Hardcoding the “Smart” Factory
The Specification of Ignorance in Robot Software

Tudor B. Ionescu
1. Introduction

In the corner of a factory, there is a prototype consisting of two robots and a small conveyor belt mounted on a tabletop. Next to it, a group of factory visitors eagerly awaits the start of the demonstration. A man in a white overall starts the production process by pressing a button on a computer screen, which triggers a robot to pick up a workpiece. Like a dancing snake, the robot moves along unexpected paths and—to the astonishment of the audience—avoids several obstacles before handing the workpiece over to a second robot that places it in a precisely defined location. Cyber-physical systems such as the one being demonstrated, the presenter explains, are essential to the fourth industrial revolution—one that the audience can only begin to comprehend.

1 The vision of a “fourth industrial revolution” (Schwab, 2017) enabled by a “smarter”, fully interconnected factory, pushed software to the forefront of its automation discourse. That conception calls upon European manufacturing companies to leverage information technologies to boost their productivity and competitiveness in an increasingly globalized manufacturing market. The vision of a fully interconnected production system that extends beyond the boundaries of a single factory or country was prompted by a growing European sense of insecurity caused by the alleged willingness and capacity of East Asian manufacturers to put European factories out of business.

2 This seemingly cutting-edge idea is far from new. The smart factory vision is reminiscent of a trend in automation from the 1980s, known as Computer Integrated Manufacturing (CIM). CIM envisioned a worker-free factory thanks to the massive rollout of interconnected industrial robots and other computer technologies. Yet, CIM struggled with the inherent limitations of then immature technologies (Heßler, 2014).
In the early 2000s, with the widespread use of the Internet and the coming of age of software technologies, a renewed attempt at developing a fully interconnected, semi-autonomous factory became increasingly compelling. In 2012, the German National Academy of Science and Engineering (an industrial lobbying organization) seized the moment and published a research policy document, known as “Industrie 4.0” (Kagermann et al., 2013), aligning the smart factory with the previous industrial revolutions (Figure 1). In Industrie 4.0, so-called cyber-physical systems (CPS) are deemed to play an important role in the realization of the factory of the future. Thanks to their advanced sensor and communication systems, CPS develop a kind of awareness of their socio-technical environment by modeling it in software.

Over the past 10 years, Industrie 4.0 has found tremendous resonance among governments and corporations all around the world. Especially European governments and companies funded a large number of research programs aimed at bringing the smart factory vision closer to reality. In practical terms, developing the “smart factory” requires submersion in various production locales to (re)code the interface between software and the sociotechnical fabric of the factory. Consequently, the Industrie 4.0 vision propelled a growing number of software developers to the factory shop floor, where they faced unexpected challenges.

My focus in this paper will be on a particular coding practice, called hardcoding. This practice arguably plays a significant role in prototyping Industrie 4.0 manufacturing systems. Hardcoding refers to the practice of writing starkly simplified code tailored to a very specific purpose in order to avoid the complexities of an algorithmic solution that may be infeasible, cost-prohibitive, or may simply not exist. While the software loses versatility, it becomes capable of doing a specific thing with relatively little development effort. In software engineering terms, hardcoding contributes to the technical debt of a system. This metaphor refers to the strategy and practice of deferring a sound implementation to advance the release of a functional yet technically flimsy version of a software program. Technical debt needs to be managed (e.g., by “refactoring”—that is, rewriting and/or reorganizing the code and improving on various technical aspects) before it accumulates too much “interest” (Cunningham, 1992).
From an analytical perspective, hardcodes can be used as indicators of various problems. By studying them, one might gain more insights into the epistemic entanglements of the developers and into how software production works in a specific organizational context. This brings up the main question I seek to investigate in this text: How and why do developers hardcode? This question is of particular interest in the context of the Industrie 4.0 vision, since it builds on cyber-physical systems that blend information technologies and industrial robotics with practices of modeling and simulation (Kagermann et al., 2013). CPS are intrinsically entangled with their operational environment—the sociotechnical fabric of the factory.

This raises a series of challenges to developing CPS: the paper will explore them by the creation of a demonstrative CPS prototype in a factory owned by a large technology company. Over the course of about two years, I observed the project in situ as a member of the development team. My observations suggest that some of the issues were epistemic in nature. These issues urged the developers to improvise solutions by hardcoding the interfaces between the software and its physical and operational environment. In this context, hardcoding seems to be a symptom of a kind of technical debt that can only be managed by speculating about potential solutions based on future technologies. Hardcodes reflect the “yet-to-be-knowns” and “yet-to-be-dones” in a software’s source code, thus conveying a project’s history of challenges and compromises.

In the studied case, demonstrative CPS prototypes, or simply demonstrators, may be regarded as models of future cyber-physical factories (Ionescu and Merz, 2018). Their development resembles a kind of costly and meticulous modelmaking, exploring the potential and limits of technologies that are specified only discursively, as it is the case with Industrie 4.0. As part of their innovation strategy, companies seek ways to bring such visions to reality in an incremental way by solving the challenges of developing them one by one. In this sense, technology demonstrators, in general, and CPS demonstrators, in particular, fulfill multiple functions. First, they reify the technological vision just enough to stimulate investments in certain research directions bearing the potential to become "hot topics" with high market potential in the near future. Second, they have an educational purpose in creating working models of technologies "in the making." Drawing on their potential to reify and educate, CPS demonstrators also act as "third mission" vectors by helping to shape broader research agendas in the industry and society in a way that is complementary to marketing activities. And third, as a form of experimental development, CPS demonstrators fulfill an epistemic function in that they specify what is not (yet) known about the supporting technologies that would facilitate a full-fledged implementation of Industrie 4.0 in various socio-economic environments, such as the factory.

This paper focuses on the third of these functions, as it seeks to investigate hardcoding as an epistemic practice in the context of experimental CPS development. Industrie 4.0 does not yet exist; it is merely a value-laden technological vision with a high potential impact on industry and society. This conflation of future and present entails a lot of uncertainty as well as management and market pressure on technology companies and their employees. How do software engineers cope with this uncertainty when they are faced with path-dependent design choices and ambitious innovation goals in smart factory projects?
The paper starts with a presentation of the case study and methodology, followed by the introduction of the theoretical framework used to analyze the ethnographic material, on the basis of which I would like to develop the main argument of the text. Then, it will explore four ethnographic moments in the work of a team of software engineers, for whom hardcoding represents both a quick solution to some technical problems and a challenge to achieving the overall goals of the ambitious Industrie 4.0 vision.

2. Case Study and Methodology

The present study reconstructs the development of a demonstrative CPS in a large technology company. The studied project aimed at transferring the results from previous research projects into a real factory. The factory is a producer of electronic components for the industrial domain, employing hundreds of workers and dozens of product design and industrial engineers. The primary goal of the project was to demonstrate the feasibility of the Industrie 4.0 paradigm to a broad company-internal audience. In line with this paradigm, the patented CPS blueprints conceived in earlier projects foresaw a fully decentralized, autonomous, agent-based architecture for the next generation of production systems, wherein the product would steer its production in a fractally decomposed and geographically distributed smart factory.

The organizational context of the study is that of a research and development division tasked with conducting applied research. Within the company’s innovation strategy, the expectation of projects like the one being studied is for a business division to take an interest in the demonstrative, proof-of-concept prototypes created by the research division and, eventually, develop them into marketable products. An essential requirement of the company’s innovation process is for research engineers to produce invention disclosures that can eventually be filed as patents. This process invites various members of the Research and Development (R&D) division to act strategically to the end of pushing their own ideas to the top of a division’s product roadmap. Once the decision to patent an invention is taken, the authors and owners of the patent often engage in different forms of dissemination and persuasion aimed at convincing business divisions to invest in the development of products based on the respective patent. In this sense, demonstrative CPS aim to reify the idea of the smart factory in a compelling way for a wide company-internal audience.

Empirically, the study is based on my participant observations of the studied project. Over a period of about 2 years, I was involved in multiple activities related to the development of different CPS prototypes as a research engineer and software architect. This paper focuses on a project aimed at developing a CPS prototype in a real factory. Consequently, the members of the project team, including myself, conducted most of their work in that factory. The team comprised 7-10 members, depending on the project phase. Most of them were software developers or architects. Whereas a developer is mainly a coder, who also contributes to the detailed design of the software, a software architect focuses on the design of the software, the development process, and the communication within and beyond the team. There was one project manager, who also acted as architect, and 2-3 industrial engineers, the latter of whom were concerned with the integration of the various hardware components (robots, assembly jigs, conveyor, etc.).
Initially, my research interests were rather technical, while only touching upon the challenges of software engineering on the shop floor. As a member of a software engineering research group, I always sought to improve the development methods that were used in R&D projects by drawing on (self-)observation and reflection, as it is common in qualitative research in software engineering. When the development work took an interesting turn, I started reflecting on the broader implications of what we were doing in a factory that increasingly resembled a laboratory (Miller and O'Leary, 1994). This suggested that a micro-perspective committed to practice-oriented science and technology studies (STS) might provide interesting insights into how software for the shop floor, driven by the Industrie 4.0 vision, is being developed in situ. Consequently, I started writing field notes focused on my work in the project and on other activities related to the CPS research agenda of the company. My attention was on our practices and our interactions with CPS technologies in the making. The result was a field protocol of about 65 typed pages that might be described as a “technography” of CPS development within the scope of my involvement in these research and development activities.

According to Paßman and Schubert (2020), “technography” is to be understood as an ethnographically inspired analysis of the production and use of technology—or, in Kline’s (2008) synoptic formulation: “technography = technology + ethnography.” In this sense, my field notes align with Rammert’s (2007) recommendation to “[f]ollow the practices and the things and describe the relations and interactivities” (p. 11, my translation), while precisely investigating as many details of things, interpretations, and activities as possible.

To distance myself from the “native” perspective, I confronted my field notes with other social scientists on several occasions as part of the PhD seminar in STS that I have attended between 2018 and 2021. In addition, I examined the themes transpiring from the field notes in light of the relevant STS and software studies debates, as suggested by Amann and Hirschauer (1997). The empirical material used in the present text underwent several reviews. My analysis of the empirical material was guided by a grounded theoretical approach (Corbin and Strauss, 2008; Strübing, 2014). The most recent iteration in this process focused on the recurring theme of “hardcoding,” which I found to be pivotal in understanding a phenomenon that I would describe as “coding in uncharted territory.” The following text should therefore be read as an exploration of the functions, meanings, and performances of hardcoding as one of many coding practices that pervade the software professions.

3. Coding as an Epistemic Practice

Coding has been described as a collaborative authoring practice (Couture, 2012) taking place in office-like settings (Schmidt, 2008). Morner and Krogh (2009) note that developers codify their explicit and tacit knowledge — the product of social interaction and communication — into source code. In this sense, open-source projects represent knowledge sources for other developers, who reproduce and alter the knowledge of their peers in their own projects. Morner and Krogh (2009) start from the observation that knowledge is situated in practice and is sensitive to the social values and cognitive categories used be the actors and their socio-material circumstances. Studies of distributed software development are concerned with processes, social interaction, and
knowledge sharing in distributed teams (Bruun and Sierla, 2007; Schulz-Schäfer and Bottel, 2017).

These views arguably align well with the modern understanding of coding and, more generally, of software production as an epistemic, collaborative, possibly distributed authoring practice, with the knowledge being invested both in the product and in the practice. Contrasting this view, Ensmenger (2010) notes that, originally, John von Neumann and Herman Goldstine used the term “coders” in reference to the women performing the “hand-work” of translating the “head-work” of men into a machine-understandable format. As Ensmenger notes,

Coding was a static process that could be performed by a low-level clerical worker.

‘Coding’ implied mechanical translation or rote transcription; ‘coders’ were obviously low on the intellectual and professional status hierarchy. (2010, p. 124)

While some authors address the broad question of what software is (Chun, 2008) and how it pervades the fabric of our society (Mackenzie, 2006), a different possible approach is to observe that there is no such thing as “code” as an amorphic substance and “coding” as an undifferentiated practice. Instead, as Ensmenger (2010) notes, there are different kinds of coding practices exercised by actors enjoying diverse intellectual statuses, depending on the historical moment and on the maturity of the different technologies used. When compiled and executed, the products of these coding practices leave the impression of a cohesive, performative technology, called software.

Against this background, this paper attempts an analysis of “hardcoding,” by the example of a project that involves both “head-work” and “hand-work” performed by software engineers. This analytical and empirical approach is committed to a practice-oriented science and technology studies (STS) research methodology: it examines software development from a micro-perspective by means of an ethnographic approach focused on social organization, artifacts, and processes.

I subscribe to the view that different codes consolidate the various social interactions and the communication of actors, whose knowledge is situated in multiple and distinct coding practices. This view is partly inspired by Morner and Krogh’s (2009) argument that software is the product of communities or organizations. As such, it embodies their knowledge, their norms, and their values. This conceptualization naturally questions the performative nature of code. Performativity is a contentious issue in software studies. Responding to Galloway (2004), Chun (2008) notes that code is performative only after the fact or after it has been pulled through “a whole imagined network of machines and humans” (p. 1). This is to say that, as a textual artifact, code is disconnected from its “command and execution” performance. Chun attributes this feature to the military history of software.

I take inspiration from Chun’s idea of the “disconnection” of code from the performance of executable software, in that the code reflects a history that cannot be told from a performance of it alone. As Ensmenger (2009) notes, “[s]oftware is history, organization, and social relationships made tangible” (p. 90). In this sense, I would argue that, by studying hardcoding, interesting histories are made visible. Hardcodes can effectively facilitate credible performances even in the lack of the knowledge required to create a software system that genuinely performs as expected. Thus, hardcodes shed light on the history of ignorance within projects and organizations.
3.1. What is Hardcoding?

The notion of “hardcoding” refers to a set of programming practices that produce program logic in the most direct, purposeful way possible, to the end of achieving an effective and immediate result. Hardcoding is generally regarded as an “anti-pattern” or considered “bad practice” because it sacrifices versatility, portability, adaptability, generality or reusability – qualities that computer programs are expected to fulfill (Smit et al., 2011). Hardcoding does render parts of a program inert in an almost mechanical way; its effectiveness shines in situations where a "quick and dirty" solution does the job. (Concrete examples of hardcoding are provided in the article's appendix.) Hence, hardcoding is controversial among software professionals.

In technology ethics, hardcoding is sometimes invoked as a means for “taming” artificial intelligence technologies, notably machine learning models, which sometimes produce results that are counterintuitive or challenge our social norms and values. Various scholars in this field see hardcoding as a potential means for introducing norms and values into software-based systems ex post facto. For example, writing about "value-sensitive design" by the example of Wikipedia, Rychwalska and Roszczyńska-Kurasinska (2017) note that “[s]ocial order value can also be promoted through hardcoding some procedures into the platform or through running bots” (p. 52). Leenes et al. (2017) note that hardcoding appears to be the method of choice when it comes to enforcing traffic rules in car software or ensuring data protection by deletion. Yet, as Leenes et al. (2017) further note, “safeguards can, in theory, be hardcoded … but hardcoded rules will often be too rigid (not allowing for context-sensitivity or multi-purpose use)” (p. 29). Referring to Blockchain technology, Sulkowski (2020) considers “hardcoding ethical rules” in so-called “decentralized autonomous organizations” as a form of “action without deliberation,” which can be aligned “with nature and natural laws” (p. 168).

While in the field of technology ethics, hardcoding appears to provide a useful construct to work with on the short term, in social studies of software it received little explicit attention. This is perhaps because, to an external observer, hardcodes can be indistinguishable from other codes. To fill this gap, I would argue that existing conceptualizations of coding do not capture the gist of the controversial practice of hardcoding. Harcodes arguably reflect expectations of a piece of software that cannot be reasonably met in a given context. In modern software production, coding and demonstration are intrinsically linked. This is also the case with Industrie 4.0 projects, in which prototyping and demonstration are means to an end of bringing a particular sociotechnical vision closer to reality.

Rosental (2011) observes that public demonstrations structure socio-economic interactions and exchanges in attempts by various individual and institutional actors to lead the world, sell products and build markets. More specifically, Coopmans (2010) and Simakova (2010) note that technology demonstrations play important roles in marketing events, such as industry fairs, while primarily targeting potential buyers and the competition as well as wider public audiences. Such demonstrations are being carefully enacted by their promoters, while strategically revealing and concealing certain aspects (Coopmans, 2010). Demonstrators are often inspired by prototype scenarios (Schulz-Schaeffer and Meister, 2017), defined as negotiation arenas between a fictional and an empirical reality, while embodying a simplified and fragmented
version of an imagined future reality (p. 11). Demonstrators may also be understood as models, the effectiveness of which is usually evaluated in consideration of their capacity or quality to "represent" objects accurately (Ionescu and Merz, 2018). Practice-oriented STS approaches address the ways in which models are being developed and used in different contexts as well as the roles they play in those contexts (Merz, 1999).

In the case of CPS, my observations suggest that hardcoding plays an important role in "the negotiation arena" between the imagined futures of manufacturing and the current realities of software practices. This balance requires identifying and concealing what is not yet known by emphasizing the promises of different technologies "in the making". In this context, hardcoding appears to be the result of a kind of epistemic pragmatism, which reduces the necessity for knowledge about how to build future-oriented systems to a minimum, by focusing on demonstration. From this perspective, hardcoding is epistemically relevant because it both helps to constructively pin down and to strategically conceal the unknowns in the process of developing CPS.

3.2. The Epistemic Relevance of Hardcoding

Böschen et al. (2010) note that when dealing with the unknown or yet to be known in their fields of research, scientific communities develop specific scientific cultures of non-knowledge (or ignorance)—a concept related to that of epistemic cultures (Knorr Cetina, 1999). In scientific cultures of ignorance, "there can be knowledge about what is not known," as Gross (2007, p. 742) puts it. As a result, Böschen et al. (2010) note, scientists develop ways of dealing with the unknown, including "strategies to react to unexpected results and events" (p. 788). Marcheselli (2019) notes that "from newspapers to funding proposals, scientists spend thousands of words describing what is not yet known" and that "the agreement on what is unknown as foundational for future scientific developments has been given different names" (p. 4). One of the names to which Marcheselli points is "specified ignorance," a concept used by Merton (1987) retrospectively in reference to what he earlier described as "the express recognition of what is not yet known but needs to be known in order to lay the foundation for still more knowledge" (Merton, 1971, p. 191); and "a first step toward supplanting that ignorance with knowledge" (Merton, 1957, p. 417).

I will not go as far as to explore the different kinds of non-knowledge that are characteristic for the studied project and team as, for example, Gross (2007) and Marcheselli (2019) do. Based on my own observations, I found the work of software developers to be not entirely comparable to that of scientists. Instead, I conceive of hardcodes as a kind of specified ignorance in the source code of CPS software. The extent and timing of hardcodes can serve as an indicator of the (epistemic) challenges encountered by coders in their attempt to make a system work. By enabling simple temporary solutions to complex problems, hardcodes work as bypasses or shortcuts, but they do not hold indefinitely. In the long term, they accumulate as a special kind of "technical debt," the "repayment" of which is contingent on technical progress rather than on investments of time, effort, and expertise.
3.3. The Notion of Technical Debt

From Ward Cunningham’s original definition as “not quite right code which we postpone making it right” (Cunningham, 1992) to its modern understanding as “the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution” (Technopedia, 2020), the metaphor of “technical debt” has become a central concept in software engineering. The versatility of the economic notion of debt led to multiple variations (e.g., code debt, test debt, architectural debt, documentation debt, etc.), which “diluted” the concept and thus required a more systematic attempt at theorizing it (Kruchten et al., 2012). A first step in managing technical debt is to make it explicit, for example by tracking the work that has been postponed along with the other tasks pertaining to the development of new features (Kruchten et al., 2012). Then, depending on the nature of the debt (e.g. code, architecture, documentation), different tools and methodologies can be used to manage it over time. As Kruchten et al. (2012) note, “the major cause of technical debt is schedule pressure,” but other causes like “carelessness, lack of education, poor processes, […] or basic incompetence” are also possible (p. 19). At the same time, “[m]ost agree that, sooner or later, technical debt will come due” (Buschmann, 2011). This, however, does not mean that technical debt cannot be used to balance different options, such as delivering fast while continuously paying “interest” on the debt, or repaying the debt “to clean up the mess”, or converting it by replacing a component with one that entails lower “interest” on the debt (Buschmann, 2011, p. 29).

In the following, I will draw upon existing conceptualizations of (technical) debt to respecify it for the case of the strategic management of ignorance in software development.

4. Hardcoding Robot Software: Four Ethnographic Vignettes

This section discusses four chronologically ordered ethnographic vignettes. The reporting technique used is inspired by that of LeCompte and Schensul’s (1999) “critical event vignettes” based on field notes, which “depict scenes that were turning points in the researcher’s understanding or that changed the direction of events in the field site” (p. 273). Each of the vignettes is followed by analytical commentary.

4.1. “Thou shalt not steal!” Hardcoding as Normative Bypass

The CPS paradigm was considered revolutionary in the company. A team of senior system and software architects developed a CPS reference architecture by closely following the Industrie 4.0 guideline. The concept presupposed a fractal decomposition as follows: a CPS was composed of one or several factories that could host several cyber-physical production centers having multiple cyber-physical production units. A CPS could thus consist of a few production units or several factories. One of the “revolutionary” principles was that the product would steer its production as it passed through different factories and units. Each of these would perform one or several production steps upon the product, as
described by a bill of process and a bill of materials. There would be algorithms that routed the product through a CPS as required by the bill of process.

First, the CPS concept was implemented in a simulation environment. In one project conducted by my research group, a simulated factory had over 90 units and was able to “produce” a virtual photo camera, among other things. The virtual factory resembled a computer game, in which each production unit occupied a little square on the screen. The product would travel from one square to another, each of which added one or two virtual parts to it. The software developers would often follow the product attentively through this cyber-physical maze. At one point, somebody noticed that one unit—a virtual robot arm—had taken a workpiece from another robot’s stack. “Look, the robot is stealing!”, the developer said. Other developers, including myself, gathered around his screen and watched and laughed nervously. Then, at the coffee machine, that robot was qualified as a thief, and we debated about how to prevent that kind of “emergent” behavior in the code. One possible solution was to associate each stack of pieces with a particular unit by introducing a notion of ownership into the system. “What if the robot is out of pieces? Would it be allowed to take from another one’s stack?” This could be implemented using another condition. Consequently, the stacks received “owner” ids pointing to designated units, and the rules preventing “stealing” without reason were coded into the algorithm using “if-then-else” logic.

As testing continued with several virtual products being commissioned for production at the same time, the workings of the system became increasingly byzantine. Why would a part go left in the maze instead of right, as the developers expected it? Or why would it disappear in one square and reemerge at the other end of the screen? One of the architects of the original fractal design observed that the agent-based, decentralized product routing algorithm continuously puzzled the developers as they tried to understand how the system made decisions. This reduced their productivity. As a result, the architect proposed to replace the agent-based, decentralized architecture with a less confusing, modular-hierarchical CPS approach. Yet, at that point in the project, it was difficult to back out of the paradigm because the Industrie 4.0 vision with its principles (decentralized production, the product steers its production, etc.) was compelling for managers and technicians alike. Besides, many problems could be fixed by adding assertions and exceptions (i.e., program structures that dealt with unexpected errors and actions). These structures, however, reduced the flexibility and autonomy of the decentralized production algorithm. At least this way, it was possible to develop early prototypes and demonstrators targeting managerial audiences.

After a few successful demonstrations of the CPS simulation environment, the demonstrators started moving from the computer screen to the laboratory.

The “stealing robot” episode suggests that, when puzzled by the behavior of experimental systems, software developers tend to hardcode common-sense norms and values into software. This is reminiscent of the way in which technology ethics scholars envision controlling artificial intelligence algorithms by promoting ethical and social order values in those algorithms through hardcoded procedures (Rychwalska and
Roszczyńska-Kurasińska, 2017) and safeguards (Leenes et al., 2017). In the studied case, however, hardcoding served a pragmatic rather than ethical purpose by allowing work to continue. The agent-based decentralized CPS architecture inspired by the Industrie 4.0 vision revealed its limitations quite early in the development process. Yet, the vision proved to be more powerful than the practical lessons learned during the implementation. In this context, given the tight project schedule, hardcoding allowed the developers to amend the system design impromptu, although some senior architects recognized the flaws of the original design and called for a fundamental redesign. The imperative of creating a simulated proof of concept for the smart factory within the allotted budget of time and money thus turned hardcoding into an accepted practice.

4.2. Producing and Inheriting Technical Debt

The first demonstrations with real hardware in the lab were successful because they were scripted to do a very specific thing. In this context, hardcoding helped to encode different demonstration scripts in a computer-executable way. In the source code, every move and action was encoded in the most rigid way, by numerically specifying all target positions of the robot’s gripper in advance and thus sticking to a thoroughly scripted demonstration. These demonstration scripts seem to be grounded in what Schulz-Schaefier and Meister (2017) refer to as prototype scenarios, that conflate a fictional and an empirical reality in a “negotiation arena” to embody a simple and fragmented future version of reality. What strikes about these demonstration scripts is that, although speculative and conflating, they are very implemented ad litteram in software with the help of hardcoding. This suggests that hardcodes are paradoxical in nature: they symbolize precision while embodying arbitrariness and fragility. This way, an imagined version of reality can be presented repeatedly in the same way to various audiences; while, at the same time, it can fall apart by changing a single constant in the prototype’s source code (which frequently happened during testing and integration, as the fourth vignette illustrates).

As an accepted practice, hardcoding provided the company with a solution to a complex, strategic problem; namely, that of legitimizing the CPS design despite the difficulties encountered during development. While ignoring the technical details of the prototypes, the project managers presented the hardcoded demonstrators as a proof of concept for the Industrie 4.0 vision. This proof ensured the continuation of the CPS research and development program. This solution, however, was a double-edged sword, as the next vignettes will show.

I was part of a team of about 12 software developers, industrial engineers, and project managers tasked with the development of a new demonstrator, the purpose of which was to show that the fully decentralized CPS paradigm also worked with real products in a real factory and with less hardcoding. The plan was to reuse the software developed for the previous lab demos. This approach seemed very complex to me in terms of the different technologies used. This complexity also arose from a desire to implement the “intelligence” of the system using one of the company’s existing software products (an industrial programming
environment) by integrating a popular robotic middleware, called Robot Operating System (ROS).

While the modules written using the company’s industrial programming environment in its proprietary language could be reused, none of the modules written for ROS was reusable. For one thing, all robot moves were hardcoded to implement the scenarios of previous demonstrations. For another, we found that the low-level design was flawed; the code looked like it was written long before the introduction of object orientation or any other modular design paradigm. For each production step, there was just one long function, composed of dozens of “if-then-else” blocks and calls to ROS’ path planning algorithm, wherein the exact target locations were hardcoded as in “moveTo(x = 123.54, y = 220.66, z = 65.345, ...).” This meant that the code could do one thing only and would not perform well in another context.

Where were the targets? What would the movement look like? It didn’t really matter at that point; that code was just a pile of technical debt. When the project manager tried to contact the original developer, the guy said he would answer our questions but would not work on a follow-up project. He seemed traumatized by previous experiences with developing CPS demonstrators. For us, this meant that we had to redesign and rewrite all the code for the new demonstrator with no prior knowledge of ROS—a notoriously difficult-to-use piece of open-source software.

I was assigned with designing a new structure for the ROS code, that was called “skill framework.” It provided several robot software skills, like transportation, assembling, screwing, etc. This framework could have been implemented in C++ or Python, both of which were supported by ROS. We opted for C++ because we still hoped that some of the code from the previous demonstrators could be reused. Now, we were stuck with C++. It required additional instrumentation compared to Python, it needed stricter compiler configurations and, in general, it was more cumbersome to work with. Besides, the robots we were planning to use (a KUKA and a Universal Robot—short UR) had their own programming environments, one based on Java and the other one on a proprietary scripting language called URScript. Thus, in total, four programming languages were used: a proprietary language, C++, Java, and URScript. The challenge seemed titanic, given that we only had about 10 months to go until the final demonstration. An intermediary demonstration was scheduled midway. This gave us about six months to implement the software, basically from scratch.

This episode illustrates how hardcoded software is being perceived as technical debt, requiring additional effort to “repay.” The team chose to use ROS to ensure continuity with previous projects. In theory, this would allow for reusing the knowledge and code developed in those projects. Yet, while demonstrations helped secure the budgets for follow-up projects, they also led to extensive hardcoding, which rendered the software unusable for other purposes.

The inherited source code provided a glimpse into the history of a project. By attempting to decipher hardcoded program logic, one gets a sense of the difficulties encountered by the creators of the code. In this case, the original developers appear to
have chosen to hardcode the demonstration script without even trying to produce a reusable component. The pressure of the deadlines and the overall atmosphere of that project seemingly determined them to take the path of least resistance. In this context, hardcoding provided a simple solution to possibly complex problems that could be applied when the issue of technical debt was not considered. Such situations can occur when the overall goal of a project is ambitious and when the implementation of a high number of features is more important than code quality.

In the case of CPS, the issue at stake was to prove the feasibility of a new production paradigm, from simulation to the shop floor. Had the demonstrations been unsuccessful, there might have been no follow-up projects at all. On this backward-dependent path, the lab demonstrators were just an intermediary step in the company’s research agenda. They emphasized the high-level algorithms implementing the Industrie 4.0 principles without focusing on low-level implementation details. Therefore, the hardcoded robot software only became a problem later when attempting to validate the paradigm in a real-world environment.

The focus of the demonstrators on high-level principles, like self-organization and autonomy, can be explained by looking into the philosophy of Industrie 4.0. The very concept of cyber-physical systems seems to be rooted in second-order cybernetics theory (Heylighen and Joslyn, 2001), which promotes the idea of self-organizing, self-referential systems. This idea is intrinsically present in the smart factory, where CPS and humans are interconnected and controlled by means of sensors and actuators. This interaction produces a continuous flow of data, going beyond the boundary of a single factory or country—thus making it controllable on a higher level. Code-level concerns did not to bother the architects of Industrie 4.0. Along with the machines, the workers and the developers were part of the self-organizing smart factory. If Industrie 4.0 were to work, then it would have to solve all problems by itself in a self-referential way.

4.3. From Headwork to Handwork: Hardcoding as Workaround

As demo day approached, we increasingly focused on integration and testing. The CPS demonstrator was installed in a factory corner. In total we had about 15 square meters. There were two desks in the back, two chairs, and one monitor. In the center of this improvised lab, the demonstrator rested on a hexagonal table. There were two robots — a larger KUKA assembly robot was placed in the center of the hexagon and the smaller UR was placed on one of the six triangles composing the hexagonal production cell (see figure 2). The UR was tasked with screwdriving one or two screws into a preassembled product. There was also a small conveyor belt meant to demonstrate the transportation skill. One had to imagine that this type of cell would be repeated many times in the factory and that assembly lines would transport parts and products from one cell to another.
Testing a feature required a series of laborious steps. First, the entire software stack had to be compiled and, considering that we used C++ as the main programming language, this took anywhere between one and three minutes. After compilation, the updated software was deployed on the robot’s control PC. Then, the robots needed to be reset and manually moved to their initial positions using their teach pendants (a hand-held device wired to the robot used to control and program the robot). The assembly parts and the unfinished products were placed in different workpiece holders (called jigs) and Kanban containers (a particular kind of Kanban container, as shown in the left hand part of Figure 2, where the workpieces slip down when removing the bottom one). The demonstrator illustrated 3-4 assembly steps performed on a preassembled printed circuit board. The parts being assembled were relatively small electronic components, like transistors and transformers. The precision challenge consisted in inserting the pins of these components into the holes on a printed circuit board. (In manufacturing, this is known as through-hole technology.) Then the central controller software had to be initialized along with all other units. Finally, clicking on a software button labeled “Start Production” would start the test.

The KUKA robot then picked and placed a transformer from a Kanban slide into an assembly jig on the conveyor. The conveyor belt started moving, producing the comforting sound of working things. The sensor LEDs turned on and off elegantly, indicating the presence or absence of a specific part in a port. This was the term we used for the precisely measured hand-over locations, where a product or part would be transferred from one machine to another. Our eyes followed the moves
of the big KUKA robot as it always seemed to choose a different path to go between the same two ports.

The funny paths were often a reason for somewhat nervous laughter. Only the developer who worked on the path planning was not amused at all when the robot suddenly hit something in its way or just stopped when it reached a safety limit, trembling like a stiff, uncanny being. Finally, the KUKA robot picked the transformer from the conveyor belt and tried to insert it into its designated position on the printed circuit board. In total, roughly 20-30 minutes were needed per test. For the whole program, in which two different products were “produced,” the process took even longer. I personally conducted at least 50 such tests.

The smaller UR robot, tasked to pick a screw from a feeder and screw it into one of the products, caused another colleague headaches. He programmed it using its hand-held controller to pick a screw from a feeder using a magnetic bit, then to go to another location, and—finally—to perform a screwing operation. It had taken two people three weeks to make this work, somehow, thanks to hardcoded logic. The project manager, however, insisted we use the same path planning library for the UR as for the KUKA robot to show that our software stack would be reusable with different types of robots in arbitrary assembly cells.

Whether the coordinates were hardcoded or dynamically calculated, nobody could tell just by watching the demonstrator perform. Therefore, trying to integrate the path planning library in UR’s codebase at this point seemed suicidal, since that which that required the integration and adaptation of an open-source ROS driver for the UR in a very short time. Nevertheless, it had to be done in order to demonstrate that all robots could communicate via the ROS middleware. When the path planning was finally integrated, we discovered that the open-source ROS driver had some issues, leading to jerky movements. Moreover, in the simulator, the robot seemed to be banging its “head” against the table from time to time. Given these manifestations, we consensually decided to postpone the integration of the path planning for the UR and to focus on more stringent tasks to ensure a successful demonstration. Consequently, the hardcoded version of the screwing operation, written in the robot’s native scripting language, was integrated into the system’s codebase and marked as technical debt.

The port measurements were an unexpected major challenge in the project. The entire system operated in reference to the KUKA robot’s coordinate system, originating in the center of the production cell. Therefore, the easiest approach was to use the robot as a port measurement tool—a purpose for which it was not designed. To measure a port, one had to drive the robot manually to a certain point, where the product was to be picked, placed, or handed over to another machine. Then one needed to carefully position the robot’s gripper fingers in the so-called grasping position and write down the coordinates of that position on a piece of paper. There were six coordinates for each port: the orientation given in Euler angles (i.e., pitch-yaw-rotation angles) and the Cartesian coordinates, which were both provided in the gripper’s coordinate system. After noting them down for each port position (there were about 15 positions in total), one had to convert them to the global coordinate system of the KUKA robot and then to quaternions.
These strange algebraic beasts were the preferred mathematical representation of orientation in ROS. The use of quaternions rendered the port measurements even more difficult because, whereas the Euler angles were somewhat intuitive to the engineers, quaternions consisted of four seemingly arbitrary sub-unitary numbers. Then, these quaternion values had to be written into a configuration file. Initially, we used a web tool to manually convert the Euler angles to quaternions until we noticed that the three decimals provided in the web computer’s result were not sufficiently precise. As it turned out, the measurement had to be accurate up to 8 decimals for the robot to correctly assemble the pieces.

Measuring one port took almost half an hour. If one kicked the hexagonal table by mistake or if something was dropped on it (which frequently happened because the robots sometimes dropped whatever they were carrying on the table when an error occurred), one had to start over. Before the integration phase, in the developer meetings, the inconspicuous phrase “someone has to measure the ports” popped up every now and then. We estimated this task would take two hours in total. In the end, the ports needed remeasuring throughout the four-week integration phase.

One week before the first scheduled demo, the situation was critical. At that point, the demonstrator consisted only of the sum of its parts. While each of the parts appeared to do the expected thing, when we tried to put them together, the system failed. The robot software would crash because of the path planning algorithm. The controller software would crash because of God knows what. One needed at least half an hour to figure out the cause. Most integration tests never finished either because software errors occurred or because the robot was not able to pick, place, or insert parts correctly. We realized that in the short time we had left, there was no way for the robot to insert a transformer’s pins into the holes on the printed circuit board. This task required submillimeter precision. This level of precision could not be achieved without some kind of automated position correction system. Therefore, we decided to go with plan B, which required us to cut off the “legs” of the transformer to make the job easier for the robot.

The overall situation revealed that the CPS paradigm had a blind spot. Even if we spent enough time and effort to make the path planning flexibly adaptable to any kind of assembly station layout (which would presumably have taken several months, if not years), one could not simply assume that the coordinates of any object in that environment would match the coordinates of its software representation. The physical environment had to be perfectly fixed and measured. Otherwise, there would be constant mismatches. Therefore, the system seemed extremely fragile to me. If you put a part one-tenth of a millimeter away from its ideal position in the port (which we thought we had measured correctly), the robot would fail to assemble the part. We had endless discussions about how a 3D camera with automatic position and orientation correction would solve these issues. In these discussions, we envisioned potential solutions involving machine learning, computer vision, and—more generally—intelligent software and algorithms that might eventually solve our problems with the cyber-physical interfaces. Yet, for the time being, we were stuck in what seemed to be nature’s
bureaucracy—a complicated entanglement of requirements and unknowns that could only be sorted out through experimentation.

This episode suggests that, although the team managed to reduce the technical debt inherited from another project by rewriting large parts of the ROS codebase, the challenge of integration testing introduced new debt into the system in the form of different types of hardcoding. Whereas the UR program for screwing was hardcoded in a similar way as the older lab demonstrators, the port measurements required a different kind of hardcoding at the interface between the CPS and its physical environment. Presumably, the architects of the port concept had not expected for these measurements to require continuous updating. This represented a pitfall in the design of the system. The necessity to continually update the hardcoded port positions to match the physically anchored (and thus hardcoded) positions of the hardware components illustrates how our coding practices transitioned from “head-work” to “hand-work” (Ensmenger, 2010) under the auspices of an ignorant system design.

When this became clear to us, there was no alternative solution in sight. An “intelligent” camera system that could solve all of our problems did not exist at that time. (We later received a camera that promised to do exactly that. Yet, we failed to integrate it because of major software issues that could only be solved by the vendor.) The hardcoded port positions thus appeared to be symptomatic of a kind of “not yet knowledge” reflected in the repetitive hand-work of measuring them. In this context, to paraphrase Merton (1987), the ports helped to “specify” that ignorance by precisely emphasizing what was needed to build the CPS and bring the Industrie 4.0 vision to life.

Technical debt may well remain undetectable by actors outside the development team, provided that the system’s behavior conforms to the specification. In general, its reduction or elimination is merely a question of technical and human resources (Buschmann, 2011). But in this instance, the kind of technical debt induced by the hardcoded port positions was arguably epistemic in nature: it could neither be hidden from the stakeholders for too long nor eliminated through technical means, even if the developers had continued to work on it indefinitely. While the team members knew how to produce a working demonstrator, they were unable to extend that knowledge to other assembly stations without remeasuring them again and again. This epistemic debt, however, did not prevent the project manager from presenting a working CPS prototype to a wider audience.

Combined with the hardcoded port positions, the missing transistor pins helped to soften the cyber-physical interface between the prototype and the products it was supposed to assemble. This showed that the complexity of an automated factory resides in the myriad of interfaces between the machines and the products they handle. A closer look at the products of manual assembly reveals how their design seeks to minimize the use of materials and space on the printed circuit board. Such a design allows for few if any spatial and material affordances that would, hypothetically, allow a metallic gripper attached to a 25 kg industrial robot arm to achieve the performances of the human hand. During the project, the developers became aware of this complexity and learned how to deal with it in a pragmatic way. However, much of that knowledge was invested in hardcoding practices.

As Merton notes, “new knowledge leads some scientists ... to become aware of other, newly identified aspects of the phenomena. There then develops a succession of
specified ignorance” (1987, p. 8). In this sense, the hardcoded port positions and the missing transistor pins specified an essential research question pertaining to the CPS paradigm: How can cyber-physical interfaces be represented in software to enable precise manipulation of small workpieces by robots?

Due to the authoritative style in which they have been initially promoted by the German Academy of Science and Engineering and then by many experts in the field, the Industrie 4.0 principles enjoyed credibility without proof. Some of the principles of the smart factory were taken for granted in the industrial automation community, notably its capacity to solve problems by itself. One of the ways in which speculative technical principles are being promoted as reliable knowledge in policy reports and recommendations is through "the conversion of ethos into logos," as Carolyn Miller (2001) puts it. Through such conversions, experts mitigate the lack of data and thus weak scientific evidence with their subjective assessments. Reputable experts resort to this technique when they cannot provide sound explanations and/or guidelines for solving issues that are anchored in technical ignorance. As a result, the production of reliable knowledge is deferred to others, who are left to confirm or infirm the top experts' hypotheses through trial and error.

4.4. Enacting the Smart Factory

The demonstrator is regularly presented to high-ranking managers and external visitors. To get to it, visitors need to pass through a long corridor animated by workers on the right side and engineering offices on the left. The improvised CPS lab is enclosed by steel bars. A sign says: “Authorized personnel only.” Beneath the hexagonal table on which the machines are fixed, there are countless cables and a few computer racks. All machines stand inert while CPS experts, dressed in white overalls, gravitate around the robots and place various parts at different locations on the table. A moment of stillness sets in, marking the transition to the ritual frame of the demonstration.

The project manager conducts the demonstration. He begins with a short introduction supported by Powerpoint slides showing how the current project fits into the company’s CPS roadmap and into the overall Industrie 4.0 vision. The manager lays out hard facts and data to win over the audience from the very beginning. Then, the production process starts while the presenter continues to explain what happens behind the scenes in more or less detail, depending on the audience. The explanations are structured around terms like “the product steers its own production”, “digital twins”, “lot size one”, “flexibility”, “autonomy”, “self-organization” and “self-optimization”. He combines them with details about the technical implementation of these desiderata. By the time one of the robots picks up the first workpiece, the members of the audience are already absorbed by the demonstrator’s performance. The presenter’s speech serves as a binding medium for ideology and technology, while the robots assemble the same products as the workers behind the audience. The factory work does not stop during demonstrations. Hammering, cutting, and drilling sounds can always be heard, rendering the experience more authentic.
The highlight of the demonstration is reached when the KUKA robot picks, then places a product within a constrained space. This requires the automated path planning to do its magic. Like a contortionist, the robot moves in unexpected ways while avoiding various obstacles to the amazement of the audience. “Why is it taking such a path? Is it optimal? Why not just take the same path every time?”, members of the audience ask. The presenter seizes this opportunity to provide some technical details about automatic path planning. Automated path planning, the presented explains, is required to support a flexible production and thus to fulfill the “lot size one” desideratum of Industrie 4.0. Lot size one means that the smart factory must be capable of producing small lots of a product variant to cope with increasing flexibility demands from customers. To the question of why the robots move so slowly, the presenter responds that the system is designed for flexibility, not productivity.

With most members of the audience seemingly convinced of the demonstrator’s ability to assemble real products, the presenter now switches to a more confident and relaxed mode in which the goal is to advertise the expertise of the development team and their availability for other projects. In doing so, he emphasizes the unique opportunity provided by Industrie 4.0 to make a new attempt at automating small part assembly. The developers, whose passive role in the performance requires them to hold the robot controllers, stop and restart the system if something goes wrong, now become central to the presenter’s discourse about expertise and opportunity. Animated by the questions of the audience, the presenter puts the demonstrator in the broader context of a series of research projects. Much expertise has been built up since the publication of the Industrie 4.0 vision in 2011. Now the team is capable of building CPS that assemble real products in real factories. Thanks to the knowledge gained in multiple projects, the company is on the right track to take it up with the competition. These messages are blended with interesting low-level technical details, such as the technology used to represent “digital twins” (i.e. virtual models and simulations of parts and machines) and the semantics of production commands.

Finally, the presenter provides a glimpse into potential future research projects in an attempt to address a few stringent questions. How can highly precise and flexible assembly capabilities be achieved with robots? How can CPS be rendered productive? How can production costs be reduced using information technologies? The solution to these problems is deferred to machine learning and other artificial intelligence technologies. Using these technologies, it will be possible to optimize and improve precision, productivity, energy use, production workflows and much more. 3D cameras will enable the automatic creation of digital twins from physical parts. Self-optimizing CPS will leverage the potential of machine learning techniques to achieve higher precision and productivity than humans. Yet, to achieve this, more research is needed.

An essential part of the discourse underlying the demonstration also addresses the “specified ignorance” (Merton, 1987) unveiled in the project. The “yet-to-be-knowns” are touched upon by the different questions of the audience as well as by the presenter himself. For example, some members of the audience noticed that the robots moved very slowly. Faster robot movements would determine the table to vibrate and...
would thus render the port measurements incorrect. In answering these questions, the presenter uses layered justifications. One such justification is that CPS are designed for flexibility, not productivity. Yet, this argument is not satisfactory for the representatives of the factory, who are part of the audience and contributed to the development, and thus know the demonstrator’s blind spots very well. Therefore, towards the end of the demonstration, the presenter preempts the doubts of the audience by pointing exactly to the deficits of the prototype. This problem is then described as a research challenge whose future solving involves the use of 3D cameras and machine learning technologies.

The presenter’s reference to machine learning is unsurprising, considering how the developers went about remeasuring port positions over and over again. This was a task of repetition with small variations, as it is common when preparing inputs for machine learning algorithms. Using a large set of annotated input data, machine learning algorithms produce executable models for solving a very specific problem. With labeled input data being its only source of “knowledge,” machine learning may be regarded as a kind of sophisticated hardcoding technique, which does not eliminate the epistemic debt of a system but only converts it into other forms. As the presenter suggests, the integration of machine learning technologies into industrial software is a long-standing expectation. The management of different forms of epistemic debt requires programmers to adapt to the new realities of their profession by acquiring new kinds of knowledge and expertise. These new realities might entail more repetitive and manual work than traditional programming.

5. Conclusion

The present case study focused on the development of CPS demonstrators in a large tech corporation. I emphasized the ways in which industrial robot programmers use hardcoding to cope with epistemic challenges and to specify ignorance in the development of CPS software for the “smart” factory. As Merton notes, “[t]he specification of ignorance amounts to problem-finding as a prelude to problem-solving” (1987, p. 10).

When the CPS software was being developed for a simulation environment and in the laboratory, technical debt accumulated fast, and “interest” had to be paid continuously. This strategy was successful because the demonstrators could be presented on schedule, and the research program received more funding. This helped to reduce that debt as development became more and more concerned with operationalizing the CPS paradigm in real factories. Here, the elimination of technical debt in the form of hardcoding became increasingly difficult as it became a symptom of the “yet-to-be-knowns,” specifically of our ignorance about how to achieve higher precision using the available technical means. At that point, the remaining technical debt was epistemic in nature. During demonstrations, it was described as a research challenge, and its “repayment” was deferred to an uncertain future in which machine learning technologies would allegedly provide solutions to problems, such as assembly precision.

Hardcoding and hardcodes can take multiple forms: as normative bypass; as executable demonstration script; as technical, then epistemic debt; and as specified ignorance in stakeholder demonstrations. These forms and roles need to be interpreted within the
frame of black boxing (Merz, 1999) that renders important aspects of a software invisible to the people who witness its performance. By effectively concealing the workings of technology in the making while compellingly emulating an imagined performance of it, hardcoding seems to provide technology creators with leeway in negotiating research funds. This comes at a cost since hardcoded "yet to be knowns" influence research agendas in ways that are contingent on individual projects and organizations, thus contributing to path dependency. Furthermore, when hardcoding becomes an expression of epistemic rather than technical debt—that is, of unsolved problems projected to be solvable in a rhetorical near future—it urges technology creators to speculate about the likelihood, potential, and provenance of such solutions, thus pushing them farther down the path of uncertainty. In the event that machine learning technologies will not live up to the promises and expectations put forth by the Industrie 4.0 vision, CPS are likely to automate the repetitive, manual tasks of factory workers only to generate more repetitive “hand and head-work” for robot programmers, thus doing a disservice to both professions.

BIBLIOGRAPHY


Kagermann Henning, Johannes Helbig, Ariane Hellinger & Wolfgang Wahlster (eds.) (2013). Umsetzungsempfehlungen für das Zukunftprojekt Industrie 4.0: Deutschlands Zukunft als Produktionsstandort sichern; Abschlussbericht des Arbeitskreises Industrie 4.0, Forschungsunion, Frankfurt a. M.


APPENDIXES

Appendix: Hardcoding in Practice

The following code examples illustrate different levels of hardcoding by the example of a function for computing the area of a circle written in a C-style programming language.

The first version of the areaOfCircle function contains a fully hard-coded formula of the circle area. This function can only compute the area of a circle with a radius equal to 30
with a high approximation error, due to the truncated value assumed for the universal constant PI.

The second version uses a parameter for the radius and is thus able to compute the radius of any circle with a high approximation error. The approximation error results from the use of a hard-coded value of PI, which lacks a sufficient number of decimals to reduce the approximation error.

The third version of the function is soft-coded because the value of PI is parameterized by reference to the Math library. This library provides a PI constant with a high number of decimal values. Using Math.PI instead of the truncated value of 3.14 reduces the approximation error of the function.

```c
float areaOfCircle_version1()
{
    return 3.14 * 30 * 30;
}
float areaOfCircle_version2(int radius)
{
    return 3.14 * radius * radius;
}
float areaOfCircle_version3(int radius)
{
    return Math.PI * radius * radius;
}
```

The use of the exemplified function is also susceptible to hardcoding. For example, the following use is hardcoded and can therefore only compute the area of a circle with a radius equal to 30:

```c
float area = areaOfCircle_version3(30);
```

To allow for more flexibility, the code above can be rewritten as:

```c
int radius = askUser();
float area = areaOfCircle_version3(radius);
```

This way, the program asks the user to provide the value of the radius, thus becoming more versatile. This example arguably shows how, through hardcoding, the potential versatility of software is constrained. This constriction may or may not be voluntary. If it is not, it may be a symptom of non-knowledge.

NOTES

1. I would like to thank the coordinators of this special issue, Gabriel Alcaras, Antoine Larribeau, and Manuel Boutet for their consideration and support. My special thanks go to Gabriel Alcaras and the two anonymous reviewers for their valuable comments and suggestions that helped to significantly improve the manuscript.

2. Ionescu and Merz (2018) draw on the same empirical material as this paper to study demonstrators as models in the context of the smart factory.

3. Qualitative research in software engineering is usually conducted by software engineers with a sensibility for the social aspects of their profession, who use action research, ethnography, case studies, and grounded theory to study software teams and organizations (Dybå, 2011). The aim of many of these researchers is not only to understand but also to improve software practices.

4. Amann and Hirschauer (1997) describe a participant observer’s difficulties in coping with her subjective interpretation of the empirical material and suggests means to mitigate that bias.
5. Schmidt (2008) seminal study of agile software development in situ looks into how programmers and managers produce software in the setting of the traditional white-collar office.

6. A middleware is a collection of software components, which provide services and communication models on top of those provided by the operating system.

ABSTRACTS

“Hardcoding” is the practice of starkly simplifying program logic to the end of producing software more effectively while sacrificing its versatility and generalizability. Drawing on concepts from science and technology studies and software engineering, this paper develops a case study of hardcoding for the “smart” factory—a kind of factory that leverages the newest information technologies to boost productivity, flexibility, and global competitiveness in existing manufacturing companies. The analysis of the ethno-graphic material collected through participant observation of a smart factory project in a large technology company suggests that hardcoding serves multiple purposes, which amount to the “specification of ignorance” in software development. In this sense, hardcodes reflect the “yet-to-be-knowns” in a software’s source code while conveying a project’s history of challenges and compromises.

INDEX

Mots-clés: technography, technical debt, non-knowledge, automation, software, smart factory

AUTHOR

TUDOR B. IONESCU

Vienna University of Technology, tudor.ionescu@tuwien.ac.at, Theresianumgasse 27, 1040-Vienna, Austria