

Maintain-ability: On Life Alongside Computer Software

Mace Ojala 

Ruhr University Bochum

Corresponding author

Mace Ojala
Ruhr University Bochum
Universitätsstraße 150
44801 Bochum, Germany
✉ mace.ojala@rub.de

Submitted: November 6, 2023

Accepted: April 28, 2026

Abstract

This paper examines what lessons can we learn about living with technology from software maintainers who struggle to keep digital infrastructures – at least most of the time – in good running order. The empirical material of the research was collected at three events as programmers convened to give testimony of disruptions of code and life. Inspired by hyphenation of *response-ability* by feminist technoscience theorists Karen Barad (2007) and Donna Haraway (2016), and in response to invitation for *broken world thinking* by Steven Jackson (2014), I theorise the concept of *maintain-ability* and demonstrate its application to foreground the capacity to not only give but also receive care which holds together more-than-human worlds at the dawn of the third millennium. The findings nuance and specify the notions of maintenance documented in science and technology studies literature, and weave it into life. Software maintainers exercise considerable agency over the material in their immediate care: code. However, in doing so they also find themselves having to articulate dynamic, interdependent and hybrid networks of relations which they are intimately entangled with, and whose durability depends on the success of their ongoing, indeterminate reconfiguration. Both the programmers and the software they maintain must continuously navigate risks of breakage, burnout, bugs or falling into obsolescence.

Keywords

software studies; software; infrastructure; maintenance; care; life.

1. Life Alongside Computer Software

I enjoy spending time playing and sculpting sound with music synthesizers. I do it almost every day, and consider it my “quality time with algorithms.” Moments of creation, self-expression and attentive listening to these specialised computers offer a welcome counterbalance to research work done in study of science and technology, which, in the interest of making room for socio-cultural critique, tends to background such validating and self-actualising, constructive relations of positive subjectification with computer technology. While programming sound, the famous concluding words of Donna Haraway’s early programmatic and arguably old-school *Cyborg Manifesto* sometimes echo in my mind: “I would rather be a cyborg than a goddess” (Haraway 1985).

In addition to enjoying making computer music itself, my artistic practice involves an ongoing intuition and interest-based exploration into acoustics, mathematics and pragmatics of various areas of synthesis, sound design, and exploration of software and hardware built or appropriated for purposes of generating, manipulating and organising sound. YouTube offers a vast repository of knowledge contributed by artists, engineers, corporations, boutique synth designers, influencers (“synthluenzers”), professors, and others. Anecdotaly, the YouTube recommender algorithm seems to do a relatively good job of suggesting interesting content for me to watch, especially in relevant areas of mathematics regarding synthesis, sound mechanics, music theory and unorthodox ideas from the history of sound synthesis and experimental music, as well as tutorials and reviews of the synthesizers I own. Much of what I am learning from this user-generated content – mediated via YouTube’s behemoth data infrastructure, as well as the search and recommender algorithms – feeds back into my artistic practice and my ongoing relationship with the software running on my computer and synthesizers.

Besides the private, of course the work parts of our biographies are bound up in personal relationships with software. While artificial separation of life into work and non-work is as necessary as it is treacherous, I encourage you to recall the moments and experiences you have shared with your word processor, web browser, PDF viewer, analysis program, and other familiar work-software which have been interwoven, sometimes more and sometimes less, with your professional commitments and projects. This is relatable and indeed the condition of all white-collar workers, including us academics.

Or consider the familiar operating systems such as GNU/Linux, macOS, Windows, Android and iOS which we teach our children to use; how to manage files, install software applications and keep them updated, customise settings, manage resources and mind cybersecurity.

As sound artists, as researchers, and as children and parents, we exist alongside computer software as users not only in a practical sense, but also existentially as we are interpellated (Althusser 1971) to take on “the user” as our identity and our assumed subject-position. Not one bundle of relationships, let’s call it *a life*, is the same as another one; each has a unique story, expanding and compressing, doings and undos of relations. Disrepair, decay and obsolescence affect not only the practical going-on of life, but goes to the core of our identities; what is it to be an artist with broken instruments, a researcher with buggy infrastructure, or a parent in a world of cyberthreats? Why does our companion software – most of the time anyway – generally seem to work? Where do software updates come from? Who is working behind the screen?

This article is on the folding and unfolding of certain biographies alternative to “the user,” and identities forming in very close and formative intimacy with software over long periods. Namely, the programmer life. More precisely, the focus is on a specific kind of programming life which is less occupied with developing new or soon-to-come software, and more or entirely with keeping existing software in good shape.

This is an article about the lives of software maintainers.

The aim of this article is to give nuance to such kind of programmer life. I do so by presenting the modal concept of *maintain-ability*: the ability to not only give but also to receive care. It is not in my interest to develop maintain-ability in quantifiable terms, that would be beside the point. Instead, my research aims to describe the texture of life intimately entangled with, and in existential sense maintained by computer software, and in particular as that life is lived

alongside and in co-dependent entanglements with existing software. At the centre of this analysis is a vignette titled “Maybe one more day per week...” which collects, illuminates and intensifies participant observation and unstructured interviews from my fieldwork at three meetups, where practicioning programmers gathered to thematise and discuss taking care of software. The vignette is an analytic construct which synthesises my fieldwork, and gives condensed character to the experience of living with software. I call this character “Stefan.” Following the vignette, the analysis will identify and interpret a selection of relations “Stefan” has with a computer program in his care. I will illuminate a fragile web of relations in the intersection of which “Stefan” is maintained as an *actor*. The article contributes to science and technology studies with an understanding of software not as a stable object but as a web of relatings, and a more holistic and fuller understanding of human experience of software life.

2. Caring about Maintenance and Repair

Questions of repair, maintenance and care have received considerable attention in science and technology studies over recent years. Journal issues dedicated to the topic include for example *Tecnoscienza* volume 6 issue 2 (Denis et al. 2015), *Nordic Journal of Science and Technology Studies* volume 9 issue 1 (Lindén and Lydahl 2021), and *Engaging Science, Technology and Science* volume 9 issue 3 (Denis et al. 2023). The late Bruno Latour recognised very sharply the importance of thinking both human and non-human life together through time (e.g., Latour 1987). Shannon Mattern has collected art and other projects into an argument which foregrounds and celebrates maintenance, usually done in the background (Mattern 2018). Beside their empirical work from the metro and streets of Paris (Denis et al. 2015; Denis and Pontille 2021), Jérôme Denis and David Pontille give a succinct answer to the question *Why do maintenance and repair matter?* (2019). Their answer is not surprising to an STS audience well versed in thinking about infrastructures of the everyday, and politics thereof.

Maria Puig de la Bellacasa built an influential posthumanist theory of care on works of Joan Tronto, Bruno Latour and Donna Haraway in her 2017 book *Matters of Care: Speculative Ethics in More Than Human Worlds*. These ideas and their call-for-action have resonated far and wide. No literature review on these themes, no matter how brief, can be considered sufficient without mention of Robert Pirsig’s 1974 hugely popular, autobiographical travelogue *Zen and the Art of Motorcycle Maintenance. An Inquiry into Values*. What this book so insightfully foregrounds is deep formative and reparative human relation with artifacts – a philosophical aspect of material culture without which any analysis seems to me lacking and unconvincing. Steven Jackson’s already canonical chapter *Rethinking Repair* (2014) collects writings on existential meanings of living with artifacts, including Martin Heidegger and Richard Sennett on craft traditions, and the familiar names of Sherry Turkle, Lucy Suchman, Katherine Hayles, Donna Haraway and Bruno Latour inviting information technologies too into the moral sphere of more-than-human becoming (*ibid.*, 232). In philosophical literature we find progress to systematise theory of people’s identity-formation from intimate, affective relations with objects (Piredda 2020; Viola 2021). There is no doubt that software are affective artifacts for my informants, as we will see below.

Other notable monographs which contribute empirically, conceptually as well as programmatically to social study of repair and maintenance include Christopher Henke's and Benjamin Sims' *Repairing Infrastructures: The Maintenance of Materiality and Power* from 2020, and Lee Vinsel's and Andrew Russell's compelling case for maintenance in their 2020 book *The Innovation Delusion: How Our Obsession with the New Has Disrupted the Work That Matters Most*. Vinsel and Russell draw from their persistent work in The Maintainers movement. The name of their book names the typical antonym of maintenance, namely innovation; and nothing invites what Vinsel and Russell call "innovation-speak" like the digital. The exciting, novel, and soon-to-come software consistently dominates the technological imagination (Balsamo 2011), masking historical and material facticity of software, and that most of software is actually old rather than new; we are already deep into the digital. My informants, software maintainers, always need to navigate their efforts against what innovation-speak, and the drive to invent and build new systems and features, rather than cleaning up and paying the "technical debt" taken during their design and development.

For programmers, maintenance has been a background concern since the beginning of modern computing in the middle of the 20th century (Ensmenger 2014). Seth Erickson and Christopher Kely theorise temporality as a vital aspect of software terms of life and evolution (2015). Indeed, thinking in terms of life, evolution and "lifecycle" are normal conceptualisations in software engineering (Ruparelia 2010; Kneuper 2017). However, it is Marisa Cohn's work based on ethnography at Cassini mission at NASA which so evocatively weaves together the lives of software and lives of people (Cohn 2016; 2017; 2019).

3. Caring for Code

Before presenting and analysing the fieldwork results in the form of a vignette, a clarification regarding the scope of this research is necessary. By "software maintenance" I focus not on all imaginable activities and practices of tending to computing systems and keeping them running. Instead, I focus narrowly on maintenance of source code. Source code is the functional specification of computer software. Code is typically composed in artificial programming languages expressed in text. In popular depictions code is often marked by dark backgrounds, monospace fonts and proliferation of parentheses (for fruitful critiques of "sourceness" of source code as well as textuality of computer programs, see for example Chun 2008 and Couture 2019).

Generally, repair and maintenance tend to be conflicting topics burdened by disinterest and stigma (Puig de la Bellacasa 2017; Mattern 2018; Denis and Pontille 2019; Vinsel and Russell 2020). So it is with software maintenance too (Ensmenger 2014). Working with old, existing systems is considered dirty toil. This sentiment is appropriated and intensified in Jaakko Koskenkorva's contribution to *Classic Programmer Paintings*, a sarcastic Tumblr upcycling known masterworks from the graphic arts by renaming them with bitter-sweet titles relevant and relatable to programmers.

The original 1893 oil painting "Raatajat rahanalaiset / Kaski" by Eero Järnefelt depicts in national-romantic style the impoverished and malnourished lower classes toiling in slash-and-burn settler agriculture in Eastern Finland at the end of 19th century (Salo *n.d.*).



Figure 1.

Eero Järnefelt, “Programmers at work maintaining a Ruby on Rails application” (1893), oil on canvas. Collaboration from Jaakko Koskenkorva.

An uncultivated wilderness behind the six figures extends as far as the eye can see. The updated title for the 21st century is “Programmers at work maintaining a Ruby on Rails application” (Koskenkorva and Järnefelt 2016). Rails is a popular programming framework for creating (and leaving behind) web applications in the programming language Ruby.

A codebase is a lifeworld from which to extract sustenance as well as identity. The larger ones have proportions of a landscape. While the measure is controversial, the size of a codebase is often measured in lines of code (LOC), and a program can span from thousands to millions of lines. Legacy codebases are chaotic, brutal, inhospitable places. Fixes must only be done locally, and expectation is that what is laboriously established here, must soon be abandoned. Getting an overview is impossible, and there is no grand plan – scraping a living is uncertain in a landscape like this. Hence the humour of *Programmers at work maintaining a Ruby on Rails application*. What matters is that any non-trivial codebase is too large for an actor to meaningfully hold all at hand and manipulate at once. Its materiality and (arte)facticity resist idealisation and Philosophies of being, inviting Philosophies

of *becoming* instead. “Maintainers are Bergsonian,” as Jérôme Denis’ and David Pontille’s inspirational cliffhanger PowerPoint at the European Association for Study of Science and Technology (EASST) 2022 panel on repair and maintenance proclaimed (Denis and Pontille 2022). Successful maintenance therefore always implies success in both knowing and unknowing over time, and articulating the string-figure, or a network of care on which the software object depends (Ojala and Cohn 2023).

Limiting my research scope of software maintenance to maintenance programming will bracket out the work of system administration, content moderation, hardware operations and repair, corporate management and many more admittedly necessary practices to keep the digital lifeworld running. These cuts are analytic rather than empirical or necessary, but as *agential cuts* (Barad 2007) they are rather enabling than disabling conditions of this research. In the tradeoff, I achieve a focus on manipulating and managing source code, in other words the single (arguably) quintessential aspect of computer life: programming. The element of maintenance focuses on materiality and diachrony of software as aspects of the actual lifeworld.

4. The Fieldwork

I attended three international tech meetups in 2018 and 2019 in Berlin, Paris and Washington D.C. At the meetings people who program convened to discuss the specific issues of maintaining existing software. All the events had the word “maintain” in their name, an exceptional move in the world of tech which tends to disregard themes of care and long-term thinking (Vinsel and Russel 2020). The events piggybacked on larger industry events as full-day, unconference style side-events. The programme was organised as a mixture of formats. Talks and presentations were given from the front of the room, while most time was spent in so-called “BarCamp” format, where participants first decide together on a topic to discuss, and then split to round-table sessions. All three took place in large, rather generic event conference centres. Between 20 and 70 participants attended each. Unsurprisingly, most participants were men, casual and informal in habitus and interactions. The gender bias is widely documented in critical literature (e.g., Ensmenger 2010), while communication styles and demeanour are often outside the focus, something I hope to partially remedy with this modest contribution. Inferring from what they said, everyone was a competent and experienced software developer. I collect and use these biases and propensities in the character we meet in the next section. All programmed for work or for pastime – or both, as that boundary tends to be rather blurry, as the section below will likewise show. Besides programmers experienced enough to get entangled with longer-term lifecycle issues first hand, there were a number of volunteer community managers from open-source projects, engineering team leads, some participants in more corporate roles in companies whose products are aimed at issues pertaining to aging software, and representatives from organizational initiatives to compensate crowdsourced labor such as open source programming (themselves venture capital backed startups, it must be added). The events were open to ticket-holding public, although targeting maintenance-oriented subsegment

of the audience of the industry events they were adjacent to. The events were conducted entirely in international English language. Indeed, many of the participants had travelled internationally to meet, myself included.

These events presented a valuable opportunity to listen to maintenance programmers give testimony of their lives alongside computer software to their peers, seek and receive empathy from one another, and share advice based on their private experiences and various socio-technical, economic, psychological and critical reflections, theorisations and world-views.

Prior to venturing out to conduct fieldwork, a colleague and I built a thematic research lexicon from desk analysis of online discussions about legacy code on blogs, Reddit, StackExchange, GitHub and HackerNews. This lexicon gave me a set of *companion concepts* to enter the field with (Winthereik 2020). To name some examples, “technical debt,” “dependency hell,” “modernisation,” and “desire to rewrite from scratch” were in the lexicon. At the events I then took notes of what literally was said, as well as my observations of various ongoingings. With the research lexicon as my companion and aide, I paid attention especially to expressions of affects, descriptions of dependence, and biographical struggles. To complement participant observation, I conducted a dozen unstructured interviews with the same companion concepts and the same research focus. After returning from the field back to the desk, I coded my notes for dynamics and ambiguities of the programmer life, as expressed by those who live it. From this and with critique from ethnographer colleagues in our STS research group,¹ I built a small handful of vignettes to substantiate the generic, synthesise the recurring, and intensify what was specific to programming and important for my informants, as well as to invalidate what I had expected but did not in fact encounter.

To present this fieldwork, the following vignette collects, intensifies and communicates what was discussed pertaining to *maintain-ability*, the ability to give as well as receive care in existentially necessary (although not sufficient) relations of becoming. As an item of ethnographic writing (Clifford 1983) the following does not represent a specific moment of fieldwork. Rather, its nocturnal setting, open mood, and informal literary and photographic style express the atmosphere and patterns of interaction among programmers at these quasi-professional events as experienced by the author, myself. The demographic biases, identities, appearances and habitus of the participants are condensed in a single archetypical character, a persona. I synthesised my observations about the affects, dependencies and biographical struggles the event participants voted to discuss in the BarCamp sessions into the social positionality and subjective life-concerns of this persona. The problem formulation and narrative the persona recounts in the vignette summarise the relational dynamics my observations and interviews revealed. After introducing my concept from the title of this article, I will uncompress the singular persona back to my material throughout the two sections “Stefan’s seven entanglements” and “Software dependencies” which follow the vignette. The specific software discussed is one of the real software applications whose solo maintainer I interviewed, anonymised, and which I present in further detail below as it exemplifies the conflicting entanglements of programmer life.

That said, let’s step in.



Figure 2.

A street milieu during warm, dark urban late Spring night. Photograph by the author.

5. “Maybe one more day per week...”

It is close to midnight, and I’m leaving the party. The warm, dark urban late Spring night engulfs me. I am satiated with culinarily quite interesting artisanal cocktail snacks sponsored by the hosting megacorp whose annual event for software developers was held earlier during the day. For this I am grateful, as I saved 3€ on dinner döner. I am leaving together with Stefan, a sympathetic ponytailed and bearded man in his mid-thirties.

As we step out, we pause on the street to finish our conversation before getting a decent sleep for tomorrow’s maintainer meeting. He shares with me something he has hesitated to tell his boss: that he would like to reduce his work from four days to three, and attend more to his pet project, a game world creation tool.

It had all started as something just for himself in his computer game creation hobby. The existing tools for world-building were too expensive. And like many programmers, he had shared the code online. During the past decade many other game creators had started using it. Growth of the indie game culture was the reason for the uptake, he tells me. While now there might be enough crowdfunding from small game development studios and hobbyists that he could afford a second day a week on it, the boss at the “real job” had already been a bit unhappy to lose his expertise one day a week.

Expressing the wish for moving to a three-day workweek might cost him his enjoyable programming day job. But the wish to better maintain and develop this world-building tool on which many computer games and their creators made themselves dependent is strong and felt like the right thing to do.

That hesitation wouldn’t be resolved tonight he concluded, as we wished good-nights and bis-danns, and walked our own ways.

(Ojala 2021)

Considering all the entanglements, we are left to ponder what will happen in this network of entanglements if Stefan proposes to his boss to go down to 3 days per week? What if he does not?

6. Maintain-ability

I theorise and work with the concept of *maintain-ability* to foreground the situated, fragile and often underappreciated capacity not only to give, but also to receive care (Ojala 2021). The concept is inspired by Karen Barad’s hyphenation of “responsibility” into “*response-ability*” (Barad 2010) and Donna Haraway’s lush use of it in her influential book *Staying with the Trouble* (Haraway 2016). What the hyphenation achieves for my analysis is to separate the activity my research focuses on, namely maintenance and especially maintenance of software, from the conditions of its possibility. It is a modal term: the ability to participate in relations of maintenance, a possibility but not more than that. Consider another *-ability* word familiar from popular discourse, “sustainability:” *sustain-ability* does not guarantee sustenance of human life on this damaged planet, it only enables it. In Bergsonian terms maintenance is virtual, and finds its actualisation (Deleuze 1988) along the lines of what makes maintenance possible, along the lines of *maintain-ability*. In other words, the concept of *maintain-ability* foregrounds what is necessary, but not yet sufficient for one thing or another to actually get maintained.

7. Stefan’s Seven Entanglements

Let’s now attend to the vignette with the concept of *maintain-ability* in mind. Here we hear the voice of a character who is navigating multiple entanglements, of which I name seven.

1. First and foremost we have the relation between Stefan and the world building tool, ie., a software as a body of code he has been primary author of for over ten years. The relation hinges on Stefan's ability to attend to the aging codebase, fix bugs, keep it hosted and available. Similarly, the source code must remain accessible, relevant and interesting to aging Stefan. Both pass through time and change in an indeterminate and open-ended process of becoming. As surely as the world building tool is same-but-different from what it was ten years ago and at each step of its biography, Stefan too is same-but-different.
2. Turning to the "what" Stefan speaks about, the proper topic of what is being said, namely to the status quo employment relation Stefan has in the company. The job pays the bills, maintaining Stefan practically at the level of subsistence via a sufficient and predicable salary payment. He does enjoy the job, therefore being maintained more deeply too. This relation depends on Stefan's employability, which is the hesitation explicit in the vignette; is a three-day employee, no matter how competent and liked, a good idea for the employer? This question concerns Stefan, and the vignette intensifies the hesitations based on feedback from going from full-time to part-time earlier.
3. Third relation, related to the previous one but more personal, relates Stefan and his current boss. Having employees and exercising decision power over them maintains the employer as an employer – a classic Hegelian dialectic of the master and the slave (Hegel 1910). This power is in no way absolute, as is evident in Stefan's apparent possibility to try to re-negotiate the employment relation (which might or might not work). However, as long as alienation is not complete, human relations are not absent from such relations. Stefan's hesitations are oriented at the psychology of the boss ("what will the boss think?"), and their higher-order intersubjective relation (e.g., "what do I think the boss thinks?," "what will the boss think I think of him," or more prosaically "is it awkward for me if he is unhappy with what I say", *etc.*).
4. Implicit are the ways in which the code artifacts at Stefan's current day job depend on his programming work with the company's products. In turn, their ability to be workable by him allows the company to exist, and by the employee relation to continue. Stefan's ability to enjoy his work suggests that the codebase is in a workable condition, and is interesting, challenging, or easy to work with, or whichever properties of the codebase enable Stefan to enjoy it. Two separate codebases are in his care, one of them in his only. Stefan is trying to re-balance his relation to both.
5. The vignette leaves unmentioned a fifth relation, namely whether Stefan has more colleagues at the company besides the boss. That is left for us to speculate, although programming is most often teamwork and team dynamics, communication and coordination, and onboarding and letting go of contributors were consistent topics at the meetups (Ojala 2021). Perhaps a three-day Stefan would be seen as a diva, a unicorn, a loser, or would be missed. He is not considering quitting, enabling us to ponder how he fits in the community, and enables his colleagues to enjoy their day job in turn. To take care of code is to also care of one's peers, whose identities too depend on it.
6. A sixth relation, one which Stefan talks about second order, is how other game developers have adopted Stefan's pet project, the world making tool. In principle, each of

the games and their developers sit in a no less entangled web of being giving and receiving care as Stefan. Consider for instance that a serious bug or security vulnerability was found in Stefan's codebase. As a popular dependency built into perhaps thousands of digital games, its implications would ricochet far and beyond the relation between the codebases of Stefan's tool published online, and other game creators' codebases.

7. Finally, let's name a seventh relation as Stefan is navigating his relation with himself. Let's call it his *identity*. Stefan is asking: "who am I?," "what do I desire?," "what is possible for me?," and "what sustains me?" More specifically "am I part of the indie gaming community?," "what kind of programmer am I?," "what does my program make me?"

From a certain perspective it is reasonable to consider the first one of the given as primary, others secondary; the vignette seems to be about Stefan's private need to maintain his long-standing pet project, and the need of that project to be cared for.

It is not my task to insist on organising relations of care in a hierarchy of primacy, but rather to analyse how software maintenance is relational, in-between, and contingent on *maintain-ability*, ability to give as well as receive care. For instance in the above, the ability of the other game developers to make themselves beneficiaries of Stefan's pet project depend (among other things) on his economic and existential dependency of the various conditions of his life, including his employment relation as expressed in the vignette, and on the world building tool being able to be incorporated into other bodies of code through its open source code, licensing, availability, tutorials, help forums, hosting the code and website online, *etc.* Likewise Stefan's bosses ability to be maintained as his boss (and perhaps more?) depends on Stefan's ability to be maintained as someone who has the ability to be depended on by other game developers.

8. Software Dependencies

A unique property of computer software is the gap between programming and execution. Computer programs are not crafted for an individual computer, but rather for a theoretical model of computation, then compiled and built as installation packages, and delivered to laptops, desktop computers, smartphones, servers, music synthesizers and other computing devices to run. Earlier distribution media of commodity software included tapes, floppy disks, CDs and online BBS systems. Nowadays installation is of course done over the Internet. Commodification alienates users from programmers: nobody can name the designers, contributors or maintainers of the digital furniture of the everyday, such as Microsoft Word (Manovich 2013) or the PDF viewer you might be reading this on. The isolation of software from hardware is a theoretical, engineering, and social achievement, as modern programming gradually throughout the 20th century gained independence from electrical engineering and professionalized, while unfortunately was made male in the process (Ensmenger 2010; Hicks 2018).

An implication of this contemporary arrangement for software maintenance is that changes can later be made to a single source code, packaged, and passed via the same online media to the sites of execution. Software updates include bug fixes, performance improvements, redesigns and rebranding, and product reorganisations in response to changes such as arrival of the EU

General Data Protection Regulation (Douglas-Jones and Cohn 2018). This is familiar from our everyday life with computers. A less salient corollary of this directed relation of software delivery is that changes can *only* be done by changing the source code: except in some exceptional cases, software issues are not solved on individual computers like the one I am writing this on, or the one you are reading on, but elsewhere in companies and open-source projects where the code receives maintenance, and results are then propagated as software updates.

Contrast this with maintenance of, for example, bridges: a crack in a bridge jeopardises that bridge and those depending on it, such as humans using it to get from place to place, a train route crossing that bridge, birds nesting on it, *etc.* Software instead resonates and scales outward from the source code onto times and places of execution. Analogically, code in need of repair is less like a bridge, and more like an *architectural plan* for a number of individual bridges built in different places and used for different purposes, and some built maybe still in the future. Under the framing of software maintenance strictly as maintenance of source code – versus, for example repair of smartphones (Nova and Bloch 2020) or as server administration – maintenance is firstly infrastructural, and secondly always dislocated, so that the causes and effects of both decay and mending are separated by the gap between programming and execution. Scaling effects of commodity software infrastructure mediated over the Internet are huge; a bug in a single software suite can impact a very large number of computers, and symmetrically an update can successfully trickle down to millions of devices.

Maintain-ability points to capacity of these rhizomes to receive care. Illustrative is the world-building tool “Stefan” maintains. At the time of writing the original draft of this article, it had 650 issues reported, and 40 incoming so-called pull requests pending for review.² The ability of this fifteen years old, now 380,000 line codebase to receive care would be improved if Stefan was able to dedicate more time to it – at least that is the hope expressed in the vignette. Reflexively this would enable Stefan to identify himself as a maintainer. Because this tool has been embedded into so many other games, indie game culture now has come to partially depend on its persistence through time. A core idea of programming is that not everyone needs to build their own world creation tools, but can build on modules made available by others. It is a standard division of labour across the digital landscape, and no code is developed up from first principles. Fixes and updates are distributed to projects which are arrangements of other projects, articulating the new with the old (Cohn 2019). The implications of this entanglement are crystal clear: the emic term for imported submodules are “dependencies.”

A few recent stories circulated widely throughout my fieldwork. One story concerned the Heartbleed bug in the OpenSSL security library. While the long-dormant bug itself was already fixed at the latest version of OpenSSL source code, severe doubts were expressed whether updates would ever reach all the devices running OpenSSL. Another story was about a Javascript package on which multiple large projects depend on too. Its sole maintainer no longer had the ability to maintain it, and passed it on to a volunteer, who unfortunately had bad intentions, and “poisoned the well” with wide consequences. The story raised controversy among my informants about moral obligations in change of open-source project ownership. Some sympathised with his decision to hand over a pet project insisting on the moral legitimacy of doing so, while other lamented the lack of trusted peers to share the workload with. Returning to Stefan, how would discovery of a deeply buried and serious

security vulnerability, or potential change of project ownership affect the seven relations listed above? Which actors would be implicated? What demands could justifiably be raised, and which ones concretely met? At what cost to whom?

The situation is more dire for software which is being run, but whose source code is unavailable. This is partially the case for all closed-source software which is, by definition, only maintain-able by those within the organisation which owns the software plus by consultants, an arrangement very familiar to the informants.³ Products fall out of support, whether by planned obsolescence (Mattern 2018; Vinsel and Russell 2020), when a company goes bust, or for whichever reason. Likewise, currently running software can have its code inaccessible because it has been lost, a concern for older legacy systems in particular, making it necessary to navigate knowing and unknowing code, constraining what repair work could even theoretically be done by the means of re-programming, and by which actors (Ojala and Cohn 2023). What these distances between code and execution mean is that code changes might or might not arrive at the site of execution. Or perhaps a codebase fails to attract care, gradually falls out sight, or is hard to work with due to design choices, accumulated grime, or use of programming languages or paradigms now out of fashion. Again using Stefan to illustrate, what would be the implications for all those other game developers, their games and their players, if Stefan ceased to maintain the world building tool? We know by necessity he eventually will.⁴ A recurring trope during my fieldwork was this: “What if the last maintainer gets hit by a bus?” The above vignette remains – as a literary technique of ethnographic writing (Clifford 1983) – mute on what his family, friends or doctor might have to say. Less dramatically, his negotiation with his boss might simply fall flat, or his colleagues could persuade him to stay on the team four days per week? These complications nuance the archetypical, differently precarious, differently anxious, underprivileged and sometimes feminized archetypes from maintenance, repair and care studies, who do not have the chance to hesitate.

It was concerning to observe that of the sessions suggested, the ones about burnout were full. We can say that while a stressed programmer is still within what is *maintain-able* by intervention by friends, colleagues, labour unions and others who care – perhaps the boss too, a burned out no longer is. Staying with the trouble (Haraway 2016) is a risky business. While the software maintainers like “Stefan” spoke of their profession often as a “sellers market,” reducing a life lived alongside software to a mere employment relation is a very narrow view, masking any deeper cultural and personal commitments, meanings, identities and *becomings*. While walking away from a project can be possible in an impoverished reduction of programming as a mere job, doing so always means more. Hence: Stefan’s project is a *pet* project – an entity one does not simply neglect or abandon without psychological and moral damage.

9. Conclusion: Fragile Relatings in a Broken World

Voice of fictional character Stefan in the vignette “Maybe one more day per week...” echoes (and by construction imitates) the discourses of the software maintainers I encountered. Like all of us, they find themselves having to articulate interdependent, hybrid and dynamic networks of relations from within. They spoke of pride, commitment, emotional and intellectual

fulfilment, and joys of working with aged software now in their care. They also voiced frustration with poorly designed or programmed code, insufficient documentation, struggles with managers and product owners to appreciate their efforts, scarcity of maintenance time, unsurmountable numbers of bug reports, communal and personal conflicts, tragedy... all aspects of a full human life. Exactly these entanglements invite what Steven Jackson called *broken world thinking* (2014); not only bodies of people and bodies of code navigate breakage, but relations do too. Further, the durability of those entanglements depends on whether they are *maintain-able*, i.e., whether they help or hinder giving and receiving care. “Stefan” is an analysis of that property in the testimonies the programmers gave during my fieldwork.

Maintain-ability is a recursive property of networks of relations which sustain lives lived alongside software. When it is thinned out, their flourishing is hindered, and duration over time is endangered; code fails to be source of running programs, and programmers fail to be maintained as maintainers. Examples: burned out maintainers have low maintain-ability, but a team of well-cared for maintainers a higher one. Closed source software has low maintain-ability as its workings are hidden behind corporate obscurity, open source has a better chance. Complicated or unusually structured codebase has low maintain-ability due to cognitive load of intervening in and improving on it, conventionally engineered and familiar style and structure a higher one. Isolated programmers have weak maintain-ability, as do those overworked (anyone remember the toxic “10x engineer” hype?) or precariously without healthy support. Installed software which can receive updates made to its source code elsewhere across the gap between code and execution is more maintainable than software which is isolated from its source (however, it is also less vulnerable to code injection attacks). When the networks of relations are robust and strong, this potentiality enables the flourishing of biographies of users, maintainers and software.

Why this matters is that life is made durable by the technologies it is entrenched in and scaffolded by (Latour 1987). A third millennium fact is that the parenthood and childhood, work, musicking with digital synthesizers, and indeed programming is lived more alongside existing and aging commodity software (Word, PDF and Excel), than soon-to-come or anticipated software (AI, care robots and VR). Building on Karl Marx’s work, Steven Jackson calls us to see beyond the alienation put in place by the logic of commodification under capitalist mode of production, and to recognise the forces at work elsewhere which global capitalism renders invisible (2014). Programmers as well as the software they maintain continuously navigate risks of breakage, burnout, bugs and falling into obsolescence, and software and digital infrastructures are therefore a matter of care (Puig de la Bellacasa 2017). And it does not need to be a matter of desperate toiling at smouldering swiddens in the periphery of a profession (see Figure 1). What I hope to have contributed in this paper to the conversation about maintenance, repair and care (Mattern 2018; Denis and Pontille 2019) is that although a possibility alone never guarantees anything, at minimum an *ability* to give, as well as receive care is necessary. I illustrated this modality with the vignette titled “Maybe one more day per week...”, which is an analysis of my fieldwork listening to people who program sharing testimony of their lives caring for code over time. We all, users and maintainers alike, live in intimate and formative relations with software, and are therefore essentially entangled with ongoing repair of this broken world. The critical question to prompt is this: “what is

necessary for these bodies of software to be able to receive care?” Singular and universalising answers like Open Source are clearly insufficient in webs of hybrid actors. As we look around the various softwares which constitute our lifeworld and reflect on our situated abilities to be cared for *by* them and to care *for* them, we see that some of the software of our lives have this ability, others less so.

Acknowledgements

I thank the convenors Giacomo Poderi and Maurizio Teli, the contributors, and the participants at the *Caring and Commoning in/through STS interventions* panel at the Danish Association for Science and Technology Studies (DASTS) conference 2022 in Aarhus, Laura Kocksch for posing challenging and productive questions, and Marisa Cohn for guidance, inspiration and more. I thank the two anonymous peer reviewers for supportive and actionable comments to clarify the analysis and core argument of this article. Writing, production and dissemination of this work is made possible by earthly critters (Haraway 2016) who maintain computer software.

Notes

¹ I've since changed project, employer and country of residence, a familiar and disruptive biographical pattern under global socio-economic conditions of academic and tech industries alike.

² 698 and 51, respectively, at the time of third review of this article.

³ As one of the reviewers of this article pointed out, some readers might very well remember the so-called CrowdStrike incident crashing Windows computers on 2024-07-19, the closing day of the EASST/4S conference in Amsterdam.

⁴ *Memento mori*.

References

- Althusser, Louis (ed.) (1971) *Ideology and Ideological State Apparatuses (Notes Towards an Investigation)*, in *Lenin and Philosophy and Other Essays*, translated by Ben Brewster, London, New Left Books, pp. 121-176.
- Balsamo, Anne (2011) *Designing Culture: The Technological Imagination at Work*, Durham, Duke University Press.
- Barad, Karen (2007) *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning*, Durham, Duke University Press.
- Barad, Karen (2010) *Quantum Entanglements and Hauntological Relations of Inheritance: Dis/Continuities, SpaceTime Enfoldings, and Justice-to-Come*, in “Derrida Today”, 3(2), pp. 240-268.
- Chun, Wendy Hui Kyong (2008) *On “Sourcery,” or Code as Fetish*, in “Configurations: A Journal of Literature, Science, and Technology”, 16(3), pp. 299-324.
- Clifford, James (1983) *On Ethnographic Authority*, in “Representations”, 2, pp. 118-146.

- Cohn, Marisa Leavitt (2016) *Convivial Decay: Entangled Lifetimes in a Geriatric Infrastructure*, in “CSCW ‘16 – Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing”, New York, Association for Computing Machinery, pp. 1511-1523.
- Cohn, Marisa Leavitt (2017) “*Lifetime Issues*”: *Temporal Relations of Design and Maintenance*, in “Continent”, R3PAIR (6.1), pp. 4-12.
- Cohn, Marisa Leavitt (2019) *Keeping Software Present: Software as a Timely Object for Digital STS*, in Janes Vertesi and David Ribes (eds.), *DigitalSTS: A Field Guide for Science & Technology Studies*, Princeton (NJ), Princeton University Press, pp. 423-446.
- Couture, Stéphane (2019) *The Ambiguous Boundaries of Computer Source Code and Some of Its Political Consequences*, in Janes Vertesi and David Ribes (eds.), *DigitalSTS: A Field Guide for Science & Technology Studies*, Princeton (NJ), Princeton University Press, pp. 136-156.
- Deleuze, Gilles (1988) *Bergsonism*, New York, Zone Books.
- Denis, Jérôme, Mongili, Alessandro and Pontille, David (2015) *Maintenance & Repair in Science and Technology Studies*, in “Tecnoscienza: Italian Journal of Science & Technology Studies”, 6(2), pp. 5-16.
- Denis, Jérôme and Pontille, David (2019) *Why Do Maintenance and Repair Matter?*, in Anders Blok, Ignacio Fariás and Celia Roberts (eds.), *The Routledge Companion to Actor-Network Theory*, Abingdon-on-Thames, Routledge, pp. 283-293.
- Denis, Jérôme and Pontille, David (2021) *Maintenance Epistemology and Public Order: Removing Graffiti in Paris*, in “Social Studies of Science”, 51(2), pp. 233-258.
- Denis, Jérôme and Pontille, David (2022) *Making Things Last: Maintenance and the Ontological Politics of Duration*, in “EASST Conference – Politics of Technoscientific Futures”, Madrid.
- Denis, Jérôme, Florentin, Daniel and Pontille, David (2023) *Maintenance and Its Knowledges: Functional Exploration, Biographical Supervision, and Behavioural Examination*, in “Engaging Science, Technology, and Society”, 9(3), pp. 53-70.
- Douglas-Jones, Rachel and Cohn, Marisa Leavitt (eds.) (2018) *GDPR: Deletion Poems*. Copenhagen, ETHOS Lab. Available at: <https://ethos.itu.dk/gdpr/> (retrieved July 7, 2026).
- Ensmenger, Nathan (2010) *Making Programming Masculine*, in Thomas J. Misa (eds.), *Gender Codes*, Hoboken, John Wiley & Sons Ltd, pp. 115-141.
- Ensmenger, Nathan (2014) *When Good Software Goes Bad: The Surprising Durability of an Ephemeral Technology*. MICE: Mistakes, Ignorance, Contingency and Error in Science and Technology. Available at: <https://homes.luddy.indiana.edu/nensmeng/posts/2014/09/20/mice/> (retrieved July 7, 2026).
- Erickson, Seth and Kelty, Christopher (2015) *The Durability of Software*, in Irina Kaldrak and Martina Leeker (eds.), *There Is No Software, There Are Just Services*, Lüneburg, meson press, pp. 39-56.
- Haraway, Donna (1985) *A Manifesto for Cyborgs: Science, Technology, and Socialist Feminism in the 1980s*, in “Socialist Review”, 80, pp. 7-44.
- Haraway, Donna (2016) *Staying with the Trouble: Making Kin in the Chthulucene* [Illustrated edition], Durham, Duke University Press.
- Hegel, Georg Wilhelm Friedrich (ed.) (1910) *Self-Consciousness*, in *The Phenomenology of Spirit*. Available at: <https://www.marxists.org/reference/archive/hegel/works/ph/phba.htm> (retrieved July 7, 2026).
- Henke, Christopher and Sims, Benjamin (2020) *Repairing Infrastructures: The Maintenance of Materiality and Power*, Cambridge (MA), MIT Press.
- Hicks, Mar (2018) *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*, Cambridge (MA), MIT Press.

- Jackson, Steven J. (2014) *Rethinking Repair*, in Tarleton Gillespie, Pablo J. Boczkowski and Kirsten A. Foot (eds.), *Media Technologies: Essays on Communication, Materiality, and Society*, Cambridge (MA), MIT Press, pp. 221-240.
- Kneuper, Ralf (2017) *Sixty Years of Software Development Life Cycle Models*, in “IEEE Annals of the History of Computing”, 39(3), pp. 41-54.
- Koskenkorva, Jaakko and Järnefelt, Eero (2016) *Programmers at Work Maintaining a Ruby on Rails Application*.
- Latour, Bruno (1987) *Technology Is Society Made Durable*, in “The Sociological Review”, 38(S1), pp. 103-131.
- Lindén, Lisa and Lydahl, Doris (2021) *Editorial: Care in STS*, in “Nordic Journal of Science and Technology Studies”, April, pp. 3-12 <https://doi.org/10.5324/njsts.v9i1.4000>.
- Manovich, Lev (2013) *Software Takes Command – Volume 5: International Texts in Critical Media Aesthetics*, London, Bloomsbury Academic.
- Mattern, Shannon (2018, November) *Maintenance and Care*. Places. Available at: <https://doi.org/10.22269/181120> (retrieved July 9, 2026).
- Nova, Nicolas and Bloch, Anaïs (2020) *Dr. Smartphones: An Ethnography of Mobile Phone Repair Shops*, Morges, IDP.
- Ojala, Mace (2021) *Maintain-ability: A Thesis On Life Alongside Computer Software* [Master’s thesis], Tampere, Tampere University, Trepo. Available at: <https://urn.fi/URN:NBN:fi:tuni-202202031820>.
- Ojala, Mace and Cohn, Marisa Leavitt (2023) *Software Maintenance as Materialization of Common Knowledge*, in “Engaging Science, Technology, and Society”, 9(3), pp. 165-185.
- Piredda, Giulia (2020) *What Is an Affective Artifact? A Further Development in Situated Affectivity*, in “Phenomenology and the Cognitive Sciences”, 19(3), pp. 549-567.
- Pirsig, Robert (1974) *Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values*, New York, William Morrow and Company.
- Puig de la Bellacasa, Maria (2017) *Matters of Care: Speculative Ethics in More Than Human Worlds*, Minneapolis, University of Minnesota Press.
- Ruparelia, Nayan B. (2010) *Software Development Lifecycle Models*, in “ACM SIGSOFT Software Engineering Notes”, 35(3), pp. 8-13.
- Salo, Erja. (n.d.) *Raatajat rahanalaiset / Kaski*. Ateneum. Available at: <https://www.kansalliskallio.fi/object/392378> (retrieved November 5, 2023).
- Vinsel, Lee and Russell, Andrew L. (2020) *The Innovation Delusion: How Our Obsession with the New Has Disrupted the Work That Matters Most*, New York, Currency.
- Viola, Marco (2021) *Three Varieties of Affective Artifacts: Feeling, Evaluative and Motivational Artifacts*, in “Phenomenology and Mind”, 20(June), pp. 228-241.
- Winthereik, Brit Ross (2020) *Is ANT’s Radical Empiricism Ethnographic?*, in Anders Blok, Ignacio Farias and Celia Roberts (eds.), *The Routledge Companion to Actor-Network Theory*, London, Routledge, pp. 24-33.