogische" Dinge tun – wenn etwa Kinder mit Kreide die Straße bemalen –, müssen Programme dafür ausgelegt sein, dass ihnen auch Eingaben zur Verfügung gestellt werden, die von Programmiererinnen und Programmierern nicht vorhergesehen waren.

Hardware

Programme werden auf Computern ausgeführt. Programme und Computer, also Software und Hardware, verhalten sich wie Geist und Körper beim Menschen. Die Hardware von Computern besteht neben dem Prozessor noch aus anderen Komponenten, die die oben genannten zentralen Aufgaben erledigen. Neben dem Prozessor, der die Berechnungen ausführt, gibt es zunächst Hardware für die Datenhaltung. Daten werden im Speicher abgelegt, auf der Festplatte, auf USB-Sticks und zunehmend auch in der Cloud. Über Anschlüsse ans Netzwerk erfolgt dann die Kommunikation mit anderen Computern, die wir in diesem Beitrag aus Platzgründen sehr stiefmütterlich behandeln. Neben Prozessor, Speicher und Netzwerkanschluss besteht ein Computer noch aus weiteren Geräten, die wir genau wie Prozessor, Speicher und Netzwerkanschluss oben schon als Hardware bezeichnet haben: Bildschirm, Fingerabdrucksensor, Temperatursensor, Batterie, Kamera, Tastatur, Lautsprecher, Mikrofon, Kabel und so weiter. Insgesamt übernimmt Software unter Verwendung dieser Hardware also die Aufgaben Berechnung, Kommunikation, Verwaltung und Speichern von Daten sowie Wechselwirkungen mit der Umwelt.

Ein- und Ausgaben sind Daten

Beim Kuchenbacken sind Eingaben die Zutaten und Ausgaben die fertigen Kuchen. Programme hingegen empfangen als Eingaben Daten und liefern als Ausgabe ebenfalls Daten. Auch die Zwischenergebnisse der einzelnen Schritte sind Daten. Es gibt Programme wie die Internetsuche, Ihr Mailprogramm oder Ihren Herd, bei denen Sie die Möglichkeit haben, etwas über die Tastatur, per Sprache oder per Gestik einzugeben, nämlich eine Suchanfrage oder eine Nachricht. Diese Programme zeigen Ihnen dann auf einem Bildschirm etwas an.

Bei einem Auto, einer Waschmaschine, einer Heizung oder einer automatisierten Bäckerei reagiert das Programm außerdem auf Reize aus seiner physikalischen Umwelt: Der adaptive Tempomat erkennt Autos vor Ihnen, die Waschmaschine reagiert auf Wasserstände und die Heizung auf Temperaturen und so weiter. Die entsprechenden Eingabedaten werden über sogenannte Sensoren zur Verfügung gestellt. Sensoren sind beispielsweise Thermometer, Kameras oder Radar. Sie übersetzen Zusammenhänge in der echten Welt, wie die Temperatur, vorausfahrende Autos oder Wasserstände, in Daten, die dann von Programmen verarbeitet werden können. Im Unterschied zu Ihrem Mailprogramm ist die Ausgabe eines Programms in der Waschmaschine oder im Auto nicht (nur) auf einer Anzeige abzulesen. Die Ausgabedaten können stattdessen auch in Effekte in der physikalischen Umwelt übersetzt werden: Beschleunigen oder Bremsen, Wasser abpumpen, Gasbrenner einschalten und Wasser erhitzen. Programme sind also überall.

Wer was wie macht: Funktionen, Algorithmen und Programme

Wir haben gesehen, dass Rezepte, Algorithmen und Programme einzelne Schritte beschreiben und damit festlegen, wie Eingaben in Ausgaben überführt werden. Für Sie als Bäckerinnen und Bäcker sind diese einzelnen Schritte wesentlich. Ich halte es nun für möglich, dass Ihre Kinder im Unterschied zu Ihnen nicht an diesen einzelnen Schritten interessiert sind, sondern nur am Kuchenessen selbst, also der Ausgabe. Nehmen wir weiter an, dass Ihre Kinder heute Lust auf Marmorkuchen haben, aber der Kühlschrank leer ist. Dann müssen die Kinder einkaufen gehen und sich deswegen auch für die Zutaten interessieren, also die Eingaben. Wenn man das Verhältnis von Ein- zu Ausgaben beschreiben will, ohne jedoch die einzelnen Schritte zu erklären, nennt man das Funktion. Funktionen kennen Sie: Die Addition beispielsweise ist so eine Funktion mit zwei Summanden als Eingabe und der Summe als Ausgabe. Wenn Sie im Kopf addieren, denken Sie gar nicht mehr darüber nach, wie man das macht – die einzelnen Schritte interessieren Sie nicht! Aber wie macht das eine Maschine? Das ist genau die Aufgabe des Programms. Ein Programm beschreibt ganz einfach, wie eine Funktion ausgerechnet wird.

Der Zusammenhang zwischen Funktion, Algorithmus und Programm ist so wichtig, dass ich ihn noch einmal zusammenfassen möchte. Die Funktion beschreibt nur den Zusammenhang von Ein- und Ausgabe, ohne genau zu sagen, wie die Ausgabe berechnet wird. Ein Algorithmus definiert die Schritte, mit denen diese Funktion ausgerechnet werden kann. Und das Programm ist die Beschreibung eines Algorithmus in einer für die Maschine verständlichen Form. Mit der Unterscheidung von „was“ und „wie“ kann man auch sagen, dass die Funktion das Problem beschreibt (was?) und der Algorithmus und das Programm die Lösung (wie?).

Programmiersprachen

Ein Programm ist, wie ein Kuchenrezept, ein Stück Text. Wenn man etwas aufschreibt, muss man dafür eine Sprache finden. Informatikerinnen und Informatiker haben deswegen Programmiersprachen entwickelt. Ein

Programm besteht aus mehreren „Sätzen“ in dieser Programmiersprache. Jeder Satz hilft dabei, Schritt für Schritt die Eingabe in die Ausgabe zu überführen.

Es gibt sehr viele Programmiersprachen, so wie es viele natürliche Sprachen gibt. Wenn Sie eine Fremdsprache beherrschen, ist Ihnen vielleicht schon einmal aufgefallen, dass sich bestimmte Sachverhalte in dieser Sprache eleganter formulieren lassen als im Deutschen und umgekehrt. Genau das ist der Grund für die Vielzahl an Programmiersprachen, denn dort ist es genauso: Bestimmte technische Schritte in der Umwandlung von Ein- und Ausgaben lassen sich in der einen Programmiersprache besser beschreiben als in einer anderen. Vielleicht können Sie sich vorstellen, dass ein Programm, das die Oberfläche Ihres Mailprogramms darstellt, leichter in einer Sprache zu schreiben ist, in der man über „Fenster“ und „Knopf“ und „Mauszeiger“ sprechen kann, als in einer, die die Entfernung zum voranfahrenden Fahrzeug in einen Befehl zum Bremsen überführt. Probieren Sie doch einmal die Programmiersprache Scratch aus (Link), die besonders gut zum Interagieren mit Figuren in Ihrem ersten eigenen Computerspiel geeignet ist. In Scratch werden die „Sätze“ übrigens durch bildliche Symbole gebildet, das geht also auch.

Teil II: Zwei echte Programme – und warum Programmieren schwierig ist!

Zwei echte Programme

Was bedeutet es nun, dass ein Programm aus Sätzen besteht, also aus hintereinander ausgeführten Schritten, um damit eine Funktion auszurechnen? Nochmals zur Erinnerung: Wir unterscheiden zwischen dem, was eine Funktion beschreibt, und wie das berechnet wird: Einem Menschen ist es intuitiv klar, wie die Buchstabenfolgen $f(x,y)=x+y$ und $g(x)=x^2$ zu verstehen sind und was sie bedeuten, nämlich „Summe von x und y“ und „x zum Quadrat“. Einer Maschine aber müssen wir sagen, wie sie das ausrechnen kann. Eine Möglichkeit für die Berechnung der Quadratfunktion ist x*x. Zusätzlich müssen wir der Maschine dann auch noch sagen, wie die Multiplikation funktioniert, die ihrerseits wiederum aus einzelnen Schritten besteht.

Sehen wir uns deshalb kurz an, wirklich nur ganz kurz, wie ein Programm jeweils aussieht, das unsere Quadrat- und Summenfunktion berechnet. Lassen Sie uns mit der Summenfunktion beginnen. Eingaben sind die zwei Summanden x und y, Ausgabe soll deren Summe sein. Lassen Sie uns für den Moment annehmen, dass wir eine einfache Programmiersprache benutzen, die zwar die Addition beliebiger Zahlen nicht beherrscht, aber die einfachere Addition von 1 und Subtraktion von 1. Wir werden jetzt versuchen, mit diesen beiden einfachen Funktionen beliebige Summen zu berechnen.

Fragen Sie mich nicht, wie man darauf kommt, aber x+y ist ja das Gleiche wie 1+x+(y-1). Wir haben also links eine Eins addiert und rechts eine Eins abgezogen, was sich gegenseitig aufhebt. 1+x+(y-1) wiederum ist das Gleiche wie 1+1+x+(y-1-1). Und so weiter. Beispielsweise ist 3+2 das Gleiche wie (1+3)+(2-1), also 4+1, und das ist das Gleiche wie (1+4)+(1-1), also 5+0. Das ist unser Endergebnis: 5.

Wir können also beliebige Summen durch Addition und Subtraktion von 1 berechnen, indem wir vom zweiten Summanden (das ist ja y) eine Eins abziehen und gleichzeitig eine Eins zum ersten Summanden, also x, hinzufügen. Das machen wir genau y-mal: Wir ziehen also y-mal eine Eins von y ab, bis das Ergebnis null ist, und addieren y-mal eine Eins zu x.

Die ursprünglichen Werte von x und y liegen im oben eingeführten Speicher, und in unserem Programm können wir die entsprechenden Werte lesen und überschreiben (dazu sagen Informatikerinnen und Informatiker speichern und noch genauer in x speichern oder in y speichern). Unser Programm sieht dann wie folgt aus.

Programm „Summe“.
Eingabe: zwei Zahlen x und y
Ausgabe: Summe x plus y
Wiederhole, solange y ungleich 0 ist:
1. Addiere 1 zu x und speichere das Ergebnis 1+x in x
2. Subtrahiere 1 von y und speichere das Ergebnis y-1 in y

Gib x aus

Als echter Code sieht das so aus: `while` (`Bedingung`) ist dabei die Wiederholungsanweisung, die so lange ausgeführt wird, wie die Bedingung in der Klammer wahr ist. ≠ ist die Ungleichheit. Mit dem Linkspfeil ← beschreiben wir den Vorgang des Speicherns: x ← x+1 sagt, dass der Wert von x+1 an die für x vorgesehene Stelle im Speicher geschrieben wird. output gibt das Ergebnis auf dem Bildschirm aus. Die geschweiften Klammern dienen nur der Gruppierung.

```
summe(x, y) {
	while (y≠0) {
		x ← x+1;
		y ← y-1;
	}
	output x;
}
```

Ein Computer arbeitet das Programm nun wie folgt ab. Nehmen wir den Fall der Eingaben x=3 und y=2.
1. Weil y≠0 ist, wird im ersten Schritt x zu 3+1=4 und y zu 2-1=1.
2. y ist immer noch ungleich null, also wird x jetzt zu 4+1=5 und y zu 1-1=0.
3. Jetzt gilt y≠0 nicht mehr, die Wiederholung ist beendet.
4. Das Ergebnis ist der letzte Wert von x, nämlich 5.

Das ist vielleicht etwas ungewohnt, aber doch eigentlich ganz logisch. Behalten Sie bitte im Kopf, dass wir hier das schwierigere Problem „berechne beliebige Summe” mithilfe zweier einfacherer Operationen gelöst haben, nämlich der Addition und Subtraktion von 1. Das ist eine Kernidee der Programmierung: Wir lösen ein schwieriges Problem mit einfachen Schritten.

Lassen Sie uns jetzt das gleiche Spiel für die Quadratfunktion spielen. Diesmal nehmen wir an, dass unsere Programmiersprache wie eben die Subtraktion von 1 zur Verfügung steht und zusätzlich die Addition von Zahlen. Letztere haben wir ja eben selbst programmiert, sodass wir dieses Programm einfach verwenden können.

x mit x zu multiplizieren ist das Gleiche, wie x-mal die Zahl x zu addieren. Denn 3*3 ist 3+3+3 und 4*4 ist 4+4+4+4. Um die Multiplikation mit der allgemeinen Addition und der Subtraktion von 1 zu beschreiben, verwenden wir wieder so einen Trick: x*x ist das Gleiche wie x+(x-1)*x. Und das ist das Gleiche wie x+x+(x-1-

1)*x. Aha! Das wiederum ist das Gleiche wie x+x+x+(x-1-1-1)*x. Und so weiter bis der Wert in der Klammer null ist. So haben wir jetzt die Quadratfunktion mithilfe der Subtraktion von 1 und der allgemeinen Addition definiert.

In diesem Fall müssen wir uns ein Zwischenergebnis merken, das wir jeweils als `z` abspeichern. Am Anfang setzen wir z auf den Wert 0. Betrachten wir jetzt das Beispiel 3^2:

1. Das ist das Gleiche wie 3*3, was das Gleiche ist wie 3+(3-1)*3. Der linke Teil der Summe ist unser Zwischenergebnis.
2. Wir addieren also diese linke 3 zu z, das ja ursprünglich 0 ist, und speichern das Ergebnis 3+0 in `z`.
3. Dann machen wir weiter: z+2*3 ist das Gleiche wie z+3+(2-1)*3, also z+3+1*3.
4. Wir addieren die linke 3 zum Zwischenergebnis z und schreiben den Wert z=3+3=6 in den Speicher für z.
5. z+1*3 ist dann das Gleiche wie z+3+(1-1)*3, was das Gleiche ist wie z+3+0*3.
6. Wir addieren die linke 3 zum Zwischenergebnis z, das nun den Wert z=6+3=9 hat.
7. 0*3 ist 0, also können wir aufhören.
8. Das Zwischenergebnis z hat sich in jedem Schritt geändert, und dessen letzter Wert ist 9. Stimmt!

Weil wir in jedem Schritt den Wert von x zum Zwischenergebnis z addieren müssen, können wir x nicht wie im Fall der Summe in jedem Schritt ändern. Stattdessen benötigen wir ein zweites Zwischenergebnis v, das hintereinander den Wert von x, x-1, x-2, …, 0 speichert.

Programm „Quadrat“.
Eingabe: eine Zahl x
Ausgabe: x zum Quadrat, also x*x
Speichere 0 im Zwischenergebnis z
Speichere x im Zwischenergebnis v
Wiederhole, solange v ungleich 0 ist:
1. Addiere x zu z und speichere das Ergebnis z+x in z
2. Subtrahiere 1 von v und speichere das Ergebnis v-1 in v

Gib z aus

Als Code:

```
quadrat(x){
	z=0;
	v=x;
	while (v≠0) {
		z ← z+x;
		v ← v-1;
	}
	output z;
}
```

Simulieren wir für die Eingabe x=3 den Computer, der das Programm ausführt:

1. Zuerst wird z auf 0 gesetzt und v auf den Wert von x, also 3.
2. Weil 3≠0 ist, wird z auf 0+3=3 gesetzt und v auf 3-1=2.
3. Weil 2≠0 ist, wiederholen wir das: z wird zu 3+3=6 und v zu 2-1=1.
4. Wiederum ist 1≠0, also wiederholen wir die Schritte erneut: z wird zu 6+3=9 und v zu 1-1=0.
5. Jetzt ist die Wiederholung beendet, denn 0≠0 gilt nicht, und es wird z=9 ausgegeben.

So einfach funktionieren Programme. Auf dieselbe Art und Weise werden E-Mail-Programme, Suchmaschinen, Steuergeräte in Autos und in Bäckereien programmiert. Wirklich!

Falls Sie sich wundern sollten: Wir haben in unsere Programme bewusst einen kleinen Fehler eingebaut, den wir hinterher finden werden.

Bibliotheken und Wiederverwendung

Wenn unsere Programmiersprache das Additionssymbol + nicht kennt, können wir es einfach durch unser Summenprogramm oben ersetzen (Änderung in Grün) und erhalten ein neues Programm `quadrat2`:

```
quadrat2(x) {
        z=0;
        v=x;
        while (v≠0) {
                z  ← summe(z,x);
                v ← v-1;
        }
        output z;
}
```

Die Verwendung von vorher erstellten Programmen in einem neuen Programm ist ein ungeheuer mächtiges Prinzip, das man erst in den Sechzigerjahren entdeckt hat. Es ist deswegen so mächtig, weil es uns einerseits erlaubt, Probleme in kleinere Probleme zu zerlegen, unabhängig voneinander zu lösen und dann die Lösungen zusammenzusetzen. Es hilft uns also unter anderem bei der Organisation unserer Programmierarbeit.

Andererseits erlaubt dieses Prinzip es auch, einmal erstellte Funktionalität anderen Programmiererinnen und Programmierern zur Verfügung zu stellen. Informatiker nennen solche Programmpakete, die anderen zur Nutzung in deren eigenen Programmen zur Verfügung gestellt werden, Bibliotheken, die wir oben mit Backbüchern verglichen haben. Vielleicht haben Sie schon einmal von der Programmiersprache Java gehört, oder von einer Sprache namens Python? Diese Sprachen werden unter anderem deswegen von sehr vielen Programmierern auf der ganzen Welt genutzt, weil sie riesige Bibliotheken für alle möglichen Funktionalitäten zur Verfügung stellen, unter anderem um Daten ins Internet zu schicken, Maschinenlernen zu verwenden oder natürlichsprachigen Text zu analysieren. Programmierer müssen diese Funktionalitäten dann nicht selbst erstellen, sondern können direkt von der Arbeit anderer profitieren, indem sie die entsprechenden Programme wiederverwenden.

Wenn Sie genau aufgepasst haben, werden Sie bemerken, dass wir zwei Konzepte für dieselbe Idee eingeführt haben. Oben hieß es, dass sich Programmiersprachen darin unterscheiden, wie sie „bestimmte Sachverhalte“ oder „technische Schritte“ ausdrücken, und dass manche Sprachen eher geeignet sind, über „Fenster“, „Knopf“ und „Mauszeiger“ zu sprechen, und andere über „Entfernung zum voranfahrenden Auto“. Hier sagen wir jetzt, dass „bestimmte Sachverhalte“ in Programmen erfasst werden, also für Probleme Lösungen programmiert werden, die dann in Bibliotheken zur Wiederverwendung zur Verfügung stehen. Ihre Beobachtung ist richtig. Das ist in der Tat zweimal dieselbe Idee in unterschiedlichen Ausprägungen: Wiederkehrende Konstruktionen sollen möglichst einfach verwendet werden können. Das zugrundeliegende Prinzip nennen Informatiker Abstraktion.

Fast geschafft: Prozessor und Maschinensprache

Vielleicht haben Sie beim Lesen den Verdacht entwickelt, dass wir irgendwie getrickst haben. Denn wir haben unsere Funktionen unter Zugrundelegung anderer, einfacherer Funktionen programmiert, was das Problem ja streng genommen nur verschiebt. Zur Berechnung einer beliebigen Summe haben wir die einfacheren Operationen „plus 1" und „minus 1" ganz nonchalant als gegeben vorausgesetzt. Im Quadratprogramm konnten wir uns gerade noch retten, weil wir die Addition vorher selbst programmiert hatten. Die „minus 1"-Operation haben wir dort aber immer noch benötigt.

Wenn Sie sich erinnern, war die Situation beim Kuchenbacken ganz ähnlich. In Rezepten werden viele Einzelschritte stark vereinfacht dargestellt, wenn etwa stillschweigend angenommen wird, dass „Backen" aus den folgenden Vorgängen besteht: „Ofentür öffnen", „alle Bleche aus dem Ofen nehmen", „Ofentür schließen", „Ofen vorheizen", „Ofentür öffnen", „Kuchenform hereinschieben", „Backen", „Ofen ausschalten", „Ofentür öffnen", „Kuchen herausnehmen", „Ofentür schließen". Dort wird vorausgesetzt, dass der Bäcker oder die Bäckerin bei komplexen Schritten weiß, aus welchen Einzelschritten sie zusammengesetzt sind.

Programme werden also unter Zuhilfenahme einfacher Operationen geschrieben. Und dann schreiben wir noch größere Programme unter Zuhilfenahme dieser einfachen Operationen, anderer Programme und der eben eingeführten Bibliotheken. Darauf aufbauend schreiben wir noch größere Programme und so weiter. Wir können Programme in diesem Sinn aufeinanderstapeln.

Ich habe aber noch nicht erklärt, wie die Ergebnisse der einfachen zugrundeliegenden Operationen ausgerechnet werden. Und vielleicht haben Sie nicht vergessen, dass wir noch gar nicht ganz genau gesagt haben, was ein Prozessor nun tut. Das können wir jetzt verstehen: Der Prozessor stellt genau diese einfachen zugrundeliegenden Operationen zur Verfügung! Er kann auf der Hardware sehr einfache Operationen berechnen wie – Sie ahnen es – die Addition oder Subtraktion von eins und die Multiplikation mit zwei oder die Division durch zwei. Er kann überprüfen, ob ein Wert null ist, kann beliebige Stellen in einem Programm als Nächsten auszuführenden Schritt auswählen und so weiter. Für diese sehr einfachen, auf der Hardware ausgeführten Operationen gibt es wieder eine eigene Programmiersprache, die man Maschinensprache nennt. Die Maschinensprache für Prozessoren von Intel umfasst ca. 1000 mögliche einfache Operationen (Quelle).

Programme in anderen Programmiersprachen wie Java oder Python werden letztlich in Programme in Maschinensprache übersetzt. Das ist tatsächlich im Sinn einer Übersetzung von Deutsch nach Englisch zu verstehen.

Vielleicht haben Sie das Wort `Compiler` schon einmal gehört: Compiler erledigen genau diese Übersetzung in Maschinensprache. Die Idee ist ganz ähnlich zur Verwendung des `summe`-Programms im `quadrat2`-Programm in der Zeile `z ← summe(z,x).` Vielleicht erinnern Sie sich an die Bemerkung zur prinzipiellen Ähnlichkeit von Bibliotheken und unterschiedlichen Programmiersprachen. Die Verwendung des Worts `summe` steht ja für die Codezeilen, die das `summe`-Programm implementieren. In Wahrheit funktioniert es ein kleines bisschen anders, aber man kann sich das so vorstellen, dass das Wort `summe` im `quadrat2`-Programm in genau die Codezeilen übersetzt wird, die dem `summe`-Programm entsprechen. Und ganz genauso stehen die Wiederholungsanweisung `while`, die Zuweisung ←, Addition und Subtraktion von 1 sowie die Ausgabefunktion `output` für kleine eigenständige Programme in Maschinensprache, in die sie tatsächlich auch übersetzt werden.

Verblüffenderweise lassen sich E-Mail-Programme, Bäckereisteuerungen und alle anderen Funktionen in diese einfache Maschinensprache übersetzen. Und ein Prozessor kann dann Programme in Maschinensprache direkt in der Hardware ausführen. Das heißt, der Prozessor erhält Eingabesignale in Form von „Strom“ und „kein Strom“, die beispielsweise die Addition von 1 zu einer Zahl 7 repräsentieren, und er liefert Ausgabesignale ebenfalls in Form von „Strom“ und „kein Strom“, die dann als das Ergebnis der Addition interpretiert werden. Das sind die berühmten Einsen und Nullen, die wir alle mit Computern verbinden.

Dass man mit Einsen und Nullen beliebige Zahlen darstellen kann, ist seit dem 17. Jahrhundert bekannt. Uns führt das hier zu weit, aber die Idee können wir kurz verstehen. Stellen Sie sich das so vor: Wir Menschen benutzen zur Darstellung von Zahlen zehn Ziffern, nämlich 0, 1, 2, …, 9. Wenn wir die Zehn darstellen wollen, benötigen wir mehr als eine Ziffer: 1 und 0, also 10. Elf ist 1 und 1, also 11. Neunundneunzig ist 9 und 9, also 99. Wenn wir die Hundert darstellen wollen, benötigen wir drei Ziffern: 1 und 0 und 0, also 100. Und so weiter. Wenn wir nur Nullen und Einsen zur Verfügung haben, können wir dasselbe Prinzip verwenden: So wie wir Menschen eine Zehn nicht mit nur einer Ziffer darstellen können, können Computer eine Zwei nicht mit nur einer Ziffer darstellen. Also wird eine zweite Ziffer hinzugefügt, und 2 wird dargestellt als 1 und 0, also 10. 3 stellen wir als 1 und 1 dar, also 11. Für die Vier benötigen wir eine weitere Ziffer, so wie wir für die Hundert eine weitere benötigt haben: Vier wird als 1 und 0 und 0 geschrieben, also 100. Und so weiter. Die Zehn wird übrigens als 1010 dargestellt, die Hundert als 1100100.

Weil die Programmschritte, die man mit Maschinensprache beschreiben kann, sehr feingranular sind, werden die entsprechenden Programme sehr schnell sehr groß und allein deswegen schwierig zu verstehen und zu warten. In der Frühzeit der Computer wurden Programme tatsächlich direkt in Maschinensprache verfasst. Heute macht das eigentlich niemand mehr, sondern benutzt die für den Menschen besser verständlichen Programmiersprachen, die mit den angesprochenen Übersetzern für Programme in Maschinensprache übersetzt werden.

Drei Bemerkungen: Zustand, drei Anweisungen reichen aus; Vermenschlichung

Erlauben Sie mir noch drei Bemerkungen zum Schluss dieses Teils. Ich finde sie wichtig, aber wenn beim ersten Lesen dieses Abschnitts nicht alles klar wird, werden Sie den Rest trotzdem verstehen können.

Erstens: Die Menge aller Zwischenergebnisse zu einem bestimmten Zeitpunkt nennt man den Zustand eines Programms. Der Zustand wird während der Programmausführung im Speicher abgelegt. Jeder Schritt der Programmausführung verändert diesen Zustand, weil Zwischenergebnisse und in unserem Beispiel auch die Werte der Eingaben verändert werden. Oft gibt es sehr viele solcher möglichen Zustände. Deren große Anzahl macht es schwierig, Programme zu schreiben und zu verstehen. Es ist also nicht nur die Anzahl der Schritte, die ein Programm komplex werden lässt (die Software in einem Auto besteht heute aus Hunderten von Millionen Zeilen Code), sondern eben auch die Anzahl der möglichen Zustände.

Zweitens: Oben haben wir erläutert, dass Programmiersprachen sich unter anderem darin unterscheiden, wie sie bestimmte Sachverhalte in einem bestimmten Kontext besonders kurz und knapp ausdrücken können: Wir haben über „Fenster“ und „Knöpfe“ gesprochen und angedeutet, dass die Ansteuerung von Maschinen mit speziellen Programmiersprachen erfolgt, die das besonders gut ausdrücken können. In unseren Programmen oben haben wir drei solcher Sachverhalte einfach so verwendet, ohne das näher zu erklären, weil sie so natürlich sind: (1) das Ablegen von Werten oder Zwischenergebnissen im Speicher, das man deswegen auch speichern nennt – das sind die Zeilen mit dem Linkspfeil; (2) die Ausführung einer Zeile nach der nächsten – das wird durch das Semikolon am Ende signalisiert; und (3) die Wiederholung – die `while`-Anweisung, die wir

schon erklärt haben. Ich finde das nach wie vor wirklich sehr verblüffend: Seit den 1960ern ist bekannt, dass alle Funktionen, die man überhaupt berechnen kann, sich mit diesen drei Sachverhalten zuzüglich Addition und Subtraktion programmieren lassen. Alle!

Warum dann der eigenartige Einschub zu Funktionen, die man überhaupt berechnen kann? Verrückterweise gibt es auch Funktionen, die man nicht berechnen kann, was kaum vorstellbar ist! Ein berühmtes Beispiel ist eine Funktion F, die als Eingabe ein Programm P erhält und für jedes „Eingabeprogramm" P ausrechnen soll, ob dieses Programm P für jede Eingabe E irgendwann mit seiner Berechnung fertig wird oder ob es passieren kann, dass das Programm unendlich lange abläuft. Wir werden gleich sehen, dass das für unsere Beispiele der Quadrat- und Summenprogramme oben im Fall der Eingabe negativer Zahlen der Fall ist. Eine solche Funktion F, die für alle Programme P und alle Eingaben E ausrechnet, ob die Berechnung irgendwann aufhört, sieht also so aus: F(P)= „ja", falls P(E) für alle Eingaben E irgendwann fertig wird, und andernfalls „nein". Für eine solche Funktion gibt es kein Programm! Man weiß das seit den 1930ern, aber darum wollen wir uns hier nicht weiter kümmern.

Drittens: Vielleicht ist Ihnen aufgefallen, dass wir im gesamten Text ganz intuitiv für Maschinen Tätigkeitswörter verwendet haben, die man eigentlich eher bei Menschen oder allgemein Lebewesen sehen würde: „auf einen Reiz reagieren", „interagieren", „kommunizieren", „erkennen", „berechnen", „lernen" und so weiter. Philosophen haben viel darüber nachgedacht. Brisant wird das, wenn man annimmt oder sprachlich suggeriert, dass Maschinen über Sachverhalte „entscheiden" und gegebenenfalls sogar „verantwortlich" oder „haftbar" sein können. Mit Ausnahme von Verantwortlichkeit und Haftung finde ich diese sprachliche Vermenschlichung eingängig und völlig in Ordnung. Verheimlichen will ich aber nicht, dass es Denkschulen im Zusammenhang mit dem sogenannten Transhumanismus gibt, die Maschinen irgendwann echte Intelligenz oder sogar Emotionen oder Verantwortung zuerkennen. Ich für meinen Teil halte das, kurz gesagt, für abwegig.

Präzision beim Programmieren; Fehler

Vielleicht haben auch Sie schon einmal ein Kuchenrezept ausprobiert, das Sie nicht ganz verstanden oder missverstanden haben, und sich dann bei genauerer Betrachtung des Ergebnisses dazu entschieden, den Kuchen vielleicht doch lieber wegzuwerfen als ihn zu essen. Manchmal liegt das an einem selbst. Manchmal liegt es aber auch daran, dass Rezepte nicht präzise genug beschrieben sind oder dass sie schlicht Fehler enthalten.

Vielleicht ist Ihnen das aufgefallen: Wenn die Eingabe y für unsere Summenfunktion oder die Eingabe x für unsere Quadratfunktion oben eine negative Zahl ist, wird das Programm niemals fertig. Denn eine negative Zahl minus eins kann niemals null ergeben: Während der Ausführung des Programms subtrahieren wir unendlich oft minus eins. Das Problem ist, dass wir vergessen haben, diesen Spezialfall zu berücksichtigen. Wir waren nicht hinreichend präzise. (Der Fehler lässt sich in beiden Programmen dadurch beheben, dass wir das ≠ in der `while`-Bedingung durch das Größer-Zeichen > ersetzen. Überzeugen Sie sich selbst. Und wenn Sie danach genau hinsehen, haben wir noch etwas vergessen: Wir haben im Programm nicht verlangt, dass die Eingaben ganze Zahlen sein müssen. Wenn man als Eingabe 4,2 eingibt, wird das Programm ebenfalls niemals fertig. Das Ersetzen von ≠ durch > behebt aber auch dieses Problem.)

Mangelnde Präzision in dieser und anderen Ausprägungen ist ein großes Problem beim Programmieren, und unter anderem deswegen ist Programmieren nicht immer ganz so einfach. In einem sehr lustigen Videoclip (Exact Instructions Challenge: This is why my children hate me) wird das Problem an einem Beispiel sehr schön klar: Ein Vater bittet seine kleinen Kinder, genau aufzuschreiben, welche Schritte er durchführen soll, um

ihnen ein Erdnussbutter-Marmeladen-Sandwich zu schmieren. Probieren Sie das ruhig einmal mit Ihren Kindern aus! Es stellt sich schnell heraus, dass das schwieriger ist als gedacht. Beispielsweise missversteht der Vater, natürlich absichtlich, die Anweisung „Erdnussbutter auf den Toast streichen“ so, dass er die Erdnussbutter auf eine Kante des Brots streicht anstatt auf die Schnittfläche. Oder er benutzt seine Hände, um Marmelade aus dem Glas zu holen, weil die Kinder vergessen haben, dazu den Einsatz eines Löffels vorzuschlagen. Oder er scheitert schon zu Beginn, weil es keine Anweisung gab, die Kühlschranktür vor Entnahme des Marmeladenglases zu öffnen.

Teil III: Wie funktioniert Maschinenlernen?

Programmieren und KI: Maschinenlernen aus Beispielen

Die einzelnen Schritte eines Programms so präzise zu beschreiben, dass nichts vergessen und nichts missverstanden wird und nichts falsch ist, ist die Aufgabe von Programmiererinnen und Programmierern. Wenn Sie das Sandwich-Spiel einmal mit Ihren Kindern spielen, werden Sie plötzlich nachvollziehen können, warum die Programme, die Sie benutzen, ab und zu fehlerhaft sind. Missverstehen Sie mich bitte nicht: Das ist keine Entschuldigung. Programme sollten natürlich fehlerfrei sein – ich will nur sagen, dass das nicht so einfach ist.

In der Tat ist dieser Sachverhalt einer der Gründe, warum die sogenannte Künstliche Intelligenz, genauer: eine ihrer Spielarten, nämlich das Maschinenlernen, sich aktuell so großer Popularität erfreut. Dort ist der Ansatz ein anderer. Anstatt jeden einzelnen Schritt genau zu beschreiben, definiert man stattdessen Programme durch eine große Menge an Beispielen. Das erklärt auch, warum man die entsprechenden Techniken Maschinenlernen nennt: Wir Menschen lernen ja auch besonders gut anhand von Beispielen.

In unserem Zusammenhang bestehen Beispiele aus Ein- und Ausgaben, die wir durch das → - Symbol voneinander trennen werden. Anstatt also die einzelnen Schritte beim Kuchenbacken zu beschreiben, betrachtet man die Beispiele (`Milch, Mehl, Zucker, Kakao, Butter, Eier`) → `Marmorkuchen` und (`Mehl, Hefe, Salz, Zucker, Äpfel`) → `Apfelkuchen` und (`Mehl, Hefe, Salz, Zucker, Zwetschgen`) → `Zwetschgenkuchen` und so weiter. Nehmen wir weiterhin an, dass es Zutatenlisten für Kompott gibt, also (`Äpfel, Zucker, Zimt`) → `Apfelkompott` und (`Birnen, Zucker, Vanillezucker, Nelke`) → `Birnenkompott`. Beim Maschinenlernen wird nun eine Funktion ermittelt, die für Eingaben, die „zwischen“ den Eingaben der Beispiele liegen, Ausgaben berechnet, die möglichst nahe bei oder zwischen den Ausgaben der entsprechenden Beispiele liegen. Für eine neue Zutatenliste (`Mehl, Hefe, Zucker, Birnen`), die nicht Teil der Beispiele war, findet die gelernte Funktion die Beispiele, deren Zutaten dieser Liste am ähnlichsten sind. Hier sind das wohl die Zutaten für Apfel- und Zwetschgenkuchen sowie Birnenkompott. Als Ausgabe gibt die gelernte Funktion etwas aus, das zwischen den zu diesen Eingaben gehörigen Ausgaben liegt – hier vielleicht Birnenkuchen.

Im Maschinenlernen unterscheidet man zwei Phasen. Die erste Phase ist das Lernen eines Sachverhalts, das in ein sogenanntes gelerntes Modell mündet, das diesen Sachverhalt abbildet. Nach dem Lernen nutzt man in der zweiten Phase das gelernte Modell wie ein Programm, indem man es mit Eingaben füttert und eine Ausgabe erhält.

Traditionell unterscheidet man die folgenden Aufgaben des Maschinenlernens: Klassifizierung, Vorhersage, Gruppierung und Assoziation.

1. Bei der Klassifizierung ordnet man Eingabedaten einer Klasse zu: Eingabebilder von Tieren etwa werden als „Hund“, „Katze“ oder „Meerschweinchen“ erkannt. Einfache Erkennungssysteme für Fußgänger ordnen einem Kamerabild die Klasse „Fußgänger“ oder „kein Fußgänger“ zu.
2. Bei der Vorhersage soll das Ergebnis keine vorher festgelegte Klasse sein, sondern ein Wert, wenn etwa der Preis eines Hauses anhand seiner Quadratmeterzahl, Lage und seines Alter vorhergesagt werden soll. Unser Birnenkuchen oben fällt auch in diese Kategorie.
3. Bei der Gruppierung (Clustering) will man „ähnliche“ Objekte in Klassen zusammenfassen, ohne die Klassen vorher zu kennen: Bilder von Tieren werden so gruppiert, dass möglichst alle Hunde, alle Katzen und alle Meerschweinchen in jeweils einer Klasse liegen, ohne dass man die Kategorien „Katze“, „Hund“, „Meerschweinchen“ vorher kennt.
4. Bei der Assoziation schließlich versucht man zu verstehen, was die Faktoren sind, die in der Vergangenheit wesentlich zu einem bestimmten Ergebnis beigetragen haben. Beispiele sind Faktoren für den Onlinekauf eines bestimmten Produkts, vielleicht die Verwendung eines iPhones statt eines Android-Smartphones beim Einkauf, andere selbst vorher gekaufte Produkte, andere von Freunden vorher gekaufte Produkte, Uhrzeit des Einkaufs und so weiter.

Es gibt sehr viele unterschiedliche Ansätze, diese Probleme technisch zu lösen. Programme hingegen funktionieren im Wesentlichen immer so, wie ich das oben beschrieben habe. Das erklärt, warum wir in diesem Beitrag den Begriff des Programms einigermaßen detailliert einführen können: Der ist nämlich ziemlich eindeutig definiert! Weil es aber andererseits so extrem viele sehr unterschiedliche Verfahren des Maschinenlernens gibt (hier finden Sie ein Überblicksbild), müssen wir beim Maschinenlernen gezwungenermaßen stärker an der Oberfläche bleiben.

Wir müssen kurz noch die Welten des Maschinenlernens und des Programmierens zusammenbringen. Beim Programmieren überlegt sich ein Mensch Schritt für Schritt, wie ein Problem gelöst werden soll. Das Ergebnis ist ein vom Menschen erstelltes Programm. Beim Maschinenlernen ist das Ergebnis das erwähnte von einer Maschine erzeugte Modell, das den gelernten Sachverhalt beschreibt. Jetzt wird es ein bisschen kompliziert. Das Maschinenlernen selbst besteht aus einzelnen Schritten und ist in diesem Sinn ein Programm wie jedes andere: Eingaben sind Beispiele, Ausgabe ist das Modell. Die Ausgabe, also das Modell, kann man nun auch als Programm verstehen, weil es eine Funktion berechnet, zum Beispiel wie einem Bild eine Tierart zugeordnet wird. Diese Modelle sind aber etwas anders als die Programme, die wir bisher kennengelernt haben. Das Modell besteht nicht aus einzelnen zielgerichteten Schritten, durch die ein Mensch verstehen könnte, wie man dem Ziel näher kommt. Das war im Gegensatz dazu in unserem Summen- und Quadratprogramm oben durchaus der Fall. Wie genau stattdessen ein Modell das Problem löst, ist eben abhängig von der Variante des Maschinenlernens, für die man sich entschieden hat.

In diesem Sinn versucht Maschinenlernen, die Aufgabe des Programmierers zu übernehmen. Weil es auch für Programmiererinnen und Programmierer sehr unterschiedliche mögliche Lösungen für ein Problem gibt, ist eigentlich zu erwarten, dass auch Maschinenlernen sehr unterschiedliche Lösungen, also Modelle, finden kann. Und genau das ist der Fall.

Warum Maschinenlernen?

Bitte machen Sie sich klar, dass für alle vier genannten Anwendungen des Maschinenlernens die oben gemachte Beobachtung gilt, dass sie sich jeweils als Funktionen darstellen lassen. Zudem kann man sehen, dass in allen Fällen entweder die Aufgabe schwer zu präzisieren ist: „Identifiziere Ähnlichkeiten“ – wann sind zwei Hunde denn ähnlich, und wie unterscheiden sich Hunde von Katzen? „Ordne einer Klasse zu“ – was

macht einen Hund denn zum Hund und eine Katze zu einer Katze? Oder die Zusammenhänge sind komplex („Was ist der Preis eines Hauses?“), oder man hat sie noch nicht genau verstanden („Identifiziere die Faktoren für eine Kaufentscheidung“).

Maschinenlernen wird aber nicht nur verwendet, weil es manchmal einfacher ist als das explizite Formulieren von Programmen. Bisweilen funktioniert es auch ganz einfach besser. Das kann man gut am Beispiel der automatischen Übersetzung natürlicher Sprachen sehen, etwa von Deutsch nach Englisch. Jahrzehntelang hat man versucht, die Regeln der beiden Grammatiken aufeinander abzubilden und hat damit letztlich nicht immer überzeugende Resultate erzielen können. Mit Maschinenlernen funktioniert das viel besser, wie das Beispiel der Übersetzungsmaschine DeepL sehr eindrücklich demonstriert – probieren Sie es einmal aus!

Ein anderes Beispiel ist die Objekterkennung in Bildern, etwa von Fußgängerinnen und Fußgängern in Kamerabildern, die von autonomen Fahrzeugen aufgezeichnet werden. Versuchen Sie einmal, genau wie im Sandwich-Fall, ganz präzise zu beschreiben, wie man einen Fußgänger erkennen kann, ohne Konzepte wie „Mensch“ oder „Kind“ oder „Schnee“ oder „Regenschirm“ zu benutzen. Diese Konzepte kennen Maschinen ja zunächst nicht. Denken Sie dabei daran, dass es Kinder und Erwachsene gibt; dass zu Fuß Gehende sich unterschiedlich schnell bewegen; dass sie Einkaufstaschen tragen und Regenschirme benutzen und zu Karneval Hühnerkostüme tragen können; dass sie in Grüppchen auftreten können und sich gegenseitig teilweise verdecken; dass sie auch von Autos verdeckt werden können; dass es unterschiedliche Licht- und Wetterverhältnisse mit Sonne, Regen und Schnee gibt; und dass zu Fuß Gehende wie aus dem Nichts auftauchen können, wenn ein Kind hinter einem Auto hervorspringt, um seinen Ball zu holen. Das ist noch schwieriger als das Erdnussbutter-Marmeladen-Sandwich!

Schließlich ist ein weiterer Grund für die Verwendung von Maschinenlernen die Tatsache, dass Programme, die solche maschinengelernten Funktionen implementieren, manchmal viel schneller ablaufen können als herkömmliche Programme.

Stolpersteine

Allerdings gibt es auch durchaus gravierende Nachteile des Maschinenlernens, über die in der aktuellen Debatte häufig großzügig hinweggesehen wird: Ob nämlich die so gefundene Ausgabe wirklich die richtige ist, das weiß keiner – und noch schlimmer: Das kann auch keiner genau wissen! Denn wir haben ganz bewusst mit Beispielen gelernt und die einzelnen Schritte nirgendwo explizit aufgeschrieben. Das ist ja gerade der Gag! Deswegen können die Ergebnisse aber auf dieser Ebene auch nicht überprüft werden. Kein Wunder also, dass es die aktuell boomende Forschungsrichtung der „erklärbaren KI“ gibt.

Beim Maschinenlernen werden häufig nicht dieselben Zusammenhänge gelernt, die ein Mensch zum Verständnis der Welt zugrunde legt. Bei der Aufgabe „unterscheide Hund von Katze“ achten wir Menschen vielleicht auf die Größe, die Form der Augen, den Schwanz, Schnurrbarthaare und so weiter. Maschinen hingegen lernen oft irgendwelche Zusammenhänge, die uns Menschen gar nicht relevant erscheinen. Verblüffenderweise funktioniert das in der Praxis trotzdem oft sehr gut. Dazu passt die bekannte Beobachtung, dass in der Regel durch das Verändern nur eines einzelnen Bildpunkts – was ein Mensch gar nicht wahrnehmen kann – aus einer richtigen Klassifizierung des Bildes als „Hund“ eine falsche „Katze“ wird. Die Frage ist dann, wie schlimm das in der Praxis ist.

Die Tatsache, dass das Maschinenlernen sich aktuell so großer Popularität erfreut, liegt auch an den großen Fortschritten in den Lernverfahren. Der Hauptgrund ist aber wohl die heute oft viel einfachere Verfügbarkeit großer Mengen von Daten. Aber täuschen Sie sich nicht: So einfach ist auch das wieder nicht. Daten sind

nämlich in der Praxis oft unvollständig und fehlerhaft oder nicht repräsentativ oder eben doch nur in relativ kleinen Mengen verfügbar. Und für die heute prominentesten Lernverfahren (vielleicht haben Sie schon einmal etwas von Deep Learning gehört), benötigt man sehr große Mengen von Beispielen, die in manchen Fällen vorliegen, etwa das Kaufverhalten bei Amazon, in vielen Fällen jedoch nicht, etwa Sicherheitsangriffe gegen Automobile. Es gibt viele andere Ansätze, auch solche, die mit kleineren Beispielmengen auszukommen versuchen, aber auch das führt uns hier zu weit. Falls Sie das interessiert, finden Sie unter anderem bei Kaggle (Link) viele öffentlich verfügbare Daten, mit denen Maschinenlernverfahren ausprobiert und überprüft werden können.

Manchmal ist es so, dass das Lernen von Funktionen sehr viel Energie verbrauchen kann (hier nur eine Quelle zu dieser erregt geführten Diskussion). Da die „Berechnung“ der Funktion nach dem Lernen des Modells manchmal hingegen sehr wenig Energie verbraucht, muss man das natürlich in Bezug setzen zu alternativen Implementierungsformen, etwa der Programmierung von Hand. Auch das führt uns hier aber zu weit.

Schließlich besteht unter vielen Forscherinnen und Forschern heute Konsens, dass rein beispielbasierte Lernverfahren nicht das Mittel der Wahl sind, wenn man den zu lernenden Ausschnitt der Realität schon gut verstanden hat – etwa die Schwerkraft, Strömungen von Flüssigkeiten oder das Verhalten elektrischer Felder in bestimmten Kontexten: Es ist nicht sinnvoll zu lernen, was man schon weiß. Maschinenlernen ist also auch aus diesem Grund nicht der eine Hammer, der sinnvoll ist für alle Nägel. Große Anstrengungen werden heute unternommen, die Welt der expliziten Regeln und Gesetze mit der beispielbasierten Welt des Maschinenlernens zu verheiraten.

Auf das aktuell kontrovers diskutierte Thema der Ethik im Maschinenlernen gehen wir später nur kurz ein, auch das würde hier zu weit führen. Heiß debattiert wird aktuell die Problematik, dass eine maschinengelernte Funktion zwar im Schnitt sehr gut sein kann, aber für einzelne kleine Untergruppen von Eingaben sehr schlecht, wie beispielsweise bei der automatischen Gesichtserkennung, wenn dadurch mitunter gesellschaftliche Minderheiten diskriminiert werden.

In der aktuellen Debatte schließlich drängt sich bisweilen das Gefühl auf, als wären mit KI beziehungsweise Maschinenlernen alle Probleme der Informatik abschließend gelöst. Das ist natürlich nicht der Fall. Insgesamt funktioniert Maschinenlernen in der Tat oft gut, aber eben auch nicht immer. Wenn man aus Beispielen lernt, ist man zwangsläufig mit den diskutierten Schwierigkeiten konfrontiert. Deswegen ist Maschinenlernen genau wie das klassische Programmieren nur ein Werkzeug im Werkzeugkasten der Informatik: Maschinenlernen kann das Programmieren nicht ersetzen, aber sinnvoll ergänzen. Und wie wir gleich sehen werden, gehört zu Software viel mehr als das „Programmieren“ von Funktionen, ob das nun durch einen Menschen geschieht oder durch Maschinenlernen.

Teil IV: Software: Wie erstellt man Programme?

Rückblick zur Auffrischung

Erinnern wir uns an dieser Stelle noch einmal, was wir bis jetzt gelernt haben. Wir wollen Eingaben in Ausgaben umwandeln. Diese Ein- und Ausgaben sind Daten: Zahlen, Sensordaten, E-Mails, Suchanfragen, Tastatureingaben, Kamerabilder, Sprache und so weiter. Deswegen gibt es im Deutschen das hinreißend altmodische Wort der elektronischen Datenverarbeitung (EDV) oder auch, unsauberer, Informationsverarbeitung. Der Zusammenhang von Ein- und Ausgabe ist eine Funktion im mathematischen Sinn, denn aus der Schule erinnern wir uns, dass eine Funktion ja immer Ein- auf Ausgaben abbildet. Wie

genau der Wert einer Funktion berechnet wird, das müssen sich Programmiererinnen und Programmierer überlegen. Die Berechnung kann durch Modelle erfolgen, die beispielbasiert per Maschinenlernen automatisch erstellt wurden. Oder sie kann durch Angabe eines Algorithmus erfolgen, der das Problem im Wesentlichen dadurch löst, dass es in einzelne Schritte zerlegt wird. Den Algorithmus formulieren die Programmiererinnen und Programmierer dann unter Hinzufügen von Details als Programm in einer dem Problem angemessenen Programmiersprache, das auf einem Prozessor unter Verwendung von Speicher, Netzwerkanschluss und weiterer Hardware ausgeführt wird. Schließlich haben wir ein zentrales Prinzip der Konstruktion von Software kennengelernt: Große Probleme werden in kleine Probleme zerlegt, die unabhängig voneinander gelöst werden, um die Lösungen dann zusammenzusetzen: Anweisungen in einer Programmiersprache wie die Wiederholungsanweisung lassen sich aus Anweisungen zusammensetzen, die ein Prozessor verstehen kann. In einem Programm kann man andere Funktionalitäten verwenden, die man selbst programmiert hat, wie etwa die Summenfunktion. Oder man setzt große Programme aus vielen kleinen Programmen zusammen, die in Bibliotheken zur Verfügung stehen.

Es ist nicht so, dass für ein gegebenes Problem nur ein einziger Algorithmus existiert, der dann auch nur auf eine einzige Art und Weise in ein Programm umgesetzt werden kann. Ganz im Gegenteil. Für das beispielhafte Problem „Sortieren einer Zahlenfolge“, das für Informatikerinnen und Informatiker immer noch ganz wesentlich ist, gibt es erstens Dutzende von Algorithmen mit unterschiedlichen Eigenschaften bezüglich des notwendigen Speichers für Zwischenergebnisse und der für die Berechnung erforderlichen Zeit und Energie. Zweitens kann man für jeden Algorithmus nicht nur wegen der freien Wahl einer Programmiersprache ganz unterschiedliche Programme schreiben, die denselben Algorithmus implementieren. Und wir haben oben gesehen, dass mit Maschinenlernen noch eine ganz andere Art unterschiedlicher Lösungen für dasselbe Problem gefunden werden kann.

Wann ist ein Programm gut?

Wenn es unterschiedliche Programme gibt, die denselben Algorithmus implementieren, und wenn es unterschiedliche Algorithmen gibt, die dasselbe Problem lösen, dann stellt sich die Frage, worin diese Programme und gegebenenfalls Algorithmen sich eigentlich unterscheiden. Algorithmen sind ja auf einer etwas höheren Ebene formulierte Anweisungen, wie ein Problem gelöst werden kann. Auf der Ebene von Algorithmen interessieren sich Informatikerinnen und Informatiker zunächst dafür, ob diese Algorithmen das gegebene Problem wirklich in allen Fällen lösen. Das war ja für unsere Berechnung von Quadrat- und Summenfunktion nicht der Fall, wenn wir uns an den Fall negativer Eingabewerte erinnern. Wenn das Problem immer richtig gelöst wird, nennen wir den Algorithmus, und dann auch das Programm, korrekt. Korrektheit ist so wichtig und gerade im Fall von Maschinenlernen so inhärent schwierig, dass wir weiter unten noch einmal darauf zurückkommen werden. Unser erstes Gütekriterium ist also Korrektheit, die man durch null fehlerhafte Berechnungen oder null fehlerhafte Programmschritte charakterisieren kann.

Beim Maschinenlernen begegnet uns die Korrektheit in etwas anderer Form: Wie gut ist die Erkennung oder die Klassifizierung? Beim Erkennen von Hunden und Katzen beispielsweise sollen einerseits alle Hunde als Hunde erkannt werden, andererseits aber keine Katzen fälschlich als Hunde. Das ist nicht dasselbe. Es gibt gute Ansätze, diese doppelte Qualität zu messen. Wie das Maschinenlernen selbst setzen sie allerdings oft eine große Datenmenge voraus – und die hat man eben nicht immer zur Verfügung. Im Fall der Gruppierung (dem Clustering) ist es sogar oft so, dass man vorab gar nicht weiß, ob die gefundenen Gruppen sinnvoll sind – denn wüsste man das schon, würde man wiederum oft gar kein Maschinenlernen benötigen.

Informatikerinnen und Informatiker interessieren sich dann insbesondere dafür, wie viel Speicher Algorithmen beziehungsweise die sie implementierenden Programme benötigen, wie viel Zeit die Ausführung in Anspruch nimmt und auch dafür, wie viel Energie sie verbrauchen. Speicher ist vergleichsweise teuer, deswegen ist weniger mehr. Intuitiv ist klar, dass eine schnellere Problemlösung im Normalfall einer langsamen vorzuziehen ist. Und dass weniger Energieverbrauch besser ist als höherer, leuchtet auch unmittelbar ein: Mit 200 Suchanfragen bei Google benötigen Sie die gleiche Menge Strom wie für das Bügeln eines Hemdes [Quelle; siehe zum Gesamtstromverbrauch von Google auch diesen Link]. Bei geschätzt 63.000 Suchanfragen pro Sekunde im Jahr 2020 [Quelle] ist es unbedingt erforderlich, hier so energiesparsam wie möglich zu programmieren, was Informatikerinnen und Informatiker natürlich auch tun. Speicher-, Zeit- und Energiebedarf sind also ein zweiter Satz wichtiger Gütekriterien.

Schließlich gibt es noch eine ganze Reihe anderer Kriterien für die Güte von Programmen. Einerseits sind das Eigenschaften, die die Nutzerinnen und Nutzer eines Programms erleben: Ist ein Programm sicher in dem Sinn, dass ein Angreifer das Programm oder seine Eingabe- oder Ausgabedaten oder Zwischenergebnisse nicht sehen oder verändern kann? Ist es sicher in dem Sinn, dass es etwa im Fall von Robotern der Umwelt keinen Schaden zufügt? Gewährt es Privatheit? Ist es einfach zu verwenden? Macht die Verwendung Spaß? Und dann gibt es Kriterien, die aus Sicht der Ingenieurinnen und Ingenieure relevant sind: Angesichts der Tatsache, dass Programme oft jahrzehntelang „leben“, ist es entscheidend, ob das Programm gut wartbar ist. Sind Änderungen einfach durchzuführen? Ist es leicht verständlich? Ist es einfach, von einer Computerhardware auf eine andere zu übertragen, was leider gar nicht selbstverständlich ist?

Software trifft zunehmend Entscheidungen, die uns alle betreffen – im autonomen Fahrzeug, bei der Gewährung von Krediten, in Ampelschaltungen, in der Medizin oder in der Polizeiarbeit. Dass das Wort „Entscheidung“ hier in die Irre führen kann, weil es Verantwortung suggeriert, haben wir schon kurz angedeutet. Am bidt kümmern wir uns deswegen auch um einen weiteren Aspekt von Güte, der sich auf ethisch wünschenswerte Überlegungen stützt und deswegen noch viel schwieriger zu bewerten ist als die anderen Gütekriterien. Leider führt auch das hier zu weit, und wir laden Sie ein, sich über unser Projekt zu Ethik in der Softwareentwicklung (und nicht nur dem Maschinenlernen!) am bidt zu informieren.

Software wird weiterhin in der Regel von Unternehmen entwickelt, die ein Interesse daran haben müssen, möglichst schnell zu entwickeln. Sonst ist die Konkurrenz schon da und hat viele Nutzerinnen und Nutzer an sich gebunden; an anderer Stelle werden wir diesen „winner takes it all“-Sachverhalt betrachten. Natürlich wollen Unternehmen auch die Kosten möglichst gering halten. Es stellt sich jetzt leider schnell heraus, dass die oben genannten Kriterien, unter anderem Korrektheit, Ressourcenverbrauch, Sicherheit und Privatheit, Nutzbarkeit und Wartbarkeit sowie Kosten und Entwicklungszeit häufig miteinander in Konflikt stehen. Gute Nutzbarkeit kollidiert oft mit hoher Sicherheit, hohe Sicherheit kollidiert bisweilen mit schneller Programmausführung; gute Wartbarkeit kann mit schneller Entwicklungszeit kollidieren und so weiter. Zielkonflikte sind ganz normal und bestimmen unser Leben: Denken Sie nur an die verschiedenen Faktoren, die eine Coronastrategie beeinflussen müssen.

Die Güte eines Programms ist also eine Kombination aus den genannten Faktoren. Dabei gibt es nicht die eine goldene Kombination, die für alle Programme optimal wäre. Software ist sehr stark abhängig von und verwoben mit dem Entwicklungs- und Einsatzkontext: Medizintechnische Produkte, Kläranlagensteuerungen, Pizza-Lieferdienst-Apps, autonome Fahrzeuge, Gartenbewässerungsanlagen und so weiter weisen ganz offensichtlich sehr unterschiedliche Anforderungen an die Güte der entsprechenden Software auf. Wir kommen später darauf zurück, wie man die unterschiedlichen Anforderungen an Güte erfüllen kann: Das ist genau eine

der zentralen Aufgaben des Software Engineering, derjenigen Disziplin, die sich um das Erstellen und Warten im mehrfachen Sinn „guter“ Software kümmert.

Was soll ein Programm tun: Anforderungen

Bevor wir zum Abschluss erklären, was Software Engineering ist und warum Software letztlich doch mehr als Programme und sehr viel mehr als Maschinenlernen ist, möchte ich auf das oben angesprochene Problem der Korrektheit zurückkommen. Erinnern wir uns: Korrektheit eines Programms ist dann gegeben, wenn es das tut, was es tun soll, wenn es also ein gegebenes Problem löst. Damit ergibt sich sofort, dass Korrektheit kein absolutes Konzept ist: Korrektheit kann immer nur mit Bezug zu dem gedacht werden, was eigentlich erwünscht ist. Informatikerinnen und Informatiker unterscheiden hier zwischen dem Soll- und dem Istverhalten eines Systems: Ersteres ist erwünscht. Zweiteres ist das, was das Programm wirklich tut, wenn es ausgeführt wird. Idealerweise sind Soll- und Istverhalten identisch.

Wir müssen vor Formulierung des Sollverhaltens genau verstehen, welches Bedürfnis wir adressieren und welches Problem wir eigentlich lösen wollen. Oft passieren in der Systementwicklung hier schon die ersten Fehler. Vielleicht kann das folgende Beispiel das illustrieren: Vor einiger Zeit habe ich mit meiner Familie in den USA gelebt. Wir hatten kein Auto, mussten also das Problem lösen, zum Supermarkt zu kommen. Sie werden das als eher unerfreuliches Problem wiedererkennen, wenn Sie selbst schon einmal mit Rucksack und schreienden Kleinkindern zu Fuß circa drei Kilometer zum Supermarkt gegangen und vollbeladen, schweißgebadet und am Rande des Nervenzusammenbruchs zurückgewankt sind. Wir haben überlegt, ein Fahrrad anzuschaffen, sich gerade entwickelnde Carsharing-Angebote wahrzunehmen oder schlicht in den sauren Apfel zu beißen und ein Taxi zu nehmen. Irgendwie war das alles nichts – und dann haben wir eines Tages den Lastwagen eines Lieferdienstes für Lebensmittel gesehen. In dem Moment fiel es uns wie Schuppen von den Augen: Das Problem war nicht, wie wir zum Supermarkt kommen. Das wirkliche Problem war, wie die Lebensmittel in unsere Wohnung kommen würden.

Im Nachhinein ist das offensichtlich. Aber vielleicht haben Sie selbst auch schon einmal festgestellt, dass Sie ein falsches Problem lösen wollten. Als Sie das dann verstanden haben, war auf einmal alles viel einfacher. Das richtige zu lösende Problem zu erkennen und zu verstehen wird in der Softwareindustrie als Requirements Engineering bezeichnet und stellt immer die erste große Hürde für ein Softwareprojekt dar. Requirements Engineering ist dabei eine Menge von Aktivitäten, die sich um das Erheben, Verstehen, Aufschreiben und Überprüfen von Bedürfnissen und Anforderungen kümmert; und wenn Sie schon einmal etwas von Design Thinking gehört haben, werden Sie sich erinnern, dass das Verständnis des richtigen zu lösenden Problems dort eine ganz große Rolle spielt. In der sogenannten agilen Softwareentwicklung spricht man deswegen auch während der Systementwicklung ununterbrochen mit der Auftraggeberin oder dem Auftraggeber beziehungsweise den zukünftigen Nutzerinnen und Nutzern eines Systems, um sicherzustellen, dass die richtige Lösung gebaut wird. Das ist auch deswegen eine gute Idee, weil die Anforderungen der Auftraggeberin oder des Auftraggebers nicht ein für alle Mal festgezurrt sind, sondern sich während der Entwicklungs- und Lebenszeit eines Softwareprodukts kontinuierlich und deutlich verändern. Es gibt Tausende Beispiele für Softwareentwicklungsprojekte, die deswegen gescheitert sind, weil die Bedürfnisse und Anforderungen nicht richtig verstanden, nicht richtig aufgeschrieben und während der unterschiedlichen Aktivitäten der Entwicklung nicht richtig kommuniziert wurden. Jede Informatikerin, jeder Informatiker kennt die Analogie der missglückten Entwicklung einer Schaukel, die bildhaft sehr schön dargestellt werden kann (Link).

Wenn das Bedürfnis und das richtige zu lösende Problem identifiziert sind, kann man in einem zweiten Schritt darüber nachdenken, die entsprechenden Anforderungen aufzuschreiben. Warum? Weil man dann den

Entwicklungsprozess strukturieren, die Arbeit in Teams aufteilen und vor allem am Ende die Korrektheit überprüfen kann. Wenn es keine klaren Anforderungen gibt, die das Sollverhalten beschreiben, dann kann es eigentlich keine Korrektheit geben. Natürlich können die Nutzerinnen und Nutzer eines Programms dieses dann benutzen und sich ein Urteil erlauben, ob es das tut, was es tun soll – aber das ist hochgradig unsystematisch, und eigentlich möchte man sie nicht mit Programmen belästigen, bei denen man weiß, dass sie noch sehr unausgegoren sind.

Fehler in Programmen finden: Testen

Um schon etwas früher überprüfen zu können, ob das System das tut, was es tun soll, geht man in der Praxis etwas anders vor: Man geht davon aus, dass die richtigen Anforderungen richtig notiert sind, und darauf aufbauend testen schon die Entwicklerinnen und Entwickler das Programm. Testen bedeutet, dass man sich für einige wenige repräsentative Eingaben vorab überlegt, was die erwünschte Ausgabe des Programms sein soll, also das Sollverhalten für diese Eingabe. Die erwünschte Ausgabe lässt sich aus den Anforderungen ableiten. Dann führt man das Programm mit genau dieser Eingabe aus und vergleicht die Ausgabe des Programms mit der Ausgabe, die man sich gewünscht hätte.

Klingt einfach, ist aber in der Praxis sehr schwierig. Denn „repräsentative“ Eingaben und damit „gute“ Testfälle zu finden, wie wir das verlangt haben, ist aus verschiedenen Gründen außerordentlich herausfordernd. Einer der Gründe ist die unglaubliche Menge möglicher Eingaben: Wenn Sie nur eine ganze Zahl als Eingabe haben, wie in unserem Quadratfunktionsbeispiel oben, sind das schon 2^{64} Möglichkeiten, das ist eine unvorstellbar große Zahl. Für die Summe aus zwei Summanden sind es schon 2^{128} Möglichkeiten. Im Universum gibt es geschätzt 2^{350} Atome, eine Zahl, die wir als Kombination von nur vier Zahlen als Eingaben schon überschreiten.

An dieser Stelle ist es interessant, noch einmal über das Erdnussbutter-Marmeladen-Sandwich und vor allem die Fußgängererkennung und das Maschinenlernen nachzudenken. Wir haben oben erläutert, dass Maschinenlernen unter anderem dann eingesetzt wird, wenn man den Lösungsweg nicht genau kennt. Gleichzeitig haben wir im Zusammenhang der Erkennung von Fußgängerinnen und Fußgängern gesehen, dass es fast unmöglich ist, das Konzept „Fußgänger“ ganz präzise zu fassen. Genau deswegen, haben wir argumentiert, wird in solchen Fällen Maschinenlernen eingesetzt. Wir benutzen Maschinenlernen also nicht nur dann, wenn das Wie schwierig zu fassen ist, sondern auch, wenn wir das Problem, also das Was, nicht genau beschreiben können. Hier passiert etwas wirklich Verrücktes: Für viele maschinengelernte Funktionen gibt es gar keine präzise Beschreibung des Sollverhaltens – denn wenn es die gäbe, hätten wir vielleicht nicht Maschinenlernen eingesetzt, sondern eine präzise Beschreibung des Sollverhaltens von Hand in die einzelnen Schritte eines Programms umgesetzt!

Wenn es aber nun keine präzise Beschreibung des Sollverhaltens gibt, wie können wir es dann systematisch testen? Die kurze Antwort ist: Das können wir zumindest im Allgemeinen nicht, und wir können es aus den genannten Gründen auch gar nicht können. Ingenieurinnen und Ingenieure verwenden verschiedene Tricks, um dieser Tatsache zu begegnen, aber hier sehen wir einen ganz frappanten Unterschied zwischen traditioneller Softwareentwicklung und Software, die auf Maschinenlernen basiert. Das erklärt auch, warum sich heute viele kluge Köpfe um das Problem der sogenannten Absicherung von KI kümmern, von der Sie im Zusammenhang mit autonomen Fahrzeugen vielleicht schon gelesen haben.

Neben dem Testen gibt es noch andere sehr nützliche Verfahren zum Finden von Fehlern, etwa das Lesen von Programmen, ohne sie auszuführen. In der Praxis zeigt sich, dass diese Verfahren sehr gut funktionieren. Nur der Vollständigkeit halber wollen wir festhalten, dass auch das leider für maschinengelernte Funktionen nicht

funktionieren kann, weil diese Funktionen ja gerade keine einzelnen Schritte beinhalten, die ein Mensch nachvollziehen könnte.

Strukturieren von Systemen: Softwaredesign

Bis jetzt haben wir drei Aktivitäten des Software Engineering kennengelernt: das sogenannte Requirements Engineering, das sich mit den durch ein Programm zu erfüllenden Bedürfnissen und Anforderungen beschäftigt; mit der Programmierung; und mit der Überprüfung von Programmen, dem Testen. Informatikerinnen und Informatiker unterscheiden bei der Konstruktion von Softwaresystemen manchmal zwischen dem sogenannten Programmieren im Kleinen und dem Programmieren im Großen. Die Art von Programmen, die wir bisher kennengelernt haben, implementieren im Kleinen einen Algorithmus oder basieren auf Maschinenlernen. Wir haben angenommen, dass wir für ein gegebenes Problem immer direkt ein Programm schreiben können. Wenn die Probleme jetzt aber sehr groß werden, werden auch die Programme sehr groß. Um dieser Komplexität Herr zu werden, müssen Probleme in Teilprobleme zerlegt werden, die ihrerseits so lange zerlegt werden müssen, bis handhabbare Teile entstehen. Das wird Programmieren im Großen genannt. Für die identifizierten Teile können dann individuell Lösungen in Form von Programmen implementiert werden, und die Teile werden dann hinterher zusammengesetzt (und das Zusammengesetzte muss seinerseits wieder getestet werden). Das haben wir auf einer eher feingranularen Ebene schon diskutiert, als wir oben die `summe`-Funktion in der `quadrat2`-Funktion verwendet haben. Hier geht es mir um eine gröbergranulare Ebene, etwa um das Auto, in dem heute ungefähr 100 Computer ihren Dienst versehen – auf denen jeweils mehrere Programme mit insgesamt Hunderten Millionen von Codezeilen ablaufen, deren Integration dann all die Funktionalitäten ergibt, die moderne Fahrzeuge anbieten. Wie man diese Funktionalität strukturiert, ist Aufgabe von Softwaredesignerinnen und -designern.

Der Punkt, auf den ich hinauswill, ist der folgende: Das Zerlegen eines Problems in Teilprobleme und das Zerlegen eines großen Systems in handhabbare Teilsysteme sind als solche kreative Akte, genau so, wie es das Programmieren und übrigens auch die Anforderungserhebung und das Testen sind. Neben Anforderungserhebung und Test beinhaltet das Erstellen von Software auch das Festlegen einer Struktur, der sogenannten Architektur eines Systems. Die ist nicht nur aus Gründen der Organisation der Aktivitäten der Systemstellung wichtig, sondern beeinflusst direkt fast alle Gütekriterien, über die wir oben gesprochen haben. Es lohnt sich zu wiederholen, dass das Programmieren und auch das Maschinenlernen nur einen sehr kleinen Teil der Aktivitäten ausmachen, die für das Bauen, Überprüfen und Warten großer Softwaresysteme notwendig sind.

Softeware Engineering

Software Engineering bezeichnet die Summe dieser Aktivitäten: Aufnehmen, Aufschreiben, Priorisieren und Überprüfen von Anforderungen; Zerlegen des Gesamtproblems in Teilprobleme und Design einer Architektur; Implementierung der Lösungen für Teilprobleme durch Programmieren oder Maschinenlernen; Test dieser Lösung gegen die Anforderungen. Und Software Engineering kümmert sich darum, wie diese Aktivitäten organisiert werden – sicher haben Sie im Zusammenhang mit Software schon einmal von „agiler Entwicklung“ oder „Scrum“ gehört. Dazu kommen noch weitere Aktivitäten, die wir schon angedeutet haben: Die Wartung solcher Systeme, die die Fehlerbehebung und die Weiterentwicklung beinhaltet; das sehr anspruchsvolle Management unterschiedlicher Versionen von Software; das Sicherstellen von Sicherheit und Privatheit sowie das Verstehen und Beheben von Sicherheits- und Privatheitsproblemen und so weiter. Schließlich beinhaltet Software Engineering auch das Strukturieren, Speichern und Verwalten von Daten. Da deren Beschreibung einen eigenen Text ungefähr der gleichen Länge wie des vorliegenden ergeben würde, verschieben wir sie auf einen zukünftigen Beitrag.

Teil V: Geschafft!

Das war's! Wir haben in einer Tour de Force viele verschiedene Konzepte kennengelernt, die wir noch einmal zusammenfassen wollen. Im Stakkato: Ausgangspunkt war ein Problem. Wenn wir die Lösung des Problems als Überführung von Eingabedaten in Ausgabedaten verstehen können, ist Software ein guter Kandidat. Das Verhältnis von Ein- zu Ausgaben lässt sich als (mathematische) Funktion beschreiben, als das Was der Problemlösung. Algorithmen sind mögliche Vorschriften, wie solche Funktionen berechnet werden. Ein Algorithmus kann dann durch unterschiedliche Programme implementiert werden, die auf der Hardware, insbesondere einem Prozessor, ablaufen. Alternativ können einzelne Funktionen mithilfe von Maschinenlernen berechnet werden. Software besteht üblicherweise aus mehreren miteinander kommunizierenden Programmen. Das Identifizieren und Verstehen des richtigen Problems, das Zergliedern in Teilprobleme, das Etablieren einer Struktur für Lösungen dieser Teilprobleme, das Sicherstellen und Überprüfen von korrekter Funktionalität und Güte eines Programms und dessen Wartung sind zentrale Aufgaben des Software Engineering. Und schließlich ist Software überall, auch in den cyberphysikalischen Systemen um uns herum.

Ich finde, wir sollten den Anspruch haben, zumindest prinzipiell zu verstehen, worum es sich dabei handelt. Vielleicht kann dieser Text dazu beitragen.

Peter Bludau und Johannes Kroß von fortiss; Manfred Broy, der statt des Kuchens lieber einen Taschenrechner gesehen hätte, Severin Kacianka und Traudl Fichtl von der TUM; Niina Zuber und Jan Gogoll vom bidt sowie Alexandra Pretschner von Boosting Change haben sehr geholfen, meine Beschreibungen des komplexen Materials zu vereinfachen. Die Idee zum Beispiel „Kuchenbacken" stammt von Gordon Mühl von Huawei. Vielen Dank!