Bavarian Research Institute for Digital Transformation

AN INSTITUTE OF THE BAVARIAN ACADEMY OF SCIENCES AND HUMANITIES

Long Read

What is Software?

Prof. Dr. Alexander Pretschner

March 2021

What is software? The word Software is on everybody's lips; software engineering is considered the key discipline par excellence and it is said that software "eats the world". In this article, we explain: what software actually is; that software is much more than computer programmes; why creating software is not so easy; and what software has to do with AI and machine learning.

We present our views on software in four parts:

1. We use the analogy of baking a cake to explain what a programme is and how it is executed on a computer.
2. These ideas are then applied to actual programmes.
3. We then show how programmes and machine learning are related.
4. Finally, we show that one programme is only ever a single small piece of functionality. Building large systems involves quite different tasks besides programming, which are addressed by software engineering.

This text has been developed in close cooperation with fortiss, the bidt, the Chair of Software and Systems Engineering and the Venture Lab Software/AI at TU Munich. Our target audience is interested laymen.

Part I: What are programmes?

Recipes for cakes are algorithms

Baking cakes has so much in common with software that we hardly need anything further to explain the central ideas behind software. Cake recipes describe which ingredients are processed in which order and how, up to the point when the cake is ready: Mix eggs, sugar and butter, add flour and baking powder, knead the dough and then bake it at 160 degrees for 30 minutes. There are the ingredients, the individual steps of preparation and eventually a cake. Also needed are tools that more than likely will not be mentioned in the recipe: Mixer, mixing bowl and oven. There are also the baker and the author of the recipe. Then there are the steps in the recipe, namely mixing, stirring, waiting and baking. In this case, you are in the role of bakers, carrying out these individual steps in the right order, ensuring that the next step can be taken using results from the previous step. With that, we now already have the essential concepts together to move on to looking at software.

In the software world, ingredients are the so-called *inputs*; our cake is the *output;* and recipes can be found in books, which in computer science are called *libraries.* The recipe describes a so-called *algorithm*. *Algorithms are step-by-step instructions on how to convert inputs into outputs.* Once algorithms have been detailed out and written down so that they can be executed by a computer, they are called *programmes* or *software*. The author of the recipe is the *programmer* and all kitchen equipment is the so-called *hardware*. You, as bakers, form an additional, particularly important piece of hardware, the so-called *processor*, often called the CPU (central processing unit). The processor *executes* the individual steps of the programme, just as you follow the individual steps of a recipe.

From your kitchen to the bakery

That's it! At least in the analogy, you have now understood what a programme does and how it runs on hardware. As an exercise, let us now transfer this basic idea before we turn to real programmes. Imagine a fully automated bakery that can bake different cakes. Before reading on, please briefly consider for yourself what constitutes inputs, outputs and hardware. Identifying programme and processor is a bit more difficult here, so before moving on, we should just make sure the concepts are clear:

- *Inputs* are the selection, such as the number, of a specific recipe and the corresponding ingredients.
- *Outputs* are the finished cakes.
- The *hardware* here, in addition to (large!) ovens, mixers and blenders, consists of egg beaters, conveyor belts, packaging machines and so on.
- These are all in turn individually controlled by *software*, special programmes that run on processors in these machines.

Now it becomes a bit more complicated. Interactions between individual hardware components must be coordinated in different ways, depending on the recipe being baked. Here, this is no longer done by you as the baker, since the entire baking process is automated. Instead, this is now done by a control computer (the *processor*), which again runs a special *programme*, namely a control program. Depending on the recipe to be baked, this programme controls the egg beater, mixer, conveyors and so on step by step in the correct sequence. In this way, it ensures that the individual stages of the bakery or baking process are supplied with the right intermediate product at the right time. Hence, the algorithms, which were originally just the recipes, are now extended in programme form to include control commands for the various machines, in other words our bakery hardware.

Software

The idea of converting inputs to outputs using hardware and software is very powerful because it can be used to describe a great variety of dependencies. I find it interesting that many mechanisms in our body can also be described in this way. For the regulation of blood glucose, for example, the pancreas continuously secretes more or less insulin depending on the current blood glucose level: If blood sugar rises due to a meal, more insulin is secreted; this reduces the blood sugar level; which in turn results in a reduction in insulin secretion. When there is too little sugar in the blood, glucagon is released, which stimulates the liver to release sugar. Thus, the “blood sugar management” system has a loop because the output (insulin, glucagon) directly affects the input (blood sugar). Similar processes happen in the heart. Simplifying even more, an increased demand for oxygen by the muscles due to exercise leads to an increase in heart rate; this increases the amount of oxygen made available to the muscles per time unit; at some point, sufficient oxygen is available and the heart rate is not further increased. The mechanisms that manage these so-called control loops can certainly be understood as programmes, although not in the sense of a sequence of programme steps.

The phenomenon of a loop from outputs to inputs is also typical in many machines. Air conditioners measure the room temperature; when the temperature is too high, they open valves to let coolant flow; this reduces the room temperature; after a certain time, this causes the valves to transport less coolant; which then causes a raise of the room temperature again; and so on. The functionality that performs this control function is implemented by programmes.

So, what exactly is software? Software is: programmes that run on one or several computers. Programmes such as Excel or your e-mail client are software that runs on a computer in the sense of a desktop, laptop, tablet, or smartphone. Programmes such as Google search, Facebook, or Instagram run, to some extent, on your laptop or smartphone to provide you with interactions. At the same time, their functionality, that is the actual execution of a search or provision of search results, images or messages, is made possible by an interplay of several, often thousands of computers. The existence of these computers usually remains completely hidden from us.

To a large extent, the functionality of your car is also provided by programmes. In a modern car, there are often more than 100 computers running completely different programmes. These control how you accelerate and brake, how the windshield wiper works, or how you move your seat. Software actually controls all the devices in our immediate environment. Because you can't see these computers running the corresponding programmes and because they are usually built into devices, these are referred to as *embedded systems*. Furthermore, because these programmes react directly to stimuli from the physical environment and, conversely, can also influence this physical environment (temperature, distance to the car in front, and so on), but at the same time virtually exist as software in cyberspace, they are also called *cyber-physical systems*. Such systems can be arbitrarily large and complex, such as the ISS space station, but also very small, such as a lighting control system in a living room that simply reacts continuously to external lighting conditions. Engineers today are working to build more and more systems on a variety of very different scales. Using networks, they can then be interconnected to enable communication between systems. You have heard the idea of connecting smaller devices under the buzzword *Internet of Things*, which enables us, for example, to automate and network lights, stoves, heating, energy supply and irrigation in our smart homes and gardens. Clearly, this poses security and privacy problems, but that is a subject for another time.

Nowadays, by the way, engineers in interdisciplinary projects usually no longer look at these technical systems in isolation, but at so-called socio-technical systems, in which human actors are also added to the interaction of various technical systems. A good example is hybrid traffic, in which both human-controlled and autonomous vehicles exist simultaneously and must react to each other. Furthermore, because people do unexpected or "illogical" things – for instance when children paint the road with chalk – programmes must be designed to provide them with inputs that were not anticipated by programmers.

Hardware

Programmes are executed on computers. Programmes and computers, that is software and hardware, behave as mind and body in humans. In addition to the processor, the hardware of computers consists of other components that perform the central tasks mentioned above. Whilst the *processor* performs the calculations, there is also hardware for *data storage*. Data is stored in *memory*, on the hard drive, on USB sticks and increasingly in the cloud. Communication with other computers is then achieved via connections to the *network*, which we will omit in this article for reasons of space. In addition to the processor as well as memory and network connection, a computer also consists of other devices, which we call *hardware* like the processor, memory and network connection above. These devices are: screen, fingerprint sensor, temperature sensor, battery, camera, keyboard, speakers, microphone, cables and so on. All in all, software uses this hardware to perform the tasks of *computing*, *communicating*, *managing and storing data* as well as *interacting* with the environment.

Inputs and outputs are data

When baking a cake, inputs are the ingredients and outputs are the finished cakes. However, programmes not only receive *data* as inputs, but they also deliver *data* as outputs. Intermediate results from the individual steps are also data. There are programmes such as Internet search and your e-mail client, or the control software in your stove. You can enter commands via a keyboard, voice, or gesture, namely a search query or a message. These programmes then show you something on a screen.

In the case of a car, a washing machine, a heating system or an automated bakery, the programme also reacts to stimuli from its physical environment: the adaptive cruise control recognises cars in front of you, the washing machine reacts to water levels, the heating system reacts to temperatures and so on. The corresponding input data is provided via so-called *sensors.* Sensors are, for example, thermometers, cameras and radar. They translate real-world phenomena, such as temperature, cars driving in front of you, and water levels, into data that can then be processed by programmes. Unlike your e-mail client, the output of a programme in a washing machine or car is not just something you read on a display. Instead, the output data is also translated into effects in the physical environment: accelerating or braking, pumping out water, turning on gas burners and heating water. In the modern world, programmes are literally everywhere.

Who does what and how: functions, algorithms and programmes

We have seen that recipes, algorithms and programmes describe individual steps that determine *how* inputs are converted into outputs. For you as bakers, these individual steps are essential. You must now consider it possible that your children, unlike yourself, are not interested in these individual steps, but only in the cake eating itself, that is, the output. Let’s further assume that your children are craving marble cake today, but the fridge is empty. They will then have to go shopping and that’s when they become interested in the ingredients, in other words: the inputs. If you want to describe the relationship between inputs and outputs, *but without explaining the individual steps, this is* called a *function*. You know functions from high school: Addition, for example, is such a function with two summands as input and the sum as output. When you add in your head, you don’t even think about how to do it, as the individual steps don’t interest you! But how does a machine do that? That is exactly the task of the programme, which *describes how a function is calculated.*

The connection between function, algorithm and programme is so important that I would like to summarise it once again. The function describes only the relationship between input and output, without saying exactly *how* the output is calculated. An algorithm defines the steps by which this function can be calculated. And the programme is the description of an algorithm in a form understandable to the machine (a “pinch of salt” is too vague to be understood by a computer!). With the distinction of “what” and “how” we can also say that the function describes the *problem* (what?), and that the algorithm and programme describe the *solution* (how?).

Programming languages

A programme, as with a cake recipe, is a piece of text. When you write something down, you have to find a language for it. Computer scientists have developed *programming languages for* this reason. A programme consists of several “sentences” in this programming language. Each sentence helps transform the input into the output, in a step-by-step manner.

There are many programming languages, just as there are many natural languages. If you know a foreign language, you may have noticed that certain concepts can be formulated more elegantly in this language than in your mother tongue and vice versa. This is exactly the reason for the multitude of programming languages. Certain technical steps in the transformation of inputs and outputs can be described better in one programming language than in another. Perhaps you can imagine that a programme that represents the interface of your e-mail client is easier to write in a language that talks about “window”, “button” and “mouse pointer” than in one that converts the distance to the car ahead into a command to brake. Have a look at the programming language Scratch which is especially good for interacting with characters in your own first computer game. In Scratch, by the way, the “sentences” are formed by pictorial symbols.

Part II: Two real programmes – and why programming is difficult!

Two real programmes

Now, what does it mean to say that a programme consists of steps executed one after the other, in order to calculate a function? As a reminder: We distinguish between what a function describes and how that is calculated: To a human it is intuitively clear how to understand the character sequences f(x,y)=x+y and g(x)= x^2 and what they mean, namely "sum of x and y" and "x squared". To a machine, though, we have to tell how to calculate this. One way to calculate the square function is x*x. We must then also tell the machine how multiplication works, which in turn consists of individual steps.

So, let's take a really quick look at what a programme looks like to calculate our square and sum functions. Let us start with the sum function. The inputs are the two summands x and y; the output is supposed to be their sum. Let's assume for the moment that we are using a simple programming language that cannot handle the addition of arbitrary numbers, but can handle the simpler addition of 1 and subtraction of 1. We will now try to calculate arbitrary sums using these two simple functions.

Let us observe that x+y is the same as 1+x+(y-1). Thus, we added a one to the left and subtracted a one from the right. They cancel each other out. In turn, 1+x+(y-1) is the same as 1+1+x+(y-1-1), and so on. For example, 3+2 is the same as (1+3)+(2-1), which is 4+1, and that is the same as (1+4)+(1-1), which is 5+0. Hence, our final result is 5.

We can thus calculate arbitrary sums by adding and subtracting 1, by subtracting 1 from the second summand (which is y) and at the same time adding 1 to the first summand, which is x. We do this exactly y-times: So, we subtract a 1 from y y-times until the result is zero, and add a one to x y-times.

The original values of x and y are handled in the *memory*, which we introduced above. In our programme we can read and overwrite the corresponding values (for this, computer scientists say *store* and even more precisely store in *x* or *store in y*). Our programme then looks like this.

"Sum" programme

```
Input: two numbers x and y
Output: Sum x plus y
Repeat as long as y is not 0:
        1. add 1 to x and store the result 1+x in x
        2. subtract 1 from y and store the result y-1 in y
Output x
```

As real code, this looks as follows. `while (condition)` is the repeat statement that is executed as long as the `condition` in the parenthesis is true. ≠ is the inequality. We use the left arrow `← to` describe the process of storing; so `x ← x+1` says that the value of x+1 is written to the location in memory designated for x. `output` prints the result on the screen. The curly braces are for grouping purposes only.

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

A computer now runs the programme as follows. Let us take the example of the inputs x=3 and y=2.

1. Because y≠0, in the first step x becomes 3+1=4 and y becomes 2-1=1.
2. y is still non-zero, so x now becomes 4+1=5 and y becomes 1-1=0.
3. Now y≠0 is no longer valid, no further repetition takes place.
4. The result is the last value of x, namely 5.

This is perhaps a bit unintuitive, but actually quite logical. Please note that here we have solved the more difficult problem “calculate any sum” with the help of two simpler operations, namely the addition and subtraction of 1. This is a core idea of programming: We solve a difficult problem using simpler steps.

Now let’s play the same game for the square function. This time we assume that our programming language offers only the subtraction of 1 and additionally the addition of numbers. We have just programmed the latter ourselves, so we can simply use this programme.

Multiplying x by x is the same as adding x-times the number x: 3*3 is 3+3+3 and 4*4 is 4+4+4+4. To describe multiplication with the general addition and subtraction of 1, we use a trick like above again: x*x is the same as x+(x-1)*x. And that is the same as x+x+(x-1-1)*x. Ah! That in turn is the same as x+x+(x-1-1)*x. And so on until the value in the parenthesis is zero. So now we have defined the square function using two simpler functions, namely the subtraction of 1 and the general addition.

In this case we have to remember an *intermediate result*, which we store as z in each case. At the beginning we set z to the value 0. Now let’s look at example of computing 3^2:

1. This is the same as 3*3, which is the same as 3+(3-1)*3. The left part of the sum is our intermediate result.
2. So we add this left 3 to z, which is originally 0, and store the result 3+0 in `z`.
3. Then we continue: z+2*3 is the same as z+3+(2-1)*3, so z+3+1*3.
4. We add the left 3 to the intermediate result z and write the value z=3+3=6 into memory for z.
5. z+1*3 is then the same as z+3+(1-1)*3, which is the same as z+3+0*3.
6. We add the left 3 to the intermediate result z, which now stores the value z=6+3=9.
7. 0*3 is 0, so we can stop.
8. The intermediate result z has changed in each step, and its last value is 9. Exactly what we set out to compute!

Because we have to add the value of x to the intermediate result z in each step, we cannot change x in each step as in the case of the sum. Instead, we need a second intermediate result v which stores the value of x, x-1, x-2, …, 0 in succession.

"Square" programme

```
Input: a number x
Output: x squared, i.e. x*x
Store 0 in intermediate result z
Store x in intermediate result v
Repeat as long as v is not 0:
    1. add x to z and store the result z+x in z
    2. subtract 1 from v and store the result v-1 in v
Output z
```

As code:

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

Let us simulate the computer running the programme for the input x=3.

1. First, z is set to 0 and v is set to the value of x, i.e. 3.
2. Because 3≠0, z is set to 0+3=3 and v is set to 3-1=2.
3. Because 2≠0, we repeat: z becomes 3+3=6 and v becomes 2-1=1.
4. Again, 1≠0, so we repeat the steps again: z becomes 6+3=9 and v becomes 1-1=0.
5. Now the repetition terminates, because 0≠0 is not valid, and z=9 is output.

That's how simple programmes work. E-mail clients, search engines, control units in cars and in bakeries are programmed the same way.

In case you are wondering: We have deliberately injected a small bug into our programmes that we will find afterwards.

Libraries and reuse

If our programming language does not know the addition symbol +, we can simply replace it with our sum programme above (change in green) and derive a new programme `square2`:

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

The use of previously created programmes in a new programme is a tremendously powerful principle that was discovered only in the Sixties. It is so powerful because, on the one hand, it allows us to break problems down into smaller problems, solve them independently and then put the solutions together. In this way, it helps us organise our programming work, among other things.

On the other hand, this principle also allows functionality once created to be made available to other programmers. Computer scientists call such programme packages *libraries*, which are made available to others for use in their own programmes. We compared them to baking books above. Perhaps you have heard of the Java programming language, or a language called Python? These languages are used by a great many programmers around the world in part because they provide huge libraries for all sorts of functionalities, including sending data to the Internet, using machine learning, or analysing natural language text. Programmers then do not have to create these functionalities themselves, but can benefit directly from the work of others by reusing the corresponding programmes.

If you have paid close attention, you will notice that we have introduced two concepts for the same idea. Above we said that programming languages differ in how they express “certain concepts” or “technical steps”, and that some languages are more suitable for talking about “window”, “button” and “mouse pointer”, and others about “distance to the car ahead.” Here we are now saying that “certain concepts” are captured in programmes, that is, solutions are programmed for problems that are then available in libraries for reuse. Your observation is correct. This is indeed the same idea in two different forms: Recurring constructs should be as easy to use as possible. The underlying principle is what computer scientists call *abstraction*.

Almost there: processor and machine language

Perhaps by now you have developed a suspicion that somehow we have tricked you. This is because we have programmed our functions on the basis of other, simpler functions. Strictly speaking, this does indeed merely shift the problem. To calculate an arbitrary sum, we have nonchalantly taken for granted the simpler operations of “plus 1” and “minus 1”. In the square programme we could just save ourselves, because we had programmed the addition ourselves earlier. But we still needed the “minus 1” operation there.

If you remember, the situation was quite similar when we were baking a cake. In recipes, many individual steps are presented in a highly simplified way, for example, when it is tacitly assumed that “baking” includes the following proceses: “Open oven door,” “Remove all trays from the oven,” “Close oven door,” “Preheat oven,” “Open oven door,” “Push in cake pan,” “Bake,” “Turn off oven,” “Open oven door,” “Remove cake,” “Close oven door. There, it is assumed that the baker knows which individual steps make up complex steps (and again, what the meaning of “a pinch of salt” is).

So, programmes are written with the help of simple operations. We then write still larger programmes with the help of these simple operations along with other programmes and the libraries we have just mentioned. Building on that, we write still larger programmes and so on. In this sense, we can stack programmes on top of each other.

However, I have not yet explained how to actually compute the results from these simple underlying operations. Furthermore, perhaps you have remembered that we have yet to tell you exactly what a processor really does. Let’s do that now: The processor provides exactly these simple underlying operations just mentioned! It can calculate very simple operations *on the hardware* like – you guessed it – adding or subtracting one and multiplying by two or dividing by two. It can check if a value is zero, it can select arbitrary locations in a programme as the next step to execute and so on. For these very simple operations executed on the hardware, there is again a separate programming language called *machine language.* The machine language for Intel processors consists of around 1000 possible simple operations.
Programmes in other programming languages such as Java or Python are ultimately translated into programmes in machine language. This is probably best understood in the sense of a translation from German to English.

You may have heard the word *compiler* before: compilers do exactly this translation into machine language. The idea is quite similar to the use of the `sum` programme in the `square2` programme in the line `z ← sum(z,x)`. You may also remember our earlier remark about the conceptual similarity between libraries and different programming languages. After all, the use of the word `sum` represents the lines of code that implement the `sum` programme. In reality, things work slightly differently, but you can think of it as replacing the word `sum` in our `square2` programme by exactly the lines of code that correspond to the `sum` programme. In exactly the same way, the repeat statement `while,` the assignment ←, addition and subtraction of 1 and the output function `output` stand for small independent programmes in machine language, into which they are actually translated as well.

Amazingly, e-mail clients, bakery controls and all other functions can be translated into this simple machine language. A processor can then execute machine language programmes directly on the hardware. That is, the processor receives input signals in the form of “voltage” and “no voltage,” representing, for example, the addition of 1 to a number 7, and it provides output signals also in the form of “voltage” and “no voltage,” which are then interpreted as the result of the addition. These are the famous binary ones and zeros that we all associate with computers.

That it is possible to represent arbitrary numbers with ones and zeros has been known since the 17th century. It would take us too long to go into any more detail about this here, but we can at least quickly understand the idea. Think of it as follows: We humans use ten digits to represent numbers, namely 0, 1, 2, …, 9. If we want to represent ten, we need more than one digit: 1 and 0, so 10. Eleven is 1 and 1, so 11. Ninety-nine is 9 and 9, so 99. If we want to represent one hundred, we need three digits: 1 and 0 and 0, so 100 and so on. If we have only zeros and ones available, we can use the same principle: Just as we humans cannot represent a ten with only one digit, computers cannot represent a two with only one digit. So, we add a second digit, and 2 is represented as 1 and 0, which is 10. We represent 3 as 1 and 1, which is 11. We need another digit for four, just as we needed another digit for hundred: Four is written as 1 and 0 and 0, so 100, and so on. By the way, the ten is represented as 1010, the hundred as 1100100.

Because the programme steps that are described with machine language are very fine-grained, the corresponding programmes quickly become very large. For this reason alone, they become difficult to understand and maintain. In computers' early days programmes were actually written directly in machine language. Today, no one actually does this anymore, but instead uses the programming languages that are better understood by humans, which are translated into machine language with the aforementioned compilers for programmes.

Three comments: Condition; three instructions are enough; humanisation

Please allow me to make three comments at the end of this part. I think they are important and even if you are still not clear about everything after your first reading of this section, you will still be able to understand the rest.

First, the set of all intermediate results at a given time is called the *state of* a programme. The state is stored in the memory during the programme execution. Each step of the programme execution changes this state, because intermediate results are changed and in our example so too are the values of the variable storing the input. Often there are many such possible states. Their large number makes it difficult to write and understand programmes. So, it is not only the number of steps that makes a programme complex (the software in a car today consists of hundreds of millions of lines of code), but also the number of possible states.

Second, we have explained that programming languages differ, among other things, in how they can succinctly express certain concepts in a particular context: We talked about "windows" and "buttons", for instance. In our real programmes above, we used three basic concepts just like that, without explaining them further, because they are so natural: (1) storing values or intermediate results in memory – those are the lines with the left arrow; (2) executing one line after the next – that's signalled by the semicolon at the end of each line; and (3) repetition – the while statement, which we already explained. I still find this really amazing: since the 1960s, it has been known that all functions that are capable of being computed can be programmed using these three facts plus addition and subtraction. All of them!

(Why the strange interjection with "*functions which one can calculate*" in the first place? That is crazy enough, but there are also functions which one cannot calculate. This is hardly imaginable! A famous example is a function F, which receives *a programme P as input* and is to compute for *each* input programme P whether this programme P will eventually terminate its calculation for *each input E*, or whether it can happen that the program P runs infinitely long. We will see in a moment how this can be the case for our examples of the square and sum programmes above in the case of the input of negative numbers. Now a function F, which computes for *all* programmes P and *all* inputs E whether the calculation will stop at some point, looks like this: F(P)= "yes" if P(E) finishes at some point for *all* inputs E, and "no" otherwise. There is no programme for such a function! We have known this since the 1930s, but best that we do not dwell on this conundrum any further here!)

Third, you may have noticed that throughout the text we have quite intuitively used words for actions of machines that you would usually attribute more to humans or living things in general: "respond to a stimulus," "interact," "communicate," "recognise," "compute," "learn," and so on. Philosophers have given considerable thought to such "humanisation" of machines. This becomes explosive when it is assumed, or linguistically suggested, that machines can "decide" facts and even be "responsible" or "liable" if necessary. With the exception of responsibility and liability, I find this linguistic humanisation catchy and completely okay. However, I do not want to hide the fact that there are schools of thought in connection with so-called transhumanism that at some point grant machines real intelligence or even emotions or responsibility. For my part, I consider this to be absurd.

Programming precision; errors

Perhaps you, too, have tried a cake recipe that you didn't quite understand or misunderstood, and then, upon closer inspection of the result, decided that perhaps you would rather throw the cake away than eat it. Sometimes that's simply your mistake. But sometimes it's also because recipes aren't described precisely enough, or because they contain mistakes.

You may have noticed that if the input y for our sum function or the input x for our square function above is a *negative* number, then the programme will never terminate. This is because a negative number minus one can never add up to zero: As we run the programme, we subtract minus one an *infinite number of times*. The problem is that we forgot to account for this special case. We were not sufficiently precise. (The error can be fixed in both programmes by replacing the ≠ in the `while` condition with the greater-than sign >. See for yourself! Now if you look carefully, we forgot something else: We didn't require the inputs to be integers in the programme. If you enter 4.2 as input, the programme will never finish either. Replacing ≠ with >, though, also fixes this problem).

Lack of precision in this and other ways is a big problem in programming. That's one of the reasons why programming is not always that easy. In a very funny video clip (Exact Instructions Challenge: This is why my children hate me), the problem is well illustrated by an example: A father asks his young children to write down exactly which steps he should perform in order to make them a peanut butter and jam sandwich. Do try this out with your kids! It quickly turns out to be more difficult than anticipated. For example, the father, intentionally of course, misunderstands the instruction to "spread peanut butter on a toast" to spread the peanut butter on one of the four edges of the bread instead of one of the two larger sides. Or he uses his hands to get jam out of the jar because the children forgot to suggest using a spoon to do it. Or he fails at the very beginning because there was no instruction to open the refrigerator door before removing the jar of jam.

Part III: How does machine learning work?

Programming and AI: Machine learning from examples

Describing the individual steps of a programme so precisely that nothing is forgotten, nothing is misunderstood and nothing is wrong is the job of programmers. If you play the sandwich game once with your children, you will suddenly understand why the programmes you use happen to be faulty from time to time. Please don't misunderstand me: this is no excuse. Programmes should be bug-free, of course – I'm just saying that it's not that simple.

In fact, this is one of the reasons why artificial intelligence, or more precisely one of its variants, machine learning, is currently so popular. The approach here is different. *Instead of describing each individual step in detail, programmes are instead defined by a large number of examples*. This also explains why the corresponding techniques are called machine learning: After all, we humans learn particularly well on the basis of examples.

In our context, examples consist of inputs and outputs, which we will separate by the → symbol. So instead of describing the individual steps in baking a cake, consider the examples (`milk, flour, sugar, cocoa, butter, eggs) → marble cake` and (`flour, yeast, salt, sugar, apples) → apple pie` and `(flour, yeast, salt, sugar, plums) → plum pie` and so on. Assume further that there are ingredient lists for compote, so `(apples, sugar, cinnamon) → apple compote` and `(pears, sugar, vanilla sugar, clove) → pear compote`. Machine learning now determines a function that, for inputs that are "in-between" the inputs of the examples, computes outputs that are as close as possible to, or in-between, the outputs of the corresponding examples. For a new list of ingredients `(flour, yeast, sugar, pears`) that was not part of our initial examples, the learned function finds the examples whose ingredients are most similar to that list. Here, these are probably the ingredients for apple pie, plum pie and pear compote. As output, the learned function outputs something that lies between the outputs belonging to these inputs – in this case, possibly pear pie.

In machine learning, a distinction is made between two phases. The first phase is the learning of a problem domain (a function), which results in a so-called learned *model* that represents this issue. After learning, in the second phase, the learned model is used like a programme by feeding it with inputs and obtaining an output.

Traditionally, the following machine learning tasks are distinguished: classification, prediction, grouping and association.

1. *Classification* involves assigning input data to a class: input images of animals, for example, are recognised as "dog," "cat" or "guinea pig." Simple recognition systems for pedestrians assign the class "pedestrian" or "no pedestrian" to a camera image.
2. In *prediction*, the result is not a predetermined class, but a value, for example, when predicting the price of a house based on its square footage, location, and age. Our pear pie above also falls into this category.
3. With *grouping* (clustering), one wants to group "similar" objects into classes without knowing the classes beforehand: Images of animals are grouped in a way that all dogs, all cats and all guinea pigs are each in one respective class without knowing the categories "cat," "dog," amd "guinea pig" beforehand.
4. Finally, in *association*, one tries to understand the factors that have significantly contributed to a particular outcome in the past. Examples include factors for buying a particular product online, perhaps using an iPhone instead of an Android smartphone when shopping; other products previously purchased by oneself; other products previously purchased by friends; time of purchase and so on.

There are many different approaches to solve these problems technically. Programmes, on the other hand, essentially always work the way I described above. This explains why we can introduce the notion of a programme in quite some detail in this article: Because it is rather clearly defined! In contrast, there are so many different methods of machine learning (here you can find an overview picture) that we are forced to stay more on the surface when it comes to machine learning.

We still need to bring together the worlds of machine learning and programming. In programming, a human thinks about how to solve a problem step by step. The result is a *programme* created by the human. In machine learning, the result is the aforementioned machine-generated *model* that represents the learned relationships. Now things become a bit confusing. Machine learning itself consists of individual steps and in this sense is a programme like any other: Inputs are examples, output is the model. The output, that is the model, can now also be understood as a programme, because it calculates a function, for example how an animal species is assigned to a picture. However, these models are somewhat different from the programmes we have encountered so far. This model does *not* consist of single goal-directed steps by which a human could understand how to move closer to the goal. In contrast, that was certainly the case in our sum and square programme above. How exactly a model solves the problem instead depends on choosing the machine learning variant.

In this sense, machine learning takes over the task of the programmer. Because there are always many different possible solutions for a programming problem, we may expect that machine learning also can find a variety of different solutions, in other words models. And indeed, that is exactly the case.

Why machine learning?

Please observe that all four applications of machine learning that we have referred to can each be understood as functions. Moreover, it can be seen that in all cases the task at hand is difficult, if not impossible, to describe precisely. “Identify similarities” – when are two dogs similar and how are dogs different from cats? “Assign to a class” – what makes a dog a dog and a cat a cat? The relationships can also be complex (“What is the price of a house?”), or they are not yet clearly understood (“Identify the factors for a purchase decision”).

But machine learning is not only used because it is sometimes easier to do than explicitly formulating programmes. Sometimes it also simply works better! This can be seen in the example of automatically translating natural languages, for example from German to English. For decades, people have tried to map the rules of the two grammars onto each other and have not always been able to achieve convincing results. With machine learning, this works much better – this very text, for instance, was translated using a translation engine called DeepL (the results are astonishing, but indeed did require quite some additional manual polishing).

Another example is object recognition in images, for example of pedestrians in camera images recorded by autonomous vehicles. Just as in our sandwich case, try to describe very precisely how to recognise a pedestrian without using concepts such as “human” or “child” or “snow” or “umbrella.” After all, machines don’t know these concepts. Keep in mind that there are children and adults; that pedestrians move at different speeds; that they carry shopping bags and use umbrellas and can wear chicken costumes at carnivals; that they can appear in groups and partially obscure each other; that they can also be obscured by cars; that there are different light and weather conditions with sun, rain, and snow; and that pedestrians can appear as if out of nowhere when a child jumps out from behind a car to retrieve their ball. This is much harder than the peanut butter and jam sandwich!

Finally, another reason for using machine learning is the fact that programmes implementing such machine-learned functions can sometimes run much faster than conventional programmes.

Stumbling blocks

However, there are also serious disadvantages with machine learning, which are often generously overlooked in the current hyped debate: No one knows whether the output found in this way is really the correct one – and even worse: No one can know exactly! This is because we have deliberately learned using examples and have not explicitly written down the individual steps anywhere – because we wouldn’t have known how to do this! That is exactly the extreme beauty of the approach! But that’s also why the results cannot be verified at this level. This, in turn, explains why currently there is so much interest in the booming research direction of “explainable AI”.

Often, machine learning does not involve learning the same relationships that a human internalises to understand the world. Given the task to "distinguish dog from cat", we humans might pay attention to the size, the shape of the eyes, the tail, whiskers and so on. Machines, though, often learn some context that doesn't even seem relevant to us humans. Amazingly, this nonetheless often works very well in practice. This is in line with the well-known observation that, as a rule, by changing only a single pixel – which a human being cannot perceive at all – a correct classification of the image as "dog" becomes a wrong "cat". The question then is how bad this is in practice.

That machine learning is currently enjoying such great popularity is due to great advances in learning methods. It is also due to tremendous advances in hardware over the last two decades. Another main reason has probably more to do with the easier availability of large amounts of data today. But let's not make a mistake here: it's not that simple either. In practice, data is very often incomplete and erroneous, or not representative, or only available in relatively small quantities. For today's most prominent learning methods (you may have heard of Deep Learning), you need very large numbers of examples. These large sets of examples are available in some cases, such as Amazon buying behaviour. They are not in many cases, such as security attacks against cars. There are many other approaches, including those that try to make do with smaller sets of examples, but that too takes us too far here.

If you are interested, you can find a lot of publicly available data at Kaggle, among others, with which machine learning methods can be tried and tested.

Sometimes it is the case that learning functions can consume an outrageous amount of energy (is just one source on this agitated discussion). However, since the "computation" of the function *after learning the model* sometimes consumes very little energy, one must of course relate this to alternative forms of implementation, such as programming by hand, which do not require much electrical energy when written but may well use a lot of energy when being executed). Unfortunately, this also takes us too far here.

Finally, there is a consensus among many researchers today that example-based learning methods alone are not the means of choice if the section of reality to be learned is already well understood – for example, gravity, flows of fluids, or the behaviour of electric fields in certain contexts. It does not make sense to learn what we already know. This is another reason why machine learning does not address all remaining engineering problems. Great efforts are being undertaken today to marry the world of explicit rules and laws of physics with the example-based world of machine learning.

We will only briefly discuss the currently controversial topic of ethics in machine learning later; that is a discussion for some other time. There is currently a heated debate about the "fairness" of machine learning. A machine-learned function can be very good on average. However, it may then happen that it is not so good for individual small subgroups of inputs, as in the case of automatic face recognition, for example, if this can sometimes discriminate against social minorities.

Finally, current debate sometimes gives the impression that AI and machine learning have solved all remaining problems of computer science. Of course, this is not the case. Generally speaking, machine learning works well, but not always. When you learn from examples, you are inevitably confronted with the difficulties we discussed above. That is why machine learning, as with classical programming, is only one tool in the computer scientists' toolbox: machine learning cannot replace programming, but it can usefully complement it. Moreover, as we will see in a moment, there is much more to software than programming or learning functions.

Part IV: Software Engineering: How to create programmes?

Refresher review

At this point, let's remember what we have learned so far. We are interested in converting *inputs* into *outputs*. These inputs and outputs are *data*: Numbers, sensor data, emails, search queries, keyboard inputs, camera images, speech and so on. That's why we speak about data processing, or information processing. The connection between input and output is a *function* in the mathematical sense, remembering from school that a function always maps inputs to outputs. How exactly the value of a function is calculated is something programmers have to think about. This *calculation* can be done by *models, which* were created automatically by *machine learning* based on examples. Or it can be done by specifying an *algorithm* that essentially solves the problem by breaking it down into individual steps. Programmers then formulate the algorithm, adding details, as a *programme* in a *programming language* appropriate to the problem, which is executed on a *processor* using memory, data transmission networks, and other hardware. Finally, we have learned about a central principle of constructing software: large problems are decomposed into small problems, solved independently, following which the solutions are assembled and integrated. Instructions in a programming language, such as the repeat instruction "while", can be composed of instructions a processor will understand. In a programme one can use other self-programmed functionalities, such as the sum function. Or one assembles large programmes from many other programmes, small and large, which are available in libraries.

It is not the case that only one algorithm exists for a given problem (or function), which can then be implemented in a programme in only one way – quite to the contrary! Consider the exemplary problem of "sorting a sequence of numbers," which continues to be quite essential for computer scientists. Firstly, there are dozens of algorithms with different properties regarding the necessary memory for intermediate results and the time and energy required for the computation. Secondly, for each algorithm one can write quite different programmes *implementing* this same algorithm. This is not merely because of the free choice of a programming language. As we have seen above, machine learning can be used to find a completely different set of solutions to the same problem.

When is a programme good?

If two different programmes implement the same algorithm and if there are different algorithms that solve the same problem, then how do these programmes and algorithms actually differ? Algorithms are, after all, instructions formulated at a somewhat high level and meant to solve a problem. At the level of algorithms, computer scientists are primarily interested in whether these algorithms really solve the given problem in all cases. This was not entirely the case for our computation of square and sum functions, if we recall the case of negative input values. If the problem is always solved in an intended way, we call the algorithm, and then the programme, *correct*. Correctness is so important and so inherently difficult, especially in the case of machine learning, that we will return to it below. So, our first quality criterion is correctness, which can be characterised by zero erroneous computations or zero erroneous programme steps.

In machine learning, we correctness takes on a slightly different form: How good is the recognition or classification? When recognising dogs and cats, for example, all dogs should be recognised as dogs. On the other hand no cats should be falsely recognised as dogs. This is not the same. There are good ways to measure this dual quality. However, as with machine learning itself, in order to be useful they often require large amounts of data, which is not always available. In the case of grouping (clustering), you often do not know in advance whether the identified groups are meaningful – because if you knew that already, you would often not need machine learning at all.

Computer scientists are then particularly interested in the extent of memory that algorithms or the programmes that implement them require; how much time it takes to execute them; and also how much energy they consume. Memory is comparatively expensive, so less is more. Intuitively, it is also clear that a faster problem solution is usually preferable to a slow one. Similarly, that less energy consumption is preferable to more energy consumption is also immediately obvious: With 200 search queries on Google, you need the same amount of electricity as for ironing a shirt [source; for the total power consumption of Google also see this link]. With an estimated 63,000 search queries per second in 2020 [source], it is imperative to write programmes that are as energy-efficiently as possible. Computer scientists are of course looking into this. In sum, memory, time and energy requirements are thus a second set of important quality criteria.

Finally, there is a whole range of other criteria for the quality of programmes. On the one hand, these are properties experienced by the users of a programme: Is a programme secure in the sense that an attacker cannot see or modify the programme or the data it handles? Is it safe in the sense that it does not harm the environment, for example in the case of robots or autonomous cars? Does it provide privacy? Is it easy to use? Is it fun to use? On the other hand, there are criteria that are relevant from an engineering perspective: Given that programmes often “live” for decades, being able to maintain the programme easily is critical. Are changes easy to make? Is the programme easy to understand? Is it easy to transfer the programme from one computer hardware to another, something which unfortunately is not self-evident at all?

Software is increasingly making decisions that affect us all – in autonomous vehicles, when granting loans, in traffic light circuits, in medicine or in police work. We have already briefly touched on the idea that the word “decision” can be misleading here because it suggests responsibility. At the bidt, we deal with another aspect of quality, which is based on ethically desirable considerations and is therefore even more difficult to evaluate than the other quality criteria. Rather than overloading you with too much detail here, we invite you to read about our project on ethics in software development (and not only machine learning!).

Software is generally developed by companies that are interested in developing as quickly as possible. Otherwise, the competitor has already conquered the market and has attracted many users. We will look at this “winner takes it all” situation elsewhere. Of course, companies also want to keep cost as low as possible. Unfortunately, it now quickly becomes apparent that the above criteria, including correctness, resource consumption, security and privacy, usability and maintainability, as well as cost and development time, often conflict with each other. Good usability often conflicts with high security; high security sometimes conflicts with fast programme execution; good maintainability can conflict with fast development time and so on. Conflicting goals are quite normal and define our lives: Just think of the various factors that influence any COVID-19 strategy.

The *quality* of a programme is therefore a combination of the factors mentioned above. There is no one golden combination that would be optimal for all programmes. Software is very strongly dependent on, and interwoven with, the development and application context: Medical technology products, sewage plant controls, pizza delivery apps, autonomous vehicles, garden irrigation systems and so on obviously have very different requirements on the quality of the corresponding software. We’ll later look at how to meet the different quality requirements. This is precisely one of the central tasks of *software engineering*, the discipline that is concerned with creating and maintaining “good” software; “good” in several senses.

What should a programme do: Requirements

Before we conclude by explaining what software engineering is and why software ultimately involves more than programmes and much more than machine learning, I would like to return to the problem of *correctness* mentioned before. Remember that a programme is “correct” if it does what it is supposed to do, in other words if it solves a given problem. This shows that correctness is not an absolute concept: correctness can only be thought of in terms of what is actually desired. Computer scientists distinguish here between the *desired* and the *actual behaviours* of a system. The former is intended and the latter is what the programme actually does when it is executed. Ideally, the desired and actual behaviours are identical.

Before formulating the intended behaviour, we must understand exactly which need we are addressing and which problem we actually want to solve. This is where the first mistakes repeatedly happen in system development. The following example maybe can help illustrate this: Some time ago, I lived with my family in the USA. We didn’t own a car, so we had to solve the problem of moving to the supermarket. You will recognise this as a somewhat unpleasant problem if you yourself have ever walked about three kilometres to the supermarket with a backpack and screaming toddlers, staggering back later fully loaded, drenched in sweat and on the verge of a nervous breakdown. We considered buying a bicycle, taking advantage of car-sharing offers that were just developing, or simply biting the bullet and taking a cab. Somehow all this didn’t do the trick – and then one day we saw the grocery delivery service van. At that moment, we realised that *the problem wasn’t how to get to the supermarket. The real problem was how the food would get to our apartment*.

In hindsight, this is of course obvious. But perhaps you yourself have already realised that you set out to solve the wrong problem. Once you figured this out, everything suddenly became so much easier. Identifying and understanding the right problem to solve, or the need to address, is called *requirements engineering in* the software industry. Requirements engineering always is the first major hurdle for a software project. Requirements engineering is a set of activities that is concerned with eliciting, understanding, writing down and reviewing needs and requirements. If you’ve ever heard of *design thinking*, you’ll remember that understanding the right problem to solve plays a decisive role there. In so-called *agile* software development, this is why you interact with the client or the future users of a system *during system development*, *on a continuous basis*, in order to ensure that the right solution is built. This is also a good idea because the client’s requirements are not fixed once and for all, but rather continually and significantly and naturally change during the development and lifetime of a software product. There are thousands of examples of software development projects that have failed because the needs and requirements were not properly understood, not properly written down, not properly followed-up, and not properly communicated during the various development activities. Every computer scientist knows the analogy of the failed development of a child swing, which can be illustrated very neatly here.

Once the need and the right problem to solve have been identified, the second step is to think about writing down the corresponding requirements. Why? Because you can then structure the development process, divide the work into teams and, most importantly, check for correctness at the end. If there are no clear requirements that describe the intended behaviour of a software-intensive system, then there can actually be no correctness (now think again about the major reason why we use machine learning!) Of course, the users of a programme can just use it and make a judgment as to whether it does what it’s supposed to do – but that’s highly unsystematic, and actually you don’t want to bother them with programmes that you know are still very immature.

Find errors in programmes: Testing

In order to be able to check a little earlier whether or not the system does what it is supposed to do, one proceeds somewhat differently in practice: One assumes that the correct requirements are noted correctly. Based on this, the developers test the programme. *Testing* means that for a few representative inputs one considers in advance what the desired output of the programme should be, in other words the intended behaviour for this input. The intended output can be derived from the requirements. Then one executes the programme with these few input values and compares the programme’s actual output with the intended output.

This sounds simple. In practice it is very difficult because finding “representative” inputs and thus “good” test cases, as we have required, is extremely challenging for several reasons. One of the reasons is the incredible number of possible inputs: If you have only one integer number as input, as in our square function example above, that’s already 264 possibilities, which is an unimaginably large number. For the sum of two summands it is already 2128 possibilities. In the universe there are estimated 2350 atoms, a number which we already exceed when using only four numbers as inputs.

At this point, it is interesting to think again about the peanut butter and jam sandwich, and to see the similarities with pedestrian recognition and machine learning. We explained above that machine learning is used, among other things, when one does not know the exact way to compute the solution. At the same time, in the context of pedestrian detection, we have seen that it is almost impossible to grasp the concept of “pedestrian” precisely enough: We cannot precisely describe our requirements. This is exactly why machine learning is used in such cases, or so we have argued. Hence we use machine learning not only when the how is difficult to grasp, but also when we can’t precisely describe the problem, the *what*. Something really crazy happens here: *For many machine-learned functions, there is no precise description of the desired behaviour at all* – because if there were, we might not have used machine learning, but would have manually implemented a precise description of the desired behaviour in the individual steps of a programme!

But if there is now no precise description of the intended behaviour, how can we test a machine-learned model systematically? The short answer is: We can’t, at least in general, and we can’t do it for the reasons I mentioned. Engineers use various tricks to counter this fact, but here we see a quite striking difference between traditional software development and software based on machine learning. This also explains why many bright minds today are concerned about the problem of so-called safe AI, which you may have read about in connection with autonomous vehicles.

Besides testing, there are other very useful procedures for finding errors, such as reading programmes without executing them. In practice, it turns out that these procedures work very well. Just for the sake of completeness, let us note that unfortunately this cannot work for machine-learned functions either, because these functions do not contain any individual steps that a human could understand and follow.

Structuring systems: Software Design

So far, we have become acquainted with three activities of software engineering: *requirements engineering*, which deals with the needs and requirements to be fulfilled by a programme; *programming*; and the verification of the correctness of programmes, *testing*. Computer scientists sometimes distinguish between what is called programming-in-the-small and programming-in-the-large when designing software systems. The kinds of programmes we have encountered so far implement an algorithm on a small scale, or they are based on machine learning. We have assumed that we can always write a programme directly for a given problem. But now, when the problems become very large, the programmes also become very large. To cope with this complexity, problems have to be decomposed into smaller sub-problems, which in turn have to be decomposed until manageable parts emerge. This is called programming-in-the-large. For the identified sub-problems, solutions can then be implemented individually in the form of programmes. The individual programmes are then assembled, following which the assembled parts must in turn be tested again. We have already discussed this on a more fine-grained level when we used the `sum` function in the `square2` function above. Here I am concerned with a much coarser level of granularity. As an example, take a car, in which today about 100 computers provide their service. On each of these computers, several programmes execute. These programmes consist of hundreds of millions of lines of code, and their integration results in all the functionalities which modern vehicles offer. It is the task of *software designers* to structure this functionality.

The point I am trying to make is this: Decomposing a problem into subproblems and decomposing a large system into manageable subsystems are as such creative acts, just as programming and, for that matter, requirements elicitation and testing are. In addition to requirements elicitation and testing, creating software also involves defining a structure, the so-called *architecture of* a system. This is important not only for reasons of organising the activities of system engineering. It also directly influences almost all the quality criteria we have talked about earlier. It is worth repeating that programming and also machine learning are only a very small part of the activities necessary for building, testing and maintaining large software systems.

Software Engineering

Software engineering refers to the sum of these activities: Recording, writing down, prioritising and checking requirements; decomposing the overall problem into sub-problems and designing an architecture; implementing the solutions for sub-problems by programming or machine learning; testing this solution against the requirements. Hence, software engineering is concerned with how these activities are organised – you've probably heard of "agile development" or "Scrum" in the context of software. In addition, there are other activities that we have already alluded to: the maintenance of such systems, which includes bug fixing and evolving the software system; the very sophisticated management of different versions of software; ensuring security and privacy. Finally, software engineering includes structuring, storing and managing data. Since describing these would result in a separate text of about the same length as this one, we defer this to a separate article.

Part V: Done!

That's it! We have learned all about many different concepts in a tour de force, which we would like to summarise once again. Our starting point was a problem. Whenever we manage to understand the solution to a problem as the transformation of input data into output data, software is a good candidate for the solution. The relationship between inputs and outputs can be described as a (mathematical) function. That's the *what of* the problem solution. Algorithms are prescribe *how* to compute such functions. An algorithm can then be implemented by different programmes. These run on hardware, in particular a processor. Alternatively, individual functions can be implemented using machine learning. Software typically consists of multiple programmes communicating with each other. The central activities of software engineering are: Identifying and understanding the right problem, breaking it down into sub-problems, establishing a structure for solutions to these sub-problems, ensuring and verifying correct functionality and quality, and evolving the software system over time. Finally, software is everywhere, in the cyber-physical systems that surround us.

I think we should strive to understand, at least in principle, what this is all about. Hopefully our text can contribute to that understanding.

Peter Bludau and Johannes Kroß from fortiss; Manfred Broy, who would have preferred to see a calculator instead of a cake; Severin Kacianka and Traudl Fichtl from TUM; Niina Zuber and Jan Gogoll from bidt; and Alexandra Pretschner from Boosting Change helped a lot to simplify my descriptions of the complex material. The idea to apply the cake baking example to the automated bakery is from Gordon Mühl from Huawei. Many thanks!