From Russia with Code

Published by Duke University Press

From Russia with Code: Programming Migrations in Post-Soviet Times.

For additional information about this book
https://muse.jhu.edu/book/70046

For content related to this chapter
https://muse.jhu.edu/related_content?type=book&id=2537289
No one knows what software needs to do. We have to discover it. That’s where a lot of people go wrong is they think this is the way a software system should be designed, how it should work, is knowable. It’s not. It is discoverable.

—RICHARD SHERIDAN, CEO and chief storyteller of Menlo Innovations; quoted in Shane Hastie, “Linda Rising and Richard Sheridan on Creating a Culture of Joy—Part 2”

In the beginning was the Bible. To be more precise, it all started with the Russian Bible search algorithm. In 1995, the future cofounders of Yandex1—Ilya Segalovich and Arkadiy Volozh—developed a demonstration product that showcased breakthrough capabilities in search and language processing. The idea was to take a classical text and use it as a database for conducting keyword searches. Why the Bible? Volozh simply answered by saying that everything else was under copyright, while Segalovich explained that the choice was driven by general cultural concerns: “The Bible is the most popular text in the world, and if anyone would need some kind of search for some array of Russian texts, that was just it” (Sokolov-Mitrich 2014, 62). As a result, the software enabled a user to run a context search through the Bible, with the algorithm accounting for the Russian language’s complex grammar. Then the algorithm started learning and two years on it became a part of the constantly growing corpus of code that unites Yandex developers today.

What makes Yandex such a unique case worth scholarly attention? It started its search engine a year before Google—in 1997—but the contexts of development in the San Francisco Bay Area and in Moscow were starkly different.2 Yandex emerged in a difficult if not hostile business environment. Not only were there too few companies...
to sustain the critical mass for a self-replicating cluster—where firms gain advantage from knowledge circulation and the experiential development of human capital—but other industries were also competing for personnel. In the years that followed, Google outgrew the garage, while Yandex outgrew the very structure of the Russian oil-dependent economy (for an overview of initiatives to wean Russia off its dependence on natural resources, see Zhikharevich, this volume).

Since Russia has indeed been well known for its tech-talented emigrants, one might think that Yandex capitalizes on a large stock of scientists and engineers with significant strengths in mathematics, hence the company’s success. But contrary to received opinions, the Russian university system—despite its strong Soviet tradition of education—has largely failed to meet the high demand for programmers through relevant training. As a result, the burden of IT training falls to the IT industry itself; Russian software companies had to become factories for turning analytically minded graduates from noncore fields into skilled software developers. Yandex, in particular, hires not only IT professionals but also mathematicians, engineers, physicists, and other techno-scientific practitioners who all need to be integrated into their corporate community. In this regard, Yandex has the same objective as any other enterprise: to break through the clusters of knowledge held by its employees and create a shared base.

But what makes Yandex, Yandex? Yandex is the idea of a search engine expressed in code. The code, and with it the knowledge that constitutes Yandex, cannot exist independently of the complex technical and physical infrastructures where it is stored. However, the code also encapsulates practices and relationships of collaboration, training, and skill transfer. This social infrastructure, which is embedded into the company’s code, is a competitive advantage. Drawing on the case of Yandex, this chapter discusses how routine engagement with code serves to transform a diversified body of newcomers with different educational backgrounds into a coherent body of IT professionals while simultaneously creating a unified community.

What kind of knowledge is cultivated in the poster child of Russian IT companies; and how does Yandex manage to breed a collective of programmers who come from various backgrounds and face the lure of foreign markets? These issues require empirical attention because education, professionalization, and the process of membership within the IT community are usually considered in relation to open source development, while the peculiarities of these processes within the corporate software sector are often understudied.
FROM COMMUNITIES OF PRACTICE TO LANGUAGE COMMUNITIES

The concepts of “communities of practice” (Lave and Wenger 1991) and its variation “networks of practice” (Brown and Duguid 2001) have been popular interpretive tools for studying engineering and software communities; however, I believe that the conceptual shift from “practice” to “language” can help us better understand the social dimension of programming by going beyond its technical level. To illustrate the potential of such reconceptualization, I provide examples from the study of the Yandex community.4

The reason why the current analysis is centered on the building of a community is that the unsuccessful attempts to establish standards for the professional development of programmers enabled the industry to consider new venues for training.5 Indeed, software development often does not fit into bureaucratically organized careers. Developers participate in activities related to the exploration, testing, and sharing of knowledge, which virtually never follows formal standards and methods that are taught in academia. On-site training, instead, helps companies provide their employees with a specific type of knowledge that stems from practice and that is always contextualized. Therefore, studying professionalization from the perspective of community-building is productive, since it helps debunk the academic division of labor and avoid discussions of “prescriptive models” as applied to the shaping of the IT profession (Shapiro 1994).

A lens for understanding how one becomes a community member through practices and interactions as the primary inputs to learning was provided by Jean Lave and Étienne Wenger. They suggest that the process of acquiring professional skills does not necessarily correspond to formal training but is closely connected to the milestones of becoming a competent member of an informal “community of practice.” However, there is an important, though often omitted, characteristic of their analysis: Lave and Wenger (1991, 29) initially intended “to rescue the idea of apprenticeship.” Therefore, their model has limited application in the case of Yandex, as a developer’s training usually lacks the most important elements of apprenticeship learning: the face-to-face interaction with a master and hands-on experience through observation.6 To explain why the Yandex community cannot be studied as a community of practices, a minor detour into how work is organized at Yandex is in order.

What I noticed during my fieldwork was that newcomers and experienced employees most often worked and communicated on terms of parity. Although horizontal communication is highly encouraged at Yandex on the
organizational level, as it fosters collaboration and the free flow of information, experienced workers themselves support this managerial strategy and give novices the chance to express their opinions, involving them in all stages of problem solving. Sometimes novices even without any practical experience have knowledge of the most recent developments in the field, allowing them to make a notable contribution. Since interaction often fosters cross- and reciprocal learning, the type of one-way learning that underlies the master-apprentice model has limitations within this context. Another constraint compromising the use of the concept “communities of practice” is the absence of practice under observation.

Situated cognition emphasizes the context dimension of knowledge, where meaning is inseparable from the situation. However, while learning through practice—as described by Lave and Wenger—requires ostensive training, in Yandex such learning through observation is difficult to find. Here, learning occurs through practice, though not in the situation where a master teaches through actions, thereby passing on tacit knowledge to the novices; instead, it occurs when newcomers learn on their own with the help of written information. One hardly ever finds employees actually showing their colleagues anything on the screen or demonstrating how they write and fix modules, while going through lines of code together with their peers. New hires have to discover by themselves what the rules of practice and the norms of participation are, since an overwhelming majority of developers’ work time is spent alone with their computers. I will discuss below the reasons why the developers’ work is mostly isolated.

The main factor that eliminates face-to-face communication is the non-geographic organization of Yandex’s development activities; in other words, teams can physically work from different locations: a different floor or building or even from different cities. In most cases, software development at Yandex does not create a situation where face-to-face communication is required. Even when Yandex employees want to discuss some work-related matter, any attempt to do so creates, as some put it, “too much fuss.” The fact is that most of the company’s workspace is an open office plan, where sound travels freely. Yandex developers are very defensive of their right to work in silence and many of them even prefer to work with their headphones on in order to keep themselves “in the zone” and not be bothered by the constant buzz of clicking keyboards. Being “in the zone” is an integral part of developers’ work, as software development demands close attention and it is easy to lose focus on an idea, goal, or particular place within hundreds of lines of code. As the rule of silence is not to be violated, all the communications
go online. Developers usually prefer to collaborate sitting in front of their own computers using messengers, the intranet, Jabber, email, or tracking changes in the code directly in the version control system. Usually, though, online communication and face-to-face conversations facilitate information sharing, rather than the transfer of skills. Furthermore, in the particular case of developers, all kinds of discussions do not involve actual learning-by-doing. Knowledge at Yandex does not come from observation, but comes in the form of code that resembles instruction and is very explicit and visible (rather than unarticulated or inexpressible, as it is in Lave and Wenger’s framework). Therefore, practices themselves, if studied through the lens of “communities of practice,” often appear as a blind spot in the analysis of software communities. Since such conceptualization does not answer how code-writing practices help to reveal and transfer knowledge within a software community, we need to find an approach that can emphasize the materiality of code-writing practices.

To explore the training modes through which professional expertise and shared understanding of practices can be transmitted, we need to take a closer look at code and programming languages in order to study the practices of code writing that govern a programmer’s performance in the virtual milieu. Code writing is a central practice in software development that varies mainly by programming language. However, code writing is not a purely technical skill but a social practice. This is because once exercised, code writing inevitably becomes socially contextualized. I will further explain how on-site training of developers is connected to code writing as a sociolinguistic practice of using the source code.

As previously noted, while practice in “communities of practice” is disembedded from its ostensive power within a digital context, another domain where we can find this referential source is language. An important aspect of this approach is that language cannot be understood as the mere transmission of information but is a form of social action. Action-oriented approach to language draws extensively on Ludwig Wittgenstein's philosophy. I give priority to Wittgenstein’s work because his theoretical framework simultaneously explains situatedness of practices, their communal nature, and learning in coparticipation. The challenge of using Wittgenstein’s theories for interpreting programming practices is the question of whether it can be applied to artificial languages.

The issue here is that the machine can only function with binary logic (operating with values that are either true or false) and cannot access the context-dependent, auxiliary knowledge about the range of meaning that
accompanies usage. In order for the machine to “understand” the code written by the developer, a compiler or interpreter “translates” it, i.e., converts it into machine code. In this regard, programming languages have a constitutive role in code-writing practices as their syntax is designed to ensure such transitions. Besides syntax, programming languages predetermine the correct combinations of symbols, data structures, comment density, number of lines, the module concept, and so forth. In other words, any programming language defines a set of actions, otherwise the machine will not understand the commands. Since programming language acts as a criterion for determining meaning, the developer must be deprived of the opportunity to misinterpret and must always use a language correctly. With such emphasis placed on the lack of surplus significance in code and its preciseness of definition, one may assume that programming languages are—in line with Wittgenstein’s philosophy—“ideal languages” (Wittgenstein [1922] 2001). If this is true, then code must be something clear cut and bug free—as there can be no misuse of an ideal language, such as, for instance, the one in which the code is written. However, bugs happen and software is not flawless, hence we need to discover what bugs show us: mistakes that are made in the human domain.

An error may occur not only because of some crucial mistake in the system’s architecture or the wrong implementation of an algorithm, but it may also creep into software by way of a programmer’s simple logical mistake, the misapplication of some technique or component, a misprint or slip-up in actual code writing, or any other failure that occurred unintentionally within the social dimension. Such errors often happen when one developer cannot understand another, or, more precisely, when one developer misinterprets the code of another. Therefore, the rules that govern code practices are looser than one might imagine and misuse of a programming language is possible. We used to think about code as something that is written for machines, rather than for people, though this issue is controversial and deserves special attention.9

Programming languages and code are not purely technical. Yet here I want to make an important distinction: the machine indeed “reads” binary code, but the human reader deals with the source code, which is converted into machine code by a compiler (or interpreter, depending on the programming language). Machine code is an object (an executable file), while source code is a constantly growing “corpus” that includes code itself and comments written in natural language.
Source code is the only stage where a developer can read code, even without comments, since it comprises only human-readable and human-written source code files.\textsuperscript{10} The readability, when considered as such, varies by programming language as well. For instance, one can read a program in Python nearly as easily as reading this text (or at least grasp the general idea of what the code does). High-level languages closely resemble spoken languages; to some extent, this explains why the entry barriers to learning them are relatively low. One can quickly pick up such languages, since they have syntax that is easy to read and to write and in comparison to other languages one need not know a massive base of specialized commands in order to get started. Another argument is that programming languages (just like natural, spoken ones) are actively developed. This occurs not only through their own evolution but also with the appearance of new languages that incorporate many features of their language predecessor. Therefore, the main reason why I believe that programming languages can achieve recognition as natural languages is that most of them are created in the semantics of spoken languages and inherit their immanent ambiguity. With all the technical rules that programming languages set, there is still room for developers to flexibly use a language, readjusting and interpreting the existing technical regulations that were imposed upon them. The crucial point here is that languages themselves are not compiled (or interpreted); rather, it is their implementations that compile code. Therefore, languages are, indeed, unambiguous and standardized as such, but there are innumerable ways of how to use a language differently for writing code. Consequently, such relative flexibility creates contradictions, making code less readable for its other users (i.e., developers).

Aside from natural-language comments (which will be discussed later), there are a lot of social elements in code, or, as some of my informants call it, “noise”:

I can immediately spot former Python developers by how they name variables. I’ve seen a couple of times how people follow Python conventions while writing code in other languages, and their code looks untidy. This noise just strikes my eyes and keeps me from focusing. It’s not the end of the world, but it shows that you don’t care about me reading it. You can, actually, edit this automatically with source code editors, but you need some add-ons that provide such functionality. And I’d rather not to be concerned with such things. (Yevgeniy, aged twenty-seven)\textsuperscript{11}
Such “noise” is completely ignored by the compiler because it is not connected with programming language but related to coding style. Coding styles are guidelines for writing software, but they serve as purely visual devices that commonly deal with the appearance of source code. Coding style is a subjective matter and depends on the preferences of an author, although it may also be a company (project or community) convention that emphasizes the necessity to improve the readability of code in order to ensure collaboration. “Noise” (i.e., differences in style) points to the fact that a convention is being violated and that the code is getting hard to read. The issue is that styles provide languages with variability. In most cases, languages support multiple ways of naming variables, presenting data, locating spaces between functions or indicating whether to put curly braces each on their own line, determining what kind of layout is most comfortable to the eye (how long a line should be, whether lines should be placed compactly or have an empty line as separator), and the list goes on. Styles are often developed for a specific programming language since, for instance, the coding style in c++ is not suitable for Python. Therefore, the code can be written differently according to style, and such an unsettling possibility of bypassing the rules influences how users perceive the meaning of code. Styles govern the process of reading and understanding, either hindering or facilitating these actions.

The difficulty here is not only whether there is (or is not) consensus on a programming style, but rests on the fact that such flexibility in representation creates a space where everything that is written is error prone. For instance, if a symbol has different meanings across languages:

Sometimes Python developers forget the curly braces and then they’re surprised that their code doesn’t run as intended. In Python, you don’t need curly braces, but in c++ curly braces explain your logic to a compiler. So, if you have more than one operator in a loop and you write this loop without curly braces, a compiler will accept that, but it will lead to a bug. Newcomers make such mistakes a lot, especially if they switch between languages.

I mean, they are not stupid, it’s about formatting properly. (Evgenii)

This highlights how a small and simple typo may alter the meaning of the code. If a coding convention is being followed, it is easier for the developer’s eye to catch such errors within what is an organized rather than “messy” code. It seems a reasonable assumption, therefore, that mistakes can be made by anyone, even experienced programmers, since many typos will inevitable sneak past the compiler. However, following the coding convention not only helps one read the code and detect errors at the earliest possible stage, but also
creates meaning. Seen in this light, style has transitioned from a mere visual device to a conductor for meaningful code production. Good style makes meaning “visible” and describes the ways in which a language is used, while the absence of style “hides” meaning and introduces ambiguity. If one has a consistent approach for writing code that corresponds to the understanding of others, then the meaning behind code remains clear and unambiguous. I argue, therefore, that programming style itself represents a liminal zone, where the “sociality” of code unfolds.

The implementation of programming languages is indeed governed by the rules of the language itself, though one cannot be reduced to the other. Instead of being automatic and standardized, the usage of programming language can be voluntarily changed by its users and is inherently problematic as well as negotiated. To collaborate effectively, Yandex employees also need to develop a shared understanding (i.e., style) of how to write code, one that is clear to every member of a particular community. Style allows us to make the perplexing character of code intelligible, since a developer, while reading source code, discovers meaning through particular details and contexts of use. In this sense, addressing the codebase is similar to a language game (Wittgenstein [1953] 2010) that implies creation of conventional systematic practices. A developer always needs to “consult” with the “source” in order to bring his code into correspondence with the codebase, with the “text” of the collective author that constitutes a certain practice.

Thus, working with the code as a learning process has two primary inputs: technology (when one learns the rules of a programming language that constitutes practices) and community (when one learns how to use this programming language according to conventions in the given context of the codebase). Yandex employees need to agree that they mean the same things not because they share the same understanding of how software should be written, but because they are predisposed to have a convention by the need to collaborate. While community conventions are consistent, they are not fixed or standardized, since they depend not only on agreements but also on adjustments that constantly arise. At this point, it is noteworthy to discuss what properties code has that allow it to facilitate the processes of learning and community building.

Code has the performative function that stems from its unique property: “Code is the first language that actually does what it says—it is a machine for converting meaning into action” (Galloway 2004, 166). Adrian Mackenzie (2005, 76) was among the first to discuss the performativity of code. Focusing on the Linux kernel, he suggests that the performativity of code
lies in its ability to objectify linguistic praxis through processes of circulation. However, while processes of circulation play a primary role for open source code, such practices of “repeating” and “citing” that, according to Mackenzie, produce performative effects are limited in the case of a company. In this respect, proprietary code is not speech, rather, it takes the form of a “sacred” text that can be accessed only by “elects” (members of a community or, in this case, Y andex developers). Yet it is not a text because it is not a fixed substance but the result of a progressive process of code writing. Indeed, a company gives code its limits, but code also constantly changes and grows with the community. In contrast to open source, one cannot decide what to write and from where to start because the company code is being written “on the shoulders” of peers and precursors, generation after generation. Therefore, a key feature that distinguishes the Y andex community from any other expert community is that developers collaborate virtually with people but practically are engaged with the fruits of their collective labor: the codebase.

The codebase is performative by virtue of the fact that it is itself an act of speech—it enacts its own description. Furthermore, the “corpus” encapsulated in the codebase preserves the intentionality of its collective author, since it cannot be detached from the context in which it is written: it is “glued” to conventions that govern code writing, data in data centers, hardware, algorithms, protocols, operating systems, virtual machines—to all the digital, social, and physical elements that are involved in its production. Therefore, such a “corpus” literally shows how the company’s software is being written, setting a pattern of code-writing practices. However, while being stable, the “corpus” is not ideal or fixed. In the case of a bug or out of necessity to update or scale certain technologies, the “corpus” is reconsidered and changed in order to bring it to stable form again. When failures in the technological domain happen, the norms and patterns of how to write software are simultaneously being revisited. The “corpus” communicates these changes to its readers and users accordingly. Therefore, it is a “monster” (Haraway 1992) that persistently resists transparency within the community while the community persistently strives to make it transparent. The ability of the codebase to simultaneously absorb and transmit the collective experience makes it a central element of community building and learning, because it facilitates learning-by-doing and learning-as-membership. The access to its knowledge means that newcomers have become a part of the community.

Such a change in approach, from communities of practice to language communities, allows us to treat programming languages and code as sites
through which we can explore how training and expertise are transferred. In what follows, I focus my analysis on code and programming languages to capture how Yandex builds a professional community around its codebase.

"READ THE SOURCE": PRACTICES OF LEARNING AND RITES OF PASSAGE

Using the conceptual lens described above, I argue that Yandex builds its community through the practice of reading the code. Readability of the source code is an important property from the business perspective, since it makes software maintenance easier and less time consuming: the code should remain readable not only to newcomers but also to old-timers, should they have to go back to it at a later date. The emphasis on readability is also important for creating a sense of community: the source code should be readable in order to continue its function as the center of the community, concentrating every employee around it. It is by being clear that the code ensures software quality, while simultaneously creating commitment among employees to the replication of the Yandex community. Therefore, the very first thing every new employee has to learn is how to make code readable and to improve its utility for human readers. At Yandex, I heard several times that code (ideally) should look like it is being written by the same person, and if the code is “unreadable, but working, it is not good code” (Artemiy, aged forty-two). As I illustrate below, this often overlooked practice of engaging with code— “code reading”—is precisely the practice that facilitates the transition from neophyte to professional.

The interviews reveal that during the early stage of their careers newcomers are primarily involved with debugging. One of my informants admitted that if he needs something to work, he quickly writes a working prototype, which is needed before being polished into a production version. Usually this kind of work is performed by less experienced colleagues:

This is how it works for a while for all newbies: I made them do all the dirty, routine work. Primarily it means that they will have to do a lot of debugging; later on they can be entrusted with fine-tuning and rewriting. From the very beginning they basically only read a lot. This way they can get acquainted with the codebase as it’s an essential skill to know how to operate within it. (Denis, aged thirty-seven)

Therefore, debugging proves itself a simple task and an effective means by which newcomers introduce themselves to the company’s codebase and
acquire all the necessary skills to work with it. This is not only the aim of on-site training but also a necessity for stable software development.Debugging is an essential routine that novices can perform, although they need to understand the corpus of code in the base and navigate freely inside the system. However, even for experienced practitioners it can be difficult to get to know a codebase and understand semantic and structural relationships between various interlinked pieces of code.

Since newcomers are not the authors of the code and they have never before participated in its development, they face the necessity of learning how to read the code and grasp all its essential moments in order to maintain it in future. Therefore, the process of familiarization with the codebase has the following trajectory: first, newcomers learn how to read the code (while being relatively passive members of the community); after this, they learn how to develop (as they begin transforming into active contributors). Such learning from the inside out comes with an important advantage. In the case of internet-based products that require stable online performance, having novices “on the bench” is potentially a way of keeping the project safe in case key contributors leave; if this happens, they are already familiar with the codebase and able to improve upon it. Once newcomers have familiarized themselves with the codebase, they can avoid inappropriate code duplicates and significant dependencies between code blocks, as well as build software following reusability principles, i.e., performing their tasks as experienced users of the codebase. Therefore, reading is a practical achievement: having the ability itself means that one understands the code and is able to operate with it. I need to note that, obviously, modifying code and then observing what happens next is the best way to learn how to deal with software—by experimenting and testing its capabilities. Yet in the case of corporate software development, programmers are limited in their freedom to “play” with code, as it might cause serious damage:

You can’t introduce any extreme changes, even if they’ll improve code performance. Very often you have to sacrifice performance for readability because you need to make sure that others will be able to maintain your code easily. (Evgenii)

In this respect, reading is not a passive act but an element of training, since by reading one prepares for the role of writing, a role that needs to be in sync with collective intention. However, the question of what exactly is to be read in the source code is a topic for further discussion.
In the previous section, I specified that the source code has two parts: human-readable code and comments. Comments that accompany source code should be convertible into documentation to the company's code, and they are written by developers-authors for developers-users. Writing documentation and comments in code is considered to be an important part of corporate software development, because it explains how a program or a system operates and how to use it. Therefore, a developer needs to comment on his code and keep track of it in order to clarify all possible inconsistencies.

Comments are used to clarify code, to make it more readable and clearer. However, at Yandex there is a conflicting tradition of not commenting on the company's code. The very presence of comments in the source code is a highly debated issue among Yandex employees. In fact, such controversy represents a conflict of interests.

The practical problem of code without comments is that it creates an uncertainty about its business purpose and direction—whether it should be actively used or removed or, more probably, changed. To put it more directly, comments to some extent justify the presence and usefulness of code. From the business point of view, software companies need documentation because management has to keep track of development. From the practitioner's point of view, comments often create confusion. While many of my informants stressed that it is necessary to write comments in order to have a proper documentation for those “next in line,” some of them argue that it is a questionable practice:

Sometimes it is better not to read comments at all as it could be extremely confusing. Ideally, it serves the purpose of clarifying code; however, as a rule of thumb, it does quite the opposite. For instance, let's say we write down a general algorithm in comments. But we execute its special case. So, what we have: you refer to one thing, while doing another. No details provided. And you have to sit and delve into it really long to figure it out.

(Aleksei, aged twenty-six)

Comments can be even more misleading when they state that code is doing one thing while in practice it is actually doing something completely different; relying only on information in comments may lead to serious vulnerabilities in the software later on. This example illustrates an interesting phenomenon: instead of functioning as a device for clarification, comments sometimes produce unexpected output and, in Bruno Latour's (2005) terminology, serve as “mediators. Comments are designed to formalize development, to make it...
transparent and explicit. Bad commenting or the absence of comments, on the other hand, deprive software of its ability to be universal in its application, to be detached from its creators.

In her study of Indian IT workers, Sareeta Amrute (2014) has shown that if a developer fails to explain the code in comments, he augments his irreplaceability. While such commenting strategies may serve to sabotage corporate rule, at Yandex, as our further discussion will show, the absence of comments has an almost ideological meaning and helps to teach newcomers how to be reliable professionals who care about the community and its replication.

“Brutal” programmers with vast experience rely mostly on technical virtuosity and mastery and do not put comments in the code: “Don’t let comments mess with your code. Excess is the enemy of good code. Nothing extra, just the essence. Comments are not necessary: if you are able to write code, you are able to read and understand it without further ado” (Denis). Such styles of coding help to test one’s proficiency and sort out candidates with potential. It is similar to what Christopher Kelty (2008, 58) calls “argument-by-technology” and “argument-by-talk,” meaning that the process of familiarization with professional practice surrounding technologies is constituted through code, as well as speech. Thus, the Yandex tradition of writing code without comments—a process that requires technical expertise—has become an organizational element that helps transform newcomers into experts. “Argument-by-technology” places initial hurdles before newcomers, requiring them to read the code and understand it:

You know, our elderly, so to say, generation is against comments in code. I think they believe we [newcomers] should suffer, while reading it [laughing]. Sometimes it’s really hard to understand what is in there [in code] without explanations and designations that are usually written in comments. They say, they want to get from us code without redundancy and that we need to learn how to understand it without tips. (Tamara, aged twenty-one)

This does not imply that luminaries in the company do not want newcomers by their side as colleagues; instead, they want them to prove their technical skills—the ability to read the code.

By reading code without comments, newcomers familiarize themselves with the structure of the codebase and the reason behind it in order have the requisite skills to improve it in the future. Thus, instead of simplifying the process of code reading, comments or their absence serve as an effective learning tool. While navigating a codebase, employees encounter difficul-
ties that capture their attention and help them memorize important moments in the code. Also, when newcomers have problems with their stack, their more experienced colleagues ask them if they read the source itself, not the comments. The very need for comments calls the explicitness of code—often hailed as the virtue of the IT world—into question. Yandex’s luminaries argue that the answer is always in the code itself. My informants with more than ten years of experience stated: “Read code and everything will be clear.”

This motto carries almost biblical connotations in the sense that one should seek all answers in the source. The knowledge of the source and the ability to read it signify that one has become a member of the group: new people join the community by learning how to make sense of the code itself, of the “corpus” encapsulated in the codebase. However, the analogy between the Bible and the codebase lies not only in the fact that both are written by a collective author, but that they both have performative force; that is to say, they cannot be interpreted.

The fact is that the comments are interpretations of actions and the code is itself an action. As noted, comments can be misleading, while the code itself is the ultimate truth: the code is not describing anything, it simply runs. Thus, by reading code without comments, newcomers learn how to understand without explanation, how to grasp the very essence of the code and then write code that literally speaks for itself:

Code tells you exactly how it was done. Comments explain to you the process or the idea behind the code. If you need to explain, it means you’re not sure. Or it means that your code is way too complex to understand. If it lacks certainty and clarity, it’s always bad code. You don’t have to explain what it is supposed to do, just do it. . . . Actually, I never read comments, as I need to see the result. (Aleksandr, aged twenty-nine)

My informants also noticed that sometimes newcomers use comments in order to “hide” ambiguities in their code, to cover their lack of experience by using comments as a “crutch”: “They often try to explain in comments what was meant to be done, instead of actually making the code to do that” (Denis). Also, such “crutches” often lead to bugs that, if detected in the main branch of the code, become not individual mistakes but mistakes of the collective author. In order to prevent such “shared” bugs, comments are often discouraged.

The absence of comments in code at Yandex provides its community with stability and ensures high-quality software by pushing newcomers to write as clearly as possible and not rely on comments as descriptions or interpretations. The absence of comments can be seen not as a drawback but as a
practical necessity that meets the requirements of professional practices. In “industrial coding” the very skill of reading lines of code written in different languages and in different styles is crucial. By “industrial coding” informants mean the specific style of code writing:

Any big company, and Yandex as well, is about industrial coding. So, code should be accurate, simple, and readable. It means that when you do industrial coding, you should write it for centuries to come. If it’s reliable, durable, then anyone can continue your work, in case you are gone with the wind. But it’s not only about us being interchangeable. Also, it’s about such situations when other groups need to borrow our code. And they should be able to understand it, if they are going to use it. Basically, if code is written in such a way, it is easier to maintain it in production when it goes public, already open for users. (Yaroslav, aged twenty-three)

“Industrial coding” implies an opportunity for long-term use and further development. Therefore, by wading through an excessive amount of code lines without comments, newcomers receive training that stems from a hands-on experience of exploring the codebase and results in a set of practices for creating readable and, therefore, reliable code.

CRITIQUING AND CORRECTING

The analysis of how Yandex builds its community through language practices gets more nuanced if we consider peer code review as another technique of training, along with code reading and commenting. Peer code review is introduced in order to detect and fix the vulnerabilities of code, but it also reveals itself as a practice that allows Yandex newcomers to evolve as professionals.

In order to create a stable, consistent, and durable “corpus” of code, its quality needs to be controlled; and peer code review performs this disciplining function. The practice of peer code review has become a key element in software development not only for open source projects, but for proprietary software as well. In particular, all lines of code that are going to be committed to a shared repository must be approved by peer code review. Yet, instead of being centralized and vertical, such control also has a communal nature: it is delegated to coworkers and dispersed among employees. Further, I will explain how peer code review helps Yandex employees to achieve a shared vision on software development.
Peer code review is closely connected to code reading, but while code reading is passive and subject to individual tasks, peer code reviewing is a collaborative practice that facilitates knowledge circulation and sharing. My field research showed that peer code review creates affinity for one another among employees, since, at Yandex, developers are allowed to review all the company’s code, regardless of the project they are currently working on. Communicating through peer code review allows different contributors to combine their ideas and track progress in other projects and also helps unite newcomers around practices of development.

Developers pay specific attention to code-writing practices, as they believe that one can evaluate professionalism by observing how the code is written:

We have a guy; he just recently came to us after his work at a bank. Well. . . . Obviously, he can’t write code. It’s not surprising, though, as he comes from a bank, but still I’m wondering how can one write so badly?! Noodles everywhere! We can’t read his pieces. A total mess! (Anatoliy, aged twenty-six)

This quotation not only highlights the distinction that this informant made between his company and the bank, but also uncovers the ethical rule for “good” code writing, which is accuracy and the necessity that code be “readable” so that everyone is able to work with it. I want to emphasize here that the concept of “good” is not a given, but something that is invented collectively and achieved during systematic examination of each other’s code, i.e., peer code review. This practice allows developers to discuss and propose changes to the codebase, as well as negotiate their code-writing practices.

When new employees come to Yandex (both experienced and inexperienced), they start writing code according to their own understanding of software development. According to my informants, code must “express” certain criteria. Usually, they explain what “good” means to them in terms of beauty: “Code might not be written in a conventional way, but if it’s still beautiful, then this is good code” (Mark, aged twenty-nine).

Yet while some of my informants defined “beauty” in terms of “nontrivial solutions” inscribed in code, others stated that “accurate design” and simplicity characterize “beautiful” code. These two different understandings of “beauty” represent two conflicting approaches to programming: mathematical and engineering. This is a very rough division, though it illustrates how different understandings of software development result in a variety of possible code-writing practices. Such imposition of a particular understanding
on code production is similar to what Michael Lynch (1985) called “disciplining,” which is when objects are brought into compliance with a certain scientific vision.

It should be noted that the division suggested above does not necessarily represent educational background, but rather subjective viewpoints on what is considered the best way to write code. In this respect, it is also interesting that Yandex developers have different preferences for using particular programming languages. Usually, but not necessarily, they tend to learn languages that “reflect” their understanding: “mathematicians” prefer high-level languages (e.g., Python) and “engineers” are adherents of low-level languages, especially C++.

Importantly, I am not suggesting that “mathematical” development equals writing code in high-level languages, or that “engineering” development implies writing code in low-level languages. There are projects at Yandex that are fundamentally engineering, but are written in high-level languages (e.g., intranet services at Yandex are mostly written in Python). Yet, the approach that programmers choose is strongly connected to how they understand development and prioritize objectives.

“Engineers” often use Python as it is a multipurpose language—it is even used at CERN and NASA— with a clear syntax and readability that make it relatively easy to maintain. However, in the case of software production within the context of a large company, an “engineering” mind chooses low-level languages for the sake of performance. Meanwhile “mathematicians” are more inclined to write code in high-level languages, since they are usually more focused on the problem itself rather than on the tools that are required to solve it. Similarly, this may be why “mathematicians” frequently start by writing “dirty,” unclear code, as they are more concerned with making it work—with solving the problem—than with long-term maintenance issues. According to my interviews, mathematicians are represented by a number of specialists with different backgrounds who focus on creating solutions that result in sophisticated algorithms. They prefer to use abstract programming languages that are “slow,” as a machine needs more time to recognize them, but “fast” in that it is easier to write code on them; they are useful for obtaining quick results during prototyping. The community of engineers unites developers who are focused on the construction of a product’s architecture. They prefer to elaborate the design of software and are concerned with how code will work on hardware. For their aims, engineers choose low-level programming languages that are “fast,” because machines...
readily recognize their symbols, while writing on them is “slow” and time consuming: “I like Python as you can write the formula you need with a couple of lines. But in the case of c++, it could take you hours of code lines to write a finished piece” (Mikhail, aged twenty-six).

This example with mathematicians and engineers illustrates the classic differentiation between “us” and “them”; and this division is possible within one company because their coding practices bear the imprint of a certain conception of how code should be written. This conception can exist in a particular community or within a certain discipline. It is often unrelated to the content of practice but closely connected to the way this practice is performed. In this respect, the quality of code is related to its readership, to the target audience that often coincides with a particular community that shares the same understanding. While all my informants denied that their community could be structured according to different languages, software communities often evolve around developers who share the same understanding of how (and which) languages should be implemented.

Therefore, there are a number of crisscrossing communities that vary based on their views about software development, their educational backgrounds, experience, and also the programming languages they use. The only thing that binds them together is the necessity to work toward common objectives. In order to coordinate their actions and make their code readable, developers need to create a convention that will ensure their agreement on meaning and coding style: if one does not follow the rule as described by coding style, others will not understand their code. The control over following such regulations is usually exercised during peer code review. But it is also peer code review that allows such conventions to come into existence in the first place. I want to highlight that such software conventions are unique to Yandex and they are not akin to industry standards since they are created during peer code reviews.

I believe this is why many of my informants admit that at the very beginning of their careers at Yandex, code development rather than code reading seemed easier for them: one is able to write code without following the convention, although it is difficult to read and understand code without knowing the rules that such conventions set.

However, the interviews revealed that code reviewing is a very hectic process that is hard to navigate. Several of my informants mentioned that review comments do not always contain useful advice. As one of my informants elaborated:
Comments in code reviews are often minor and, to be honest, poorly reflect upon shortcomings in code. Sometimes they can even highlight for you a specific place in code and just write “Won’t work.” That’s all. Since brevity is of the essence. . . . And, so, you have to spare no effort in this and rely on yourself. (Aleksey)

Informants also emphasized that one needs to take greater responsibility for all, even minor, changes, as they might result in greater difficulties down the line:

You know, sometimes code will be reviewed after code commit.19 Well, generally it is before. . . . Well, you know, we usually fight a lot over this: before or after. It is debatable. Anyway, for the sake of time, it might happen after you already entered all your changes. If it has bugs, it might also affect other modules. . . . All that happens next is a horror show: we are trying to rewrite everything around a troubled spot as it all starts crumbling. (Yaroslav)

The quotation above reveals that the consequences of peer code review have often become confrontational, and it pushes newcomers to change. The absence of a conventional approach to peer code review across projects (pre-commit or postcommit), the habit of providing quick and undetailed review comments, and the scalable character of reviewing practices facilitate professional learning. Thus, peer code review practices force developers to start exploring and learning the company’s codebase faster while simultaneously gaining practical knowledge about the corporate environment.

In addition, I want to clarify one more aspect that helps boost the on-site training of Yandex newbies. As previously mentioned, any Yandex developer is able to review all the company’s code and write a review of any new piece that appears in the shared repository. Also, it is possible to trace the author of code through an intraorganizational network called “Staff,” where one can find employees’ profiles along with all their contact information, position titles, and the structure of projects and divisions within the company. Moreover, Yandex has its own system of blogs (“Etushka”) with a built-in ranking mechanism: the most popular and widely discussed posts are displayed at the top. Yandex’s internal blogs are used to report on various nondisclosure-agreement topics: they discuss their new software releases, beta testing, the company’s business strategies and plans, and sometimes things that are not work related. While being a platform able to facilitate knowledge sharing, the system of blogs often serves the function of expressing public shaming:
Code review is a hot topic. It happens very often that somebody posts a piece of code with no comments at all, just lines of code. As a rule, things tend to go south: the authors are publicly accused, made fun of and usually they have to try to explain what they plan to do initially, then fix all the bugs, rewrite a piece, and post a link to the latest review. It is a motive in itself. (Aleksey)

This illustrates the role that the practices of critique and correction play in ensuring the stability of the “corpus,” which is central to community building at Yandex. The tradition of criticizing and correcting as part of peer code review creates a unique, constantly changing “corpus” that differentiates the IT community from any other sci-tech community.

Unlike other fields and disciplines, software development as well as computer science do not have a disciplinary core—a solid tradition with “founding fathers” and unity of writing. The algorithms that are created within computer science are constantly revisited and improved. When the algorithms are implemented in the industry they are completely changed by adjustment to specific needs. And software development itself persistently seeks its origin across different approaches and practices. The case of Yandex has shown that software development does not occur in a rarified, abstract world with fixed concepts and rules, which is probably why it is so hard to strictly define its origin. Instead, we can find a number of communities within the field that create their own, unique repositories of knowledge.

The perpetual return to the codebase, which I described earlier, is similar to what Michel Foucault ([1969] 1984) discussed in “What Is an Author?” An important distinction exists though: Foucault argued that such “return to the origin” is what sustains a discipline that is dependent on the work of an individual author. He draws on examples of Marx and Freud, arguing that they have become not just the authors of their own works but “founders of discursivity”; that is, their texts produced “the rules for the formation of other texts” (Foucault [1969] 1984, 114). This also creates a space where, in one’s attempt to link oneself to a particular discipline, one cannot go beyond these original texts, since writing has become enabled by the “author function.” In this sense, while psychoanalysis and Marxism are “closed” around a fixed original text or corpus written by their founders, the “corpus” at the center of the Yandex community differs in that it remains “open” to criticism and correction. The unity of writing at Yandex is not imposed by the existence of some inceptive work that eliminates contradictions a priori; rather, such unity is achieved by resolving ambiguities on a permanent basis through
peer code review. Therefore, Yandex employees have a moral obligation to correct the “corpus” because it ensures the stability of their community.

It is worth mentioning that while writing, according to Foucault, is defined in terms of the absence of the author, at Yandex writing never happens without the collective author, since the community extends into the “corpus” itself. Thus, code writing is the curious form of practice that does not reduce an ongoing process of knowledge production to one finalized text, since it is always tangled into data and context. While most forms of modern knowledge work result in the creation of abstract “immutable mobiles” (e.g., scientific articles) that are the same in different locations and cultural settings (Latour 1990), “corpora” that occur in software production cannot be detached from authors and spaces.

Code exists in and produces around relationships that are both of social and technological nature. However, these relationships, no matter how dispersed or fixed they are, still carry their “situatedness”—imprints of where, how, and by whom they were set in motion. While the Yandex community is built around the codebase, Russian civic hackers studied by Ksenia Ermoshina are mobilized around a different kind of circulating “corpora” of code (Ermoshina, this volume) and the source code at Goldman Sachs is part of relationships between yet other types of technological and social infrastructures. The multipurpose nature of programming languages provides a developer with the potential to implement them everywhere, though the knowledge of how they can be implemented in a particular place or space is what makes code-writing practices intelligible and “located”; therefore, it is possible to argue that they do not travel around countries or companies as easily as we used to think.

CONCLUSION

Drawing from a study of Yandex’s practices of training, this chapter has discussed how routine engagement with code serves to transform a diversified body of newcomers with different educational backgrounds into a coherent body of IT professionals. In order to collaborate, employees need to share a common understanding of software development that does not arise from any particular methodology imposed from above; rather, it is achieved and discovered through everyday routine processes of engaging with the codebase.

The example of Yandex’s codebase has shown that code and the knowledge it encapsulates can no more be detached from machines and technological infrastructure than from the social relationships and practices of Yandex
employees. Studying code from the perspective of language allows us to reveal complex relationships of training and skill transfer that the hybrid codebase mediates. A developer needs to write complementary code that reproduces the “rules” of the existing codebase. Such “rules” articulate themselves in meeting the requirements of both the community and the technological environment wherein the code runs. Thus, when one reads the source code one also prepares to perform the role of “writer” by learning the conventions of the community, all of which are inscribed in the source code. Along with code reading, Yandex developers build a community around the company’s codebase through peer code review and commenting on code. Such mundane and routine practices, exercised across all the projects and teams at Yandex, allow employees to obtain practical knowledge and gain the shared understanding of how software should be written.

There is one amusing footnote to this study of Yandex. Earlier in the chapter, we discussed that the company’s codebase resembles a sacred corpus in the sense that the emphasis is on a textual assemblage whose reading is one of the key “rituals” of the Yandex community. New people join the community by reading and learning how to make sense of this corpus (i.e., the codebase), and it is this corpus that passes on knowledge through the generations and remains after all the current players are gone. But unlike other sacred corpora, Yandex’s accumulates knowledge and grows with the community. If the story of Yandex’s creation hadn’t started with the Bible search directory, the story would have to be invented; for where two or three have gathered in the name of code, Yandex is there in their midst (Sokolov-Mitrich 2014, 61).

NOTES

1. Yandex (NASDAQ: YNDX) was the first Russian technological company that made its initial public offering on NASDAQ.
2. On the complex technological ecosystem that is behind California’s economic success, see Arora, Gambardella, and Klepper 2005; Etzkowitz 2008; Gilson 1999; Kenney 2000; Lerner 2009; and Saxenian 1994.
3. See, for example, Coleman 2013; Coleman and Hill 2004; Kelty 2008; Kogut and Metiu 2001; Singh 2012; Steinmacher et al. 2015; Takhteyev 2012; and von Krogh, Spaeth, and Lakhani 2003.
4. With permission from the public relations and human resources departments of the company, I conducted my field study in Moscow, where the headquarters of Yandex are located. During my fieldwork (from January 15 to March 1, 2014), I carried out twenty-six semistructured, in-depth interviews with developers.
and four expert interviews with managers, along with workplace observations during the interviews and follow-up discussions with several informants.


6. On the applicability of Lave and Wenger’s model, see also Cox 2005; Kimble 2006; Storberg-Walker 2008; Takhteyev 2009, 2012.

7. Version control systems are used to track changes to software development projects.

8. On comment density, see Arafat and Riehle 2009.

9. In 1984, Donald Knuth, one of the world’s preeminent computer scientists, introduced a novel approach to programming, which he called “literate programming” (Knuth 1984). While literate programming has not become mainstream practice, it represents an important shift in attitude: the idea that code is intended for human readers rather than machines.

10. I want to emphasize, though, that under consideration here is the collection of source code without comments. Comments are descriptive notes aimed only at human readers, and they do not “participate” directly in the software production, as they are not enclosed in output (i.e., a program runs regardless of their presence or absence).

11. All interviews with IT workers were conducted in Russian and translated by the author.

12. For more on code as speech, see Coleman 2009, 2013; and Stallman 2002.

13. They are skillful and knowledgeable developers who know all the ins and outs of Yandex technologies. Richard Sheridan (2013) calls such experts “towers of knowledge,” meaning that if they leave, a project could fall apart.

14. In addition, my informants told me that commenting strategies may depend on programming language as well. For instance, comments in programs written in high-level languages are always undesirable, because code in such cases should be self-documented.

15. Code in production is the stage when the actual development is finished, the code has been already released, and now it is being maintained.


17. Compare with Thomas Kuhn’s (1962) discussion on the role of aesthetic considerations in the acceptance of new scientific paradigms.

18. The Conseil européen pour la recherche nucléaire (European Organization for Nuclear Research) and the National Aeronautics and Space Administration.

19. A “commit” is the making of a set of tentative changes permanent.

20. Debates over relationships between computer science, mathematics, and software engineering, along with discussions about the vague division between theoretical and applied regions of computer science, are an inherent problem within the field. See for example Dijkstra 1976; Graham, Knuth, and Patashnik 1994; Hoffman and Weiss 2001; Knuth 1968, 1996; Mahoney 1992; Priestley 2011; Tedre 2013; and Winograd 1991.
REFERENCES


This page intentionally left blank