Of Cyborg Developers and *Big Brother* Programming AI

Carol V. Alexandru
Software Evolution and Architecture Lab
University of Zurich, Switzerland
alexandru@ifi.uzh.ch

Harald C. Gall
Software Evolution and Architecture Lab
University of Zurich, Switzerland
gall@ifi.uzh.ch

1. Wednesday, 13. November 2047 – A day in the life of a software analyst

Meet Ana, AI data integrator for INGSOFT. The time: 07:32 am. Ana's personal assistant, Lucio, has been monitoring her heart rate and breathing throughout the night using just his microphone and notices that now is the ideal time to wake up: outside of REM sleep and after roughly 8 hours of sleep – a duration which has been determined from the experience of countless previous nights and consisting of pre-sleep activity, detected mood in the morning, performance throughout the following day and over a dozen other biometric and behavioral indicators. The assistant sounds a gentle alarm at the appropriate volume. After a brief shower, Ana reads her edition of PG (the Personal Gazette). Given the work cycles of her teammates, Lucio suggests to work from home this morning. INGSOFT's main product is 'Metadapt', an AI that dynamically adapts running AI agents for new situations. For the past few weeks, she's been immersed in deploying Metadapt on the cities traffic system. The 2048 Summer Olympics are coming up, and the AI routing self-driving cars and their vintage counterparts is projected to react poorly to the unexpected change in traffic volume and travel patterns. Ana sits down at her workstation and is assigned the first task for today. 'Gone are the days of issue trackers and SCRUM meetings' she thinks, briefly reminiscing her days as an intern at one of the derelict 'coding zoos' as they've come to call the old-fashioned, cubicle-compartmentalized office buildings of the past. The biocam, installed in her workstation and unobtrusively monitoring Ana's heart rate, skin temperature, eye movements and pupil dilation, notices her mind wandering and turns on the coffee machine in her kitchen. Ana's been tasked with talking to the people at Publitrans, who run the cities personal public transport system. They have some vital information regarding anticipated changes in cab traffic during the Olympics. Ana frequently gets these assignments, as she has excellent communication skills and also knows how to speak to people such that Metadapt can follow the discussion. As the smell of her favourite brew permeates the apartment and Ana moves to the Kitchen, Lucio takes the opportunity to tell her that the meeting will take place at 9am. INGSOFT has recently subscribed to WaveVR, that new business communication tool consisting of a few tiny laser projectors and a set of comfy gloves. 3D images of her two contacts at Publitrans, captured using their biocams, is projected into Ana's eyes as she moves around her office. The gloves help to facilitate personal contact by inducing haptic feedback where necessary and help the biocam to accurately track movements such that participants can easily manipulate other objects projected into the virtual 3D scene. VR has been around for over 30 years, and the experience can still be flaky at times, but it beats wasting hours on daily commutes, plus it saves a lot of energy – with the biosphere going to hell and whatnot. Ana's job as a data integrator consists of figuring out which pieces of additional information can help Metadapt make the right choices in governing AI, and also implementing the interfaces between different data sources. Sure, there are ongoing efforts to facilitate the automatic, need-driven data exchange between the countless AI systems governing every aspect of the computerized world – like they've implemented between Canada, the US and Mexico, but a global solution is still impeded by slow-moving political and social structures. Plus a fairly unique event such the Olympics still requires significant human intervention. The meeting starts, and as one participant struggles with the automatic volume adjustment of his mic ('how have they still not solved this problem!', Ana thinks), the other two start discussing the topic at hand. Metadapt is listening in on the conversation to capture those details, which it recognizes as containing new information. This allows Ana to later formalize the conversation and guide Metadapt in synthesizing the necessary interfaces. As the meeting ends, Lucio preemptively checks if any of Ana's friends happen to be in the area for lunch today. He prepares a few suggestions for the time and

location for lunch, based on the daily offers publicized by nearby eateries, how busy they are, as well as Ana's current dietary needs and preferences. Ana finalizes the formal interface specification connecting Metadapt to Publitrans' data endpoints by applying necessary corrective measures and as she gets ready to leave, Metadapt is already scanning Publitrans' historical data relevant to improving traffic flow in the coming year.

2. Introduction

For millennia, humans spent the majority of their lives performing manual labor to provide for themselves and their community. The advent of the industrial revolution initiated a brief, but radical shift to a service economy where fewer and fewer people are employed to provide for our basic needs. Many jobs were lost, but innovation also created countless new professions. Now, with the advent of the information age, humanity is experiencing another great shift, where menial service jobs are replaced by software, en masse. Like during the industrial revolution, critical voices fear that the loss of jobs may outpace job creation [4]. They argue that new innovations in information technology often have an immediate, global impact and can generate tremendous value, while only involving a few people in their inception. For example, the automotive industry has been a strong driver of innovation throughout the industrial revolution, and by 1979, General Motors employed over 800 000 workers and generated 12 billion USD of revenue (adjusted for inflation) [7]. In 2016, Alphabet (a.k.a. Google) earned 19 billion USD, while employing just 72 000 workers [8]. Many service industry jobs are already being replaced by AI: Middle management, accounting, risk assessment, human resources – in all these professions, humans are competing with artificial intelligence. Software developers have so far been spared, but this is likely subject to change [4].

Traditionally, computers were known for excelling at those kinds of tasks, which humans find difficult, such as complex mathematical computations or operating on large datasets. The current wave of machine learning, however, specifically aims to imitate human skills. Problems such as recognizing objects in images or distinguishing spoken words from background noise come naturally to us humans, while previous attempts to manually replicate these skills in software have been met with limited success. Today however, as we are no longer writing explicit solutions for these problems and instead let computers learn by themselves, they are rapidly catching up with us.

In most cases, the models imitating human skills are trained on what essentially constitutes 'previous experience'. That is, the models are provided with data and some human decisions, such that they learn to make the right decisions autonomously, given specific circumstances. Ergo, we argue, all that is needed for these approaches to start imitating human programmers is the quality of observational data available to the systems. Consequently, we make the following assumption and pursue its consequences throughout this paper:

> A modern AI that can observe every interaction of human programmers for long enough will eventually be able to imitate their work.

Based on this assumption, we discuss the impact of the continuing progress being made in machine learning, and how it will affect the software engineering profession. We particularly reason for three specific consequences that will profoundly change the field of software engineering:

1. Software developers become tightly integrated with artificial intelligence. They become, to a certain degree, *Cyborg developers*.

2. To facilitate this process, invasive but unobtrusive monitoring methods will become more widespread.

3. Teaching software engineering re-orients itself to focus on innovation and the requirements of not the current, but the future generation.

3. Learning by watching a million teachers

The vast majority of effective AI systems today use some form of supervised machine learning. Large datasets of high quality are necessary to train these systems, but their resulting capabilities are exceptional.

A common example is the 'Captcha' mechanism on websites where human visitors are asked to perform a small task that is hard to automate using hand-written software: for example, reading a blurry line of printed text, or selecting all images that contain a certain object, like a car or a road sign. These tasks are used to prevent simple crawlers from accessing and wasting online resources. However, a beneficial and certainly not unintended side effect of these 'Captcha' mechanisms is that the human inputs can be used to actively train an artificial intelligence on the given task. By this

mechanism, properly scoped tasks are now being solved using machine learning, one problem at a time.

But not only low-level tasks such as these are learned by machines. Higher-level service industry jobs are also, *today*, being imitated and replaced by AI: a US company has built a project management software which, given a formal description of a task, autonomously hires freelancer with the necessary skills and coordinates their effort. In a pilot study, their product oversaw the creation of a 124-page research report for a Fortune 50 company, involving 23 freelancers, including writers, topic experts and proofreaders [6]. Apart from the actual end result, tools such as this may, in the future, be able to not just take over coordinative management work, but at the same time determine the skill level and quality of work of individual contractors, such that it can prefer better workers over weaker ones. Not only that: as it splits a problem into clearly defined tasks, it can start to correlate tasks and their human-supplied solutions, slowly learning to provide certain solutions on its own, without the need for a contractor.

What these systems have in common is that they allow, or even actively elicit, human problem solving skills for narrowly scoped tasks. And depending on the problem domain, it may take fewer or more examples to train a model that can imitate human behavior, but for many of the menial management jobs and day-to-day maintenance tasks, it appears to be just a matter of time.

4. The 'Big Brother' programming AI

Given the trend of artificial intelligence 'watching' humans perform certain tasks, it is not far-fetched to assume that a similar trend will occur in software engineering.

Imagine a software comprising an issue tracker and a version control system. Issues are specified as formally as possible, i.e., not just consisting of plain text and severity, but capturing accurate information on affected artifacts, actual and expected outcomes and other rich metadata. Likewise, any change submitted to the version control system is atomic (not mixing up several changes in one commit), and described with additional metadata, as well as the issue it relates to. The software classifies issues based on their metadata and distributes them to individual developers. Over time, given enough examples, and correlating the consequences of each change to future issues, it can not only learn the strengths and weaknesses of individual developers but also learn to correlate issues and their codified solutions. After a while, simple issues (such as "prepare repository for next major release") can be performed automatically, while issues requiring human intervention can be routed to those employees most likely coming up with the best solution.

But why stop there? Given that there are millions of developers using the same programming language and similar frameworks (say, Java and the Android Development Kit), it is likely that any one problem has been solved dozens of times by different people, somewhere in the world. Assuming a hypothetical 'Big Brother' software is able to track every keystroke, every application used and every website visited, it should be able to discern signal from noise by comparing interactions of different users and consequently imitate relevant behavior. Tie in user-reported errors (e.g., mobile app ratings), and the system becomes able to exhibit self-correcting behavior. Once such a 'Big Brother' programming AI has been developed and widely distributed, for example as part of a major IDE, it is only a matter of time until some proportion of issues will not require human intervention to be solved anymore.

Naturally, there are several problems inhibiting the development of such tools. There exist ethical and legal issues with monitoring employee activity [5], thus the blanket-surveillance of developer interaction with the machine may be unfeasible in the near future. With developers switching between work items and brief social media- or communication breaks, discerning signal from noise may also be difficult.

5. Augmenting human developers

That said, early adopters are already willing to let artificial intelligence peek over their shoulder. For example, FlowTracker [12] captures all developer activity (keystrokes, mouse movements and scrolling), analyzing it on a personalized basis to determine how busy a developer is. Several hundred test subjects used FlowTracker in an industrial, long-term study, and 80% of questioned users suggested that they will continue using the software beyond the duration of the study. Biometric sensors, which monitor the physical state of a developer are being implemented in early lab studies. For example, heart rate variability and electrodermal activity have been directly linked to perceived task difficulty [10], and may help the computer to provide adaptive help.

Indeed, while workforce monitoring is overcoming privacy- and other related issues, customers of products such as Facebook or GMail willingly open their private and public lives to data-driven analysis already. Products such as 'Google Assistant' only ever get better with increased usage.

If we can solve privacy-related matters, a ubiquity

Table 1. An example of plain-text input, lexing instructions, resulting tokens, and corresponding annotations

Input	`List<Throwable> errors = TestHelper.trackPluginErrors();`
Lexing	`000110000000011 000001 1 00000000011000000000000000001111`
Tokens	`[List] [<] [Throwable] [>] [errors] [=] [TestHelper] [.] [trackPluginErrors] [(] [)] [;]`
Annotation	`[ClassOrInterfaceType|14] [TypeArguments|15] [ClassOrInterfaceType|18] [TypeArguments|15] [VariableDeclaratorId|15] [VariableDeclarator|14] [Primary|19] [Expression|17] [Expression|17] [Expression|16] [Expression|16] [LocalVariableDeclarationStatement|11]`

of self-improving AI agents could be highly beneficial to software developers. It could 1. prevent mistakes 2. distribute work-loads on a personalized basis, exploiting the strengths and considering the preferences of any specific developer, and 3. over time, given a 'Big Brother' type AI, even provide partial or complete implementations.

6. Can computers learn to program?

Of particular interest (and considerable difficulty) is the last of the points above. Can we teach computers how to program? What does it mean *to program* in this context? In dealing with these questions, we performed two experiments which we briefly summarize here.

6.1. Imitating how humans read code

In one experiment [3], we first considered how humans learn to program. Any respectable programming tutorial will start with a "Hello, World!" and explain the different parts of the respective source code. For example, in Python, the instructor may write `print("Hello, World!")` and explain that `print` is a function call, the opening and closing brackets hold parameters to be passed and that "Hello, World!" is a string, as indicated by the double quotes. Thus, the first thing a student learns is to recognize different pieces of code and how they relate to each other.

We were wondering if an AI could learn to imitate this 'first lesson' of learning how to program. To answer this question, we trained a recurrent neural network to 1. recognize individual tokens in source code, and 2. annotate each token with its type and probable depth in the original AST. To obtain the necessary training data, we wrote a data-extraction tool to download and parse 1000 Java projects from GitHub. While parsing the source code, the tool generates the following data on a line-by-line basis:

1. the original plain-text source code,
2. lexing instructions indicating the token boundaries in the plain-text sequence,
3. the source code tokens (as delimited by applying the lexing instructions), and
4. the type of each token as well as its depth in the parsed AST.

The tool provided us with over 50 million samples which we fed into two separate recurrent neural networks; one to recognize the token boundaries in the original plain text (with a per-word perplexity of 1.11) and one to annotate the tokens (perplexity: 1.28). The two models work in unison to tokenize and annotate plain text source code, as shown in **??** – much like a novice programmer would do.

This result begs the question, what else we could teach these models. Could we teach how variable assignment works? How types are resolved? How imports are sourced? One small-scoped problem at a time, we may teach an orchestra of models to not only process source code like a traditional compiler (which is essentially just a 'dumb automaton'), but to gain some sort of inherent understanding of software, like a human programmer has. Based on such a learning process, we may obtain combined models which are able to autonomously make higher-level decisions which are yet reserved for human developers.

6.2. Character-level source code generation

Complementing our experiment on reading source code, we performed another study with the goal of writing source code [2]. Previous work on natural language text has shown that recurrent neural networks are able to internalize not only syntactic and grammatical structures, but, to some degree, semantic relationships in natural language texts. For example, Sutskever et al. [11] trained a deep RNN on a text corpus from Wikipedia, obtaining a model that is able to synthesize a stream of text which convincingly resembles human-written text, containing few grammatical errors and comprising semantically coherent sentences and paragraphs.

We applied a similar model to a corpus of plain-text Java source code, resulting in a model which synthesizes a continuous stream of Java code given a few seed characters. The model has no explicit knowledge of syntax and grammar, yet it is able to correctly chain single characters into long sequences of what looks like human-written source code at first glance, including comments. While the generated source code is not compilable (and frequently contains complete, perfect recitals of the Apache license, which appears in many source files), it indicates that neural networks have the capacity to build an internal representation of what

constitutes the Java programming language.

6.3. Conclusion: we need data

Both these experiments illustrate two important, although unsurprising, realizations about how machine learning applies to programming tasks:

1. The artificial intelligence is able to match humans in *qualitative* terms. This means that, for correctly scoped problems, the machine can perfectly imitate, but hardly outperform a human in terms of the provided solution. Naturally, it can produce solutions at a much higher speed, but it is doubtful how this helps with programming issues.

2. Large amounts of high-quality training data are required. Bad examples result in a badly trained model.

However, compared to other industries, software engineering actually has a good chance at capturing all relevant data, as our profession is largely digital in the first place. However, today's issue trackers and version control tools rely mostly on simple plain-text descriptions. Likewise, specifications are usually informal or at best semi-structured. If we want to enable the next generation of recommender systems, we need to start thinking about how we can enable machine learning to increasingly learn from the day-to-day behavior of software developers in the wild.

7. Symbiosis of programmer and AI

Assuming a trend towards 'Big Brother' type AI systems, which continuously monitor developer activity, we expect that one consequence would be a gradual shift in the responsibilities and day-to-day activities of regular software developers. Today, software developers have to make a lot of decisions. They are heavily involved in bug triaging and make all the choices regarding utilized technologies, work-flows, deployment, etc, all the while relying on small-scale, narrowly scoped recommender systems (e.g., code completion, test selection, service discovery). As AI is able to take over more and more of these responsibilities, the role of the developer will not disappear, although it will move, and possibly shrink. In a way, a role inversion will slowly occur, as the AI is still unable to consider overarching functional requirements and other complex interdependencies within a software project. While today, the developer guides these intelligent systems to the required solution, in the future an AI may guide the human to perform the last, necessary tasks and supply missing pieces of information which it is unable to gather on its own. Last but not least, there is evidence suggesting that humans readily accept instructions from a computer, sometimes more willingly than from other humans [9]. Apart from a core team of data scientists, who curate, maintain and extend the AI, regular software developers (who are not working directly on the AI systems) will increasingly find themselves acting in two roles:

1. as the *glue* holding together different autonomous recommender systems, synthesizers and other adaptive systems and their data sources, and

2. as the primary interface to other human actors.

8. Limits of AI

Why then, have software engineers not yet been replaced with programming AIs? Apart from the obvious problems with privacy and concrete implementation, software development, to a certain degree, is already a largely non-repetitive, and often thoroughly creative profession. As mentioned previously, machine learning can generally match humans qualitatively (and outpace them quantitatively), but it can rarely outperform them, because it learns from human examples. Reinforcement learning may improve upon this in certain situations, but even assuming a 'Big Brother' type of AI, it is hard to imagine it coming up with a new app to solve a new and unique every-day problem, or design a new game, or come up with a new story-line that is not quickly recognized by humans as being recycled from previous examples. Undoubtedly, attempts to do so will be made, and for certain niches (say, children's entertainment or education), success is not unlikely. However, attempts to train deep learning models to generate music or visual art are severely limited and much less impressive than their properly scoped problem solving counterparts being introduced in industry today.

9. Information age software engineering education

While it is natural to interpret some of the predictions made in this paper as bleak and detrimental to the software engineering profession, AI may actually free us from many of the difficulties that make software development tedious. However, the software engineering profession must adapt to the inevitable change heading for it.

If any problem that has been solved a sufficient number of times will become solvable by an AI, does it really make sense to teach existing programming

languages and best practices to our students? Does it make sense to teach Java and design patterns? An AI that learns from real-world developer behavior will certainly weight interactions of experienced programmers (causing fewer regressions and needing less time to implement solutions) more strongly, while novice programmers may tend to introduce additional noise and faults, which the AI will weigh less. A junior software engineer may need several years to attain expert status, while learning from a handful of peers and from their own experience. However, an AI being trained using examples provided by thousands or millions of developers will certainly outpace the junior software engineer sooner rather than later.

Today, the world still lacks millions of software developers and information system experts: a 2015 report estimates a shortfall of 1.5 million experts in the information security domain alone [1]. However, observing the long-term trend, many of the people we educate today may become jobless half-way through their careers. Today's need for software developers is as much a symptom of slow-moving higher education in the past as their anticipated obsolescence in the future. We certainly need developers to keep an AI running, but today already, these are often mathematicians and other domain experts, which have little in common with the average software engineer. But assuming that AI can take over more and more repetitive tasks, the need for "code monkeys" will decrease rapidly. Instead, we need to educate innovators and, naturally, experts in machine learning and related fields.

10. Conclusion

Early experimental research suggests that many every-day software engineering tasks, such as bug triaging, reverting changes that introduce regressions, or even writing code, can likely be imitated by a sufficiently informed AI. Implementation and privacy issues are likely to be a temporary hold-up, as the sort of paradigm shift we currently experience cannot typically be stopped by legislation and as examples like Facebook show that many people will drop privacy concerns at their convenience.

We did not delve into the overarching question regarding the eventual economic outcome of the ongoing revolution. Some say that with productivity further outpacing available labour, we may eventually find ourselves in an economy with few jobs and an increasingly poor population that cannot afford the goods it produces. Some say we need some form of basic income.

The jury is still out, but one thing is certain: The progress of AI is unstoppable and it will heavily influence the way we develop software in the decades to come.

References

[1] 2015 (isc)² global information security workforce study. https://iamcybersafe.org/wp-content/uploads/2017/01/FrostSullivan-ISC%C2%B2-Global-Information-Security-Workforce-Study-2015.pdf, 2015.

[2] C. V. Alexandru. Guided code synthesis using deep neural networks. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Seattle, 2016.

[3] C. V. Alexandru, S. Panichella, and H. Gall. Replicating parser behavior using neural machine translation. In *25th IEEE International Conference on Program Comprehension (ICPC)*, Buenos Aires, Argentina, 2017.

[4] E. Brynjolfsson and A. McAfee. *The Second Machine Age: Work Progress, and Prosperity in a Time of Brilliant Technologies*. W. W. Norton & Company, New York, NY, USA, 2014.

[5] V. J. Ella. Employee monitoring and workplace privacy law. *National Symposium on Technology in Labor & Employment Law*, 2016.

[6] D. Fidler. Here's how managers can be replaced by software. https://hbr.org/2015/04/heres-how-managers-can-be-replaced-by-software, 2015.

[7] Fortune. Fortune 500: General motors. http://archive.fortune.com/magazines/fortune/fortune500_archive/snapshots/1979/563.html 3.5B USD revenue, adjusted to 2016 USD: 12B, 2007.

[8] Fortune. Fortune 500: Alphabet. http://fortune.com/fortune500/alphabet/, 2017.

[9] M. C. Gombolay and J. A. Shah. Increasing the adoption of autonomous robotic teammates in collaborative manufacturing. In *International Conference on Human-Robot Interaction (HRI) Pioneers Workshop*, 2014.

[10] S. C. Müller and T. Fritz. Using (bio)metrics to predict code quality online. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 452–463, New York, NY, USA, 2016. ACM.

[11] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 1017–1024, New York, NY, USA, June 2011. ACM.

[12] M. Züger, C. Corley, A. N. Meyer, B. Li, T. Fritz, D. Shepherd, V. Augustine, P. Francis, N. Kraft, and W. Snipes. Reducing interruptions at work: A large-scale field study of flowlight. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 61–72, New York, NY, USA, 2017. ACM.