WORKSHOP 1: Copyright Infrastructure - Digiskills and metadata 1 December 2020

Maria von Kügelgen: Update your digital skills! (translation of transcript to English)

"the communication with the software developers is much more fruitful if both parties use, at least to some extent, the same language. Personally, I have often acted as something of an interpreter between lawyers and coders."

So the background for all of this is that the world has seen a tremendous change in recent years and decades. More and more, in a continuous fashion, work is done more by computers than by human beings. In some respects, computers do a far better job than humans: they can rapidly process huge amounts of data. As a rule, they work particularly well when the rules for processing the data are clear. In principle, they are flawless; if the algorithms made for them and the rules given for them are flawless, in principle, the computers either will not… they don't make the same kinds of errors as humans do. They don't get tired either: humans have a strange need to rest and sleep, unlike computers. In turn, there are some things with which humans do a better job than machines: such as creative thinking, interaction skills, and empathy – such human characteristics. The goal, of course, is to be able to combine these things in the best way possible, in that the computer would do the things it does well and people could focus on the things they're good at. What has resulted from this is that working life has seen a drastic change. In the past, the IT field used to be its own little world: the coders and the IT people stayed in their own silo, and then there were the others. But recently, they have begun to come together in a way that, in the IT field, there has emerged lots of people that are focused on a certain area of expertise. On the other hand, there are needs in different areas of expertise for people who understand something about, for example, software development and different technologies.

And indeed, the need for experts to participate in software development is growing continuously. A good example is my own professional career: working in the Tax Administration, I participated in the software development of inheritance and gift taxation, as a client. And there, too, I felt a lot of frustration about the way I was just presented with a data model that they had on the software – does the data model look good to you? I was like: what on earth is a data model? What am I supposed to see from this? On the other hand, in my current work – I'm currently working for KPMG under the title of Head of Legal Technology – it is my job to develop our lawyers' work processes with the aid of technology. Work like this did not exist before; I have worked there for a couple of years, and the position was created when I started working there. So working life is changing tremendously. Even if one weren't actively involved with developing the software, in any case, it is useful to understand what the benefit of technology is and what it can be used for. Naturally, when you are involved with that, the communication with the software developers is much more fruitful if both parties, at least to some extent, the same language. Personally, I have often acted as something of an interpreter between lawyers and coders. When you are able to speak both the coders' and the lawyers' languages, many things (progress in a much smoother way). I can ask lawyers the questions to which the coders need answers in order to do their own work. So firstly, things get sped up, and secondly, the result is closer to what the need is. By the way, feel free to interrupt me if you have any questions.

Yes. So I thought that we could begin by discussing what software actually is. It's much discussed, but if you are not involved with IT (information technology), the actual nature of it may be a bit obscure. Software consists of code in which computers are given different kinds of commands. In order for the rules to be useful, there is also need for data that is processed by the code. That's a simple way to describe what software is. In practice, in software development, coders are usually involved – they write the code. You may have come across the terms like libraries and frameworks that are used extremely often. Libraries are ready-made strings of code that the coders can make use of. Different kinds of libraries can be implemented in the software project. For example, you can have mathematical libraries that do different kinds of separate things. You can, say, ask the library to give you something… or you can give them parameters in certain situations: for example, give me a random number between one and ten – or one and six, if I want to develop a fun little dice-rolling piece of software, so you give it a range in which you want random numbers. Software frameworks are kind of similar, but how they work is that they make up the body of the software, and there are spots for supplements which you can fill with your own content. Then, for data, in order for the data to get stored between uses, a database is needed for storing it. There are different kinds of databases, too. One should consider which would fit the project based on the needs. Pieces of software can communicate with each other via interfaces. You often hear people refer to it by the abbreviation API – application programming interface. Similarly, within one piece of software, you can have APIs - software architecture is built in such a way – which is often the case – that there are different kinds of building blocks, each of which have their own interfaces via which they communicate with other. Anna also asked me to discuss AI briefly. AI per se has not been clearly defined as to what it is. But basically, it can be divided into two: first, there was a rule-based AI that's basically ordinary coding with rules and commands that are similar to those of other, non-AI pieces of software. In a way, it's even kind of random as to what is and what is not called AI, when it comes to rule-based AI. In practice, it's only based on having complex rules and lots of computing ability. The chess software Deep Blue, for example, is a rule-based AI that is based on the computer being able to process so many kinds of chess situations so fast that it ended up beating the best human chess player in the world. In principle, a human with pen and paper could achieve what the Deep Blue does, it would just take a ridiculous amount of time, and there is no time for that in a chess game. Another thing that is mostly discussed these days when people talk about AI is machine learning. Unlike AI, machine learning has been defined much more specifically as to what it's about. In practice, a human being codes a machine-learning algorithm. In other words, they tell the machine how… well, there is always a teaching material.

You cannot see it from the picture in the slide very well. On the top left, you have labelled material, with those kinds of different figures, and they have been given labels, presented below: triangle, circle, and square. Then the machine-learning algorithm is given this material. It's kind of similar to how children learn and how people learn things – they're told: that's a ball, that's a square, and at some point, the machine-learning algorithm begins to see: right, when those things are around, it's a square, and when those things are around, it's a triangle. After you've taught the model to it successfully, it must be given a test material that's different from the teaching material, so that you can test whether it actually understands what those things were. In the example presented here, the machine-learning algorithm did a great job: it was given a green ball and a yellow triangle, and it was able to label them correctly. And when it comes to these things, you have to keep in mind that if the data fed to the model is bad or biased, so is the model. In the book, we had this example: in the teaching material, it had been taught to identify a cow from a picture. When it was tested with the test material, it came to light that in all, or almost all, of the pictures in the teaching material, the cows were on lawns. Hence the model learnt that whenever there was lawn on the picture, it is a cow. No one meant this to happen, of course, and I guess the cow example is kind of harmless, but say, if you teach… I think at least in the USA, there have been examples of how they have developed an automatic system for the judicial system, trying to weed out racial data, for example. But since the address data was included, and there have been… for example, in certain predominantly black residential areas where there have been more crimes, in practice, it has ended up with black people getting more severe punishments, even though (--). And there are numerous similar examples. And since people don't necessarily understand or have the visibility to how the machine has come up with the model, a problem with transparency emerges: people don't necessarily see how biased the system is, there is no understanding on (the source of the problems). But let's move forward, for the time is limited.

To wrap things up, I have several slides actually, but I thought I would quickly go through how software is made. So the thing is all software projects basically have the same stages. They (don't necessarily follow the same order), and in smaller projects, one person may do everything. In bigger projects, each task may be assigned to several people. In any case, at least one should map out the need as to whether it's necessary to make the software in the first place. If the decision is that it is a good idea indeed, you have to define what it does. You have to design it technically, how it's realised (--) and test whether it works as it should (--) deploy it and maintain it. On the right-hand side, there are two pictures. The top one describes a so-called Waterfall model, in which you finish the stage for the entire software, then freeze it, and move on to the next stage. This was the way to make software for a long time, but in actuality, this doesn't work very well, because at the point you start doing things, you don't have a clear picture of what the desired outcome is. Secondly, at the point the software is ready, it's most likely already outdated. The bottom picture describes an agile method called Scrum that is often used. The idea of that is that all the stages are repeated iteratively, continuously; you do a little, you do a little more, you do a little more, and at each stage, you ask for feedback from the client – what does this look like, is this the right direction? In case something needs to be fixed, it's fixed immediately, unlike in the Waterfall model where you do that only when everything's complete and fixing things up is extremely costly. Now briefly the different stages. So like I said, the first stage is where you think about the problem that needs to be solved, what solution alternatives exist, and whether there is need for software to begin with or whether it would be a better solution to change the company's internal processes, or to make use of a piece of software that is already in use. If you decide to proceed with it, you have to first define what functionalities it should have. In the defining stage, first of all, functional requirements, but what's at least as important and what's often forgotten is non-functional requirements, i.e. so-called quality requirements. So for example, how the software has to be available to the users, at what times of day, how big numbers of users it must withstand. Things like that.

So in technical design, the software architecture is considered: the building blocks it consists of, how they communicate with each other, what programming languages are used, and what kinds of databases are used. Like I said, there are different kinds of databases, and they have different purposes. There are also different environments in which the software is made. So there is the development environment where people basically code. Then there's often one or more test environments where you can test different things. For example, the functionalities themselves, but also things related to performance: will the software crash if 5,000 users suddenly turn up? After testing and making sure that… and the test environment should be as close as possible to the final use environment. After that, it's taken to the production environment. Sometimes the smallest projects only consist of the production environment, but there is always the risk that, in case there's a bug in the code, it's immediately (visible to the) users.

What you do is you code. You find the errors in the code. One important part is documentation, so that later… which in particular makes it easier to make later-stage changes, as well as version management – the use of GitHub, for example. So that you have a place for the master code, and different coders can make change things there themselves. Only once they've made their changes and tested them, it's introduced in the main code. Then there's the testing, the goal of which is, of course, finding errors. Testing is done along the journey on different levels: from individual small strings of code to entire systems and how systems work together. Finding an error does not always mean that it will get fixed. Sometimes it is decided that it's, for example, way too expensive to fix, so a conscious decision not to fix it is made. As for the deployment and maintenance stage, in the Waterfall model, the deployment only happens at the end. There's also a… an often-used agile method is called DevOps which practically means that everything proceeds to the production stage automatically whenever there is new code. It requires that the code is stored on the cloud. Every time you introduce new stuff in the master code, the automated tests are run, and if everything is fine, it proceeds to the production stage automatically. Of course, there are also hybrids: for example, new versions (--), or if need be. In the maintenance stage, the question is how problems are handled, how it is monitored whether (things work properly or not). If the systems are critical, the monitoring has to be significantly more massive (--), say, a game (that is not even time-critical). (--) how to react to it: whether you fix the code, or what's causing the error in the first place, and how it gets fixed. The maintenance level is (largely) defined by the quality requirements. So if it is meant for continuous use and if there must never occur a downtime of more than a minute, the requirements are naturally very different from a robot (vacuum cleaner) in a living room corner. It is often defined by the service level agreement, i.e. the SLA.

Thanks, questions?