Engaging Science, Technology, and Society

THEMATIC COLLECTION: MAINTENANCE & ITS KNOWLEDGES ORIGINAL RESEARCH ARTICLE

Software Maintenance as Materialization of Common Knowledge

MACE OJALA
RUHR UNIVERSITY BOCHUM
GERMANY

MARISA LEAVITT COHN
IT UNIVERSITY OF COPENHAGEN
DENMARK

Abstract

While development of software always implicitly takes place in contexts of inherited entanglements and legacies, its maintenance deals explicitly with what is already present. Software maintenance locates itself in media res, in the middle of things. Maintaining software typically involves intervening in the material archive of source code, documentation, and software tools. Doing so successfully requires relevant situated knowledge of how the software at hand already hangs together, and how to effectively put this knowledge to use. This knowledge builds on first-hand experience, acquired in practice over shared lifetimes of people and code. For code to continue to endure over time, ongoing articulation of its entanglements is externalized and materialized across contributing programmers and software development tools, each themselves vulnerable and in need of maintenance. This paper analyzes how this process of externalizing and materializing knowledge is negotiated. We conclude that the common knowledge which suspends the string figure of software in time and in a broken world (Jackson 2014) is always a locally hybrid assemblage which carries this knowledge forward. Hence, to maintain software well is to add on to its legacy.

Keywords

maintenance; software; software studies; epistemology; biography; string figure

Software Already Exists

Despite the tendency to associate software with what is new or soon-to-come, plenty of software already exists. Most software is not cutting edge, but rather exists in some contingent relation to maintenance. In standard models of software development and engineering, maintenance is considered a late phase of the software life cycle, following development phases of specification, design, coding and testing (Ruparelia 2010). This linearity implies that all software which has come out of earlier development phases has successfully entered the state of maintenance. This is, however, too optimistic and removed—it will not surprise a maintenance and repair studies reader that not all software actually gets looked after—it has been built. The blanket assumption about a stable and continuous maintenance stage obscures the ongoing efforts of keeping an inventory of what software exists. This then raises epistemological questions about the status of software that ontologically exists, but slips out of what is known by anybody.

Much of the work of software maintenance we will write about here is precisely the ongoing work of knowing what exists. Software maintenance, we show, is a set of epistemic relations and practices of

To cite this article: Ojala, Mace, and Marisa Leavitt Cohn. 2023. "Software Maintenance as Materialization of Common Knowledge." *Engaging Science, Technology, and Society* 9(3): 165–185. https://doi.org/10.17351/ests2023.1325.

To email contact Mace Ojala: mace.ojala@rub.de.

coming to know and making existing software present to the organization or community of programmers, many of which are characterized by a diversity of labor commitments, turnover and high churn rates. This work of knowing what software exists includes appreciating that the entire body of code consisting of hundreds, thousands, or millions of lines of code is not knowable *in toto* by one individual no matter how well they are tooled up. This means that working to know what code exists is also about recognizing what is sufficient knowledge for the sake of practical maintenance. Based on testimonies at meetups and online forums we analyzed, we will show that balancing what and when to know and maintain, are mutually entangled since forgetting or making code obsolescent is inevitable.

An essential premise, then, of the epistemic work of software maintenance is that some of the code will be forgotten, and that forgetting is a form of planned or unplanned obsolescence, and of produced *agnōsis* (Proctor and Schiebinger 2008). Epistemic practices have gained much attention in science and technology studies—inquiring into how the collaborative work of science makes its objects knowable, "visible, legible, mobile, accountable and actionable" (Vertesi 2015). Often this work has even focused on the role that software systems play in "appresenting" (Knorr Cetina and Bruegger 2002), that is bringing present to action on screen the object of science or work concern (see also Goodwin 1995), and draws attention to the active role that retooling, testing, and recalibrating software play in constructing the scientific object (Vertesi 2015; Paine and Lee 2017). This too is part of software maintenance work because, like the ocean floor or the stock exchange, software must be mediated by other software to be made present to the organization (Cohn 2016). While software's source code itself is often available for inspection, it is also inaccessible by virtue of the impracticality of inspecting all of a code base line by line (often viewed as a last resort "ground truth" to discover what the code is doing, if all other ways of knowing fail). This means that software is paradoxically available to scrutiny by the maintainer, but also remote from the maintainer who wishes to know more it. The maintainer may actively maintain this remoteness, as a practical achievement, or code may also quite passively slip out of maintainers' knowledge. These different kinds of forgetting (active and passive) have quite different effects and consequences for software maintenance. Less attention has been given to the relation between software forgetting and unknowing, or indeed how software itself resists being knowable. In this paper we detail the dynamics of handing down code knowledge between actors who care for it over time. This matters, because changes to software are made by modifying its source code (Krysa and Sedek 2008). The role of software in the sciences can often be to bring remote objects like the ocean floor or a distant planet present, however, bodies of software code too are a kind of remote object, residing and operating on servers and code repositories, without necessarily being known and available to action. As prior research has shown this work of "keeping software present" to organizational accounts and action (Cohn 2019) is ongoing and also tied-up in the relational work of software maintainers.

Here is the epistemic problem of maintenance: how to organize collaborative, interpretive knowledge over time? The questions we take up here are what work is involved in knowing *what* code exists, and *who* and *what* is doing this knowing? Given that in total software is impossible to know due to its size, complexity and process nature of running code, how do maintainers negotiate what is deemed acceptable, sufficient, and proper knowledge (or forgetting) of software? This is in part a question of how an organization, community, collective or lineage of programmers chooses which parts of a system to maintain or allow to fall into disrepair, what is deemed wasteful or too costly to maintain, and what forms of not-knowing are acceptable. But it is also a question of the social work of shaping, translating and disciplining

knowledge of code for the maintenance of careers. An organization can know some parts of the code too poorly, leading to breakdowns; an individual can know some code too well in ways that will defract relevance and defer career progression. So, the ongoing effort to know what code exists becomes a question of *what* is good to know, *what* are proper ways of knowing code (Cohn 2019), and *who* or *what* does the knowing? Is for instance, the original programmer the best long-term maintainer? Can tools hold on to code-maintenance when humans cannot? These normativities arise in ideas of how best to materialize by writing and documenting code so that it is knowable and maintainable in the tools that programmers use to do this work, as well as in how contributors are brought in and out of the maintenance effort.

Epistemic work in software maintenance produces binds, as programmers can become tightly bound to the code that they know well. Often the question of how to sustain software systems transforms to the question of how to keep around the people who embody the knowledge of it, or what to do if someone leaves a project or dies (as often phrased "what if the sole maintainer is hit by a bus?"). This binding is part of the felt lifeworld of programmers who often complain about being too bound to the code they have written or the projects they contribute to, and weigh career decisions in terms of these binds. Thus, while much work is put into assembling tools, documentation—and despite the obduracy of the code itself—there is an acceptance that vital knowledge resides in the maintainer.

This article makes a conceptual contribution to research on maintenance by building upon existing theorization of maintenance as a matter of collective knowledge. Particularly with regards to software, we show how a conceptual distinction between different orders of shared knowledge is helpful to explain the epistemic work of software maintainers as they approach the ambiguous status of the object in their care. We substantiate this sociology of knowledge by attending to the ways in which the epistemic work of software maintenance is: distributed across human and non-human actors, noting the fragility of epistemic achievements, and considering how software objects resist being cared for / known about beyond their implementation. We explore two primary types of epistemic work in coordinating software maintenance. Initially, we provide an overview of the diagnostic tooling programmers employ to locate which parts of the code are in a state of decay (forgetting) and require maintenance or are no longer maintainable. We focus on two kinds of tools, namely version control systems and automated tests which enable reanimating the material-semiotic archive of the software and open it up for re-interpretations when changes to black boxed source code become necessary. Secondly, we explore the peopling of maintenance through onboarding and upskilling, employee churn (retaining and letting go of people), and how the weaving of epistemic binds between people and code are imagined. As we show how these aspects of software maintenance knowledge are inseparable—as tools used to know software materialize individually held knowledge and beliefs—we consider the entwining of the biographies of software artifacts and coders over time.

What will become clear is that there are many conditions through which software can fall out of maintenance, and various ambiguous states of maintained-unmaintained. Software practitioners need to discern, and we in turn differentiate analytically, between different states of unmaintained code. This epistemic effort to maintain code requires finding a proper balance between the known and unknown status of code in different arrangements.

While it is unsurprising to find that maintenance knowledge of software is performed collectively across humans and non-humans, we further theorize that this string figure (Haraway 2013, 2016) of maintained code relies upon a recursive relationship between knowing what is maintained and knowing how

best to maintain it. This recursivity of software maintenance knowledge has practical implications for maintainers who can find themselves in a relationship to unmaintainable code when shared knowledge of code is not both mutually and commonly held.

Knowing Together

Repair and maintenance have attracted scholars of science and technology studies, and for good reason. When things break, the taken-for-granted jumps into view and spurs action (Heidegger 1977; Graham and Thrift 2007). This has set a fruitful scene for studying infrastructures underpinned in the everyday (Star and Ruhleder 1996; Star 1999; Bowker and Star 2000; Henke 2000; Graham and Thrift 2007; Denis, Mongili, and Pontille 2015; Henke and Sims 2020; Denis and Pontille 2020, 2021). In review of socio-historical research, the work of maintenance is often low status and pay, stigmatizing, and the labor of minority populations (Bowker and Star 2000; Mattern 2018; Denis and Pontille 2020; Vinsel and Russell 2020). Relatedly, insights from research of repair and maintenance have been put to use as a foundation for building critiques of and alternatives to innovation, consumption and perpetual economic growth (Jackson 2014; Mattern 2018; Vinsel and Russell 2020). Further, feminist scholarship in STS has developed and advanced theory towards the affective and political registers of care, ethics of care, and care as an important analytical lens (Tronto 1998; Puig de la Bellacasa 2011; Haraway 2016; Puig de la Bellacasa 2017; Conrad 2019; Lindén and Lydahl 2021). While some of this literature considers computer systems the maintenance of existing code remains predominantly unaddressed.

Ethnomethodological research of what Christopher Henke and Benjamin Sims call "socio-technical repair" (2020) supports theorization that technical knowledge is never enough, but "when people do work, they are moving through, relying on, and modifying networks of people, ideas, and material artifacts" (predominantly Henke 2000, but also Kocksch et al. 2018; Henke and Sims 2020; de Wilde 2021; Denis and Pontille 2021). Summarizing epistemology from the literature above, maintenance practices are knowledge practices, and rich sites of unique epistemic activity. Maintainer knowledge is necessary for material artifacts to last and the continuation of the societies the artifacts participate in (Latour 1990; Denis and Pontille 2021).

Maintenance studies attend to the knowledge maintainers require for objects to be made to endure, i.e. to assess their fragility in order to perform their stability (Denis and Pontille 2017). This work raises the questions: how does something remain the same despite its continuous mutability as it undergoes maintenance under many hands; and how do maintainers make known the material fragility of the objects in their care without making that fragility visible to users or others (ibid.)?

> [M]aintenance enacts what we might think of as two-sided objects, fragile in the eyes and hands of maintainers, reliable in the eyes of users. (ibid., 3)

Software, in our view, is similarly two-sided, even to its maintainers. This is because software's fragility resides not only in its resistance to enduring, but oftentimes in its recalcitrant endurability that resists knowing. While software can become over time nearly impossible to maintain without the right knowledge, tools, and people, software can in some ways endure too readily, in the sense of lines of code that remain unmaintained but are never fully forgotten, deleted or removed. Software can exist in various states of falling out of maintenance. Code can be relied upon to run and perform actions on machines without being actively

maintained. On the other hand, some existing code will continue to exist, although it will never run. Examples of this kind are an error recovery routine whose error condition never activates, a feature which used to relevant but isn't anymore, and a code sketch left undeveloped. In this sense, software's durability holds a somewhat paradoxical relationship to maintenance knowledge. Some of the epistemic work of software maintenance is to allow for not knowing some of a code base despite its endurance so that other parts of the code can be known adequately. The fragility of software is made known to maintainers when it is appresented by tools or animated by active contributions by other maintainers. However, fragility also arises as the failure to animate the material archive of the code base, and the code base slips into the wrong kinds of forgetting.

In science and technology studies it is a truism that knowledge is socio-cultural. Beyond the view that culture influences or even determines what individuals can and do know, individual humans as analytically most relevant agents of knowledge has been challenged in favor of collective forms of knowledge. For instance through the anthropological work among marine engineers and machine operators onboard large ships in *Cognition in the Wild*, Edwin Hutchins (1995) argued that material-cultural systems such as naval vessels and their crews have cognition of their own which does not reduce to what the agents individually know. Similarly we find that to last over time as software depends on more-than-human cognition.

During our research on software maintenance, we have found it illuminating to draw upon a conceptual distinction of shared knowledge down to mutual knowledge on the one hand, and common knowledge on the other. Both kinds of shared knowledge are socio-epistemological concepts, reaching beyond individuals to collectives as epistemic agents. The basic, first-order mutual knowledge refers to what everyone knows individually—"everyone knows that *p*," where p is a placeholder for any proposition is insufficient for coordination. Further, this itself can be made available to others as higher-order common knowledge—"everyone knows that *p*, plus everyone knows that everyone knows that *p*." In other words, the latter expands the former to include knowledge about knowledge others hold. For example, two independent programmers might mutually be aware of a certain bug, but only by announcing it on a bug tracking system they both use, is the bug made common knowledge—the bug no longer exists with the two programmers individually, but in their co-relations mediated via the tracking system. Announcement allow individuals to relate to one another as epistemic peers, and intersubjective "I know, that you know, that I know…" recursion produces "collective knowers" (Vanderschraaf and Sillari 2013). This analysis explains how knowing can soundly be attributed to teams, projects, companies, cultures, or indeed to a lineage of maintainers.

In this section, we make the case that the "string figure" of maintained software is only possible when a more-than-human *collective knower* of the software is achieved (and known to all). Mutual knowledge of the code base is not enough—it is common knowledge which allows coordination of software maintenance, and for creating robust multi-agent knowledge.

Passing on a String Figure

In this section we consider the work that is involved in knowing *what* code exists, and *who/what* participates in this knowing? Given that software is impossible to know *in toto*, how do maintainers negotiate what is deemed acceptable, sufficient, and proper knowledge (as well as forgetting) of software in order to maintain

it? We ask further how these epistemic practices of coordination of knowing code become imbricated with the politics of organizational work?

As with other forms of maintenance, software maintenance too remains marginalized, poorly resourced, understudied, down prioritized and unappreciated while at the same time being known to be vital for the longevity and success of software as discussed above. Basic recognition of software maintenance as a concern dates back to mid-twentieth century, before computer science and software engineering developed into independent professions and fields of research (Ensmenger 2014). However, distaste for this everyday maintenance permeates among programmers (ibid.). Maintenance programming is often a chore delegated to juniors and poorly performing programmers (ibid.), PhD fellows (Boscoe and Scroggins 2019) and the like, or left to some "random person on the internet" who will ultimately remain imagined, unspecified and unaccounted for in task delegation (Munroe 2020). Unfortunately, little consideration for maintenance seems to be included in computer science and software engineering programs.[1]

Secondary artifacts like diagrams, flowcharts and technical documentation have been used as surrogates for knowing code (de Souza, Froehlich, and Dourish 2005), and as conscription devices for involving people (Henderson 1991). However, these non-code artifacts have variously been ignored or rejected by programmers (Cohn, Sim, and Lee 2009; Ensmenger 2014). We show that tools like version control systems and automated tests are similarly de-centered. The prioritized status of code in software programming introduces peculiar ambivalences through valorization of code as materialized maintainer knowledge and code as the most valuable contribution. Meritocracy, an ideology which programmer culture holds dear, translates long-term maintenance experience to epistemic authority, and therefore power, for those who have stuck it out.[2]

We see maintenance knowledge as a relational capacity built over time and articulated into higher order common knowledge (Vanderschraaf and Sillari 2013). This knowledge does not exist in the abstract, but is materialized in bodies and tools. Part of the epistemic practice of software maintenance is gaining an idea of what is commonly known by the participating individual knowers, what is known by the collective, and knowing how various human and non-human actors are participating in holding that knowledge together. In addressing our questions of the work involved in producing and materially sustaining this common knowledge we are reminded by Adrian Mackenzie that both machines and people are the recipients of code (Mackenzie 2006). Thus negotiations over what are the acceptable and proper ways of knowing and/or letting go of knowing is often a nonlinear exchange among people and tools.

Viewing software as a *string figure* (Haraway 2016, 2013) , a metaphor inspired by the "cat's cradle", an open-ended and co-operative game of manipulating a loop of string on the players hands, Haraway invites to see to knowledge as something contingent, fragile, as well as created through intra-action,

[1] Also the programs provided at our universities.

[2] Geiger et al., point to the tensions present in the accumulated status of open source maintainers. Maintainers can become minor celebrities by virtue of how many people depend upon their work. This can lead to maintainers being somewhat "cursed" by the growing demands, so much so that they cannot walk away (Geiger, Howard, and Irani 2021).

suspension as well as letting go. For us, this concept also points to the ways that software maintenance knowledge exists in tension between knowing and forgetting, as the tools that are used to (re)animate the material archive of existing code and the programmers who people a project, weave the strands of software together and suspend it in space-time. String figures are passed along from one coder to another lineages where attrition, forgetting and loss are an acknowledged (and at times highly visible) part of the epistemic work.

In the following section our listening to programmers sharing their experiences of handing these string figures i.e. how participants learn to know what others know comes to the fore. We analyze programmers articulating a process through which common knowledge materializes by attending to the role of "tooling" and "peopling" in software. Tooling relates to the ways that the material archive of the code that exists is (re)animated as the string figure of common knowledge through various software tools. "Peopling" relates to how software requires humans to hold and suspend the string figure of common knowledge, and under what conditions (as well as tensions and affects) this string figure can be held together and made to last like the shapes in the game of cat's cradle.

Listening to Programmers

To investigate how sufficient knowledge for maintaining software is acquired, organized, distributed and kept alive, we conducted a two-phase research project. The first phase started online from forums where software maintainers discuss their experiences. We identified key concepts from these discussions and collected a research lexicon. The lexicon was then used to further navigate related online discussions. Some of the most productive terms included the quasi-economic *technical debt*, the adjective *legacy* which carries a negative value, the metaphor of *code smell* an experienced programmer would be able to trace to its decaying source, and the name of a particular programming language *COBOL*, widely considered to be a *dead* language (Ritasdatter 2020; Marino 2020).[3]

In the second phase we headed out to the field. Of the two authors of this paper, Mace attended gatherings of software maintainers in Paris and Berlin, and we both attended a gathering in Washington DC (they each lasted a day). The first two were organized adjacent to a major industry event, and the third was a track in a large academic conference of maintenance. All three targeted those concerned with questions of software maintenance. Participation in each event was in the dozens, with participants popping in and out throughout the day. The programme was organized largely in a self-organizing unconference format with a casual atmosphere, and a familiar mode of gathering in tech circles. During the morning, the participants would propose relevant topics, possibly merge similar proposals, and then vote the topics to proceed with in the afternoon. We attended the discussions, and followed up with a handful of unstructured interviews during the breaks. We took notes through the events which were later transcribed and organized afterwards.

[3] The sheer number of terms used to describe how software ages such as—decay, rot, smell, grime, debt, pollution as well as the terms for managing aging code came up such as—refactoring, bootstrapping, strangling, harnessing. This broad vocabulary of code speaks to the variety of practices, methods, and tools that exist to know and maintain software.

Informed by the first phase of online research within forums, our note taking at the gatherings focused on identifying concerns pertinent to software maintenance, formulations of problems and potential solutions, and the subjective experiences of software maintenance.

Next up, we synthesize what practicing programmers consider necessary knowledges for successful maintenance, if maintenance is at all possible. We focus on the dynamics of this multi-agent common knowledge at the inflection points of when knowledge is translated across tools and people, or to put it another way, when the string figure changes hands.

Tooling to Know a Material-Epistemic Archive

The sheer prevalence of tools and techniques to know code, map its architecture, test it, and locate what is rotten or unmaintainable reveals the challenges of software maintenance work. The participants of the gatherings agreed that software maintenance involves the appropriate arrangement and application of these tools. Tim Ingold (2000), anthropologist and theorist of material culture argues that cultural scholars mustn't ignore the relational work of learning to live with tools. With that in mind, we discuss: (1) version control systems and (2) automated testing because both tools are central in illuminating what code exists.

Version Control Systems as Sites of Making Contributors Out of Contributions

Version control systems are crucial to contemporary software maintenance work (Yuill 2008; Geiger, Howard, and Irani 2021). These are, in short, systems for storage of code and for tracking code changes. While keeping code under version control is widely accepted best practice, and the systems for doing so are a useful and beneficial element of the maintenance infrastructure, some features of specific version control systems however attract controversy. Unsurprisingly, the concerns center on the dominant forum is GitHub®, which integrates revision control with other features such as bug reporting, discussion forums, project documentation and developer profiles earlier done on mailing lists and web forums (Yuill 2008). In addition to keeping a record of the history of every single change to the lines of code available for revisiting when technical problems emerge, contemporary, cloud-based software development platforms such as GitHub®, keep each change connected to a user profile which thereby objectifies by making those persons visible. This centralization and "platformization" (Helmond 2015) of socio-cultural processes of software development entangle and ensnare biographies of people with biographies of code. Our informants spend considerable effort debating the ambivalent affects in response to parallel platform mechanisms of surveillance and of attribution.

In an enlightening moment of fieldwork an informant remarked how regrettable the name of one of the commands, *git blame* is; the exact same functionality to check from the version control system who precisely has made changes to an individual line of code could just as well had been called "git credit," he said in a session. No matter how serious or not the affect of blame was (when the command was designed), its normative materialization was criticized by our informant, with others around him nodding in agreement.

At the same time, however, a programmer's portfolio of contributions associated with their GitHub® profile builds a valuable asset of social capital to trade in the political economy of the flexible, entrepreneurial precariat. Analogously, a body of code performs its vitality, youthfulness and aliveness by attracting contributions and being cared for (for a similar argument about performing taken-cared-ofness

see Denis and Pontille 2021 about graffiti removal). Data visualizations and elaborate videos are produced by maintainers that visualize and dramatize contributions to software projects; depicting crescendos of action and denouement as projects stabilize to maturity. The significance of these contributions however requires qualitative evaluation of their content in the context of the body of code. Falling out of love and into obsolescence (Peters 2015) only connotatively signifies absence of recent contributions or negligence of code work. Public listings of issues on open source software projects are similarly ambivalent as records of both work done and undone. How to judge a maintainer of a project with hundreds of open bug reports? Does it suggest that the project is widely used, vital and hence accumulates feedback? Has the programmer been headhunted to new projects? Or perhaps they fell victim to burnout, layoff, war or death? This interpretive flexibility of crude metrics concerns programmers involved in them.

The debates about what is visible on integrated version control systems, and what incentive structures that visibility creates moves us to encounter the enduring questions of what counts as a "valuable contribution?" To expand the scope of what is deemed valuable, our informants mentioned attempts to persuade the platforms to include recording non-code contributions such as the community work of onboarding new maintainers. The hope is that more kinds of contributions would be (ac)counted on the platforms. These attempts connect software development and maintenance with other visibility struggles in other maintenance work, like nursing (Bowker and Star 2000). Ironically, recording non-code contributions might serve to further platformize (Helmond 2015) participation in this digital culture.

Automated Testing as a Hopeful Techne of Amnesia

Besides reading the primary source code and studying its genealogy tracked in the version control system, coming to know software involves prodding, poking, and testing code. In software engineering some elements of testing have been formalized and materialized in secondary code that tests the functioning of the primary software. These automated tests are propositional, they are code about code, metacode so to speak, designed so that the code can either pass or fail.

As Geiger et al., point out, existing test suites can be "staggeringly large," sometimes multiple times the size of the primary code they check. Some "software libraries for programming languages [. . .] have tens of thousands of tests, and programming languages [. . .] can have hundreds of thousands of tests" (Geiger et al. 2021). However based on our research listening to maintainers online and in person, rather than being huge, test suites are often small—even too small. Being computer programs, tests are of course a maintenance concern in and of themselves. Disappointed and frustrated maintainers complain that test suites tend to be non-exhaustive, ambiguous, outdated, buggy, absent and the test infrastructure itself won't run for any number of reasons.

Automated testing is a pragmatic workaround around the theoretical, awkward fact famously proven by Alan Turing in 1936 that still haunts computer science today: knowledge about computer programs is mathematically incomplete (Turing 1937; Sipser 2013). As secondary software, automated tests place a harness on the primary software, constraining and bonding it. The test harness serves the function of an electric fuse; the tests short-circuit before the primary code breaks, thereby signaling issues to be fixed. According to a canonical software engineer and author Michael Feathers, often cited in our empirical material online and in person, carefully maneuvering a legacy codebase under a testing regime is the preferred way to come to know it and to maintain it (Feathers 2004). The hope of testing is to check that code

behaves as intended, and does not misbehave. As more of a technique of debugging than proving paradigm (Payette 2014), testing code dodges the theoretical facts about unknowability of programs in general. Tests are instead a pragmatic rule-of-thumb engineering response to the accepted existence of bugs; an application of anthropological recognition of the limits of knowledge, and the affective importance of sense of confidence. Tests are a way to manage workload as well as the labor of others, not only by making software easier to maintain especially as it scales up, but sometimes by creating barriers to change, to shield maintainers from incoming requests (Geiger et al. 2021). Both our online material and fieldwork at the gatherings highlight testing and debugging and de-emphasize theoretical and formal proofs as preferred *modus operandi* of software maintenance.

From the temporal, diachronic perspective of maintenance, tests serve two complementary epistemic functions for coming to learn and to forget code. Firstly as executable and re-executable programs they check that the primary software *still* behaves as before. Secondly tests serve as documentation, a rephrasing of what the main software's intended behavior *was* at the time it was last programmed. Testing is a *technē* of amnesia—to automate testing is to offload the knowledge work of remembering what software has been written to this secondary software, the test suite. As such tests add to the local archaeological record and legacy, and become found objects of people now already gone. The test suite is another body of code within the black boxed, material archive from which some knowledge can potentially be re-constructed for practical purposes of receiving, appropriating, and maintaining and continuing their inherited legacy. The immediate aim of this offloading is to undo the primary code as a matter of concern, to black box it. As the box is laboriously closed, it gains its factuality, its reality, and can recede off-stage into the background. The indirect, humane aim of black boxing is gaining confidence, and removal of maintainer anxiety. When the amnesia technique of automated testing succeeds, the burden on maintenance is taken up by the tooled knowledge infrastructure. Confidence is achieved and anxiety lessened when forgetting is done well, and the box remains shut.

As we have analyzed, version control systems and testing frameworks aim to construct and expose two views to a local, material-epistemic archive which suspend a body of code in time.[4] Ideally, this archive provides a genealogy, and a living and open para-text for creating interpretations of the primary code. A version control system resists closure of the code-as-it-currently-is as a final, ahistorical truth about it (Yuill 2008), while a testing framework defers closure of the code-as-it-was-written the final truth about the intent of the design. Both enable interpretation and partially or even completely rewriting the code, perhaps by a new generation of programmers, while allowing the software to remain functionally the same. The code at hand thus remains one of many possible shapes the software could have taken, and the task of

[4] Akhil Gupta (2018) describes the temporality of living with unfinished or abandoned infrastructural projects and asks how we might theorize from this modality of 'suspension'. [add first names] Gina Neff and David Stark (2004) point to the social and economic effects of software being permanently suspended in ongoing beta testing. [you haven't introduced beta testing at all in your opening about testing] While these works point to the effects of living with systems and infrastructures that are continuously built, we draw attention here to what this work is like for developers and maintainers as part of their epistemic practice.

successful maintenance is to pass on a legacy which programmers can pick up, interpret, claim their own, and carry on.

As our fieldwork shows this archive succeeds only partially, and is itself in a vulnerable process of becoming. Our informants reported experiences of abandoned code repositories, outdated contact information, reported but unclosed issues, lacking test coverage, misleading documentation, alien programming practices, inaccessible closed source code, forgotten programming languages, need to convivially "hold the hand" of software as it decays (Cohn 2016), perverse economic incentives, unfavorable socio-economic structures plus of course a full breadth of human life in maintenance as tragedy, drama, comedy, and romance (Ojala 2021). The ideology of archived and automated amnesia is blooming and indeed flourishing with human life. Maintenance remains a wonderfully human concern.

Life in software maintenance takes place amidst a local bricolage of tools such as version control systems and test suites. When organized appropriately, this network carries the legacy code and the legacy of those who have been bound with it through time. In the next section we describe how maintainers carefully pass this string figure (Haraway 2013, 2016) from hand to hand as maintainers are onboarded, retained, and let go of in various projects.

Peopling and Biographies. Making Contributors Valuable

Based on substantive interviews, Geiger et al., describe the many forms of work that comprise software maintenance. While much of this work is in working directly with code to manage versions and automate and integrate tests, as described in the previous section, it is also the "more than technical" work of managing the labor of others, recruiting and retaining both users and maintainers, as a project grows (Geiger et al. 2021). In our fieldwork we too observe practitioners both online and at the gatherings defend against the asocial, even an anti-social idea that a life with software is a life of typing code, rather than maneuvering contributors into and out of the sociotechnical sphere, what we refer to as the "peopling" of software. While Geiger et al., focus on the problem of scaling-up labor and trust over time, there is also the corollary problem of attrition; i.e. the loss of people from a project and how to sustain common knowledge of the code in order to maintain it. Many more projects stagnate and wither than scale up (Coelho et al. 2020).

Software maintenance is about code standing the test of time. Passage of time itself is not the concern, but the pressures of the dynamic fabric in which code is embedded; its use, other software surrounding it such as its dependencies, and the amount and quality of attention paid to it. A maintainer is someone who witnesses the murmuration of the fabric and makes choices about appropriate changes to the code. Being present at these trials, and participating in them is where the valuable experience is acquired. Correspondingly, these trials must be attended for software to surpass them. What are some of the ways which bring maintainers to attend to these trials? Who gets to participate? To be clear, "a contributor" pertains to individual people while "a contribution" pertains, somewhat troublingly, to a collection of lines of code. The former are individuated and valuated from a stream (or often a mere trickle) of the latter.

Onboarding

Regardless of whether a programmer comes to participate in a body of software through company employment, through consultancy, or on a voluntary basis typical of open source software, they must get onboarded. The crux of onboarding is to achieve sufficient and relevant knowledge to make valuable

contributions. While we witness ongoing controversies about what kinds of contributions are valuable; as well as *how* and *if* they are recorded in version control or other management systems, our informants tell us that changes to the source code are conventionally considered to be the most valuable. These are simply called "code contributions." It follows intuitively from the notion that to change software is to change its code (Krysa and Grzesiek 2008). However besides code, contributions might include reporting bugs, editing documentation, supporting users, raising funds, doing managerial work, and onboarding new programmers. As one of the informants put it:

> As maintainers we all looooove [sic] docs and project management. How do we celebrate this?

At another instance during the fieldwork, Mace was told by the CEO—of a growth-phase, high-energy tech startup during a lunch break—that they plan to contribute to an upstream open source project (that their product depends on) by allocating their competent graphic designer to redesign a better logo for the project. Would a new logo be received as a significant contribution, or one of minor value?

Version control systems and testing frameworks both have a role to play in weaving newcomers into the string figure. At one of the gatherings we observed a community leader—from a very large international open-source company—present best practices of inviting new volunteer maintainers with bite-size contributions, while lecturing on the importance of educating regional community management teams on the questions of diversity via strategy documents translated to local languages. In a frustrated response another maintainer sighed in private to Mace conducting fieldwork that "not everyone is Kubernetes!"—continuing that "two years ago it was Docker list [*of contributors*] everyone wanted to be on" (paraphrased in fieldnotes). In contrast to the attractive and prestigious, major infrastructure projects, his now single-person project had absolutely no resources of such scale. He was desperate to find anyone at all to aid with dealing with the piling backlog of technical issues in a programming language now fallen out of fashion. This string figure had so to speak, thinned out.

Contributions from new maintainers usually go through a peer review process. Review animates the existing knowledge infrastructure and involves various tools such as compilers, syntax and style checkers, and indeed testing frameworks described earlier in this paper. Besides tooling, the present maintainers pay attention to the proposed changes and scrutinize them in the light of their local experience and resources (Bialski 2019). Not only machines, but humans too are rendered as recipients of code (Mackenzie 2006). Scrutinizing code is costly, but if a community of programmers can invest in evaluating which contributions are valuable, they also learn to tell the valuable contributors from "drive-by contributors," and configure them as people the software depends on. Successful onboarding produces valuable contributors out of their contributions.

Retaining

During its lifetime a body of software might come to be touched by many programmers. This isn't the case universally—the long tail of codebases is only ever touched by one or very few people. Either way, programmers come to work on many softwares in their lifetime in paid work, pet projects, personal development studies, and fixing someone else's code (Sollfrank and Soon 2021).

Pointing to the precarious economic reality ("the hustle") which mature software must navigate, one of our informants claimed that "[patronage based] sponsorship is all nice-nice, but any serious money is in consultancy jobs." Another informant characterized software production as "get paid once, maintain forever." While long, stable careers do exist in software engineering (Cohn 2016, 2017; Ojala 2021), "tech" is marked by high mobility. The implication for maintenance is that the configurations of contributing people are unstable, and retaining well socialized contributors is a practical problem throughout the patchwork of digital infrastructure.

Letting Go

The trials to which software is put in the dynamic world continuously question the binds on which software depends on for its continuity. A crucial trial is when a valuable contributor leaves. Reasons for leaving mentioned both by programmers online and those at the gatherings were no less diverse than life itself, including: waning interest, starting a more exciting new project, retirement, a company merger, mutiny, bullying, burnout, or death. A senior founder told us that he had been gradually sidelined on a project he co-started, eventually becoming "a cleaning lady," and (gendered) maintenance was something "he could still do," before dropping off the project altogether. Our colleague has conducted fieldwork on a tech company with Ukrainian developers, whose life-priorities have obviously shifted radically at the time of writing in 2023. In any case, a leaving maintainer takes valuable, vital knowledge with them.

Software whose last maintainer leaves slips into a haunting state of *obsolescence* (Peters 2015, 90) between being maintained and being finally gone. Particularly open source software tends to linger on online, unmaintained well after its valuable contributions have ceased and valuable contributors have left. Most code on GitHub® quickly becomes unmaintained and abandoned (Coelho et al. 2020). Many developers admit having left behind unmaintained software projects, and the authors of this paper do too. With no valuable contributors the string figure is inanimate, no longer passed on hand to hand, and no longer held together by the tensions of doing so. Software without maintainers has fallen into obsolescence, and has fallen out of love.

Materializing Common Knowledge Manifests the String Figure and Enables Coordination

As expressed in the very notion of "source code," software is ideally entirely defined by its quintessential material: code (Krysa and Sedek 2008). In this view, code does not describe or represent the software at hand, but code is the software. This view implies that to know a software system, would be to trace it back to its source. Hence, programmers tend to prioritize code contributions as the valuable contributions. We do not wish to categorically deny this view; we agree with software studies scholars that code have been marginalized in socio-cultural studies, which is why including it would be worthwhile (Mackenzie 2006; Marino 2020). At the same time, the very boundaries of code are fluid and contested (Couture 2019), have evolved throughout the history of computing, and continue to do so. While the historical movement of programming language development—to make code more like "natural language"—partially succeeds, it at the same time masks the fundamentally nonlinear nature of software. Code expressed in a higher level programming language such as COBOL or Python® might have the appearance of words, statements and even sentences, but that is hardly enough to accept it as a language proper. (Marino 2020). Instead code branches, forks, loops and folds back onto itself. Furthermore, sections of code might never execute due to

deprecation, obsolescence or bugs. Therefore the notion of knowing code by individually "reading" it is misleading.

Maintaining software over time poses unique epistemic limits: expecting individual people to "know" thousands of lines of interacting and branching code is unreasonable, even when they are its sole author. It is utterly hopeless to strive to know software in some final sense, to give an inventory of facts or a conclusive account of it. Such requirements aim at epistemic closure and timelessness. Instead, maintenance is oriented towards time, timeliness and capacity to add on to that time.

We have therefore argued that it is more appropriate to appreciate the collective and interpretive quality of software maintenance knowledge. Undoubtedly real regularities across software projects exist: the market hegemony of GitHub®, agreement about programming languages and coding standards, style, design patterns, documentation practices, code analyzers, fashions and trends, cultivated and disciplined aesthetics, as this paper has shown. However, each project is also unlike others; what materially gets archived in the version control system, what code behavior is disciplined by the test framework, and what experience new, mature or senior maintainers acquire, is unique to each software project. Bodies of code thus develop their own biographies, to be interpreted amidst hybrid assemblages which bind multiple human and non-human knowledges together. To negotiate the politics and pragmatics of how to achieve a knowledge infrastructure for the interpretive, cooperative task of sustaining code and the relations it is suspended by then would be crucial activity. Since a final truth about software is relatively irrelevant for the task of maintenance, this knowledge is cobbled together piecemeal from a set of incomplete and even conflicting practices and beliefs of tools and people. This paper has analyzed and drawn together debates about what is recorded by version control systems, why aren't tests comprehensive, whom to onboard as valuable contributors, what incentives to offer to current ones, and why do old maintainers leave.

When experience acquired through trials is sufficiently articulated, common knowledge i.e. higher order collective knowledge can gradually be achieved by the network of actors which supports the software. Individual tools as well as people can set aside from their working memory (to use an emic term from computation) what the software is, and pragmatically coordinate collective action. As maintenance experience is gradually materialized and made common, each agent can trust—or at least hope—that others know too, and that one is not the sole knower of any one thing. Knowledge can thus be encapsulated (another emic term from computation), black boxed (Latour 1987), and effectively forgotten with a reasonable hope that it can be retrieved, opened and remembered later by others who need to intervene in code at some unpredictable future trial. This common knowledge is not held by individuals but by collectives (Mathiesen 2007; Vanderschraaf and Sillari 2013). Importance of non-human actors in the work of knowing must be acknowledged. For instance when unit tests know a code function well, the test framework can be trusted to give a warning before the function breaks. When a module is given a good name, a human can be trusted to be able interpret what it is expected to do. When an experienced maintainer has good reasons to stick around, they can be trusted to attend to future trials. While individual tools and people forget, this common knowledge held by the string figure remembers and suspends software in time.

Conclusion

Steven Jackson's text *Rethinking Repair* called for *broken world thinking* (Jackson 2014; for critique see Ritasdatter 2020). We find that the software maintainers embody this thinking. As we have seen,

maintenance of software is not only about keeping code going, but finding ways to know and unknow the code you already have. As tool decay over time, communities struggle to onboard new contributors, and experienced maintainers leave, codebases risk falling out of maintenance. However, if maintenance knowledge has successfully been made available so that others know what each knows, i.e. *common*, it does not rest entirely on these individual agents, but is collective in kind. We have called this the string figure that software depends on.

The need to maintain software systems often comes from articulation needed between new development and the existing, older systems (Cohn 2019). Frustration mounts when programmers would wish to make the existing simpler, clearer, and more elegant, but management, clients or users fail to see intrinsic value in what doesn't add new functionality. Continuous pressure to legitimize basic care work is among the familiar themes of repair and maintenance studies. Further, as documented by feminist social historians decades ago and known by marginalized groups since forever, ongoing invisible labor makes that which is visible, possible in the first place. Moreover in tech, the bug fixes, performance improvements and security patches as well as other incremental changes to software tend to get co-opted into innovation-speak (Vinsel and Russell 2020) as upgrades, next generation and disruptions. Disruption always comes as a cost, and this cost falls on maintenance. Individuals are left alone to deal with the risks of growing too close and intimate with software which might become "legacy," while neoliberal capitalist calculus casts such biographical concerns over obsolescence aside as economic externalities. To think along with Sara Ahmed's poetic analysis of the word "use", they have been used up (Ahmed 2019). It follows, that to use software is to use its maintainers. Code, like people, risks breaking down. Some breakdowns are catastrophic such as the event-stream hijack exploit (@adam-npm 2018), or a personal burnout—a topic our informants consider a constant threat and wanted to dedicate entire sessions at the unconferences. More often breakdowns are gradual decays into disuse, disinterest and disrepair. Intimate bonds between people and artifacts are relations of vulnerability, and entanglement of biographies of people and of software can drag both down when one falls.

In this paper we have analyzed experiences of practitioning programmers who participate in software maintenance. We focused on their epistemic struggles to know the code in their care. Our theorization shows that when individually held knowledge circulates between actors (for example re-expressed as tests, announced on bug trackers, and while onboarding new contributors), a higher order social phenomenon emerges: common knowledge. Adopting a metaphor from Donna Haraway, and calling this contingent more-than-human achievement a string figure highlights its active, cooperative, dynamic, and also vulnerable nature. With this, we primarily contribute the insight that maintenance knowledge is collective and intersubjective in kind. Additionally by discussing the roles of auxiliary software, precarity of commitments and atomic contributions, and an object which resists knowability, we hope to have substantiated some of the fascinating particularities and dynamics of computer software. While doing so, we wish to persuade social epistemologists with STS insights to accept technology and hybrids as legitimate knowers, or at least quasi-knowers (Freiman and Miller 2020), worthy of analysis.

The implication of viewing software maintenance as a contingent and productive web of recursive, intersubjective relations is that when we call for explainability of algorithms, demand accountability of computational systems, expect long-term sustainability of digital infrastructure, or ask who is welcomed

into communities of computation, we should not turn to individual people, but to the rhizomatic, living legacies which suspend code in time.

Acknowledgements

Jérôme Denis, David Pontille and others organize an ongoing panel series at EASST where in 2020 we presented an early version of this text, and received valuable comments. We thank the two anonymous reviewers for their comments which helped us streamline and substantiate our argument, and the editors for enabling this work. We are grateful for the software maintainers—especially in free and open source—without whose continuous efforts the production and dissemination of this text would be absolutely impossible.

Author Biographies

Mace Ojala is software studies scholar researching maintenance, meanings and cultural techniques of software, and is currently a PhD fellow at Virtual Lifeworlds research centre at Ruhr University Bochum, trying to stare down his arch-enemy the PDF file format by theorizing it into oblivion.

Marisa Leavitt Cohn is an Associate Professor in the Technologies in Practice research group and director of the ETHOS Lab at the IT University of Copenhagen, researching software maintenance, technological obsolescence, and temporalities of infrastructural decay through ethnographic, research-through-design, and feminist-STS approaches.

References

@adam-npm. 2018. "Details about the Event-Stream Incident." *Npm Blog* (Archive). November 27, 2018. Accessed February 23, 2024. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident.

Ahmed, Sara. 2019. *What's the Use?: On the Uses of Use*. Illustrated edition. Durham: Duke University Press. https://doi.org/10.1215/9781478007210.

Bialski, Paula. 2019. "Code Review as Communication: The Case of Corporate Software Developers." In *Communication*, edited by Paula Bialski, Finn Brunton, and Mercedes Bunz, 93–111. Lüneburg: Meson Press. https://meson.press/books/communication/.

Boscoe, Bernadette, and Michael Scroggins. 2019. "Maintaining FITS in Two Careers." Presented at the Conference Maintainers III: Practice, Policy and Care, October 6–9, 2019, Washington DC.

Bowker, Geoffrey C., and Susan Leigh Star. 2000. *Sorting Things Out: Classification and Its Consequences*. Cambridge, MA: The MIT Press.

Chun, Wendy Hui Kyong. 2008. "On 'Sourcery,' or Code as Fetish." *Configurations: A Journal of Literature, Science, and Technology* 16(3): 299–324. https://doi.org/10.1353/con.0.0064.

———. 2011. *Programmed Visions. Software and Memory*. Cambridge, MA: MIT Press.

Coelho, Jailton, Marco Tulio Valente, Luciano Milen, and Luciana L. Silva. 2020. "Is this GitHub Project Maintained? Measuring the Level of Maintenance Activity of Open-Source Projects." *Information*

and Software Technology 122: 1 – 16. https://doi.org/10.1016/j.infsof.2020.106274.

Cohn, Marisa Leavitt. 2016. "Convivial Decay: Entangled Lifetimes in a Geriatric Infrastructure." In Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing, CSCW '16, 1511–23. New York, NY: Association for Computing Machinery. https://doi.org/10.1145/2818048.2820077.

———. 2017. "'Lifetime Issues': Temporal Relations of Design and Maintenance." *Continent* 6(1): 4–12.

———. 2019. "Keeping Software Present: Software as a Timely Object for STS Studies of the Digital." In *DigitalSTS: A Field Guide for Science & Technology Studies*, edited by Janet Vertesi and David Ribes, 423–46. Princeton, NJ: Princeton University Press.

Cohn, Marisa Leavitt, Susan Elliott Sim, and Charlotte P. Lee. 2009. "What Counts as Software Process? Negotiating the Boundary of Software Work Through Artifacts and Conversation." *Computer Supported Cooperative Work (CSCW)* 18(5): 401–443. https://doi.org/10.1007/s10606-009-9100-4.

Collin, Finn. 2019. "The Twin Roots and Branches of Social Epistemology." In *The Routledge Handbook of Social Epistemology*, edited by Miranda Fricker, Peter J. Graham, David Henderson, and Nikolaj J. L. L. Pedersen. Routledge Handbooks Online. New York, NY: Routledge. https://doi.org/10.4324/9781315717937-3.

Conrad, Lisa. 2019. "The Organization is a Repair Shop." *Ephemera: Theory & Politics in Organization* 19(2): 303–24. https://ephemerajournal.org/contribution/organization-repair-shop.

Couture, Stéphane. 2019. "The Ambiguous Boundaries of Computer Source Code and Some of Its Political Consequences." In *digitalSTS: A Field Guide for Science & Technology Studies*, edited by Janet Vertesi and David Ribes, 136–56. Princeton, NJ: Princeton University Press.

Denis, Jérôme, Alessandro Mongili, and David Pontille. 2015. "Maintenance & Repair in Science and Technology Studies." *TECNOSCIENZA: Italian Journal of Science & Technology Studies* 6(2): 5–15. https://doi.org/10.6092/issn.2038-3460/17251.

Denis, Jérôme, and David Pontille. 2020. "Why Do Maintenance and Repair Matter?" In *The Routledge Companion to Actor-Network Theory*, edited by Anders Blok, Ignacio Farías, and Celia Roberts, 283–93. London, England, and New York, NY: Routledge.

———. 2017. "Beyond Breakdown: Exploring Regimes of Maintenance." *Continent* 6(1): 13–17.

———. 2021. "Maintenance Epistemology and Public Order: Removing Graffiti in Paris." *Social Studies of Science* 51(2): 233–58. https://doi.org/10.1177/0306312720956720.

Ensmenger, Nathan. 2014. "When Good Software Goes Bad: The Surprising Durability of an Ephemeral Technology." Paper presented at the Conference MICE: Mistakes, Ignorance, Contingency, and Error in Science and Technology, October 2–4, 2014, Münich. Accessed February 23, 2024. http://homes.soic.indiana.edu/nensmeng/files/ensmenger-mice.pdf.

Feathers, Michael C. 2004. *Working Effectively with Legacy Code*. First Edition. Upper Saddle River, NJ: Pearson.

Freiman, Ori, and Boaz Miller. 2020. "Can Artificial Entities Assert?" In *The Oxford Handbook of Assertion*, edited by Sanford C. Goldberg, 414–34. Oxford, UK: Oxford University Press.

Geiger, R. Stuart, Dorothy Howard, and Lilly Irani. 2021. "The Labor of Maintaining and Scaling Free and Open-Source Software Projects." *Proceedings of the ACM on Human-Computer Interaction* 5(1): 1–28. https://doi.org/10.1145/3449249.

Goodwin, Charles. 1995. "Seeing in Depth." *Social Studies of Science* 25(2): 237–74. https://doi.org/10.1177/030631295025002002.

Graham, Stephen, and Nigel Thrift. 2007. "Out of Order: Understanding Repair and Maintenance." *Theory, Culture & Society* 24(3): 1–25. https://doi.org/10.1177/0263276407075954.

Gupta, Akhil. 2018. "The Future in Ruins: Thoughts on the Temporality of Infrastructure." In *The Promise of Infrastructure*, edited by Nikhil Anand, Akhil Gupta, and Hannah Appel, 62–79. Durham, NC: Duke University Press. https://doi.org/10.1515/9781478002031-004.

Haraway, Donna J. 2013. "SF: Science Fiction, Speculative Fabulation, String Figures, So Far." *Ada. A Journal of Gender, New Media & Technology* 3. Accessed February 23, 2024. https://scholarsbank.uoregon.edu/xmlui/bitstream/handle/1794/26308/ada03-sfsci-har-2013.pdf?sequence=1&isAllowed=y.

———. 2016. *Staying with the Trouble: Making Kin in the Chthulucene*. Illustrated Edition. Experimental Futures. Durham, NC: Duke University Press.

Heidegger, Martin. 1977. *The Question Concerning Technology, and Other Essays*. Translated and with an Introduction by William Lovitt. New York, NY: Harper Torchbooks.

Helmond, Anne. 2015. "The Platformization of the Web: Making Web Data Platform Ready." *Social Media + Society* 1(2). https://doi.org/10.1177/2056305115603080.

Henderson, Kathryn. 1991. "Flexible Sketches and Inflexible Data Bases: Visual Communication, Conscription Devices, and Boundary Objects in Design Engineering." *Science, Technology, & Human Values* 16(4): 448–73. https://doi.org/10.1177/016224399101600402.

Henke, Christopher R. 2000. "The Mechanics of Workplace Order: Toward a Sociology of Repair." *Berkeley Journal of Sociology* 44: 55–81.

Henke, Christopher R., and Benjamin Sims. 2020. *Repairing Infrastructures: The Maintenance of Materiality and Power*. Cambridge, MA: MIT Press.

Hutchins, Edwin. 1995. *Cognition in the Wild*. Cambridge, MA: MIT Press.

Ingold, Tim. 2000. *The Perception of the Environment: Essays on Livelihood, Dwelling and Skill*. London, England, and New York, NY: Routledge.

Jackson, Steven J. 2014. "Rethinking Repair." In *Media Technologies: Essays on Communication, Materiality, and Society*, edited by Tarleton Gillespie, Pablo J. Boczkowski, and Kirsten A. Foot, 221–40. Cambridge, MA: The MIT Press.

Knorr Cetina, Karin, and Urs Bruegger. 2002. "Global Microstructures: The Virtual Societies of Financial Markets." *American Journal of Sociology* 107(4): 905–50. https://doi.org/10.1086/341045.

Kocksch, Laura, Matthias Korn, Andreas Poller, and Susann Wagenknecht. 2018. "Caring for IT Security: Accountabilities, Moralities, and Oscillations in IT Security Practices." *Proceedings of the ACM on Human-Computer Interaction* 2: 1–20. https://doi.org/10.1145/3274361.

Krysa, Joasia, and Grzesiek Sedek. 2008. "Source Code." In *Software Studies: A Lexicon*, edited by Matthew Fuller, 236–43. Cambridge, MA: MIT Press. https://doi.org/10.7551/mitpress/7725.003.0036.

Latour, Bruno. 1987. *Science in Action: How to Follow Scientists and Engineers Through Society*. Cambridge, MA: Harvard University Press.

———. 1990. "Technology Is Society Made Durable." *The Sociological Review* 38(1): 103–31. https://doi.org/10.1111/j.1467-954X.1990.tb03350.x.

Lindén, Lisa, and Doris Lydahl. 2021. "Editorial: Care in STS." *Nordic Journal of Science and Technology Studies* 9(1): 3–12. https://doi.org/10.5324/njsts.v9i1.4000.

Mackenzie, Adrian. 2006. *Cutting Code: Software and Sociality*. Frankfurt am Main: Peter Lang.

Marino, Mark C. 2020. *Critical Code Studies*. Cambridge, MA: MIT Press.

Mathiesen, Kay. 2007. "Introduction to Special Issue of *Social Epistemology* on 'Collective Knowledge and Collective Knowers.'" *Social Epistemology: A Journal of Knowledge, Culture and Policy* 21(3): 209–16. https://doi.org/10.1080/02691720701673934.

Mattern, Shannon. 2018. "Maintenance and Care." *Places Journal*. https://doi.org/10.22269/181120.

Miller, Boaz, and Ori Freiman. 2020. "Trust and Distributed Epistemic Labor" In *The Routledge Handbook on Trust and Philosophy*, edited by Judith Simon, 341–53. New York, NY: Routledge.

Munroe, Randall. 2020. "Dependency." *Xkcd: A Webcomic of Romance, Sarcasm, Math, and Language*. Comic feed. August 17, 2020. Accessed February 23, 2024. https://xkcd.com/2347/.

Neff, Gina, and David C. Stark. 2004. "Permanently Beta: Responsive Organization in the Internet Era." In *Society Online: The Internet in Context*, edited by Philip N. Howard and Steve Jones, 173–88. Thousand Oaks: SAGE Publications, Inc. https://doi.org/10.4135/9781452229560.

Ojala, Mace. 2021. "Maintain-Ability. A Thesis on Life Alongside Computer Software." Master's Thesis, Tampere: Tampere University. Trepo. https://urn.fi/URN:NBN:fi:tuni-202202031820.

Paine, Drew, and Charlotte P. Lee. 2017. "'Who Has Plots?': Contextualizing Scientific Software, Practice, and Visualizations." *Proceedings of the ACM on Human-Computer Interaction* 1: 1–21. https://doi.org/10.1145/3134720.

Payette, Sandy. 2014. "Hopper and Dijkstra: Crisis, Revolution, and the Future of Programming." *IEEE Annals of the History of Computing* 36(4): 64–73. https://doi.org/10.1109/MAHC.2014.54.

Peters, John Durham. 2015. "Proliferation and Obsolescence of the Historical Record in the Digital Era." In *Cultures of Obsolescence: History, Materiality, and the Digital Age*, edited by Babette B. Tischleder and Sarah Wasserman, 79–96. New York: Palgrave Macmillan. https://doi.org/10.1057/9781137463647_5.

Proctor, Robert N., and Londa Schiebinger, eds. 2008. *Agnotology: The Making and Unmaking of Ignorance*. Stanford, CA: Stanford University Press.

Puig de la Bellacasa, Maria. 2011. "Matters of Care in Technoscience: Assembling Neglected Things." *Social Studies of Science* 41(1): 85–106. https://doi.org/10.1177/0306312710380301.

———. 2017. *Matters of Care. Speculative Ethics in More than Human Worlds*. Minneapolis, MN: University of Minnesota Press.

Ritasdatter, Linda Hilfling. 2020. "Unwrapping Cobol: Lessons in Crisis Computing." PhD Thesis, Malmö: Malmö University Publications. http://urn.kb.se/resolve?urn=urn:nbn:se:mau:diva-17847.

Ruparelia, Nayan B. 2010. "Software Development Lifecycle Models." *ACM SIGSOFT Software Engineering Notes* 35(3): 8–13. https://doi.org/10.1145/1764810.1764814.

Sipser, Michael. 2013. *Introduction to the Theory of Computation*. Third Edition. Boston, MA: Cengage Learning.

Sollfrank, Cornelia, and Winnie Soon. 2021. *Fix My Code*. Berlin: Eeclectic.

Souza, Cleidson de, Jon Froehlich, and Paul Dourish. 2005. "Seeking the Source: Software Source Code as a Social and Technical Artifact." In *Proceedings of the 2005 ACM International Conference on Supporting Group Work*, 197–206. New York, NY: Association for Computing Machinery. https://doi.org/10.1145/1099203.1099239.

Star, Susan Leigh. 1999. "The Ethnography of Infrastructure:" *American Behavioral Scientist* 43(3): 377–91. https://doi.org/10.1177/00027649921955326.

Star, Susan Leigh, and Karen Ruhleder. 1996. "Steps Toward an Ecology of Infrastructure: Design and Access for Large Information Spaces." *Information Systems Research* 7(1): 111–34. https://doi.org/10.1287/isre.7.1.111.

Tronto, Joan C. 1998. "An Ethic of Care." *Generations: Journal of the American Society on Aging* 22(3): 15–20. https://www.jstor.org/stable/44875693.

Turing, Alan Mathison. 1937. "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society* s2–42(1): 230–65. https://doi.org/10.1112/plms/s2-42.1.230.

Vanderschraaf, Peter, and Giacomo Sillari. [2001] 2013. "Common Knowledge." In *The Stanford Encyclopedia of* Philosophy Archive, edited by Edward N. Zalta. Metaphysics Research Lab, Stanford University. Accessed February 23, 2024. https://plato.stanford.edu/archives/fall2021/entries/common-knowledge/.

Vertesi, Janet. 2015. *Seeing Like a Rover: How Robots, Teams, and Images Craft Knowledge of Mars.* Chicago, IL: University of Chicago Press.

Vinsel, Lee, and Andrew L. Russell. 2020. *The Innovation Delusion: How Our Obsession with the New Has Disrupted the Work that Matters Most*. New York, NY: Currency.

Wilde, Mandy de. 2021. "'A Heat Pump Needs a Bit of Care': On Maintainability and Repairing Gender–Technology Relations." *Science, Technology, & Human Values* 46(6): 1261–85. https://doi.org/10.1177/0162243920978301.

Yuill, Simon. 2008. "Concurrent Versions Control." In *Software Studies: A Lexicon*, edited by Matthew Fuller, 64–69. Cambridge, MA: MIT Press. https://doi.org/10.7551/mitpress/7725.003.0011.