media/rep/

Repositorium für die Medienwissenschaft

Paula Bialski

Code Review as Communication: The Case of Corporate Software Developers

2019

https://doi.org/10.25969/mediarep/12006

Veröffentlichungsversion / published version
Sammelbandbeitrag / collection article

Empfohlene Zitierung / Suggested Citation:
Bialski, Paula: Code Review as Communication: The Case of Corporate Software Developers. In: Paula Bialski, Finn Brunton, Mercedes Bunz (Hg.): *Communication.* Lüneburg: University of Minnesota Press 2019, S. 93–111. DOI: https://doi.org/10.25969/mediarep/12006.

Nutzungsbedingungen:
Dieser Text wird unter einer Creative Commons BY-NC 4.0 Lizenz zur Verfügung gestellt. Nähere Auskünfte zu dieser Lizenz finden Sie hier:
https://creativecommons.org/licenses/by-nc/4.0

Terms of use:
This document is made available under a creative commons BY-NC 4.0 License. For more information see:
https://creativecommons.org/licenses/by-nc/4.0

DFG Deutsche Forschungsgemeinschaft

Philipps Universität Marburg

[3]

Code Review as Communication: The Case of Corporate Software Developers

Paula Bialski

> Communication is a rich tangle of intellectual and cultural strands that encodes our time's confrontations with itself. To understand communication is to understand much more.
>
> —John Durham Peters

In early August 2017, a Google employee published an internal memo titled "Google's Ideological Echo Chamber." Most of us recall the content of the memo as well as the buzz that it stirred in tech circles in Silicon Valley and beyond: the memo outlined the reasons that Google's diversity initiative was fundamentally wrong. One section in particular struck me—an ethnographer of corporate software developers—as particularly misleading: "as Google was trying to make software engineering more people-oriented with pair programming and more collaboration," Google's initiative was fundamentally limited by "how people-oriented certain roles and Google can be" (Motherboard, August 8, 2017). One of the memo's

messages was to "deemphasize empathy" in tech companies. As far as software developers are concerned, who needs empathy anyway when building technical systems?

While I acknowledge that this memo stirred up a lengthy debate around gender biases in the tech industry, about what women are good and not good at, and about the general differences between men and women in the first place, I will, for the purpose of staying on topic, leave out this debate and focus on one of the article's claims: that communication and empathy should not be stressed as assets among tech workers.

Communication in digital cultures is multifaceted. We know that it can exist between humans, mediated by digital technologies. It can occur between a human and machines (the way we record our voices or type into a screen). It can exist machine to machine (the way Finn Brunton so eloquently underlined in chapter 1 of this volume).

What the author of the Google memo overlooked was that software is perhaps one of the most complex examples where digital media and human sociality indeed intertwine. Human empathy in a team meeting sits alongside merging pieces of code written by separate people. In this chapter, I would like to zoom in on the processes of a large-scale software development corporation to uncover the way in which human–machine cohabitation and co-creation call on various modes of communication. In previous chapters, Brunton explored communication and digital technology through questioning the alienlike dialogue between technical entities—focusing more on communication between machines—and Bunz looked at the way in which these technical entities are designed to address us, the user, and what kind of effect this type of design (childlike and patronizing) has on us. This chapter will draw you, the reader, into an ethnographic field to see the way communication between machines, between humans and machines, and between humans building machines plays out. Doing so, I hope, will give you a grounded example of the multifaceted nature of communication in digital cultures.

Charlie

I met Charlie about two years ago during one of Noah's film nights in Berlin's Neukolln district. We were all sitting on Noah's living room couch and discussing their software company—a place where I later started conducting my long-term fieldwork project. The company focused on building mapping and navigation software and had approximately six thousand employees. One thousand of them were software developers of all shapes and sizes. I worked with Charlie on the "navigation" team, which worked on, among other things, how to find the best route from A to B (which is a much tougher problem than it sounds).

I just assumed Charlie was a developer. And he played the part. He spoke knowledgeably about code, bugs, breakdowns, and updates, and I somehow just jumped to the conclusion that he knew a lot about programming. It was only in the recent weeks, when I started working in his department, that I noticed something fishy. During a product owner[1] meeting, he was there. Sitting at the table. He was very engaged in the meeting, very vocal, very un-developer-like. I first figured that he was a lead developer, or somebody just keen enough to sit at the product owner meetings, until I realized that he was, in fact, a product owner. This mix-up—the fact that I didn't know what an employee was doing exactly—was quite a common mix-up, even for any regular employee. This was due to the fluidity of both his role (he himself didn't know if he was a product manager, product owner, error manager, or a number of other possible positions) and the amount of technical knowledge he had (his knowledge of the software system they were building was somewhere in the gray zone of being technical and not-so-technical-at-all).

Charlie had the poshest of English accents. This accent would surprise me every time he opened his mouth. Perhaps it was because this software development company hires a mishmash of nationalities, creating a sort of Globish clang around the office. To overhear a real native English speaker was quite odd. And the type of English

he spoke was a Merry Poppins sort of English, using sentences like "I'm just taking care of some bits and bobs" and "we're far away from everything being tickety boo." Despite the fact that he looked cheery all the time, Charlie always explained that he was stressed and that there was a lot of work on his shoulders.

Charlie was also very understanding of my role as an ethnographer and spoke to me softly and with true interest in what I do. That afternoon, after I asked him a few times about the "fire" he was trying to put out, he offered to show me what he was working on. He grabbed a coffee cup and his laptop, and we walked over to the common area near the staircase, with nice wooden picnic tables. He opened up his laptop to show a screen of software called Gerrit—a free, open source, web-based code collaboration tool.[2] Gerrit displayed a list of *merges*—meaning updates or additions to the code base. This list included the type of update, who updated it, and the status of the update (whether it passed or failed). During merging, the code would get automatically tested, and various problems could arise. I learned that it was his job as error manager (an additional job he was assigned to do, on top of his product owner job) to monitor these updates, especially during the last week of production. That day was *feature complete day*—which meant that all features or changes to the new software version should be finished and merged into the code base. Merging doesn't happen seamlessly. Charlie anticipated that something wrong would happen and, during days like these, always kept his computer open and monitored the progress of each merge.

Looking over his laptop, he started pointing at a list of bars colored red and light blue, delineating the status of the merge process. Each of these bars was one "piece" of code belonging to one developer who pushed this code into the rest of the build. The bars would change color depending on whether the code merged, or worked, with the rest of the code. I looked at one of the bars that turned red and said "merge error," and I asked him, "So is it kind of like putting two pieces of a puzzle together, but one person chopped off one arm of the puzzle piece, so it doesn't fit into the other piece as it was intended to?"

Charlie explained, "You have to imagine that there is a Google Doc that two people are working on. And at the same time, one editor makes changes to what the other editor was working on. When they merge these two documents, their changes would conflict."

Charlie explained that this is called a *merge conflict*—when a change, or *job,* that one developer merges influences another line of code that another developer merged. We were both looking intently at the screen at a long list of around fifteen jobs, some of them with "merge conflict" marked next to them. Throughout the next fifteen minutes of sitting there together, I could see him monitoring this screen intently. He explained that each time a developer wants to merge what he or she was working on into the main repository, it would take around forty minutes for the server to test the merged code. In monitoring this screen, Charlie was waiting for a developer, and the server, and the code base to communicate something to the rest of the team (including him) regarding the status of the code. And there were also other conflicts that could arise aside from merge conflicts.

I noticed myself starting to get a small sense of how programmers worked together. And the way they worked together was through various forms of communication, avoidance of communication, or analyzing communication. The term *communication* was constantly floating around the office. If we take Charlie as an example, just these few minutes of interacting with him displayed all the verbal, nonverbal, machine-based, human–machine, or team-based communication that takes place in a software development setting. Charlie's type of English compared to the English of his colleagues posed a communication issue. The way Charlie's client expressed what they needed Charlie's feature to do for the user posed a communication issue. The way one developer's code communicated to the code base posed a communication issue. The various ways two developers worked on the same piece of software posed a communication issue.

Moments of Unfolding

The question of how digital media technology affects the social and how the social affects digital media technologies can best be studied in such hard-to-grasp, always slippery, always changing moments of unfolding. My goal with this initial fieldwork anecdote was to briefly introduce you to the highly complex feat of communication that software development entails. Mixed in between servers; shyness; Globish; and software standards, testing standards, and customer expectations are people collaborating on the same project. This collaboration consists of a deep intermingling of human and machine. As Mercedes Bunz pointed out in her chapter in this volume, digital communication needed to be conceptualized on more than one layer, because software operates always with more than one interface. These interfaces are between machine and machine and between machine and user. As I will point out, there is also another layer, that between the writers of code (software developers) and other software developers. The overarching goal in this chapter is to speak to the issue of communication in digital cultures, by providing a rather descriptive map of the forms of communication that occur in corporate programming environments. In doing so, I aim to explain exactly how software development is organized as a practice and how communication fits into this practice. This communication that I will unfold includes a number of objects and people. This communication takes place

1. machine to machine
2. programmer to machine
3. programmer to organization
4. programmer to programmer

As part of a constant assemblage of human and machine, these three sides—organization, producers, and the material resistance of software—interact with one another in iterative processes, shaping together digital media, which are never not just filters but, in John Durham Peters's (2015, 2) words, "vehicles that carry and communicate meaning." The methodological challenge is then in

understanding how all these pieces communicate with each other and to observe such relations in moments of their unfolding.

In this chapter, I mainly draw on the ethnographic data I collected during my ongoing research of corporate software developers. In August 2016, I began an organizational ethnography at a large, six-thousand-employee Berlin-based software company, BerlinTech. At the time this chapter was published, I had spent more than eighty working days immersed in this company—attending meetings, observing the developers develop software, and holding both formal and informal interviews in the office kitchens, over lunch, or during long walks to and from work. My general focus within my research is to uncover technologically mediated sociality. I frame my research within cultural, social and media theory, science and technology studies, software studies, infrastructure studies, and the ethnography of software developers.

As I mentioned in a previous paragraph, this company makes key mapping infrastructure for third parties such as social networking sites or various apps in need of embedded maps. I spent two months among front-end app developers who built the company's "consumer" product—meaning the app that users download for use on their smartphones and desktops—and am currently situated among the back-end developers who build in-car navigation systems and maintain the routing infrastructure. The larger motivation behind my ethnographic project stems out of the belief that studying the mundane, quite "average" software developer brings us closer to the way in which our digital infrastructures work. To study everyday software developers—the masses of engineers who build the infrastructures that power our mobility, our sociality, our consumption (the list goes on)—is to study of how communication between humans and machines functions, and is changing, in our everyday, digitally networked lives.

In this chapter, I attempt to unpack how developers communicate with one another, and how machines communicate to developers, as well as the imperfect tools (such as the code review software)

that are invented to assist the communication between machines (code to code).

This research aims for (1) an ethnographically rich understanding of digital sociality that (2) focuses on moments of unfolding; (3) is based on a trimodal way of communicating between developers, the code they build, and the software they use to communicate with; and (4) last, but not least, looks at the interrelations between these three sides. I am aware, of course, that my field has other sites where communication takes place, especially between designers, management, and users. Programmers use a variety of collaborative management tools that are designed specifically for developers themselves (such as code review systems and workflow systems) or for the labor force at large (such as electronic calendars, intranet systems, and online spreadsheets). For the sake of this topic, I narrow down our case study of developer communication to one tool, Gerrit, that strictly relates to a practice called *code review.*

Despite ethnographies that look at the intricate ways in which programmers collaborate, an ethnographic mapping of programming practices in corporations in relation to the notion of communication has not been done. Many of these topics have also been addressed extensively in software engineering literature, but not from an ethnographic angle.

Describing Gerrit

After two months of working in the field, my field diaries left me with lingering questions. The luck of my field was that my informants weren't an isolated tribe, cut off from the rest of the world, but were software developers, who, as you could imagine, were often sitting in front of their computers, often with their various chat systems (Skype, Spark, or Slack) open, instantly accessible. This provided me with a sort of never-ending feeling of field access—as if I had a lifeline to my field that hardly ended after I left the office.

That being said, this chapter is made up of a variety of pieces of data I collected throughout a two-year period: a number of field notes before I entered the field, notes collected during my fieldwork, and more recent online exchanges with my informants over various chat applications.

The conversation between Charlie and me illustrated the basics of a code review system: it is based on a culture of peer review. The team of one hundred developers I currently research creates routing infrastructure for navigation systems (so how the fastest or most optimal route from A to B is calculated). These developers are then split up into subteams. The team I directly sit with is the electric vehicle (EV) team, comprising five developers. They collectively have to solve a routing problem typical for EVs: how to optimize a driving route for a car that has to secure charging stations every few hours? Before their project is finalized and is implemented into an in-vehicle system, their work is split into small subtasks. Before the project starts (the project's *sprint period*), the group of developers sit together in a room with their manager, or *product owner,* and define the subtasks that need to be done to complete the project. These subtasks are called *tickets.* Then, the next day, each developer takes a ticket and starts working on it. A ticket, in the case of the EV team, could be, for example, to match the library for EV charging stations to a route library. Maria would then take this ticket, work on it, and then upload it or "push" it to the Gerrit code review system for review. Her colleagues would then give her a score between +2 and −2 (the exact meaning of which I shall explain later).

Noah, one of my main informants, explained that the main purpose of the code review[3] was to collectively monitor if each line of code that a developer tries to upload into his team's main repository actually fits, and will work, with the code that the other team members have created. There are two main threads of communication at play here: the communication between developers, both offline (e.g., developers often peek over their desktop screens and yell out to a colleague, "Can you review my code please?" or "Hey,

why did you give me a −1!") and via their code review software (i.e., in the form of a code review), and the communication that lies between the technical system itself, for example, Noah's lines of code have to speak to another developer's lines of code, and these collections of lines have to speak to one another, so to speak, when running within the entire system. While I will get into more detail in a later section, I just want to underline that in this process, communication in the right way, using the right syntax, for example, is highly necessary.

After being a year in this field (with only latent knowledge of how software is built), I have come to the understanding that code review systems are an inherent part of a corporate software environment—a crucial part of a sort of production pipeline. It is necessary because a software product has to be shipped to its users within a certain time frame for the software company to remain competitive on the market, and that shipment has to actually work, without major bugs or breakdowns. Large-scale corporate software environments, made up of teams of dozens of developers, also need core review systems to make collaboration (and communication) with a large number of software updates easier. Michael, a developer who used to work at BerlinTech and who has since moved to a small start-up company, explained that the code review system at BerlinTech is a good example of how a big company deals with the review process. After moving jobs, Michael and his team of two to three people don't have a review process, but rather, the team communicates with one another by talking before starting their job, agreeing on something, and cooperating throughout the development process. Merging a change into the whole system becomes more of a formality in his case, rather than a necessity.

In noncorporate contexts, where software developers work on a small team without a strict deadline, they might use an informal code review system—giving each other feedback much like a band of musicians would give each other tips and tricks on how to make a song better. But a code review system in a corporate, large-scale

software project environment is a formalized, software-based assembly line. Additionally, the code review system is a way of monitoring the code that goes into building a system but also becomes a surveillance tool for upper management to monitor what type of work the programmers are actually doing. And as I shall explain in a later section of this chapter, it also becomes a sousveillance tool for programmers to monitor other programmers.

At BerlinTech, Gerrit is only one of many code review systems in use among software developers. The company itself has more than one thousand developers, and communication between developers happens in a number of ways, including through code review systems, but also in face-to-face, daily, fifteen-minute standup meetings; at lunchtime; at other collaboration meetings; and via all sorts of internal (meaning set up by the company for office use only) and external (meaning set up by team members for office use and other issues) platforms. Some (and this list is in no way exhaustive) include the company's internal chat system, called Yammer (which is limited for social interaction), which is the company's "official" internal communication tool for all work-related issues, and collaboration software like Cisco Spark, Slack, or even Skype chat. Each team, depending on what they are building, has a different type of communication tool, and even a different type of code review system. Keeping track of these channels of communication was a dizzying and oftentimes frustrating task for the developers with whom I worked. Some collaboration tools emerge bottom-up, initiated by the collaborators themselves, and some collaboration tools were initiated top-down, by the management of the company. While the ways in which sociality emerges in these meetings—in specific working groups or on each individual platform, with their specificities and power politics—are relevant to the theme of communication here, I will refrain from addressing them for the sake of keeping a more narrow focus on code review tools and the communication inherent within them.

Returning to Gerrit, this is a code review system that is free and open source, originally authored by Google—and it is not the only

system used at BerlinTech. As Noah explained, the reason many different teams use different code review systems is "mostly because not everybody completed the transition to Gerrit. Some people prefer other ways, because their project is small, or they're not aware of Gerrit, or they just don't like it" (field notes, June 2017).

Simon, one team leader, explained to me, "Gerrit is a key part of our culture. If a developer has a piece of code, he uploads it to Gerrit. You collaborate together to make one commit happen. This is not like competition for making code. It is trying to work together. Trying to transfer information and knowledge" (field notes, August 2016).

Sebastian, another iOS developer (building the application for iPhones), uses the metaphor of a tree to explain how code review works: "It's like a tree and every coder adds another branch to that tree. In order to merge their changes (branches), there has to be certain tests done. Only after these tests can the branches be really incorporated as part of this tree" (field notes, August 2016).

Using Sebastian's metaphor, when a programmer adds a "branch" to Gerrit, it is visible to other developers, and the code waits for at least two developers—plus an automated bot—to approve the code. At BerlinTech's front-end team, developers were encouraged to look at an incoming review every one to two hours, although one developer informed me that this rule "often didn't happen anyways, but a review of your work did take place every twenty-four hours" (field notes, June 2017). A review in the Gerrit system has five variables:

−2: Do not submit

−1: I would prefer that you didn't submit this

0: No score

+1: Looks good to me, but someone else must approve

+2: Looks good to me, approved

Michael, a web developer, half-jokingly once confessed that on Fridays, when he feels like leaving work and running off for a beer, he would quickly go through the code review system and just add "+2, +2, +2" to all the tickets waiting to be reviewed (field notes, August 2016). How much of this is actually true is a mystery, but it alludes to the way in which variables such as fatigue, the weather, the time of day, and the relationships between the developers themselves factor into their ratings.

Communication Models

At this point you, the reader, are perhaps beginning to understand the general purpose of a code review system and some basics of how Gerrit functions. If we unpack how communication works within this system, we can start to see three communication models at play. First, the code review system mediates communication between people, in this case, programmers working on code. This type of model of communication is between two humans and is dialogic and interactive. In the case of Gerrit, developers review one another based on a culture of reciprocity: if somebody has reviewed your code, you are obliged to review her code in return. If a developer requests a code review and then, upon gaining a review, refuses to reciprocate and review code in the future, this behavior is viewed as antisocial. Moreover, code review is also done in conversation. For example, a developer might write, "To be honest, this whole solution looks a bit . . . hacky. Could you please describe how it works and why this is a correct solution?"[4] This developer will not approve the "committer's" (programmer submitting a code update/solution to be reviewed) code until he gains an answer to his question. Code review is a continuous back-and-forth between committer and reviewer and must reach a level of mutual understanding.

Additionally, it might not come as a surprise that interaction between two developers is often easier through a system like Gerrit, minimizing the social pressure of face-to-face interaction. Michal recalled a situation where "there was one review that had

20+ patches [fixes] and a lot of discussion in the comments just between two guys. And as I see it, it would be way easier for them to chat (one was working remotely) and decide on a solution and then just fix it with one patch. But they were both pretty introverted so I'm guessing they 'preferred' to handle it through Gerrit" (field notes, July 2017).

As Peters (1999, 23) has noted, this model of communication is also sometimes defined by "open and frank talk between intimates or coworkers. Here communication does not mean simply talk; it refers to a special kind of talk distinguished by intimacy and disclosure." Michael also added that in the best iteration of communication during code review, the two developers share a sort of "respect": when the reviewer asks questions about the solution, the committer attempts to answer them and fix the problems that the reviewer mentioned (field notes, June 2017). This respect can form an intimacy between two developers, where two developers form a sort of rapport through their code review process.

Communication in this case is thus reciprocal, takes place between two developers, and involves explicit discussion and questions that one developer directs toward another. The rating system I mentioned in an earlier section (+2, +1, 0, −1, −2) is also part of a more implicit communication system between developers, and each developer might give his own subjective understanding of what each score means without explicitly describing this meaning to the other developer. This type of communication also can mean a sense of "transfer or transmission . . . of ideas, thoughts, or meanings" (Peters 1999, 15).

I asked Michael, "OK, so let me get this straight. I can't give Mike a +2 if a little piece of his code is wrong, so then I comment on a code block, and then Mike patches it, and its still wrong, so I comment again?" Michael responded by stating,

> Well you can—it depends on the reviewer style; my style was, "I can give you +2 even if something is a bit wrong, as long as I will highlight it and you will fix it in the future,"

> because my main goal with code review was to make the other person a better programmer for the future. But I know some asshole reviewers who would see a very small bad piece of code . . . for instance, a variable name they didn't agree with, because it wasn't descriptive enough, and they would give the committer −1 and tell them to fix it. I would in that case give them +2 and say "make the variable more descriptive in the future" . . . but I can imagine the other side would say "I care about the codebase and I'm strict with reviews, but some other people [like Michael] allow bad code to go through and assume that people will fix it." And a "lazy reviewer" I'm assuming would give a −2 to code without explaining what's wrong. (field notes, June 2017)

In this example, Gerrit's rating system takes on different standards for different developers. Where the so-called asshole developer gives a −1 to a developer who forgets to provide enough description of a variable, Michael chooses to waive this lack of description for the benefit of encouraging the developer to work on it next time. Technical standards take on different meanings in practice, and developers who understand the meaning of what Michael is communicating with his +2 will be all the more annoyed or surprised when they receive a −1 from him.

Broadcasting

The second model is the broadcast model of communication: rather than being a reciprocal exchange between dyads, code review becomes a one-to-many performative act. The developer's review itself broadcasts meaning to a group of (often) unspecified others.

For instance, Gerrit functions in such a way that each review is tagged with the name of the reviewer. At BerlinTech, one development team I was working on was expected to conduct at least two reviews per hour. While this standard was not kept up, some developers would just engage in code review to keep up their

hourly quota. Note that neither the content of the code review comment nor the rating of the code review itself was important. Rather, the mere gesture communicated to others "I am reviewing, thus I am part of this team." If Noah, for example, conducted a review, he was communicating to his fellow developers, as well as to his supervisor, that he is indeed a team player and does care about the quality of the team's project.

On the other hand, I observed that some developers found code review to be part of their procrastination regime, as it was easy and "mindless." Engaging in code review would help communicate to their team as well as their boss that they were engaged and working, while in practice, they were merely conducting this "mindless" work to avoid their more challenging job. The reviewer is thus not engaged in a meaningful dialogue with the committer but is rather clicking through the reviews quickly and mindlessly.

As I also mentioned earlier, code review becomes a surveillance tool for upper management to monitor what type of work the programmers are actually doing. Gerrit is designed in such a way that makes work highly transparent to anyone in the company, including management. It also becomes a sousveillance tool—a way for colleagues to implicitly monitor one another's work. The mere online presence of a manager within the Gerrit system, as well as the mere presence of a colleague, broadcasts or communicates to other developers "I am here, and I am watching you."

Communicating Code

A third form of communication present within a code review system is communication between pieces of code, which are essentially lists of commands or solutions that communicate meaning to a machine of other commands or solutions. Code can be seen as a sort of agent, commanding the computer's performance.

My previous two definitions of communication involved human communication. But in the case of code review, code has to speak to other code for it to function. Peters (1999) noted that while

communication can mean a sort of exchange, or transfer, involving some sort of reciprocity, these factors do not have to take place between humans: "Communication can mean something like the successful linkage of two separate termini, as they say in telegraphy. Here simply getting through, as in delivery of mail or e-mail, is enough to constitute communication" (16). In this book, my fellow author Finn Brunton eloquently analyzes Claude Shannon when discussing this type of communication between machines.

To illustrate how this functions in a code review system, take a look at a snippet of conversation between a team of developers who are discussing a small functionality failure in a feature they are developing for a website. This conversation was cut from a Skype conversation they were having (as mentioned, developers often discuss their issues using various chat systems):

> NINA: There is one failure, which I think not caused by you, @John—"Scenario: Clicking on map markers and PDC links" . . . when I check in the commit of Michal I still see it failing.
>
> JOHN: well, the current run is my submit. Let's see what is left after that, but yea, worth taking a look at as well
>
> NINA: @Piotr, I see that the step "And I expand the place panel contents if not expanded" which you added. I wonder if the change you did is the reason of failure of "Scenario: Clicking on map markers and PDC links."

In this instance, John submitted a change (often called a *commit* or a *submit*) to the functionality of a side panel that is affecting various links and markers on a map on their website. In this instance, the code in John's commit is not communicating to the code in Piotr's submit. Perhaps this code has a syntax error—which could mean that the line of code is not merging or communicating with the other code through this error. Unlike humans, who can infer meaning even when something is mumbled or slurred, or when a word is spoken in a different language, when various parts of one machine speak to one another through code, this code has to be written

using the correct syntax. One can say that the main purpose of the code review process is for humans (programmers) to assist machines (various parts of an app build) in their communication process.

Code communicates to other code through these processes of merging. Maria, the back-end developer I mentioned previously, explained the way in which code interacts, by stating, "Yes, we are constantly breaking each other's stuff. What you are creating communicates with other code others are building. And we are all doing it at the same time. Its like building a house. When building a house, the plumber works on something, and then the carpenter. Often this doesn't happen at the same time, because their work would conflict with one other. But we do work on stuff at the same time. And it does conflict."

One can imagine a sort of Tower of Babel, with code communicating to other layers of code. Although each line of code, or each commit, can be traced back to the author who created it, code has to communicate with other code even when developers are away, on vacation, or asleep. This communication brings the machine (in the case of my developers, a routing engine) to life.

IIIIIIIIIIIIIIII

Technical systems are incredibly social. A code review system, based on something that should be seemingly mechanic, is a highly variable communicative process both because of its technical limitations and also because of the culture of communication that develops in the setting in which it is situated.

Understanding how developers communicate to one another through a code review system can reveal how a technical system structures cooperation, how standards of communication develop, and the idiosyncrasies of human–human as well as human–machine communication. It can help us understand how digital media at large impacts communication. Understanding how producers and designers of technology create their products can reveal the cultural, political, and economic motives that structure

and determine the design of technology. Understanding the material resistance and viscosity of software, its bugs, breakdowns, and technical dependencies, helps us to understand some of the properties of digital media technology itself.

This chapter has focused on one case study: the code review process among corporate software developers. I did so to understand the various nuances and forms that communication in digital cultures can take. Although my communication models were in no way exhaustive here, I focused on "communication as interaction" between programmer and programmer, "communication as broadcast" between programmer and the rest of the organization, and "communication between code" as the way in which code is seen as an interactive process between other code. This analysis, I hope, brought you, the reader, closer to imagining the communicative interrelationship between machines (compilers, databases, processors, memory, servers, clouds), (2) programmers, and (3) the infrastructure within which both programmers and machines function.

Notes

1 Those less technical members of a development team who held the developers responsible to the customer and the product they were creating.

2 See http://www.gerritcodereview.com/.

3 As one of my informants clarified, the code review has many unofficial roles as well, in building shared coding practices, initiating junior developers, building a shared view of the team on how code should look, elucidating which part of code they are putting emphasis on as a group—style, efficiency, testability, and so on. In less collaborative teams, one can see how code review creates conflicts that should have resolved prior to coding.

4 https://github.com/facebook/react-native/pull/14259.

References

Peters, John Durham. 1999. *Speaking into the Air: A History of the Idea of Communication.* Chicago: University of Chicago Press.

Peters, John Durham. 2015. *The Marvelous Clouds: Toward a Philosophy of Elemental Media.* Chicago: University of Chicago Press.