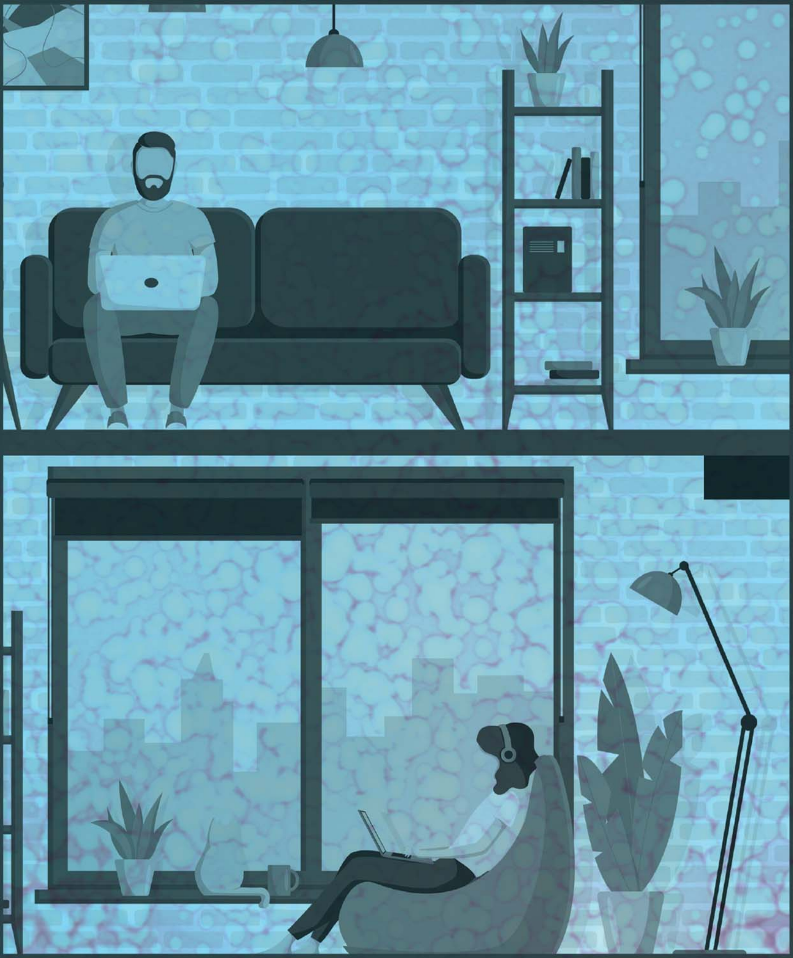
CODERSPEAK

THE LANGUAGE OF COMPUTER PROGRAMMERS



GUILHERME ORLANDINI HEURICH

UCLPRESS

Coderspeak

Coderspeak

The language of computer programmers

Guilherme Orlandini Heurich

UCLPRESS

First published in 2024 by
UCL Press
University College London
Gower Street
London WC1E 6BT

Available to download free: www.uclpress.co.uk

A CIP catalogue record for this book is available from The British Library.

ISBN: 978-1-80008-600-5 (Hbk.)
ISBN: 978-1-80008-599-2 (Pbk.)
ISBN: 978-1-80008-598-5 (PDF)
ISBN: 978-1-80008-601-2 (epub)
DOI: https://doi.org/10.14324/111.9781800085985

To the memory of Chris Seaton

Contents

Acknowledgements

Thanks to Chris Penfold at UCL Press who agreed that this was an interesting book to write. Thank you for being precise and fair throughout the editing process. I hope you can read my future books in one sitting as well.

Thank you to the Leverhulme Trust and to UCL for jointly funding the research that gave rise to the materials in this book. Being able to focus on an entire new area of research has been very valuable to me and it wouldn't have been possible without this funding. As Head of Department at UCL Anthropology, Professor Susanne Küechler truly encouraged me to develop the project further. Thank you, Susanne: your enthusiasm for the initial idea and your continuous support for this project have been truly inspiring. As mentor to this project, Professor Hannah Knox constantly nudged me to think carefully about what I was trying to say, to be open about ethical questions and to be firm in my opinions. Thank you, Hannah: I've learned a lot from you. Managing this project would not have been possible without the help of the best Finance and Research Grants Officer any university department in the world could have. Wendy Chandler, thank you. You've helped me so much, always in a kind and honest way.

Over the past few years I've been very lucky to share great moments, conversations and laughs with the most interesting scholars and thinkers. Thank you Ashraf Hoque, Dalia Iskander, Lewis Daly, Alison Macdonald, Carey Jewitt, Rafael Schacter, Tone Walford, Allen Abramsom, Ludovic Coupaye, Miranda Johannson, Timothy Caroll, Rik Aadrians, Martin Holbraad, Heidy Geismar, Sahra Gibbon, Helene Neveu Kringelbach, Aaron Parkhurst, Nicholas Lackenby and Sara Nila. Hope you enjoy the book.

Workshopping and exchanging ideas in various formats have been crucial to the development of this project. Thanks to Luis Felipe Rosado Murillo, Jan David Hauck, John Pelan, Mose Giordano, Ellen Webborn, Pontus Stenetorp, Jon Corbett, Ranjodh Singh Dhaliwal, Daniel Temkin, Sareeta Amrute, Hannah Knox, Ron Eglash and Chris Ball for their massive contributions to different iterations of this project. Thank you also to Brittany Martin for the invitation to speak on the Ruby on Rails podcast and to Professor Carey Jewitt for the invitation to talk about my research on the UCL Minds podcast.

Thank you to the British Library for providing peaceful, beautiful spaces in which one's thoughts can be shaped into words. Most of this book, from first draft to finished piece, has been written in the Asian and African Studies Reading Room on Floor 3. It's the best reading room, for there is a water fountain and a toilet nearby.

Thank you to the Arvon Foundation for organising a brilliant course on 'Writing Fiction and Non-Fiction'. The course provided me with the tools to re-imagine what this book could be; tools that I picked up from the brilliant workshops given by Will Menmuir and Kim Sherwood. Thank you to Ali, Chris, Anita, Kate, Jules, Jude, Jenny, Jazzy, Jay, Julia and Lis for a lovely week of storytelling and lifesharing. Hope we'll meet again in Darkest Devon one day.

Thank you to all the programmers who designed, created and continue to support the development of Emacs. Doom-Emacs has been my text editor since I found out I could write computer code and books in the same place. This research was put to together using Emac's Org-Roam and the writing has been done, exclusively, in it. If you like 'second-brain' tools, remember that Emacs will always be free and open source – whereas the flashy alternatives may not.

If it weren't for you, programmers who shared their stories with me, this book would have never been possible. You are the kindest, most generous people. You had the patience to listen to my questions and to trust me to handle your data with care. I thank everyone at Upstream, who not only had to endure my questions but also to review my pull requests. I thank the Ruby community, the most welcoming community of all, and I hope to have told a story that makes sense to you.

I was lucky to receive the support of many amazing developers during this journey. One of them, Chris Seaton, left us way too early. He taught me, among other things, the ways in which Rubyists embrace the chaos of computing. Thank you, Chris. I dedicate this book to you.

Writing a book has a been a dream of mine for a long time. I never thought the first one would be about programmers. Nor did I ever think it would be written in a foreign language. But I knew that no one would have supported me as much as Julia did in writing it. Our daughter Rita and you, Julia, are the most important things in my life. I wouldn't be anything without your love.

Introduction

There is a deck of cards in your hands. You need to organise it. You pick the first card – it's a Queen of Hearts – and you place it on the floor. (You should be sitting on an imaginary floor for this thought experiment to work.) You pick the second card, a Seven of Clubs. You put it on a separate column; you don't want to mix Clubs and Hearts. The next card is a Four of Clubs. Easy: put it before the Seven, on the Clubs columns. You proceed until the deck is finished and there you are: you have sorted the cards. Each card is now in one of four columns, from the lowest to the highest value.

Computer programs can implement this process of sorting cards (or anything else) in many ways. You did it by going over each element one by one and comparing them to the ones you had already seen. This process of going over each element and doing something with it, this action of picking a card and applying some thinking to it, is at the heart of one of the quintessential structures of the Ruby programming language: the 'block'. One way of doing this, in Ruby, would be:

```
card_deck.each do |card|
  hearts.push(card) if card.suit == 'Hearts'
  clubs.push(card)  if card.suit == 'Clubs'
end
```

A Ruby block is everything that happens between those two magic words: 'do…end'. A block is like a mini tool that you can re-use as many times as you want. It is 'ike one of those little Hello Kitty boxes they sell at the mall that's stuffed with tiny pencils and microscopic paper', as one Ruby programmer poignantly wrote.[1] In this analogy such boxes can be used in 'covert stationery operations', although 'blocks don't require so much squinting'. A block is thus a tool that implements what you wish to do to a group of things, be they deck cards or tiny pencils.

A programming language is made of two things: the language itself and its community. When Yukihiro Matsumoto started working on the Ruby language in February 1993, he had full control over the first aspect. 'Matz' – the name that Matsumoto-san goes by – had been searching for a language that would make him happy, but failing to find one. He wanted an object-oriented scripting language, but was not happy with Perl. Object-oriented programming languages rely on digital objects called classes, able to send and receive messages to and from other classes. Object-oriented programs thus consist of many different classes interacting with each other. But Perl, thought Matz, did not explore this philosophy enough. It seemed like a toy-language to him. He wanted a programming language that was truly object-orientated – one in which 'everything' was an object. Python, another language that already existed at the time, didn't work either. Python's object-oriented features seemed like an after-thought, a kind of add-on to the language. Like many other programmers before and after him, Matz decided to build what he couldn't find.[2]

It was another developer, however, who came up with the name. Keiju Ishitsuka suggested ルビ in a private chat message. 'Why Ruby (ルビ) ?' asked Matz. 'Because of Perl,' Ishitsuka-san replied. After a brief chat about what kinds of shellfish actually generate pearls, both programmers agreed on a provisional codename for the language. 'Indeed, Ruby is okay,' wrote Keiju. 'But Coral is also okay … tsk tsk,' replied Matz. After a few days Keiju wrote again to say that he had remembered that Ruby was his birthstone. Matz, now convinced, replied 'Ruby, then.'[3]

From the very beginning, Matz thought the language should be open source. There was no point in keeping it to himself and not allowing other people to use it. If he hadn't open sourced it, Ruby might have followed the path of many other languages that are now completely dead; languages that only lived in the machines of their creators. Thinking back on it, Matz says that Ruby only became widely used 'because of the decision to go open source, along with a fair amount of luck'.[4] That 'fair amount of luck' was the second thing that makes programming languages, the one over which Matz had little control: the Ruby community.

Making software open source does not automatically guarantee its longevity. Many open-source projects perish. In fact, most of them do – because they don't create a community around them. For an open-source programming language to thrive, it needs to generate daily interest and conversations around it. It needs to be picked up by developers, not just for personal projects but also for commercial products or academic experiments. A programming language without its community is just a collection of files destined to end up in the recycling bin. Still , the communal element of Ruby was outside of Matz's hands. He had no way of knowing whether the language would have a community at all. In less than 15 years, however, the language that he created would generate one of the most vibrant, fun, quirky, weird and successful programming communities that the software industry has ever seen. Ruby's success went way beyond Matz's expectations. A language that he designed for himself, according to his own personal taste, resonated with many people in Japan and beyond. People who got involved in creating the Ruby community.

One of the philosophies of the Ruby language (and of its community) is that there are many ways of doing something. Everything is open for modification in Ruby. Elements that most language designers wouldn't allow programmers to change, Ruby allows. With Ruby you are free to add what you want – although, as we'll see, not everyone agrees that's a good idea. Being able to extend the language is key to Ruby's philosophy, however, and blocks are a key structure for the programmer to exercise this freedom to extend the language. You can create your own way of operating repeatedly over data, just like you did with your card_deck. You can design blocks to iterate over things in any way you want. Blocks were even called iterators at some point, but they were soon freed from playing this relatively limited role. A block can do anything: that's why it is the one mandatory structure in the Ruby language.[5] If you think in blocks, then Ruby is for you. It fits your brain and it makes you happy.

The world of programmers and programming is immense. There are around 24 million people working as software engineers, developers, programmers, coders or just 'devs' in the world today.[6] How do you make sense of such a large and diverse population of people? As I've learned during the past few years of being in and among the people who inhabit the world of software, you have to cut the problem down to a manageable size. Reduce it to something that fits your brain, so you can describe it easily. You might not have the solution for that problem yet, but at least you know what it is. My way is the anthropological way. I've cut through the programming world with a conceptual knife that gives me a slice I can just about fit in my head: programmers who write code in Ruby. For almost two years I worked as part of a team of Ruby developers in a London company. I've learned from them and built things with them but, most of all, I've tried to listen to their stories. Through them and their connections, I was able to reach out to many other Ruby developers in the UK, Europe, the Americas and Japan. It is the combination of my experience as a Ruby dev and the individual trajectories of many other programmers that sits at the heart of this book.

Anthropology and programming might seem like very different ways of engaging with the world, but they share one principle: you should learn things by doing. Just as programmers value tinkering and experimenting more than a bookish approach to technology, so anthropologists favour learning directly from the people they study, rather than sitting in an armchair and reading about it. That is why anthropology's favoured method is ethnography. Ethnography is founded on the idea that to understand a particular group's experiences of the world, you have to do things as they do them. In other words, ethnography is anthropology's way of 'learning by doing' – which in the case of programming means embedding yourself in that practice. To do as programmers do, however, you need to program in a specific environment, given that the overall majority of programmers practise their craft as part of a team within a company. They build things together, share their knowledge and their differences, all the while creating a product. It is in this situation that an understanding of how programmers experience their world makes sense.

In preparation for this research, I went into training and learned to program in one of London's many coding boot camps. It was a tough, intensive, four-month long training course in which we not only learned the basics of programming, but also wrote real-world software applications. Ruby was the first language that we learned, followed by JavaScript, and most of us also explored other programming languages as well. From the very first moment Ruby became the natural choice as a research subject because of its easy to use qualities and its welcoming community. The boot camp experience not only gave me basic programming knowledge; it also allowed me to build the necessary connections to find a place to conduct my research. Through them I found Upstream[7] (a pseudonym), the company where I conducted the fieldwork for this book. Although access to corporate environments is notoriously tricky to negotiate – because companies are often suspicious of research – presenting myself as a researcher who could actually program helped me to get my foot in the door. It allowed me to experience what other programmers in the company did, to gain access to internal communications and to explore different relationships between management and programmers.

At Upstream, my week consisted of having two days entirely dedicated to programming – sometimes by myself, often with other programmers and always in conversations with engineers, testers and managers. I would spend the rest of the week chatting and interviewing programmers in the company, learning about the history of programming languages and studying Ruby code. It was a solid 18-month stretch in which I was able to expand the network of research participants, which started in London and ended up in Japan.

Much of my fieldwork was done during the Covid-19 pandemic, and therefore remotely. The pandemic hit, the research was ongoing and there was no turning back. I was lucky to have already been part of the Ruby team at Upstream. Negotiating access from scratch, during a global pandemic while everyone adjusted to working from home, would have been tough. My colleagues at Upstream knew me, and they knew about my research already. Even so, when I recorded private conversations – as I did with almost everyone in the team – I made sure to explain the research one more time, and to share with them an information sheet and a consent form.

Interestingly, Upstream's team of programmers already included many remote workers, even before the pandemic. A good portion of Ruby developers were outsourced from the Ukraine, and having people joining remotely was already part of the company's process. In this regard, adapting to WFH life was easier; we already had a constant 'Zoom' connection in our meetings, even before Covid. When some workers in the UK started returning to work, Upstream remained remote: there were no offices any more. As with many other companies, paying for office space made less sense, so the company decided to stay as it was. Although the pandemic certainly affected the interpersonal development of my relationships with people at Upstream, it was a reality that I couldn't escape from. I chose to continue instead of pausing this experience. In research, we don't get to pick the deck of cards: we only get to work on how we sort it.

This book is not a technical book on the language – there are plenty of those out there. This is the story of Ruby on the ground, of how programmers write Ruby to build software that is used by millions of people every day. A story that has repeated itself many times over in the histories of hundreds of small and large companies that chose Ruby as their main language. They chose Ruby and tailored it to their specific uses. Twitter, Airbnb, GitHub and Shopify are just some of them. Because Ruby allows for things to be implemented in so many ways, it is said that each workplace creates its own unique Ruby style. And just as a solution can be implemented in various ways, this book is only one among many possible ways to understand Ruby and its community.

Multiple, the Ruby way is.

Notes

[1] _why. ‘why’s (poignant) guide to Ruby’, 19.
[2] Matz. ‘[ruby-talk:00382] Re: history of ruby’.
[3] Ishitsuka. ‘[ruby-dev:5173] Re: to_i,to_s の素朴な疑問’.
[4] Matz. ‘The man who gave us Ruby’.
[5] Venners. ‘A conversation with Yukihiro Matsumoto’, Part III.
[6] /DATA. ‘Developer Nation Report 2020’.
[7] The company’s name is fictional and does not refer to any current or past existing software companies in London or elsewhere.

Part I
Open source

1
Principal engineer

In July 2019 I took the London Overground train to Liverpool Street Station and walked for about 20 minutes towards the Old Street Roundabout. This is the centre of London's Tech City, an area where many technology firms have their offices. The area has increasingly become one of the tech hotspots of Europe; developer numbers here have superseded other European capitals famous for their tech scenes, such as Helsinki, Berlin or Paris.[1] As I reached a small alley just off the main road, a smartly dressed 20-year-old walked past me. Blond beard, mustard and black North Face jumper, green trousers, white shoes. Yes, this must be the right place.

I walked up four flights of stairs and saw the big logo drawn on top of a double door: ‘Upstream’. I rang the bell and said ‘Hi, my name is Guilherme and I’m here for an interview.’ I tend to use my full name in these circumstances, although I quickly change to Gui – pronounced Ghee, like clarified butter – as soon as possible. People have always called me Gui. Someone buzzed me in and came to greet me on the other side of the double doors. The sun was streaming through one of the windows and illuminating the floor in front of me. What a lovely summer’s day, I thought. Am I really in London?

After a few minutes of waiting, a person with a long moustache and wavy short hair came over. ‘Hi, please come with me.’ I could barely make out what was said behind the enormous cookie sweeper, but followed along through a big, open-plan office. We walked past ten rows of long desks on either side. Three people on each desk, glued to their screens. It was about 11.00 a.m. and the working day was well on its way. As we veered left at the end of the rows of desks, I briefly glanced towards the back of the office in which a long table and a kitchen space were divided by a wall. We entered a small, enclosed office and sat down on two armchairs, facing each other. In the middle was a small table with a glass of water. The interview was about to start.

> 'I'm Charles, I'm the Principal Engineer here at Upstream … Hmm, we've reviewed your code test … So, why do you want to work here?'

I had rehearsed this question at home. Upstream is a start-up trying to change how people bought their groceries. It focused on organic and local produce made as close to London as possible. It sold exclusively online. I said I believed in the company's mission and that it sounded like the perfect place to work as a Ruby programmer. He looked at me with a slight grin. He knew I had prepared that answer. He seemed nervous, constantly looking at his notes. His nervousness actually relaxed me. He looked at his notebook and, barely lifting his eyes from behind his glasses to look at me, he said 'You have an unusual background … hm, why do you want to work in tech?'

He was right, my background as an anthropologist with 15-odd years of experience working in Indigenous Amazonian societies was unusual. I explained that I needed a break from academia and that I hoped working as a programmer could give me the experience I needed to design projects that could help Indigenous peoples in the future. 'Oh, so you don't want to work at Upstream for a long time?' he asked, grinning as he did so. Damn, wrong answer.

Before I could try to reply, he moved on. He asked me about different programming languages and what kind of applications people usually built with each of them. Suddenly someone else walked into the room, greeted me, introduced themselves as Muneeb and sat down on a high bench a metre or so away from us. It was strange. I felt watched, which I guess was the game they were playing. Muneeb asked me about the boot camp I went to and wasn't surprised when I said it was intense but amazing. I mentioned how there was a big focus on Test Driven Development (TDD), then I went off on a tangent. In very academic language I talked about TDD being counter-intuitive, making you write tests before the first lines of the actual program are on your screen. They looked puzzled. They were probably wondering why they ever let an anthropologist into the building.

I think it was my bizarre answer that led Charles to ask his next question: 'I notice you didn't use Rails in your code test.' Rails is the most popular Ruby 'framework' – a collection of software libraries that allows you to build applications by following an established pattern. Built on top of Ruby, Rails has a major influence in the community. Some say that the Ruby community and the Rails community are the same, which is probably going too far. 'If I had to guess, I'd say that 80 per cent of us do Rails,' a famous Rubyist and podcaster told me. But that depends on location: the Americas perhaps lead as the place with most Rails developers, followed by Europe, and then Japan probably at the bottom of the list. Japan, which is where Ruby was created, is perhaps the place where Rails is less influential and least used. 'They use Ruby for everything in Japan,' an American programmer who lives and works there told me. 'They even use Ruby to launch rockets.'

There have always been many alternatives to Rails – even if they never described themselves as such. Sinatra is one of them. A lightweight Ruby web framework, its logo is a drawing of Frank Sinatra's hat. It is notoriously friendly when it comes to building an application because of its minimalistic syntax.

Some people find that very beautiful.

```
require 'sinatra'
get 'index' do
  'Hello world!'
end
```

Sinatra creates bespoke routing functions out of well-known ways of accessing a webpage, like 'get' and 'post'. It uses that familiarity and merges it into Ruby's block construction. In just four lines of code, you've got something running: 'Hello world!' appears on a browser.

Things are different with Rails. Rather than creating the files yourself, Rails generates them for you. Quite a lot of them too. A simple, new application will generate 1,772 files. If you immediately spin up a server, the only thing those files will give you is the same 'Hello world' as the four lines of code of a Sinatra app. What's the trade-off, then? Rails, of course, creates many important things that you will probably need anyway: test files, database configuration files, basic front-end skeleton files, etc. There is nothing wrong with a framework that creates hundreds of files – in a sense, that's the whole point of using a framework such as Rails. You want the basic skeleton of things generated for you. However, you have to choose your tools carefully. Rails might be good if you are developing your company's main app – again, that's what frameworks are good for – but maybe Rails is too big a tool if you are only writing a small program for a job interview. I chose Sinatra for Upstream's code test because of its minimalism. It did the job for what I needed, a simple program to shorten URLs. Now I had to explain my decision.

‘I notice you didn’t use Rails in your code test. Why?’
‘Because Rails would be overkill.’

I realised I had made a mistake as soon as the words came out of my mouth. Upstream used Rails; they were probably expecting me to exalt Rails in some way or another. After all, they had chosen it as a framework and were (possibly) hiring me as a Ruby developer to work on the back-end of their main app – which was, of course, a Rails application. Sitting on that chair in an office in the middle of London’s tech city, I started to sweat. I closed my eyes only to hear Charles say, ‘Yeah, I agree, Rails would be overkill. Thanks for coming, we’ll be in touch.’

Wait, what?

Imagine you are shopping for groceries. You go to a shop’s website and start scrolling. Maybe you’ve got a list, maybe you haven’t. You might be in a rush. You’re definitely not very excited about having to do this. You start adding things to your basket, considering whether or not you should go for the ‘three for £7’ cheese offer. As you can imagine, a lot of things are going on in every action you make. Some of them are visual and obvious, like the little number on the right upper corner, which increases every time you add something to your basket. However, a lot of things are not that visible. What happens if you go on a mad binge and start adding a million cheeses? Will the shop have that many available? Will the website crash? If they don’t deliver, will they give you a refund?

My first task at Upstream was to implement a date field called ‘Available On’. It allowed us to add products to Upstream’s catalogue without making them immediately available on the shopfront. You could only buy something if the product’s ‘available_on’ was sooner than the date you were shopping for. I understood the task, but I had no idea how to implement it – this was only my first week and I still had to get used to the codebase. This included the main Rails app and several other smaller services, all built with Ruby. And that was just the back-end: the front-end was React, Swift and Kotlin, but I hadn’t been hired to work on that. (I would at some point touch some React code and break it.) As I flicked through different codebases, installed the necessary dependencies on the new laptop and spun the right Docker containers to make it work, I realised I would need help. This was a bit embarrassing. As they had hired me to do a job, why couldn't I do it? Why couldn't I find my way to start implementing a bloody attribute field? The answer had nothing to do with technical ability. This was the first time I felt an infamous programmer trait: impostor syndrome.

As I sat at my desk in one of the many rows of tables in the large open plan office, I decided to slide my chair across and ask Charles for help. He sat a few places away on the same row, so rolling on my chair all the way there didn't feel too strange.

'Charles, could I ask you something quickly?'

He seemed to be deep into something.

He took a deep breath and said 'Yes'.

He shifted his chair towards me and I briefly glanced at his screen. The resolution was so high – or the font was so small, or both – and I wondered how on earth he could read code like that. I told him about the 'available_on' task and he matter of factly replied:

> 'You need to look at the product files on the API, create a database migration for the field, then do something similar on the Product Catalogue app. Oh, and don't forget the serialiser.'

By the time I had finished writing down what he had just said, Charles was already back on his desk, reading tiny code on his screen. I worked it out in my head. The database needs a change to be able to store the values for the new field. The attribute needs to be created on both the API and the Product Catalogue to keep them in sync. The serialiser will format these attributes in a way that other apps can understand them. Brilliant. Charles helped me figure out what I had to do in just a few minutes. It didn't cost him anything. Or maybe I distracted him from something very important and it did actually cost him something. In which case he would never admit it.

Charles made me feel I could do that task without uttering any words of encouragement. He demolished my impostor syndrome by acknowledging me as a programmer. I was obviously just a junior, but that didn't matter to him. He treated me as an equal and ushered me in. Born the same year as me, Charles had heaps of experience already, but he treated me, a developer in his first job, as if we were on the same level. He was never condescending, never patronising. Even though sometimes it was hard to understand what he said because of his monstrous moustache, his words were always spoken with respect.

Well, almost always.

As I juggled multiple applications to do the ‘available_on’ task, I stumbled again. This time I didn’t even have to say anything. The swivelling chair rolling along towards him was enough.

> ‘Do I need to add the date field to the Basket app as well or is it enough to just have it on Product Catalogue and the monolith?’
>
> ‘Yes.’
>
> ‘Thanks, Charles.’

I rolled back to my desk, only to realise that he’d given me a non-answer. I was experiencing Charles’s clever humour for the first time. Some people loved it. I certainly enjoyed it, but a few people at Upstream didn’t. It was dry, very dry. He always had a clever comment to make if you didn’t use the precise and correct meaning of words. If there was room for ambiguity, he would point it out.

‘Do we need to implement this now or are people still debating?’ someone wrote on Slack once.

Charles’s response was ‘Yes’.

Like many other Ruby programmers his age, Charles suffered from ‘Post Traumatic Java Disorder’. After years of writing Java, he once described his transition into Ruby like a breath of fresh air. With Ruby, code was actually readable. ‘Suddenly you don’t have to have all these curly braces everywhere,’ he told me when we spoke about his early days in computing. ‘It actually reads like – not completely like English, but, you know, it’s a lot more readable than Java.’ All you did in Java was repeat the same bits of code in several places in order to make the compiler happy.

Java wasn’t totally bad: it had some nice things. It did all the memory management for you. You didn’t have to worry about getting a load of memory – actual physical rows of memory – and tell the computer you’d put your data in there. As Charles remarked

> ‘I could just say “here is an object” and then when I’ve stopped using the object, the garbage collector will clean it up. I didn’t have to worry about memory management at all.’

Java was crucial at the time. It was embracing Object Oriented Programming, taking an academic language like Smalltalk and making it more commercial. At a time when Python wasn’t object oriented and Perl didn’t do anything like that, Java seemed cool. But it left many programmers with a sense that, somehow, somewhere, code could be written in a more friendly way. For a lot of them, the friendly way was the Ruby way. Few people went back to Java after finding Ruby.

Charles found Ruby in 2014. The community was already known for being fun and playful, a welcoming place for misfits and whimsical people. It had conference titles like 'Keep Ruby Weird' and 'Ruby on Ales'. Such playfulness impacted everything, including the names that people chose for their software libraries. (A library is a bundle of computer code that can be used by other computer programs.) You had libraries like 'rake' – a contraption made of Ruby and Make, and not a gardening tool; or 'hpricot' – an HTML parser that tastes like an apricot. If contemporary software engineering relies on library managers to install and keep libraries in sync, it seemed that the Ruby community was taking the mickey.

In Ruby, at least since 2010, RubyGems has been the tool to add and manage other people's software in your own projects.[2] Accordingly, a Ruby library is called a 'gem' – in keeping with the mineral theme – and you install and download one by typing 'gem install [library_name]' on a terminal. So you can imagine how people go a long way to find an easy name that people can remember. You can't just call your library something like 'abhorrent_nameresolver3000' because you risk no one using it. People will just use some other library, like 'name_jam'. When Charles joined the community, it seemed to him that it was more important to find a funny name for a gem than focusing on what the gem did. He once wrote a library that organised a type of file called 'yaml' and called it 'befuddle'. He loved the idea that someone would type 'gem install befuddle' and install his library to befuddle their yaml.

I remember my interview as if it were yesterday. And I know exactly why. It is because of Charles, the role that he played at Upstream and how I became part of that small world. I felt I was accepted into a community that I didn't really know at all. Not because Charles was a gatekeeper of the community – he really wasn't involved in conferences or meet-ups, for instance – but because he treated me as someone he could work with, build things with. Later on we would spend a few afternoons trying to optimise a nasty database query or try to encrypt and decrypt information with the proper elliptic curve algorithm – but it was still the initial weeks of treating me as a programmer that gave me the confidence to believe that maybe, perhaps one day, I could fix broken programs.

I know that it was easier for me to feel accepted than it might have been for many people. Although an immigrant, it is certainly much easier for a white, middle-class Brazilian man who speaks good English to be accepted in the computational boys' club. Later in my research interviews, and also at Upstream, which had a fairly diverse team, I would hear stories from women and non-binary people of colour who have not only struggled much more than I have to break into this club, but have also had to work extremely hard to remain in it. The linguistic, gendered and racial make-up of the world of computing made it easier for me to feel at home among the geeks.

Charles wasn't around when Upstream started, but he blamed most of the problems with the codebase on one initial bad decision: someone had decided that it was a good idea not to build the application entirely from scratch. Instead, they would build it on top of a free e-commerce platform called Spree. They grabbed this existing open-source project and decided to expand it into their own version – they forked it. The idea was that once developers started working on the code, it would become increasingly detached from the original Spree fork. Ideally, the code would be different enough in future for Upstream to present it as a completely alternative solution. However, things sometimes don't go to plan. In this case, Upstream's developers never managed to get rid of the original Spree code. As it evolved, disentanglement didn't really happen. One app became many apps, and the curse of the initial fork was always present.

In a sense, Upstream's history mirrored the history of the Ruby community, which could never disentangle itself from the Ruby on Rails framework. Although there was never an intention to separate Ruby and Ruby on Rails – to some people, the communities are one and the same – the creation of the Ruby on Rails framework is one of the single most important events in the history of the Ruby community. 'Ruby would be a very niche language if it weren't for Rails,' Charles told me. Rails catapulted Ruby into the world and created one of the community's most enduring mythologies: that Ruby is the perfect language for building things quickly. The perfect language to use if you want to build, say, a blog in 15 minutes.

However, what made Ruby on Rails possible in the first place started many years earlier. It is, perhaps, quite hard to imagine the world of software development as it is today without this major development in the history of computing. A global, multi-faceted and participatory movement that coalesced around the late 1990s. It's something that every company today uses, perhaps without even noticing it. Companies such as Upstream would never be able to survive without it. Like old furniture that we have in our houses, sometimes we don’t even see it any more. It has faded into the background, become old news. It is known as free and open-source software.

Notes

[1] Quetteville. ‘Teaching tech’.
[2] Elmendorf. ‘RubyGems’.

2
Open source

'You can't talk about free software if you don't talk about what was happening in Seattle during the late 1990s,' Luis Felipe told me. He was an early Linux adopter in the mid-1990s and has been researching open-source communities for a long time. We've now known each other for over 20 years. I reached out to him because I wanted to understand the cultural history of free and open-source software. I needed the point of view of someone who knows a lot about anthropological attempts to understand software. In his experience, everything began in Seattle – a city whose name would echo profoundly in the Ruby community and in the world of free and open-source software.

'Is free software the same as open-source software?' I asked him when we met over Zoom to have a chat.

'They sometimes go together, but they mean different things. Free software is software that respects basic software freedoms. The word "free" is not about money at all, it's about writing code that people can study, modify, share and use how they want it.'

'And open source?'

'Open source historically has become more about the licence that you use for giving people permission to use, modify and redistribute software. The licence needs to say that a user may change, use and study that code however they want.'

'They sound pretty similar to me …,' I said.

'I know, right?' Luis Felipe laughed. 'First it was called "free", but then people decided to re-brand it as 'open source' to be more palatable to the corporate world. Open-source advocates sometimes emphasise 'permissive licences' that do not require for licensees to reciprocate when they take a piece of permissively licenced software.'

'Who changed it?'

'Oh, a bunch of people, but key names include Bruce Perens, Eric Raymond, Tim O'Reilly and Linus Torvalds.'

O'Reilly now owns one of the major media companies in the software industry, publishing mainly technical books. He was crucial in promoting the term 'Web 2.0' in the early 2000s, mainly as a way of changing the public perception of the internet. In 1998 the tech stock market collapsed, bursting a bubble that had been building for a while, and slashing the confidence of people in the new-ish internet. The 'Web 2.0' was a way of re-branding the internet, and open-source software was a crucial part of that re-invention. Linus, for his part, is famous for a few of his software inventions: Linux, the operating system, and git, the version control management system. They are major figures in the history of the free and open-source software movement – often referred to as F/OSS. The history of the movement can be traced back to the early 1980s, when ideas around freedom and computing started floating around the Massachusetts Institute of Technology's AI lab in Cambridge (USA). Ideas which, ultimately, derive from the development of the Unix family of operating systems in the 1970s at Bell Labs in New Jersey (USA).

There is, of course, a long history to the development of open-source software, but we can sum it up, for now, by saying that:

'It's basically BL and AL, isn't it?' I asked Luis Felipe.

'BL?'

'Before Linux and After Linux?'

He laughed, shook his head and said, 'Sure, let's go with that.'

'You were talking about Seattle, though. Do you mean the anti-corporate globalisation movements?'

'Yes, but also the creation of the Independent Media Centre.'

'Why that, specifically?'

The answer to that was complex.

> 'Because there was a lot of back-end and front-end software development involved in creating that centre in Seattle, which then trickled down to the creation of these "Indymedia" centres all over the world. It was a collective effort to create software that could be used by anyone for political purposes. There was someone who was involved with these guys who became very relevant in the way we think about software in anthropology: Gabriella Coleman. She started her PhD in 2000, in Chicago, and she got to know IndyMedia

> activists from Seattle. These political activists were very keen free software activists as well. Their stories are very much connected with the experience of folks in Brazil, who were in turn connected with other people that organised a global independent media space during the World Social Forum in 2003. In this community space, there were a bunch of machines that we installed that were running a version of GNU/Linux.'

I was interested in the role of Gabriella Coleman. 'And she comes to that event?' I asked.

'She comes to the International Free Software Forum as part of her study of the Debian community, which becomes her PhD research. And she starts publishing, years later, about the liberal bases of the free software movement. She connects this experience of creating free software with the liberal tradition of free speech, you know? As if free software was a of form of free speech.'

'How about the experience with Free Software in the Global South? I wonder what she thinks of that?'

This drew another complex reply from Luis Felipe, revealing the movement's international scale.

> 'Basically, she helped us understand that the free software movement is grounded in "liberal" sensibilities in the Euro-American context. The interesting and mostly unexplored thing is that we have different political and historical experiences with "software freedom" in countries such as Brazil. The F/OSS community also took different shapes in Japan, Mexico, India, Spain and many other countries. In our experience in Brazil, we were very much aware that we lived in the periphery of the periphery of capitalism. We would change, adapt and subvert everything that came from the metropoles in an instant.'

Coleman takes 'cultural liberalism' to be the main cultural staple of programming.[1] The things that programmers talk about, such as privacy and the power of the individual, have been central themes of the North American liberal tradition. What programmers – or hackers, as she usually calls them – have done is to rework those ideas in the context of computing. But what hackers do isn't only to repackage liberal ideas in the context of computing. There is also a communal, collective side to it.

> The lived experience of F/OSS hacking is more populist and communal, and at the core of F/OSS practice is an awareness of connection with a community of developers who make all code possible: the source code of others is easily available for use or re-use; source code repositories, Internet Relay Chat, mailing lists, bug tracking software and other technical applications facilitate all work; and all the while your fellow coders are at hand, ready to help when difficulties arise and willing to serve as an attentive audience to view and admire the finished product.[2]

Understanding these and other approaches to software is crucial to connect the history of open source to the experiences of contemporary software developers.

This was obviously a key point. 'The relationship between "free speech" and software wasn't as important in Brazil, then?' I asked Luis Felipe.

'Think about Richard Stallman and the Free Software Foundation. When he came to Brazil, it wasn't because people were importing everything he said about politics, technology and software. He was more of a canvas, on which people would project the discussions that we had with local governments about developing software as something that concerns local autonomy and technical sovereignty. What I mean is this: Stallman doesn't come to colonise us; he comes to be cannibalised, and for Free Software to be re-imagined as Software Livre according to our own political projects.'

I am still a little confused. So I ask, 'But it's not like the movement in Brazil was against free speech, right?'

'Of course not, but the problems we faced were different. We improvised and hassled to put things together. We tried, best we could, to resist the influence of corporations and to create our own things. Our approach, if anything, was anti-liberal. We wanted to put forward the political project that we cultivated, locally.'

When Luis Felipe was a teenager, Microsoft Windows was the king of operating systems. There was no Android and very few people had access to Macs. If you wanted to use a computer in the 1990s, you had to deal with the white clouds of Windows 95 or the green hills of Windows XP. 'Absolutely rubbish operating systems,' he told me, 'but they were everywhere.' Windows is still everywhere, but in the late 1990s the first versions of Linux appeared and everyone went nuts. A bunch of young people, in the southernmost tip of Brazil, erasing Windows machines and installing Linux.

In 2000 some Linux enthusiasts, alongside civil servants and the local Labour Party government in Porto Alegre, organised the first Free Software International Forum (Fórum Internacional de Software Livre, FISL). Richard Stallman was there, running around the stage with no shoes on. The State governor Olivio Dutra came, confusing everyone with a speech about 'force code'. He probably meant 'source code'. A year later, the first World Social Forum (WSF) occurred in Porto Alegre. Created as an antithesis to the World Economic Forum in Davos (Switzerland), the Forum would become one of the most important series of events in the history of grass-roots organisations. It was massive, especially for the development of the free software movement.

'There was one guy who was really influential around that time,' Luis Felipe told me. 'Eric Raymond. He published an essay in 1999 about Linux, open source, and free software.[3] He is a very controversial guy in the community because he took over the Hacker Jargon Files,[4] which had been created by a bunch of hackers at MIT. It's a collection of entries that describe concepts and ideas of hacker culture. It circulated around the networks and bulletin boards in the 1990s. Raymond became the first self-titled editor of the Jargon Files, though, and began to influence the free software movement quite a lot.'

'Influential through his essays?'

'Yes, absolutely. Because he was writing about being pragmatic and commercially orientated right at the moment that ideas around "open source" were put together.'

'The idea of open source starts to become a thing of its own – it detaches itself from "free software"?' I queried.

'Something like that,' Luis Felipe replied. 'The open-source narrative brought this idea that free software should be branded in a way that is more palatable for commercial enterprises. It was around this time that companies like Red Hat established themselves by making their own Linux distribution and their IT services a commercial success.'

In his essays, Eric Raymond writes that the hacker ideology of open source is like liberal ideas about common-law and land tenure. According to common law, there are three ways to occupy a piece of land: you take it, if no one owns it; you buy it; or you occupy it if it's been abandoned. 'This is equivalent to the general culture of licensing within the [software] community,' Raymond writes. 'The idea is that you own an open-source project if you found it, if it's given to you or if it's been abandoned.'[5] He agrees that hacker culture could also be more 'market-friendly', following the lead of Linus Torvalds, who spearheaded a new tradition within hacker culture. Raymond describes a 'pragmatist' tradition that combined technical prowess and commercial mindset, and that gently did away with 'the more purist and fanatical elements [of free software]'.[6]

From Google to the start-up that your cousin created last year, everyone uses open-source software. Undeniably free software dominates a good part of today's computing ecosystem. Upstream, the London company that I worked for and researched, used an open-source e-commerce software called Spree. Spree allows you to set a shop on the internet and sell your stuff. You don't have to pay anyone to use it, but there is an expectation that you might contribute to the development of Spree itself. There is a sense of community here, in which you are welcome to use and modify things, on the expectation that you should, if you can, fix any issues with the software and release an improved version back to the community. However, most companies don't spend a lot of time and money in contributing to the community – the tension that arises between companies and communities lies at the heart of this. While free software advocates might look at this as a conflict, open-source enthusiasts argue that private companies and the software community can benefit each other.

The debate around free software and open source is crucial to understanding the trajectories of people working at Upstream and the people who are part of the Ruby community. Ruby probably wouldn't exist if it weren't for the wider hacker community that grew during the early 1990s. Upstream wouldn't exist were it not for Spree, Ruby and Rails (which we'll come to in the next chapter).

However, one of the most striking characteristics of free and open-source software debates is that Raymond, in his essay, draws on an academic concept developed in France almost 100 years ago.

'The real problem, in that essay,' Luis Felipe told me, 'is how Raymond interprets the idea of the "gift".'

'Do you mean "THE" gift? Marcel Mauss's gift?'

'The very same one, yes. Raymond understands the gift economy from the point of view of American liberal economics,' Luis Felipe told me. 'He thinks people engage in a gift economy purely because it gives them an ego boost. An ego boost that raises your value, your …'

'Prestige?' I interrupted him.

'Prestige, yes. It's a very utilitarian reading of Mauss. And Mauss was launching an attack on utilitarianism, so it's very misguided, Raymond's reading.'

Marcel Mauss was a French anthropologist. About a century ago, he wrote an essay about gift exchange and how it works in society. This is still probably one of the most read essays in anthropology. His focus was not on the gift that your cousin Eric gave your brother-in-law Richard last Christmas. Society is not just a collection of individuals. There is something else, something more, that produces relations between people. Social life, he thought, should be understood as a system of relations. Individuals are, in a sense, the result of social relations, not the opposite.

Marcel Mauss writes about the social, contractual and economic logic of gifts that people give to each other. He is not focused on one specific gift exchange: his problem is not the present that Eric gave Richard. His problem is how this specific exchange fits within a wider network of relationships. Eric and Richard's relationship, given that they are brothers-in-law, doesn't include just them: it has to do with their families and with their class, their gender, their occupations and their race. In one gift exchange, all of that comes together.

'I don't understand how giving gifts to one another has anything to do with software,' I told Luis Felipe.

'Think about the free and open-source communities in general. What are people giving to each other? Sometimes whole programs, web frameworks, libraries, documentation, tips, tricks, direct and mutual help – but mostly they are circulating patches. I use your software, find a bug, fix it and send a patch to you, so you, the maintainer, can fix it for everyone, not just me.'

'Sure, but, if I remember correctly, Mauss is saying: "Look, there is very little freedom in the way that economics works. Receiving and giving gifts are not acts that people do willy-nilly".'

'True. People don't give and receive gifts out of the goodness of their hearts. It is not a tit-for-tat in gift economies. They are bound by social obligations. It's like they have to. It's not as free as it might look at first. It is, simultaneously, interested as in implicated in people's social web yet also disinterested, as in not being primarily motivated by individual needs and goals.'

We usually think of people and the objects that they possess as separate things. Objects are things: people are not. Objects can be made, shaped and given away; people cannot be made, moulded or parted with. This separation is at the heart of how world economies work: it's the idea of a commodity, something that can be extracted, moved, sold and bought in vast quantities – irrespective of the connections it has with a particular land or people.[7] Marcel Mauss pioneered a different way of thinking about the economy. He changed the perspective by questioning this core idea: what if we could never separate an object from its creator? If I write a piece of software, isn't there a part of me that is always in there? We might be tempted to say that this 'part of me' is just a projection onto the thing itself. It's not the software in and of itself, it's how I feel about it that creates that illusion of a connection. If I give this software to you, there is nothing of me in there: it's all yours now. Maybe. But then how do we explain the fact that people feel an obligation to give back? When we receive gifts, don't we feel we need to reciprocate, at some point, at some date? We do. But where does that come from? Mauss has an answer.

> What imposes obligation in the present received and exchanged is the fact that the thing received is not inactive. Even when it has been abandoned by the giver, it still possesses something of him.[8]

He is borrowing this idea from Polynesian societies, but he wants to make a point about every economy in the world. There is always a 'spirit' within a 'thing' – physical or not – that is being exchanged between people. It is something attached to me, the giver, and it will be permanently attached to me, even if you pass my gift along.

We can think about software in this way. Programmers talk about reciprocating gifts all the time. When they contribute to open-source projects, they often say they want to 'give back to the community'. Is the community asking them, individually? Not at all. Then why do they feel the need to give back? Where does this sense of obligation come from? It comes from the connection between the programmers who wrote it and the program they created. Mauss might have replied, it's the spirit in the machine.

'There was a point when I was reading everything I could about free software,' said Luis Felipe. 'And one person writing a lot about these communities and the kind of social relations they created through software was a guy called Chris Kelty.'

‘Weren’t you one of his students?’ I asked him.

‘I was. Gabriella Coleman put me in touch with him and he really supported what I wanted to do. At the time, very few people were looking into this relationship between anthropology and programming.’

‘Well, that’s still true today.’

‘It is.’

‘But how did your research interests fit with his?’

‘Chris connects well the history of software development with the growth of free software and open source as part of a massive cultural shift. It’s the creation of a social and technical collective – a “recursive public”, he calls it – that changes the way we think about knowledge and power. He uses this idea that we were talking about earlier, that code should be public and open to change, and he says that this changes how we see the way that private companies use software. There is an expectation, after open source, that code should be more out there, that it shouldn’t only be the property of a private enterprise.’

‘A cultural change that massively changes the economy of software, right?’ I ventured.

‘It does. What Chris does is interesting. He wants to understand software beyond a simple utilitarian thing. He is talking about a moral economy, you know?’

‘What do you mean?’

‘Utilitarianism is this idea that the individual is at the centre. Everything stems from individual action. Self-interest and rationality are supposed to be what people mobilise when they make their economic decisions. We calculate and reason, making economic decisions that are meant to maximise our happiness. Chris, building on some of the stuff that Mauss and other moral philosophers said, is very critical of this approach. People are not only self-interested individuals; they make rational economic decisions because they are also tied to social moralities and obligations that shape the decisions they make. And the ones they don’t make as well. The market itself is not the pure land of rationality; it is filled with rituals and myths.’

‘And how does that fit in what Chris is working on?’

‘His approach is very clear when he is talking about the invention of GPL.’

‘GPL?’

‘GPL is the licence that Richard Stallman, from the Free Software Foundation, and others invented. It’s not just one licence, but more like a whole bunch of them. The basic idea is that anyone can use, learn from, share and modify a piece of software under a GPL licence. It's where the whole thing of copyleft comes from, you know?'

The General Public Licence (GPL) emerged in 1989 as a way of unifying the software licences that existed at the time. At its core, the GPL states that a human readable version of source code must accompany any distribution of software. It all started when Richard Stallman needed to use some software which had been released free to use, but had, since then, been closed and sold off to a private company. Stallman wanted to make sure everyone could use a piece of software once it had been made available to the public. The genie could not be put back in the bottle.

'What was I saying?' Luis Felipe returns from his daydream.

'That the GPL is not a stroke of geniality that emerges from Stallman as an individual?' I suggest.

'Exactly. He is a great programmer, of course, but he didn't come up with this single-handedly, right? There was a very complex social situation that emerged out of a practical problem. He needed the driver to use for a video display on Emacs, but it had been created by a guy who sold it to another company and now he couldn't use it anymore. How do you solve that? And Chris, in his book, shows how the whole imbroglio between companies and programmers created a lot of confusion about who was contributing what and where. And this situation creates the perfect condition for GPL to be invented, because it takes the idea of personal property out of the equation. If it doesn't matter who owns a particular piece of code, since a lot of people contributed to it, then people can share and modify software more freely. And what Chris is pointing out is that behind any technical innovation lies a hugely complex social and technical process which we should be looking at.'

My conversation with Luis Felipe was crucial to understanding the links between open-source software and wider cultural issues. It made me think not only about how companies such as Upstream use free software, for example the platform Spree, to kick-start their own businesses, but also how small companies can rarely afford to give back to the community. They simply don't move resources into contributing to open-source projects, even though they benefit from them. On the other hand, many developers mentioned their desire to make more contributions. They feel the need to reciprocate – they sense the 'spirit of software' emanating from the code they use; they hear the echoes of the developers who wrote that code.

Unlike Eric Raymond, they are not in it for the prestige only. They wish to be part of this larger community that built things such as the

Independent Media Centres running Linux on desktop machines during a gathering of grassroots organisations such as the World Social Forum. A community that enabled many things to happen, including one of the most significant events in the history of Ruby: the launch of a web framework called Ruby on Rails. An event directly connected to the political and technological initiatives that a small bunch of Linux hackers kicked-off in the south of Brazil in the early 2000s. People that Luis Felipe has known all his life, who gathered to discuss the politics, techniques and economics of free software. It was there, almost 20 years ago, that Ruby on Rails emerged for the very first time.

Notes

[1] Coleman. *Coding Freedom*.
[2] Coleman and Golub. 'Hacker practice', 263.
[3] Raymond. *The Cathedral and the Bazaar*.
[4] Jargon File.
[5] Raymond. *The Cathedral and the Bazaar*, 65.
[6] Raymond. *The Cathedral and the Bazaar*, 70.
[7] Gregory. *Gifts and Commodities*.
[8] Mauss. *The Gift*.

3
The myth of Rails

Myths are not ancient history, not just stories of a long forgotten past. They offer a virtual space that holds a group's cherished values and bring their full weight to shape the world in which we live. Myths provide a backdrop against which you construct the present. They structure our world through narratives of events that could recur again and again.

On 2 June 2005, at 8:45 p.m. local time, David Heinemeier Hansson began a presentation that would echo through the Ruby community and provide one of its foundational myths.[1] It became inescapable. It captivated and brought people into the community; it made them interested in the Ruby programming language. The presentation was called 'How to build a blog in 15 minutes' and was later described as one of the talks that would make non-programmers think you are a magician.[2] In it DHH, as he is usually known in the programming world, presented to the world his framework for building web applications. A framework is a program that creates other programs: a meta-program, a program generator. It was called Ruby on Rails.

DHH uses Rails to create a fully functioning blogging website in just 15 minutes. By the end of his talk, users of the blog could post, edit, delete and update posts, as well as add comments to the posts of other people. He gave the talk during what was then the largest open-source conference in Latin America, the Free Software International Forum (FISL)[3] – the one organised by Luis Felipe, who we met in Chapter 2, and his friends. Hosted every year since 2000 in Porto Alegre, Brazil, the FISL conference had a crucial role in kicking off the debates around free and open-source software. At the time the city was also the host of the World Social Forum – the huge initiative that promoted local political movements as alternatives against the rise of neoliberalism, a politico-economic philosophy promoted by the World Economic Forum in Davos, Switzerland. In contrast to the World Economic Forum and neoliberalism, the World Social Forum upheld a ‘think global, act local’ ideology in which local political action was perceived as paramount in tackling global problems such as climate change, poverty and inequality. The free and open-source movement was very much in line with such initiatives, and the focus on fairer information technologies echoed those ideals. Participatory initiatives that helped to shape the emergence of ‘web 2.0’ – the new phase of the internet, in which people could interact and create things on the web rather than be passive readers of text in webpages. It is in this context that Ruby on Rails came to be.

DHH blazes through his talk to get everything done in 15 minutes. He needs to move fast to get everything finished in time, and many commands that he executes are only partially explained: code seems to appear out of nowhere. He creates a magical atmosphere. ‘Look at all the things I’m not doing,’ he repeats constantly, at one point warning the audience: ‘It goes fast, so don’t blink’. After every one of his ‘magic tricks’ in which code magically appears, DHH shouts ‘whoops’ as if to say ‘how did that happen? I don’t know, it’s magic!’ He says it so much that the talk itself is known as the ‘whoops’ video. Surprise, speed and magic would become his trademark over the years: quick-paced screen casts showing new features of his web framework. It is almost like he’s been re-building the same blog example over and over. He can’t get past it; he’s been swept along by the myth.

Now almost 20 years old, DHH’s talk inspired many programmers to try Ruby on Rails for the first time. Programmers today still mention it as the major influence at the start of their careers and one of the reasons they stayed in the Ruby community. They saw the magic in the presentation and stayed to figure out what was behind it. For some, the video was not only their first contact with Rails but their first contact with Ruby as well. A French developer told me about his experience of watching DHH’s video for the first time.

> ‘I looked at it and thought “Oh, wow, this is (written) in a programming language that I don’t understand anything about, but it looks a lot like what I’ve been learning about in Python. It looks really cool and there’s a whole framework which abstracts all the kinds of stuff that we’ve had to do over and over again.’

He was quite right. Rails enabled you to build an application quickly, with a neat logic and code that looked cool.

But why was the video about a blog?

If you were born after the internet, it may be hard to appreciate what the first iteration of the web was like. In its infancy, the internet consisted of text-based webpages, some animated content, images that took a while to download and basic search engines to find content. It was the era of internet relay chats, where you could create your own chat room – and wait for someone, anyone, to join – or join the chat rooms of other people. And then there was Gmail. Launched in 2004, Gmail was shrouded in secrecy at first. You could only get an account if you were invited by someone already there – the same with Facebook. And from that moment on, the web changed; it became known as 'web 2.0'. Or at least that's how big tech might want us to see it. Many of the technological developments that underpinned this new era came from the efforts of the free and open-source community. It wasn't the result of a single company's product. It was many people, building software together, collectively, and sharing the results.

To clarify what 'web 2.0' means, Tim O'Reilly wrote an extensive appraisal of what companies such as Google and Amazon were doing differently back then (in 2005.) Among other things, he praises the ability to harness the collective intelligence of the internet, the millions of user changes that happen all the time. Here he praises open source as 'an instance of collective, net-enabled intelligence'. He adds:

> There are more than 100,000 open-source software projects listed on SourceForge.net. Anyone can add a project, anyone can download and use the code, and new projects migrate from the edges to the center as a result of users putting them to work, an organic software adoption process relying almost entirely on viral marketing.[4]

'Web 2.0' is still the current era of the internet. Weirdly, the change from 1.0 to web 2.0 had nothing to do with technical updates of the internet's infrastructure. What changed was the way in which people started to interact with websites – something that we might take for granted now. From 2000 onwards websites became dynamic, allowing for more interaction with users. Instead of plain texts that you could only look and read, web 2.0 pulled us in. It told us to create our own content. Through so-called 'web applications', web 2.0 absorbed us all, and blogging became the quintessential thing to do on the internet. It was a way of putting your ideas out there without needing any knowledge of programming. Although web 2.0 started in the mid-noughties, things only really got going about eight or ten years later. By the early-2010s Ruby and Rails conferences could be found all over the world. In just a few years, many web tools and applications started using Rails and spreading the word. A few years later, it felt as though Ruby was taking over the whole (tech) world.

In many ways, Ruby and Ruby on Rails are the hallmark of the web 2.0 era. Not only because there was a lot of blogging, but also because, for a while, it seemed that everything on the internet was made of Ruby. Many of the applications that millions of people use today were created with Ruby on Rails, all of them around the start of the century. Twitter (2006), Shopify (2006), SoundCloud (2008) and Airbnb (2008) all use Rails – in some cases, as with Shopify, the main application is still a Rails app, 17 years later. Of course, some companies such as LinkedIn (2002)[5] have moved on from Rails towards other technologies. Granted too, many other companies of this era have no Ruby at all in their codebase. Google, YouTube and Facebook, for instance, have zero Ruby; they rely mostly on C++, Java and Python. But think about the programming infrastructure developed around that era. The two main applications where people store code – GitHub (2008) and Gitlab (2014) – are Rails apps, while the main package manager for MacOS – Homebrew (2009) – is written entirely in Ruby. No wonder Rubyists refer to this era as 'the time that Ruby was building the web'. It might not be technically true – there was (and probably there still is) more PHP code on the web than anything else – but for some people it felt as if the web was made of Ruby.

On the day that I was watching DHH's video for the zillionth time, I received an email from someone called Mike. He wrote that he had come across my research and he wanted to talk about the Ruby community with me. We set up a call. I explained the research briefly and gave him my usual spiel: tell me your story as a developer. Mike froze. I thought it was the Zoom call, but no, it was him. 'Hm, I don't really like to talk about myself,' Mike said. I'm so glad that he did.

Sometime in 2014 Mike joined a consulting company. They worked with start-ups and helped them to navigate the world of web development. It seemed to him that everyone at the time was using Ruby and its tools. They helped people at Twitter in fixing performance issues with Ruby. They navigated through many GitHub repositories built with Rails. They deployed everything with Heroku. As Mike explained

> 'It felt like this really magical time where all of these new tools were available to us. We were beating all these tools to death, and they

> were all working and pretty much doing what we wanted. And they were all implemented in Ruby.'

People working in these companies all knew each other. Some of the companies, like GitHub, had been invented during Ruby conferences. It felt like a small world.

> 'It was just a really amazing period of time where it felt like you were part of something. We felt like the cool kids. The kind of feeling you get … like, it felt really good, it felt nice to belong … like, I don't know how else to describe it other than it was a really exciting time.'

These days Mike works for one of the largest Ruby companies in the world: Shopify. Many Rubyists have recently flocked to Shopify. Rails and Shopify have been threading the world of web development for a long time. It is said that DHH gave Toby, Shopify's founder, a flash drive with the files of what was to become the first version of Rails. Today Shopify not only uses Rails and Ruby, but it has the money to pay a big team to work exclusively on the Ruby language and on the Ruby on Rails framework. They don't even work on company code. Some of them have actually never seen it. They just work to make the language and the framework better. No business logic, no deadlines for new features, no customers. Rather, other programmers are their customers. They write code to make it easier for other people to write code.

Rafael França, a Brazilian programmer, put this team together. He has been contributing to Rails for more than a decade and I reached out to him one day. I wrote the email in Portuguese as we're both from the same neck of the woods. Rafael told me that he hesitated to reply, as he doesn't really like to put his voice out there.

> 'I'm more of a laid-back programmer, I take it slowly. I never wake up before 9.00 a.m. and usually work until very late. I'm constantly working. I never speak at conferences, but I've been trying to change that. That's why I said yes to have a chat with you.'

Rafael told me that part of his shyness has to do with not being a native English speaker in a world dominated by this language. These language barriers prevented him from speaking at conferences, being invited to podcasts and sharing his views on Ruby, Rails and the communities around them.

At the beginning of 2012 Rafael made a new year's resolution: he would contribute to Rails every day of the year. After three months he became an official 'committer' – what open-source contributors are called. After seven months he became a member of the Rails core team. He continued on this path until the end of that year, then started 2013 on the same vibe. The next year, 2014, came and went; so did 2015, and it's now been 10 years of daily contributions to one single open-source project 'with the exception of weekends and the occasional holidays, of course'.

He heard about Ruby while at university. He had an assignment and tried doing it in Java, but it was too much boilerplate – too much repetitive code. He then found a Java framework which was inspired by Rails, leading him to investigate this new tool that he'd never heard of. That was only the start, as he explained. 'But what really brought me in was the community around Rails.' Rafael had heard of software engineer Martin Fowler and his book *Refactoring*.[6] As Fowler had been suggesting people take up Rails, Rafael decided to invest in it. That was in 2009, and the rest is (commit) history. However, it wasn't just the big names that brought Rafael to the community. At the end of the noughties he started working at Plataformatec, a major IT company in São Paulo. 'Plataformatec was where I started in the Ruby community,' he told me. 'There were people there who already contributed to the community, like José Valim.' Already a Rails core member by then, Valim would later create the Elixir programming language, directly inspired by Ruby and its community. Rafael acknowledged the importance of his influence.

> 'With José's mentorship, I became interested in open source. It was never a formal mentorship, but it created in me a sense of obligation … or maybe an objective, really, to contribute to a framework that gave me the will to work as a programmer again.'

It is amazing to think that sometime around 2010 two of the most impactful members of the Rails community, Rafael and José, were working in the same Brazilian company. And that both felt a sense of obligation to give back to the community. Marcel Mauss would be proud.

At the time, Rafael was very uninterested in his university degree in IT and was thinking of giving up, something that he eventually did. However, working with Valim and others gave him the spark that he needed to work as a Ruby dev and to make that fateful resolution on New Year's Eve. Rafael had tried to contribute to open source before, but he felt unable to access the communities around the development of the Linux kernel – the most successful open-source project of all. The Linux community was just too big and too scattered around to give him confidence to contribute. On top of that, there was also the language barrier. 'I am a shy man, and I never formally studied English,' he told me. If the Linux community was too big and English-centric – as most programming communities were, in fact – Plataformatec provided a friendly and contained environment that inspired him. It gave Rafael a way into Ruby and generated in him the will to give back to the community.

Rafael's words echo something that I encountered again and again in my Ruby journey. A developer's contact with the wider programming community is often through their work colleagues. In a sense, the community 'is' the company for which they work. Many developers have relayed to me how for them the Ruby community is the people they work with and the occasional newsletters they read. If they are lucky, like Rafael was, those people are also involved in the wider community (i.e. interested in open source, present at conferences, part of core teams, etc.) and that might inspire you to do those things as well. However, that might not be the case – which doesn't mean you won't still benefit from your local community. Maybe people give internal talks, perhaps there might even be a book club, like the one we had at Upstream. Most certainly the company's coding style will shape the way you work.

There isn't just one Ruby community: there are many, many Ruby (and Rails) communities. Perhaps not as much as one per company, but certainly more than only 'one community'. The reason being that Ruby managed to create a generative philosophy in which there are multiple ways of doing things. Incredibly, that is something that works in code and in the community, with each reinforcing the other. You can create a new framework because the language itself – its blocks – allows you to do that easily. The many ways of writing code blocks mirror the many shapes of the community.

At the 2020 Ruby World Conference in Japan, Matz and DHH hosted a Q&A about Rails and its relationship to Ruby. Being able to change something, DHH told Matz, 'is part of the wonders of Ruby for me. The core language allows someone else to come up with their own dialect that makes the language better for them. That is what I've done for 18 years.'[7] According to DHH, Ruby's malleability also applies to Rails: if you don't like something, you can just open it and change it. As he observes

> 'It means that what we have is not like a beautiful sculpture made in marble. It's made in clay. And if we want it to be different, we can just shape it different.'

'There's a sense in which there's a Rails community inside Ruby,' Andy Croll told me when we spoke over Zoom at the beginning of 2021. We'd met at the Paris RubyConf in February 2020, just two weeks before the first UK Covid-19 lockdown. Andy had given a talk about mental health and programming at the Paris RubyConf and I thought he might be a good person to talk to about the community. I introduced myself and told him I was planning a bit of research about the cultural values of the community. 'Oh, interesting,' he replied. 'I have many theories about the Ruby community.' Brilliant.

About a year later we continued our conversation. 'So, there is Rails and Ruby, but there are many offshoots that sort of almost define themselves in contrast to Rails,' Andy continued, set on his exploration of what he calls the 'concentric circles of the community'. He emphasises the importance of Rails, commenting

> 'There is the "dryer" part of the Ruby ecosystem, there is Roda, which is one man's brain turned into a framework [...] and then there are lots of alternative frameworks, but none quite have the heft of Rails in terms of mind share.'

In other words, if Rails is only one of many frameworks, it is also very much the most dominant one.

Rails has brought many people into the community and helped to shape the era in which the web was made of Ruby. It created one of the lasting myths of the community: that you can come in and create your own dialect of the language. You can pick up some blocks, twisting them into shape, and build your own clay house. As a myth, though, it also established the limits of the community. It is the place where the community tends to go, where it tends to sit. But what happens when one of the many ways of doing something becomes the dominant one? If Rails is 'the' web framework, can it ever be challenged? Rails might be made of clay, as DHH says, but what happens when the clay hardens and you can no longer shape it into something else?

In 2021 I came across a rant by Ukrainian developer Victor Shepelev on Twitter. I immediately reached out to him and we had several conversations about Ruby, Rails and his life as a developer. Among other things, Victor made me understand what the so-called ‘Rails magic’ was and why he disliked it so much. It confused things, he thought, because it made people think that Ruby and Rails are the same. His thread used the same metaphor that DHH used to describe Ruby – clay – but he had a different view on the matter.

> 1. Ruby’s unique proposal: find the way your ideas is expressed the best way, design your dictionary, combine it in different ways, see where the combinations lead you. It is the proposal of a modelling clay: you have it as a material and can go wherever you want.
> 2. Rails’ unique proposal: somebody took the modelling clay, created a language in a shape that was natural to them and gave it to you as a ready-made (baked) frame, with small holes you now can fill.
>
> (@zverok)[8]

Clay is one of the oldest materials humans have used. It can be anything. You can build a tiny pipe; you can build a large house. Because Ruby is made of clay, you can mould Ruby’s structures into different shapes, creating a ‘dialect’, a language within the language. But when a dialect looks so much like the language itself, when Rails became so dominant that people can’t separate it from Ruby, it becomes harder and harder for other projects to have their time in the sun. There is a problem when people look at Rails and think it’s Ruby, when they can’t tell the house from the clay.

As any potter knows, once clay is fired no change can be made. What comes out of the kiln remains. To Victor (and others), Rails has hardened the clay. Rails has covered the Ruby clay with its own glaze, marking it with an aesthetic form. The only thing left is to present it ‘ready-made’ and ‘baked’, as Victor writes. ‘Sometimes I wish that the Rails “hype” died to the ground and, for a few years, people stopped to try building water towers of modelling clay.’ Only when ‘it became half-forgotten’ could Rails be ‘rediscovered for its true value. But that’s just me being old and grumpy.’

Grumpy he may be, but the points Victor makes are crucial at a time when DHH’s public comments on the web have turned people away from Rails, and perhaps even from Ruby too. In April 2021 the so-called ‘Basecamp debacle’ took over Twitter. Jason Fried and DHH – founders of Basecamp – circulated messages within the company to discourage anyone to talk about politics in the workplace. The messages leaked. People were outraged, not least because DHH himself often talked about politics online. The result was that over one-third of Basecamp employees left the company, including people who had been there for over 15 years. In his summary of the main events, Richard Schneem, a long-time Rails contributor, wrote that he was 'anxious over the future of Rails and the Rails community'.[9] Schneem felt that the potential effects of the Basecamp debacle weren't being addressed by the community. A Ruby newsletter of that same week didn't mention it at all, simply adding 'Ruby is not Rails' as the newsletter's email subject. On another blog post, Eric Schultz wrote of his fears that the

> 'positive work in the Rails community on diversity and inclusion has been put at risk. The work by Rails Girls brought so many people into our community who would otherwise have not participated. […] In a practical sense, all of these problems boil down to one thing: Ruby on Rails is simply too associated with Basecamp and DHH.'[10]

Schultz makes a great point: this tight coupling between Rails and DHH/ Basecamp is not just bad for the community, but bad for Basecamp as well. Each of them should be able to change without having to worry about changing the other.

The debate is perhaps as old as the community; it flares up every now and then. It shows the diversity within the Ruby community, but also the grip that Rails has on it. Ruby would never be what it is today if it weren't for Rails. But Ruby is, in many ways, bigger and very different to Rails. Many prominent Rubyists have never worked with Rails, including Matz himself and other members of the Ruby core team, the people who maintain the language. 'There are a lot of alternative frameworks,' Andy Croll told me, and even though none of them have quite the gravity that Rails has, 'everyone is kind of interested in the ideas coming from these frameworks'. These different approaches highlight how the Ruby community is not 'a' community but many: concentric blocks with Ruby at the centre, with Rails forming the outer block and a whole bunch of others coming in between.

Ruby wouldn't be what it is without Rails – but can Ruby ever become something else if it wanted to? David Heinemeier Hansson's presentation on how to build a blog in 15 minutes showed to the world that Ruby allows for many things. It allows you to create your own bespoke language, a language that hides away everything that you don't want to show. At the end of the presentation, DHH showed that only 58 lines of code were written to get the blog up and running. At the time this was unheard of; no web framework could do this. Even today, few would give you so much with such little code.

The dominance of Rails is like a mirror that shapes the community. It certainly doesn't limit anyone to create other things – and the many blocks of the community are here to prove that. Yet it is a limit, in the sense that you constantly need to work against it. You can always develop new things, but Rails will be lurking in the background, perhaps waiting to say, 'That's nice, but I've implemented that already.' No matter what you do, Rails will be there. You can pile up the clay and create a brand new house. Then Mr Rails, your next-door neighbour, will come and greet you, saying

> 'Oh, that's a lovely little house. But come here, come see my house … this can be your house too … Old house? Oh no, this house is not old, it's almost new. It was only created 1 year ago.'

The rise of blogging is one of the celebrated features of 'web 2.0'. With a combination of new technologies such as RSS and permalinks, blogging has allowed the development of conversations between people writing on different pages, commenting and post about each other. It all may sound a bit dated now, but the 'blogosphere' was all the rage when Ruby on Rails was coming about. As O'Reilly observes,

> '‘[T]urning the web into a kind of global brain, the blogosphere is the equivalent of constant mental chatter in the forebrain, the voice we hear in all of our heads.'

If we move aside the brain metaphors, which seemed to be a bit too abundant today, it becomes clear that Rails emerged by harnessing the possibility of spinning up production-ready web applications able to tap into the heralded potentials (truthful or not) of the new era. My intention, however, is not to bust the myth of Rails. What we need is a new definition of myth; myth as a 'place' in time, a moment in which certain cultural developments came together – open source and the internet, blogging and collective conversations. Rails as a myth, as the best possible way of putting your idea out there – a bet that many companies, including Upstream, felt was worth a go.

Notes

1 Heinemeier-Henson. ‘Ruby on Rails demo’.
2 Visser. ‘What tutorials would make a non-developer think you’re a magician?’
3 Takhteyev. ‘Open source, open world’.
4 O’Reilly. *What is Web 2.0*, 2.
5 Hoff. ‘LinkedIn Moved From Rails to Node’.
6 Fowler. ‘Refactoring’.
7 Ruby World Conference, RWC2020 基調講演 2 David Heinemeier Hanson 英語.
8 Shepelev. ‘A long rant about Ruby’.
9 Schneem. ‘The room where it happens’.
10 Schultz. ‘Effect of the Last Week on Ruby on Rails’.

4
Half-broken monoliths

The icon you tap on your phone is an illusion. It looks like a single, contained thing, but is in fact multiple. An app is usually many apps. It might have started as one small application, but as it grows it splits into many. A large app begs for multiplication. The larger it gets, the harder it is to maintain it. Smaller things are always more manageable. The sheer size of a large application makes it more complex. It is much harder to keep the app's logic in your head. Breaking it down could, potentially, turn a big, complex thing into many smaller, simpler ones.

The process often goes like this. You create an app and start doing business. It grows, the app becomes a bit too big, and then you start splitting it up into smaller apps, each responsible for one domain or area of the original app. Ideally, if extractions go well, your app makes the transition from being a monolith to becoming several microservices. The key word here is 'ideally', because often this process doesn't go so smoothly. If it doesn't go to plan, you end up having to manage a few apps that still rely heavily on the original big app. In other words the process gets stuck somewhere in the middle, resulting in an app that has failed to become a collection of smaller services: a half-broken monolith.

'What are you working on these days, Dmytro?' I asked him, just to make conversation. I already knew the answer.

'Just doing API stuff,' he replied.

Dmytro's updates had been the same for the past few months. At Upstream we had an all-hands tech-team meeting every morning, known as 'stand-ups'. One by one, we'd tell the others what we were currently working on. Not Dmytro. He'd usually say the same thing: 'Just doing API stuff.' Sometimes he would go into more detail, but not often. I don't think he ever ‘stood up’ during our daily calls either. It was hard to tell as he’d always worked remotely from Kyiv, dialling in on the stand-up call. Maybe he did stand up, we just couldn’t see it. He was one of the software engineers from a team of outsourced Ukrainian developers working at Upstream. Over the years this team would grow to become almost the entirety of the company’s tech team.

Upstream programmers called the company’s main app ‘the API’. They sometimes called it ‘the monolith’ too. Any application can be designed as an API, or ‘application interface’ as the acronym actually stands for, but that didn’t matter at Upstream. Whenever someone mentioned ‘the API’, everyone knew that it referred to the original Rails application, built when the business started. Sometimes I had the feeling that the definite article ‘the’ also implied that the original application was ‘the’ most important one. It certainly was important because it did the bulk of the back-end work. But it was also ‘important’ because everyone complained about it all the time.

Over the years, the API had accumulated many responsibilities; it had simply grown too much. For many developers working on the front-end applications – the website, the iOS and Android apps – the API had become too slow. Painfully slow. This was a real problem, because the front-end applications had to go through the API in everything they did. The API provided the data they needed to display to the user. To some of these developers, the issue was not only that the API was bloated. They also blamed the language it was written in – Ruby.

‘The API’ started as a Ruby on Rails monolith. It was built on top of Spree, the open-source e-commerce platform. The idea was that the Spree code would be slowly replaced as the business and the code took shape. The Spree code was never completely removed, however, and it created significant obstacles when the time came to start breaking down the big monolith. Because Spree had been designed to be one single application, breaking it down would have already been difficult, even without all the extra Upstream code on top of it. With that additional burden it became almost impossible.

Building an app on top of an existing e-commerce application is more common than one would imagine. Dmytro explained how he saw similar issues in the new company he works for when we talked about his life as a developer over a Zoom call in 2021. ‘It is a Rails application based on Solidus, which is based on Spree, which is exactly the problem that Upstream had,’ he commented. They struggled to move away from it. A new service written in Go – a newish programming language – couldn’t be completely detached from it. He felt that people at Upstream – and in his current company – simply didn't put in the time to understand the philosophy behind Spree. 'When you look into it, it actually makes a lot of sense.' Even though understanding Spree's logic could have helped, however, to expand on Spree and go free style can be tricky.

'It sounds like a terrible idea to use Spree, Dmytro,' I told him.

'It's always that trade-off of choosing a system like that and then having to go with it … having to do something with it, having to start modifying it in a way that just means like, eventually, it's so tedious that in fact you're actually fighting against it, rather than adding enhancements to it.'

'Why do it, then?'

'I remember talking to people who were saying "Well, it got us to where we are now, so clearly it's been valuable in some way" – but even at the time there was a regret that it had been put in.'

The problem of a growing monolith is usually described as a problem of growth and entanglement. Apps grow too much and become harder to reason with; code becomes too 'entangled' and hard to change. If it's hard to keep all the different parts of the app in your head, then you're probably heading that way. You can read thousands of blog posts about this problem, usually with a preferred way of preventing or overcoming these issues. Microservices architecture – 'many little interconnected apps' – is often suggested as the antidote to having a big, clunky, monolithic app. Sometimes, a modular monolith – essentially 'a few apps within a big app' is also a chosen path. These are valid solutions, of course, and carry their own specific challenges. To an anthropologist, however, what is more interesting is how developers talk about these issues and how they live through them. How some things just seem impossible to solve, even if (or because) you throw more money and more people at them.

When Upstream developers described the process of creating new applications based on parts of the API, they often used sculpting metaphors. A monolith is like a stone. You carefully chisel some of its bits. A monolith is something that you 'break down', 'carve pieces out of' and 'extract functionality'. Such metaphors were slightly shifted when I spoke to Amir, an experienced programmer working at a major Ruby company. He reached out to me during the research for this book; he wanted to share his insights about the Ruby community. I was also interested in his life, of course, because it is from stories like his that a partial picture of this community could be composed. As Amir described his current work situation, I couldn't help but think about what I had seen at Upstream. He explained:

> ‘A big part of our [team’s] mandate is paying attention to the code base and making sure that if we continue on this track for another hundred years, those apps are still going to be malleable, they are still going to be pliable, and we can still ship stuff.’

To ‘ship stuff’ is to deliver new features to an application. Amir wasn’t talking about carving out pieces of a block of stone, but only because he was looking at it from another point of view. He wanted to avoid code becoming stone-like. ‘My main job,’ Amir told me, ‘is to prevent code from ossifying.’

Dictionary.com defines ossification as a process in which something ‘hardens like bone’ or ‘becomes inflexible in habits, attitudes, opinions’. I like the idea that computer code could become inflexible in its ‘attitudes’. It gives the cold letter of the code a human persona. There is a limit to how much we can – or should – think of computers and software as humans, but I think a little bit of anthropomorphising never hurt anyone. It reminds us of the fact that writing code is not a human-only activity; it depends quite a lot on the moods, the state and the history of that specific codebase. We write code, sometimes, despite the difficulties presented to us by the codebase, despite its affordances. In other words, we ignore some of the code’s opinions and we tolerate the machine’s attitudes, in order to change its habits.

Working with the API at Upstream was a constant battle with a codebase that seemed to be hardening before my eyes. I didn’t have a concept to understand it before Amir described it as ossification, and it made me wonder how that happens. How does a codebase start turning into bone? When an application gets too big, does that mean it has ossified?

‘Well, not necessarily,’ Amir told me. ‘I think it's a function of the size, but it’s also a function of inappropriate coupling going on.’

‘What’s inappropriate coupling?’ I asked.

‘It’s when two separate parts of an application have become intertwined. You change something in one place, and something breaks in a place that you didn’t expect it to. You can’t change part of your app without having cascading test failures everywhere else.’ I saw this happen a few times at Upstream: if you change one thing, something unrelated breaks. This is often the case with Ruby on Rails, says Amir, because Rails makes it so easy to design an app, to make it grow – but then ends up with too many things talking to each other. Yet size is also a factor in defining when a codebase ossifies. If you don’t know that touching something in one place will break something in another, then it might also be because the application has grown to the point where you can't hold it in your head any more.

'And that's when you need to make sure that you're paying attention,' Amir explained.

'It's not just small start-ups that struggle with this, then?'

'Not at all. I'll give you an example. Say there is an area of the monolith that we would like to scale up. Sometimes we can't because everything is so tightly coupled together and it's weird. Everything breaks, and then what we do is put some stuff in the secret drawer.'

'A secret drawer?' I had to ask.

He laughed. 'It's bad. Basically, anything that doesn't fit – and something else – goes in there. And so, of course, everything then depends on it, and it can call out to anything else. And okay, that's bad. So this year, what we're trying to do is we're trying to introduce some component boundaries within that component.'

It's bad, of course, but it does put things into perspective. If even his company, with its myriad of engineers, can't crack these things easily, it makes places such as Upstream seem more normal. As I talked to Amir, things started falling into place. I couldn't help but tell him that his description matched what I'd been seeing. I described to him how Upstream's main app had all these other apps dangling around it, all of them only partially extracted from the main application. I didn't know if Upstream's situation was common, but after his description of his own workplace, it certainly seemed less unique.

'You know,' I said, 'people talk about microservices architecture a lot these days. Some people are against it and want to defend the majestic monolith; others want to re-write everything into smaller apps. But I think some companies are just stuck in the middle, and it's very hard to move away from that. It's like they have half-broken monoliths. Does that make any sense, Amir?'

'What you are describing is what has happened in every single organisation I've ever worked for.'

A crucial part of any ecommerce application is to be able to handle different products, and to know how many of which one can sell at a specific point in time. At Upstream the logic to handle this was in a part of 'the API' called the availability service. The availability code had two main parts: a service class and a calculator. When front-end clients asked for the availability of some products, they would send the product ids to the API. The API inherited its way of handling products from Spree, where a product has many variants. Milk, for instance, can be a product that has two variants: a 200 ml bottle and a 400 ml bottle. Each product has an identification – an ID – and each variant has one as well. They were called spree_product_id and spree_variant_id. All these IDs are stored in the database and contain information regarding that product or variant: how many batches of the variant are stocked in the warehouse, what the average shelf life of the product is, what its dimensions are, etc. Both product and variant ids stored in the database had normal numbers: 1, 2, 3 …

> Milk bottle 200ml, spree_variant_id = 42
> Milk bottle 400ml, spree_variant_id = 43

On top of this relationship between a product-id and its variant-id(s), a service called Product Catalogue was created to handle the logic regarding purchasing bundles of products. In this new service, each variant was given a product_catalogue_id. However, this new id was not an ordinal normal number: 1, 2, 3, etc. Instead each product_catalogue_id was uniquely generated following an encoding format called 'Universally Unique Identifier' or UUID. These IDs are randomly generated. It is a safer system than a normal ordinal count of IDs because it prevents different things sharing the same ID by mistake. Using the UUID format, the chance of two things having the same ID is close to zero.[1] But having another set of IDs on top of the existing ones created more complexity. This in turn made it very difficult to understand them when it was time to extract the availability service from the API as a service on its own. If you wanted some information about milk, for instance, you would have to know three ids:

> A product id of ordinal type: '2'
> A variant id of ordinal type: '42'
> A catalogue UUID: '123e4567-e89b-12d3-a456-426610000242'

In 2020 Diego, Carly and I were tasked with implementing a new service to handle the availability of products. Following Charles's design, our job was to extract the availability logic from the API into a new service. Ideally, this would prevent the front-end clients from hitting the API with too many requests for the availability of products. One day, on Slack, Carly commented 'Hey! I think we should look at the legacy_product_id name as it's a bit confusing'. Legacy_product_id was yet another way of dealing with IDs. Supposedly, it would work as referring to the spree_product_ids, but by changing its name – signalling that it shouldn't be used from that moment forward. Supposedly, of course. Carly then suggested we should abandon it.

'What are our thoughts on using just 'spree_variant_id ?' she wrote. She also posted a message from Charles, from a direct conversation with him:

> the names we use for things are a bit of a mess. lots of bad decisions. the plan was to move away from spree and the idea of variants. in the product catalogue everything is a product, and variants are the relationships, whereas in spree a product is a parent object, and a variant is what we actually sell. but once we get rid of spree, we'll have to get rid of legacy_product_id, so we may as well have used the spree_variant_id.

'Once we get rid of Spree': I heard that sentence over and over again. The shiny light at the end of the tunnel: a Spree-free territory. We'll get there very soon, once we get rid of Spree. The promised land.

'The reason for spree_variant_id is the front-end and product catalogue are already using it,' replied Diego.

'That sounds good to me, especially if nothing else is using legacy_product_id,' wrote Carly.

We decided to keep using spree_variant_id as the main way of referring to our products in the new availability service. The product_catalogue_ids would remain in the product catalogue app, while the legacy_product_id wouldn't be used at all. Faced with the growing complexity of referring to the same things in different applications, we would continue to use the original IDs that had been in use since the beginning of time: the IDs we inherited from Spree. Sounded like it wouldn't be this time that we would 'get rid of it'.

The complex situation with IDs echoes a joke that some programmers at Upstream pointed out to me. It goes something like this. Two programmers look at the current situation of a system's design and see that there are three different ways of doing the same thing. 'This is absurd,' they think. 'We should be doing this in the same way everywhere. We need to develop a standard that covers all use cases and implement that!' The programmers agree that creating a new system to replace the current three-way system is the way to go and get to work. And that is how the fourth way of doing the same thing is created. In a conference presentation in Kyiv, Dmytro talked about monoliths, microservices and the problems of working with big Rails applications. He added more variations to the joke.

> 'You have a problem and decide to use threads. Two now problems have you.'
> 'You have a problem and decide to use functional languages. Now your problems are immutable.'
> 'You have a problem and decide to use Java. Now you have ProblemFactory.'

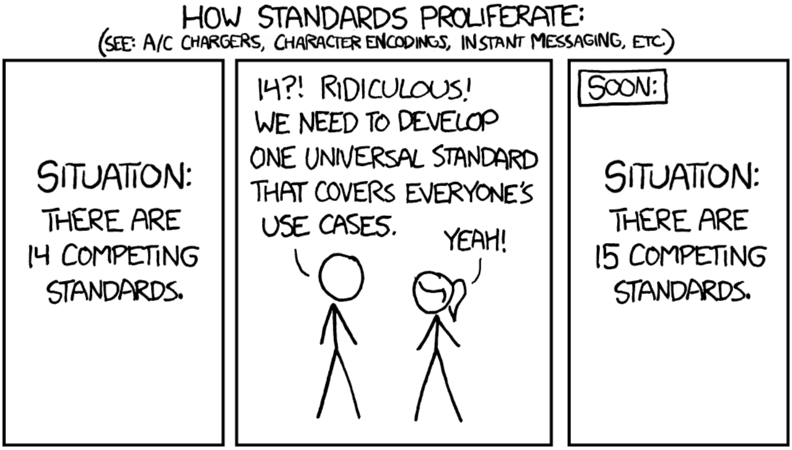

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 4.1 XKCD. 'How standards proliferate'. XKCD.com. CC BY-NC 2.5.

Battling with spree_product_ids, spree_variant_ids, legacy_product_ids, product_catalogue_ids and the id that we created for the new availability service sounded just like that. Sure, we weren't trying to replace any of the old ids, only trying to find a way of moving away from the API and the Spree logic. The question is, how do you do it? It is certainly not an easy task. You have to create some sort of continuity with the old system while at the same time creating something that is simpler and makes more sense in the new one. You must create boundaries, but you also need to maintain the connection. You need a fence, but you also need a gate.

Martin Fowler, one of the most influential software engineers and a long-time Rubyist, wrote that microservices architecture are not something so new; in fact, its roots go back to the way Unix was designed.[2] Unix is a system developed during the 1970s at Bell Labs which has influenced the field quite a bit, to say the least. A crucial component of Unix philosophy is the idea of modularity and Fowler thinks that microservices are a way of implementing that idea.[3] He mentions the move towards microservices by companies such as the *Guardian* newspaper, Netflix and Amazon during the mid-noughties, but stresses how there are also downsides to this architectural choice.

When everything is split into different processes, communication between them takes longer than if they were all in the same place. In addition, any change of responsibilities between components becomes harder to do, because now you are not just moving within the same process but also 'crossing process boundaries'. In this talk of microservices versus monoliths, it seems that the issue of boundaries is crucial. Boundaries are hard to enforce when a monolith application, and there are many out there, sits somewhere in between, half-broken. How do you enforce boundaries when there are so many holes in the fence?

We might be tempted to see a stalled transition from monoliths to microservices as a failure. If the goal of a clear transition from point A to point B didn't occur, it is easy to regard it as a botched attempt to succeed. This makes sense – but to see it as a failure requires that we adopt the point of view of an ideal. An ideal in which programmers could easily migrate a whole software architecture from A to B, from monolithic to microservices. Upstream's architecture, for example, would be very far away from that ideal and could be thought of as a failed architecture. But what if instead of thinking of half-broken monoliths as failed attempts, we consider them for what they are? Real-world architectures with which programmers must grapple daily. These are not failures, but realities. Majestic half-broken monoliths, if you will. They attempted a transition, but have struggled to complete it. Perhaps treating them as a more permanent feature, rather than just a transitional state, might help us to understand better what they are and how to work with them.

Upstream's architecture fits here too: different apps struggling to find the 'right ID' and to communicate with each other. My guess is that Upstreams situation is far from unique: many companies are probably in this situation as well, as Amir observed.

A boundary is usually thought of as something that creates a divide between two things, for example a fence that demarcates the boundary between two fields. It's interesting that Amir's work of preventing code ossification is actually one of creating boundaries. If enforcing boundaries is a way of preventing code from ossifying, then how do you create boundaries in a half-broken object? How do you fence off a shared space? Depending on the scale, the cohesiveness of software can change. What seems like a half-broken monolith from one perspective might seem like a rounded piece of code from somewhere else.

Notes

[1] Wikipedia. UUID.
[2] Fowler. 'Microservices'.
[3] On modularity, see McPherson, 'U.S. operating systems at mid-century'.

5
A new service

In 1974 Vint Cerf and Bob Kahn came up with a way of sending information from one computer to another by dividing it into small packets. Since then, virtually all information on the internet is passed around like that. Instead of sending a big chunk of 0s and 1s, which is what computer information is, you slice it into small blocks, send each one as a packet and put them back together at the other end. Small packets to be bundled up somewhere down the network. Sadly, conversations between humans never really work quite like that.

'Can you pass me those crisps?' I asked Charles at a Leytonstone pub sometime in February 2021.

'Things are not great at Upstream,' I said.

'I don't think we're hiring at my place at the moment,' Charles said.

Every time I complained about Upstream, Charles seemed to think I was asking him for a job. Upstream made Charles redundant a few months before. It wasn't long until he found a new job. He didn't like it that much.

'I'd love to hire you, you know.'

'Charles, that's not why I'm saying this. I mean, thanks, I appreciate it, but really, not asking for a job. Not that I wouldn't like to work with you.'

Oh God, what a mad conversation. Why can't we humans just be like transistors sometimes? Why can't we just process information in a clean and efficient way?

A few months before Charles left Upstream, he put me in a team to build a new program – a new 'service'. The new availability service. There was already a plan to build this, but it suddenly became very urgent when we started getting the same problems almost every day with the big monolith, the API. They were timeout errors. The API kept having, or 'giving', as they say, timeout errors. People would click on a certain product, add something to their baskets – then it would just spin forever. The spin of death. The reason was that the website requests for the availability of products weren't completing in time. They timed out, leaving clients with no data, no numbers of how many products were available that could be sold to the people shopping for their groceries. It's hard to sell something if you can't tell the customer that it's available.

'I think I know what it is,' Charles told me one day at Upstream. 'It's that damned sequel query. I'll show you.'

He sat down and I slid my chair over to his desk, like I'd done so many times. I had used SQL (i.e. 'sequel') before; it's a raw language, used to get stuff out of the database. Every retail app has a database, a place to store data about your products, users, orders, etc. This information is spread out into different tables, each one with columns and rows. Products table, variants table, stock batches, inventory and merchant shipments tables. It will probably also have information about orders and deliveries, so you have tables that hold data on line items and delivery slots. Then you use that data by building connections and relationships between these tables. You join them, in various ways. You join all the users that live in a certain postcode with all orders above £100, for instance. It's like you're applying a filter, telling the database which bits of information you need at a particular time. You 'query' the database. As Charles opened the file with the SQL query he wanted to show me, I tried to recollect what I knew about SQL. It wasn't much.

'Here it is,' he said.

```
WHERE merchant_shipments.id = $1
     AND (
          spree_products.held = false
        OR
          (
            spree_products.held = true
          AND
            shipment_line_items.quantity > 0
          )
         )
     AND spree_variants.is_master = false
     AND…
```

The query was 214 lines long. It had loads of things written in caps, like WHERE, AND, OR, SELECT, FROM. It read as a sequence of steps, each of which had a condition attached to it. Get the shipments that have this id, look at the available products in there, make sure we've got some of them, make sure the variants are not master variants. Wait. What the hell is a master variant?

'That's just something we inherited from Spree. But, you know, once we get rid of Spree, we won't have to worry about that,' Charles explained.

Oh, great. Once we get rid of Spree.

Charles and I spent a few days on that query, trying to make it better. We rewrote bits of it, extracted things to other queries, did things to the make it faster. We actually made it slower. How hard can it be to find out how much of a product is in the warehouse? Surely this shouldn't take longer than me going there and counting it myself? It wasn't that simple. Conditions that were inherited from Spree, combined with bespoke business logic, were forcing the query to timeout. There didn't seem to be a way around it, it was just too hard to find out how many products we actually had available. There were too many things to check and, over the years, they had all been coupled together in such a way that it was hard to disentangle. It was time for a new service.

'I'm putting you in a team to build the new availability service, Gui.'

Whoosh, impostor syndrome. A new service, me?

'Diego and Carly are going to lead it, but you'll do most of the work.'

I see.

So it was that, after only a few months at Upstream, I was placed in the team that would build a new service. The new availability service. The team was made of three: Diego, Carly and myself. Carly would leave quite early on. She had bigger fish to fry, or so it seemed to Diego and I, who remained until the end. The others had both joined the company after I did, but they were way more experienced than me. Diego had worked in several different companies and had been made redundant a couple of times. Carly had only worked for a couple of places, including a start-up that became an enormous company in only a few years. While we tried to figure out exactly what we needed to extract from 'the API', the necessary discussions on how to implement the communication between different services started as well. Could this be an opportunity for us to experiment with something new? Maybe a different programming language, perhaps some other form of sending information between apps?

While setting up the new service, Diego talked to other people in the wider tech team about what they had done while implementing other services. After talking to Muneeb about how the basket service had been implemented, he came back to tell us on the messaging platform Slack:

> Diego 11:22 AM:
> So after speaking with Muneeb. The Basket uses dry-container and dry-auto_inject. Dry-rb have since come out with dry-system which uses both of the gems mentioned and more – https://dry-rb.org/gems/dry-system/0.12/ – Take a look and let me know what you think? I really like the way dry-system boots files only when you need them making the booting of the service much quicker as well as the tests and the convenient plug-ins.
>
> Carly 11:34 AM:
> Worth giving it a go I reckon. Regarding the file loading, I guess it may be desirable to have different behaviour in dev vs. production similar to how Rails does eager loading of files in production but not in development by default. So that in prod the app can boot fully before it starts accepting requests, for better performance.

Building a new service from scratch is quite an exciting thing. It is not something that developers do very often. Most developers these days work for a company for two years and then move on. In that relatively short time, it may be that no new services will have been implemented. Building a new service is exciting because there is a bit of room to implement or work with technologies that you might have wanted to work on for a while but hadn't yet had the opportunity. Again, most developers come for two years and work on whatever they need to work on: if it's a Rails app, then Rails it is; if it's a pure Ruby app, then that's what it will be; if it's JavaScript, God forbid, then you'll have to work with that.

In her book about Indian programmers in Berlin, Sareeta Amrute observes that 'coding can be a tool to extend and think through human possibilities'. She also notes that many programmers use different strategies to 'carve out spaces and times in the office to pursue their own coding projects that allow them temporary ownership over their work'.[1] And Martin Fowler writes, 'You want to use Node.js to stand-up a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine.'[2] There are options, but there are trade-offs.

'Should we write it in Elixir?' Carly suggested to Diego when we had a chat about the new service.

‘Why?’ he replied.

‘I really want to learn it. And it would be faster than a Ruby app.’

‘Sure, but what happens after you leave the company? Who would maintain it?’

It was a good point. Someone else could potentially learn the language, but that takes time – something that companies probably won’t have.

‘I think we should stick with Ruby. But maybe we could use a different framework?’

We eventually settled on a combination of different gems in a pure Ruby app – in other words, we wouldn’t use any web framework such as Rails at all. While Diego worked on setting up the basic skeleton of the app and Carly focused on setting up the deployment pipeline, I worked on creating the JIRA tickets. JIRA is a project management tool and creating the task tickets was a classic task job for a junior developer like me.

The tricky thing about building this service was understanding firstly which information the new service needed to get from where, and secondly which information it needed to send to whom. There was no clear plan laid out and we had to chase people for answers. After a meeting with the chiefs, Carly came to relay some of these answers to us on a Friday afternoon.

> Carly 5:43 PM:
> Re the order events – after exchanging a few more messages with Muneeb earlier and running it past Charles, a meeting may not be necessary unless you guys have any concerns about Muneeb’s suggestion. Charles seems to think it sounds OK.
>
> The suggestion being that we don’t consume the events published by basket service, but instead wait for orders service to be live so we can consume the new order events it will be publishing.

Neither Diego or I replied during the weekend, so she asked again on Monday.

> Carly 9:58 AM:
> Morning guys, what are your thoughts on the above? Does this sound fine or do you think we should try and avoid the dependency on the orders service going live? (I don’t know how realistic the 1–2 month timeline is.)

> I suggest we go along with this for now and we can change our minds later if it's looking like orders service won't be ready in time
>
> Diego 10:05 AM:
> I'd like to set up a call with front-end clients Andy, Muneeb and Charles, so we can discuss all the points we have. [edited]
> The orders service also involves front-end clients.
>
> Gui 10:10 AM:
> Hey guys, good morning Given that basket is currently unstable and that Muneeb is the only person dealing with that and given that he is the only one working on the order service, I think it's unlikely this will be finished in the next couple of months
>
> Diego 10:13 AM good to know:
> Lets see what we can get out of them in the meeting.

At this point, maybe a month into the building of the new service, things between Diego and Carly started to get a bit difficult. They had the same experience, but Carly had a better job title. Diego told me once that 'just because of her title, she can decide things and I can't. And I'm the lead in this project!' That was true. Diego's title was not a Lead Engineer – someone who manages people – but he was the 'lead' in this project. Soon Carly started taking over, having meetings with Charles and Ivan by herself, completely side-lining Diego. He and I often felt that we were not part of any decisions, even though there were only three of us working on the new service. Then one day, Carly wrote on Slack that Charles, Ivan and she 'agreed that we shouldn't wait for the order service to be ready'.

Diego got really upset. Not being invited to decide on this was too much. He reached out to me and we had a call.

'Every time I talk to someone, everyone has a different opinion,' he told me.

'About the plan for the new service?'

'No. Well, yes, that too, but like the order service, for example, only yesterday it was mentioned we should be using that and now we're not. It's getting very confusing.'

'But what about the plan?'

'I've always said the plan for end of June is to move the calculator to the new availability service and then implement the warehouse management system. But no, people wanted to wait for the order service to be ready.'

‘Are we going to?’

‘I don’t know. That’s why I’m saying we should get everyone on the same call.’

He had a meeting to go to, so we hung up. Later in the day he DM’ed me about it:

> ‘It seems like no one knows what the order service is. So it would be a good time for all of us to know what it is and how it links into the current architecture. (edited).’

The issue with the order service was a terrible distraction – it took our attention right away from the task. In particular, it took Diego away from feeling in control of his own project. He messaged me again the next morning.

> Diego 10:11 AM:
> Morning. I’ll find out the hell is going on with this order service. We’re doing it … we’re not doing it … we’re doing it … we’re not doing it … we’re doing it.

We were not doing it. The availability service would have to rely on the current state of things – which was probably what we expected at the beginning. We lost a month on the whole order service shenanigans, and we still had to decide how we would send data from the new service to the other apps. The conflict and the lack of communication between the two senior developers certainly damaged the project. It created a state of confusion and uncertainty that would last until the end. But it wasn’t just the leadership of the project that had communication problems. The code we were writing did too.

Conway’s Law – a famous concept in software engineering – states that there is a predictable relationship between the structure of communication within an organisation and the systems that such an organisation designs. The way that people communicate with each other is replicated in the way they design anything that needs to have communication channels. In other words, if five people are working on a project, the project will have five different parts.

> Examples. A contract research organisation had eight people who were to produce a COBOL and an ALGOL compiler. After some initial estimates of difficulty and time, five people were assigned to

> the COBOL job and three to the ALGOL job. The resulting COBOL compiler ran in five phases, the ALGOL compiler ran in three.[3]

We built the availability service to plug into the existing way in which apps communicated with each other at Upstream. They used something called Kafka. Kafka is a messaging system – a way of transferring data between two places, using the internet. Kafka can handle a high volume of messages and it works in a distributed way: there is no central structure that holds all the messages, but instead there are several. LinkedIn built Kafka in 2009–10 to handle the complexity and size of its databases. They open-sourced it in 2011. LinkedIn had different systems with different databases that needed to talk to each other. They named it after Franz Kafka to describe the (Kafkaesque) situation they were trying to escape from: difficulties in communication.

So far, so familiar.

Kafka was always a controversial topic at Upstream. Some people liked it, other people hated it: everyone used it.

'I hate Kafka so much,' Diego told me one morning after our daily stand-up. 'I just don't understand why people want to use it. It's impossible to test, you never really know if the messages are being published or consumed. It's just too much of a black box to me.'

'Can't we just use something else? Can't we just do like normal API calls and stuff?' I asked, trying to sound clever.

'I wish, but I don't think we can really. Kafka is everywhere in our system now.'

It sounded like we needed a solution to get out of this Kafkaesque situation that Kafka had created for us. Talk about naming karma. But as Diego said, there wasn't much point in debating. We had to suck it up and just do it, and the result was that I spent months writing code to handle Kafka 'events'. A Kafka event is something that an app wants to share with the world (of other apps in the system). If 'the API' wants to tell everyone that a new stock batch has arrived at the warehouse, it publishes a Kafka event. If a new order has completed, the Basket service will probably want to broadcast that to the world. Other apps will be on the lookout for events like that. When a new stock batch is in, the availability service would need to know and change the stock numbers accordingly. After an order has been finalised, the availability service would also want to hear about that, and probably diminish stock numbers accordingly.

You need quite a bit of code to handle this back and forth of messages. I spent weeks building these message handlers. Because it was so repetitive, I made mistakes all the time. I didn’t particularly hate doing it. I didn’t have a strong opinion about Kafka. It sounded like a clever way of handling a bucketload of messages, although it was tricky to test it. What actually made me angry was having to continue using Spree. In this new service, not only did we have to keep using it; we had to put it back in. One Kafka handler, for instance, replaced any mention of a ‘legacy_variant_id’ variable with a ‘spree_variant_id’ variable.

```
module Kafka
  module Consumers
    module Handlers
      class VariantUpdated < Base
        include Import["repository.bucket"]

        KEYS = %w[legacy_variant_id].freeze
        MAPPINGS = { "legacy_variant_id" => "spree_
        variant_id" }.freeze

        def call(payload)
          bucket.upsert(
            prepare(payload),
            unique_by: %i[spree_variant_id in_hub_
            at sell_by]
          )
        end

        private

        def prepare(payload)
         keys = payload.slice(*KEYS)
         keys.transform_keys { |k| MAPPINGS[k] || k }
        end
      end
    end
  end
end
```

We were, literally, putting Spree back in. But hey, once we get rid of Spree …

Diego and I worked on this project for months. We wrote about 5,000 lines of code. We tested it, the testers tested it and it seemed to work just fine. There were data discrepancies, but that was to be expected, given that the new service could never be exactly on par with the current one. It just didn't have all the historical data. We did import it, but there were things in the data accumulated for years in the API that we just couldn't replicate. Slowly, we started to lose confidence in the work we had done. The new availability service provided results which were very close to the old service inside the API. But they weren't the same. They were very close, but not equal. And we needed equal.

The project was called off.

Months after the project was shelved, Diego and I had a call to chat about what had happened. Diego blamed Kafka for it. It bloated the codebase – adding loads of repetitive or very similar code. Kafka was also horrible to test; you never knew if the messages were being sent and received properly. Diego didn't like how Kafka just created a whole load of code that needed to be added to the codebase (which always means more code to maintain).

'You know, you have the consumers, you have the events, you need to encrypt the message. So, then, you start having loads of other crazy like file structure of the payload that needs to go into it. You need 4/5 different files to actually send the message itself. Yeah, I just, I can't see … We are either using Kafka wrong at Upstream – we're not using it how it's meant to be used – or I just don't like it. Or both.'

'I get that, but I don't understand why we ended up shelving it? I know that there were some data discrepancies, but they were minimal, and they were to be expected as well.'

Diego suddenly paused, and asked:

'Do you, do you actually don't know?'

'I don't, no.'

'So, you know how difficult it was to keep all the data in sync, right? There were so many ways of allocating the variant on specific days, there were the block-listed days, the multiple places it was causing so many problems … I did mention to Ivan that I didn't feel comfortable with the service. And can you remember how many times we kept finding new things that people didn't know about?'

Of course I remembered, and it was bizarre. People just used the systems in so many ways that it was hard to understand it.

'To me, personally,' Diego explained, 'I felt like there were so many unknowns. There are still so many unknowns. I mean, does it give you confidence that even people in the company don't know how certain things work? Does that make you feel confident?'

I told him that we had managed to tackle the edge cases, that it was hard to keep in sync before launching it, that I felt that we could have gone ahead, it would have been hard for the first month or so, but we would have dealt with it when it was live. He disagreed.

After so many months of hard work, it was frustrating not to see it go ahead. Diego lost confidence in himself because communication within the team was so poor. We lost confidence in the code we were writing because its different parts weren't passing data around correctly. Communication was a bit off; data was a bit off. We didn't go ahead, and the new availability service would remain what it is now: dead code never to be used – 5,000 lines of it.

Notes

[1] Amrute. *Encoding Race, Encoding Class,* 22.

[2] Fowler. 'Microservices'.

[3] Conway. 'How do committees invent?', 28.

Part II
Meta languages

6 Language dreams

There were usually around 20 developers in Upstream's 'tech team'. When I joined, roughly half of the team was outsourced from the Ukraine. Not too distant from the UK time-zone wise, with a rich computational tradition and costing half the wages of a London programmer, no wonder tech companies love the country. Dmytro told me once that virtually no one in the Ukraine works for local companies: everyone works remotely for foreign ones. He was the team lead for outsourced employees at Upstream. 'I sleep about three hours a night,' he told me. 'I'm always working.' Over the years, the number of outsourced developers at Upstream increased. The Covid-19 pandemic, by making remote work more acceptable, reinforced that trend. To me, the opportunity to listen to the life stories of Ukrainian developers was crucial, even though this book is not focused on international labour dynamics.[1] It was crucial because it spoke, in a clear way, about the dynamics of human and programming languages in the world of computing.

Upstream's team was divided into smaller teams, each responsible for a different area or application. One group worked on developing Android applications, another focused solely on the website and so on. The biggest section of the team provided back-end functionality that enabled other applications to run – this was where I worked, as part of the Ruby back-end. Overseeing all this, a quality assurance team of engineers – known simply as 'QA' – made sure that any change in the code base wouldn't break things and cause disruption for users or customers.

One of the QA developers was called Oleksandr. A Ukrainian, he lived and worked out of Kyiv and considered both Ukrainian and Russian to be his native languages. But the computer code he reads, analyses and tests every day is written in Ruby. Of course Ruby is not based on Russian or Ukrainian, nor does it have Japanese keywords. When Matz created Ruby, he based it on English – like most programming languages in the world.

Over the past 70 years, English has come to dominate the world of computing. It has achieved this to such an extent that one needs to look hard to find programming languages or computer systems that are not based on English syntax, grammar or vocabulary. From a language designer's perspective, creating a programming language in languages other than English has become not only impractical, but perhaps even unthinkable. When the developer Ramsey Nasser created قلب, a programming language that uses only Arabic characters,[2] he quickly realised that even though the language is perfectly good for creating any computer application, modern software development requires that your language interacts with libraries and tools that already exist. The majority of these are written in English.[3]

English permeates computing all the way through, from the content of webpages to keywords in compilers. More than half of webpages are written in English; Russian comes second with about 7 per cent.[4] This dominance is not restricted to the words on webpages; it extends to the level of operating systems and, of course, to computer source code. The barriers created against non-English languages can even extend to the characters that you type, given that software will often have a hard time dealing with non-ASCII characters. The American Standard Code for Information Interchange (ASCII) is a character encoding standard for electronic communications; developed in the 1960s, it supports 128 characters from the Latin alphabet. It wasn't until the early 2000s, for instance, that Windows started supporting Arabic fonts, while Word, PowerPoint and Excel only fully supported Arabic and Hebrew in 2016.[5] In Ruby, for instance, you couldn't name a Class with non-ASCII characters until 2018.

Throughout his working life, Oleksandr has used programming languages written in one of his native languages, Russian. However, programming in those languages makes him feel uneasy. There is something strange and eerie in those languages – something that makes him cringe. Compared to languages written in English, those languages don't look serious to him. He feels embarrassed by them. As he told me

> 'I have my own experience because I speak Russian. And I know that we have at least one popular programming language in Russian which is called 1C and it's terrible, and people keep laughing at it. I don't know how that feels for native English guys, but in Russia,

> when you code in your natural language, you just start laughing. They cannot read it. … Their minds start blowing up because they read it as a natural language. But it's not natural. It's technical. So what the fuck?'

Well, what the fuck indeed.

How can you feel embarrassed by reading code written in your own language? What would it take for something like that to happen? How can natural languages and programming languages intersect and mix in such a way?

Programmers usually focus on technology, but they should think about the people using their language as well. As Matz declares: 'We are humans, we are people, so we have minds and feelings.' His advice for anyone contemplating designing a programming language is that they should focus on human psychology as well as on the technical side of things. Rubyists love to say that the Ruby language is exactly like English, that it reads like English. They say this makes code more readable, which should always be a good thing. According to Matz, 'humans invented programming languages because there is a limit to the human understanding of machines'. If we could understand what machines say, if (only) we could understand binary code, we could just write everything in machine language. But we don't and we can't – not at least in any productive way. And so we need high-level languages such as Perl, Python or Ruby.

Larry Wall, who created the Perl programming language in 1988 and is one of Matz's programming heroes, says that Perl is much closer to a human language than most computer languages. It tries to access a deep, fundamental level. This fundamental level is the way that people use language and expect it to evolve over time: the myriad of ways in which language is used.[6] Wall recognises the diversity in speech communities and inserts that into the community spirit of Perl. He explains that a good language designer should take that into account; they should also 'stay out of the face of the programmer' and give programmers many ways of expanding the language. Let them use it as they please. This is the role model to which Matz is so attracted. The leader who can give something to the community and then step back and watch it grow. Ruby adopts this view and translates it to the notion that there are multiple ways of implementing something.

But it seems that Ruby also wishes to access a different level: the way people feel about that language, the sensations they have in using it to write code. This is something that Oleks also manifested, though in a rather different way. If people in the community often talk about Ruby's ability to 'fit our brain' or to be the 'language of our thoughts', Oleks's feelings suggest that this has something to do with Ruby being written – and then having been shaped, over the years – to be intuitive for people who speak English. 'We speak natively,' says why the lucky stiff, one of the most prominent historical figures in the Ruby community; but perhaps only if we speak English natively as well. 'It means that you don't have to make a lot of effort to use it. It's like using a second language, but that language is actually English,' a member of the Ruby Core Team, the group that decides what happens to the development of the language, told me. Coderspeak is not a language that you are born with – no languages are. Coderspeak is something you learn.

Language designers often point to the other languages that inspired them to create their own programming language. In the case of Ruby, the languages that are usually invoked are Perl, for the philosophy of doing many things; LISP, for the focus on high-level functions and closures; and SmallTalk, for the use of Object Orientation as the main architectural philosophy. However, I think we can add some other languages to the mix as well. Languages that are perhaps a little bit less talked about these days, languages such as FLOW-MATIC, BASIC and Self. Along with Ruby, these languages share the dream of creating a programming language that is as close to English as possible. The reasoning is that this would make them more approachable – which is why we need to add a little something to understand how languages such as Ruby fit the brain of programmers. Something that is as important as neuronal connections in the brain, but lies at the surface level of Ruby's syntax. We need to sketch a brief history of programming languages that dream of being like English.

Programming languages only became known as 'languages' after the creation of FORTRAN in 1957 at MIT by John Backus and his team. Before that, they were known as machine code. At first, 'language' meant the language of engineers, as FORTRAN's syntax is more akin to algebra than to any human language. The success of FORTRAN is tied to its ability to hide the specific workings of the machine behind a code that allowed engineers to solve their own problems. It allowed them to focus on themselves, not on the machine – FORTRAN somehow echoing Ruby before Ruby even existed.

Computing historian Paul Ceruzzi has observed that 'Fortran's success was matched in the commercial world by COBOL (Common

Business Oriented Language).’ COBOL replaced algebraic symbols with English commands. For instance, the sign ‘>’ became ‘GREATER THAN’ – and the idea was that managers and businesspeople would be able to read the code of the programs they were using.[7] Jean Sammet, one of the minds behind COBOL, says that the companies involved in developing the language were quite proud of it. However, that didn’t stop companies who were not involved from having a ‘smug attitude of “I could have done it better”’.[8]

When FORTRAN was being developed, Grace Hopper and her group at the Rand corporation developed a system called AT-3, which later became MATH-MATIC. It had the same intentions as FORTRAN: a language to write code like mathematical equations. In the event only FORTRAN survived, but Grace and her team had something else under their belts, FLOW-MATIC. This has been described as ‘“the first English-like language for business data processing […] released in 1958’.[9]

In 1953 the engineering department at Rand Corporation in New York proposed to management that mathematical notation should be used for mathematical programs, whereas English statements should be used for data processing programs. The department also offered to create a compiler for each. Almost two years later, it was done.

> We got our little compiler running. It wouldn’t take more than 20 statements. And on the back of this report, we put a nice little program in English. And we said,
>
> > "Dear Kind Management:
> > If you come down to the machine room, we’ll be delighted to run this program for you."
> >
> > And it read: INPUT INVENTORY FILE A; PRICE FILE B; OUTPUT
> > PRICED INVENTORY FILE C. COMPARE PRODUCT #A WITH PRODUCT #B.
> > IF GREATER, GO TO OPERATION 10; IF EQUAL, GO TO OPERATION 5; OTHERWISE GO TO OPERATION 2. TRANSFER A TO D; WRITE ITEM D; JUMP TO OPERATION 8. REWIND B; CLOSE OUT FILE C AND D; and STOP.
> >
> > Nice little English program.[10]

The team at Rand quickly realised that the program was a bit too short for the big budget they had requested. They therefore wrote another program, which translated the English words into French, and sent a similar letter out: “Dear Kind Management …”. Next they wrote one in German, but this time the management got suspicious. How could the computer understand German? ‘That hit the fan!!’ remembered Grace Hopper in a speech. ‘It was absolutely obvious that a respectable American computer, built in Philadelphia, Pennsylvania, could not possibly understand French or German! And it took us four months to say “no, no, no, no! We wouldn’t think of programming it in anything but English”.’[11]

The language they came up with, FLOW-MATIC, came with a brochure. It promised to take you ‘directly from Flow Chart to Finished Program’, the main selling point being the English-like pseudo-code that would generate the program to be stored in a magnetic tape: ‘To program a new application, the user merely describes his systems flow chart in the English-language instructions of FLOW-MATIC’. But the number one ‘unique saving’ of the language was that it eliminated almost all your ‘coding load’ by shifting the ‘emphasis of programming effort from detailed coding to problem definition and systems analysis’.[12] The idea of reducing ‘coding load’ through a friendlier interface is a characteristic of FLOW-MATIC that echoed in subsequent programming language design.[13] In other words, the idea was that English-like syntax reduces the mental overload and allows you to think about the problem. Fewer braces (in the code), more spaces (in your head).

Invented at Dartmouth College and closely linked to the evolution of time-sharing computers, BASIC is another language that dreams of being understood by a wider audience. The desire to build it came from the realisation that people with little computing experience make the key decisions in business and government. Yet how could they make correct decisions about computing if they couldn’t understand a word of the code used in those machines? The development of ASCII and cheaper terminals allowed researchers at Dartmouth to build ‘a system that would be friendly and easy to use. We had absolutely no doubt about the easy-to-use part,’ wrote one of BASIC’s designers.[14]

They chose commands that encouraged non-technical people to use the computers, such as HELLO and GOODBYE. They felt these were essential if the number of BASIC users was to grow. A crucial idea was that the code actually generated by their program would be totally hidden. They could write their program, compile it by typing RUN, receive neat English error messages if anything was wrong – all without ever looking at the outputted code.

Let’s take a look at an early BASIC program.

```
Instr. no.     Operation    Operand
10             LET          X = (7+8)/3
20             PRINT        x
30             END
```

We can see right away that it doesn't go as deep into the English-likeness that FLOW-MATIC, COBOL and other languages were going for. It's a table, a spreadsheet, rather than a sentence. But, in a sense, creating something that reads like a sentence wasn't their main aim. English was a tool to do something else: to get more people into computing, to create a community. 'We did not design a language and then attempt to mould the user community to its use,' said Thomas Kurtz, one of BASIC's designers. This is the reason why hobbyists could write their own compilers and change anything they wanted; they were never restricted by a language specification.[15] A philosophy that echoes again in Larry Wall's notion of letting the programmer be, of not forcing things into people's minds. Ruby would follow a similar approach to community building.

FLOW-MATIC, COBOL and BASIC shared yet another trait: the dream of breaking communication barriers between programming and non-programming groups. For the first two, perhaps because the languages were designed inside a company, the aim was to create a language that would unite these two groups. As the FLOW-MATIC brochure says, they wished to create a world in which 'flow charts and codes are made intelligible to the non-programmer as well as to the programmer'.[16] FLOW-MATIC and COBOL wanted managers and programmers to be the same, or at least to speak the same language.

Today most programmers would agree that businesspeople shouldn't be too close to the logic of the applications. They believe that there should be a neat separation, a wall – usually provided by product managers – that shields the development of applications from the development of business ideas. Business should specify the needs and features, but the details of implementation should remain in the programmers' hands. You can ask any programmer on how becoming one means unbecoming the other: how managing usually means not having the time and head space to focus on programming tasks. Programmers and managers will never be the same.

BASIC was different – which is why, by the early 1980s, it had become the most widely used computer language. BASIC aimed at bringing 'ordinary people' into programming by 'removing unnecessary technical distinctions', by providing defaults when the 'user doesn't care' and by showing that 'error messages can be made understandable'.[17]

This sounds a lot like Ruby, and something starts to click when we realise that BASIC was in fact Matz's first language. Ruby is not for managers at all, but it is a language to bring people into programming. It is a language for programmers, one that 'reads (almost) like English' and that wishes to reduce the mental load through a friendly syntax. But if FLOW-MATIC went too far on this – it conflated programming languages and natural languages; it wanted them to be the same – Ruby seems to have got it just right. The key to having a programming language that is easy on the mind and the eyes is not in having it look exactly like English. It's sufficient that it only resembles English, that it pretends to be English.

Computers pretend all the time. They pretend to be sentient, to work perfectly, to think about you, to make you feel wanted. Pretence is at the heart of computing. That's not necessarily a bad thing. Programming languages also pretend to be things. They can pretend to be English, for example, by mixing tokens from the language with programming structures. Ruby does so by using special keywords that echo the language, such as unless – perhaps the only programming language to use that word. Along with other English tokens, unless allows you to write Ruby code like this:

```
return card_deck.first unless card_deck.empty?
```

In other words, give me the first card of the card_deck unless the card_deck is empty. Ruby programmers use expressions like this one all the time. It is very much one of the preferred styles of writing contemporary Ruby – it is idiomatic Ruby. It is because of function names that are 'like English', such as 'first', 'unless', and 'empty', and a syntax that resembles English that Ruby programmers often say that Ruby reads almost like English. The key here is 'almost'. It creates a connection between something you know (English) and something you want to master (Ruby) – but at the same time it adds just a slight bit of difference. Almost English.

Rubyists talk about their language in ways that you wouldn't expect from a programmer. They say thinks like 'Ruby fits my brain', or it allows me to 'express myself', or it's easy to 'translate my thoughts into code'. 'The logic I saw in my head,' says the Ruby programmer Avdi Grim, 'transferred into program logic with a minimum of interference.'[18] Of course, programmers and machines read programming languages in a very different way, but Ruby doesn't work on the computer's terms. It is not designed to accommodate the computer, says the programmer called why the lucky stiff. We shouldn't even call Ruby a computer language, he writes, because we are not using it to translate what the computer thinks. We are not 'foreigners, seeking citizenship in the computer's locale'.

> But what do you call the language when your brain begins to think in that language? When you start to use the language's own words and colloquialisms to express yourself. Say, the computer can't do that. How can it be the computer's language? It is ours, we speak it natively! We can no longer truthfully call it a computer language. It is coderspeak. It is the language of our thoughts.[19]

Somehow the ways in which Ruby programmers think and wish to express themselves is echoed by the language's design. This link between Ruby and its programmers, between language syntax and thinking, runs deep in the community. Coderspeak: a language that is neither human nor 'machinic' but actually both. A language that pretends to be both; a language that builds on a sense of kinship between what happens in a computer and what happens in people's brains, emerging in the late 1970s as part of a rising 'computational culture'. This culture afforded new ways of linking information processing, computer programming and the human mind.[20] If the 'mind' of a computer works in binary terms, that is, if a computer thinks in 0s and 1s, human minds seldom work like that. Human values are better described with labels such as 'almost' or 'not exactly', rather than with straight answers. That's not a problem, it's just the way it is.

Although much has been written for and against the idea that our brains are like computers, there is something more interesting in the way that humans learn languages which in turn applies to the way we learn programming languages. In this sense, it brings them together. The most important thing when learning a new language is that two different 'people' can exchange symbols in a way that enables the learner to put themselves in the perspective of the other.[21] What matters is that one person can feel that there is an actual exchange. In other words, that the programmer can have an appreciation of things from the computer's point of view.

We might think that's not possible because we can't understand machine language – but we are not talking about understanding when talking about learning a language. In some ways, it's impossible to understand things from someone else's point of view. It's theirs, it belongs to them. But that doesn't mean we can't have an appreciation, a partial understanding, of what it feels like. That's what well designed error messages are for. That's what English-like keywords and symbols are for. We can exchange them. If the programmer why the lucky stiff questioned how Ruby could belong to a computer if we can speak its words and colloquialisms, it is precisely because such words are colloquialisms; they are part of a conversation between programmer and computer. They sit somewhere in between the two. They are something in between, a mix-in. Ruby is not a computer language, but nor is it a human language – it's both. Well, almost both.

The idea that coders had various ways of 'speaking' pointed to the linguistic diversity that we see in natural languages: the different accents, registers, languages and dialects. But it also pointed to a connection between humans and machines. Some sort of underlying language of languages, that made it all possible. The discussion was all getting very meta. I needed to understand a bit more about this whole business of languages and how they function in society. I needed to bounce this off someone, so I reached out to another academic friend.

Notes

[1] This was before the Russian invasion of Ukraine in 2022.
[2] Nasser. 'The قلب Programming Language'.
[3] Nasser. 'A personal computer for children of all cultures'.
[4] Lawrence. 'Siri disciplines'.
[5] Stanton. 'Broken Is Word'.
[6] Wall. 'Larry Wall: Why Perl Is like a human language'.
[7] Ceruzzi. *Computing*, 62.
[8] Sammet. 'General Views on COBOL', 345.
[9] Sammet. 'History of IBM's technical contributions', 522.
[10] Hopper. 'Keynote address', 17.
[11] Hopper. 'Keynote address', 17.
[12] Remington. 'Introducing a new language for automatic programming', 3.
[13] Marino. *FLOW-MATIC*, *Critical Code Studies*, 146.
[14] Kurtz. 'BASIC', 519.
[15] Kurtz. 'BASIC', 534.
[16] Remington. 'Introducing a new language for automatic programming', 4.
[17] Kurtz. 'BASIC', 535.
[18] Grimm. 'Confident Ruby', 2.
[19] _why. 'why's (poignant) guide to Ruby', 32.
[20] Turkle. *The Second Self*, 229.
[21] Tomasello. *The Cultural Origins.*

7
Meta-programming

Language is not just an underlying cognitive structure in the brain. Language is a social activity; it is bound to a social context. Language is in essence communicative; it is about what we say to each other and how, and when, we do it. In programming, communication also abounds. The difference, perhaps, is that conversations happen not only between coders using human languages, but also between them and their machines. Coderspeak, the language that connects them, is made of all that: the human chatter, the programming language syntax, the social and the computational contexts and the meta elements that make it all possible.

Language has different functions. There is the literal meaning of words and the potential poetic effect of them – but there is also the social context that binds an utterance. For example, to what extent can we separate the language we use to write our software in from the wider political context of the world? Can we even consider writing software in anything other than English? Who would use that software if we did? Even programming, a small example of how we do things with words,[1] is integrally connected to the social lives of languages. Everything is context-sensitive and everything is tied, in some dimension, to a language ideology:[2] the understandings that speakers have of language in relation to the political and social context that surrounds and shapes the way we communicate to each other.

'It's not just about the prevalence of English in business contexts,' I said to Jan. I'd been telling him about Oleksandr's story. 'It's also the dominance of English in the development of programming itself, which is fascinating. But what happens when English becomes a global language? How many different "Englishes" are out there?'

I reached out to Jan to get his view on what I'd been hearing from people in the Ruby community. I've known him for a very long time. He's an expert in linguistic anthropology, especially in things such as linguistic ideologies and meta-pragmatics, fields which I thought could help me understand what I was seeing. He's been researching language and society in Paraguay for the past 20 years. My conversation with Oleksandr had had a big impact on me. I wanted to understand a bit more about how linguistic anthropologists think about cultural perceptions of language and stigma. How could someone be embarrassed by his own language? How can software provoke such feelings?

'I remember the first time I was in Paraguay,' Jan told me. 'I had an interview scheduled with an anthropologist in Posadas. I'll never forget him. We sat down to chat, I turned on my little recorder, but the battery suddenly died. He immediately knew why. "Oh, you probably bought the cheap batteries that they sell here, they're no good. Look, just go down to the market, buy new batteries, but make sure they're Alkaline. I'll be waiting here. There is no rush." He was very patient with my inexperience.'

'What did you and the Paraguayan anthropologist talk about?' I asked.

'Argentinian, he was.'

'Sorry.'

'We talked about all the progressive, leftist governments in South America of the time – it was around 2005, I think. Lula in Brazil, Kirchner in Argentina, Mujica in Uruguay. There was a strong push for regional integration between these countries, and the guy's research was about the grassroots organisations that were trying to resist the economic policies that these countries were trying to implement.'

'Why were you there?'

'I was part of an interdisciplinary project that focused on the cultural integration between those countries in relation to economic integration that was happening in the region. And my part in the project was to think about the role of the Indigenous Guaraní language in these attempts to integrate.'

'Why Guaraní?'

'Because it's a language spoken in all of these countries, even though Paraguay is the only one to have made it an official language. Most people in Paraguay speak Spanish and Guaraní, both are used in official documents and stuff.'

'You thought that Guaraní would be the glue between the countries, linguistically?'

“Yes, one of the cultural elements that they have in common. It turned out that I was a bit wrong, though. Guaraní is a language that, essentially, Paraguay claims for itself. It’s what differentiates Paraguay from Brazil, Argentina and Uruguay. And this view goes back a long time. So this idea that Guaraní was a language that united these countries through the same Indigenous heritage was totally wrong. The opposite was the case: only Paraguay claimed Guaraní as its national thing.’

It seemed that Jan and I were getting far away from any discussion about linguistic anthropology and programming. However, there was something in chatting about Guaraní, an Indigenous language from South America, that I thought might help me understand the languages of this other tribe, the programmers.

‘After coming back from Paraguay, I started looking into linguistic anthropological theory – concepts like indexicality and language ideology – to help me understand what I’d seen,’ Jan continued. ‘Most people think about language primarily as a way to communicate content. But language actually has many different functions. For instance, people use language to differentiate themselves from others.[3] We understand ourselves as part of a group, or as different from other groups, through the language or the dialect that we speak. And the good thing about linguistic anthropology is that you can talk about all these things together: you don’t need one theory that explains linguistic structure and the meaning of words, then another theory to make sense of the social, cultural, economic and political aspects of using those words. You have one theory that explains both: how words refer to things in the world and how they refer to the people and groups of people that use them.’[4]

Language theory and computer science have been walking, hand in hand, for a long time. In the 1950s, when both modern linguistics and modern computer science became more established, there was a lot of borrowing between the disciplines. We might think that there is a ‘natural’ connection between the way human languages work and the way programming languages work, but there isn’t. Programming ‘languages’ only started to be called languages after FORTRAN in the 1950s – before that, there was only code. In those days coders, mostly women, would pull levers and fill in punch cards with coded instructions that made computers work. As the job of programming moved away from physical levers and holes in paper card, it became connected more and more closely to the written word.

‘What Chomsky was trying to do in his linguistic theory was to break everything down into digital information,’ Jan told me.

‘So he was saying that in the end words don’t really matter – what matters is the structure underneath them?’

‘And that structure is binary,’ he replied.

‘That is, of course, the essence of how computers work today: binaries. People have argued that software isn’t even that relevant, because “everything in computing”, at the end of the day, relies on the binary structure of transistors. Everything in computing boils down to the 0s and 1s in the transistors, and therefore whatever you have on the surface, whatever language you use to write software, doesn’t really matter. At the end of the day, it’s all binary logic on the core. Which ties back to Chomsky, I guess.’

‘It does.’

‘In the sense that Chomsky might be saying: “Different languages don’t really matter because there’s only the kernel, the true universal grammar”.’

‘But not really, right?’ Jan intervened.

‘Right!’

‘That is the core difference between Chomskyan approaches to language and linguistic anthropology. As linguistic anthropologists, we recognise that if you really want to understand language, you need to consider context. The way in which you use language, the effects of words, the social meaning people associate with words. All of that matters.’

‘Absolutely, and there is a similar problem with computing, hardware and binaries. There isn’t only hardware. The way people build software, the decisions that they make, the language that they speak, the way they use it, the kind of bugs they introduce, the kinds of ways in which they solve the bugs, the social consequences of software, all of that also matters.’

‘I think,’ Jan said and paused. ‘I think …’ and then he paused again. ‘I think …’

‘I can see that you’re thinking, Jan.’

He laughed and said ‘I think that the broader question here is about linguistic diversity. I think a lot of people, including linguists and computer scientists, assume that, no matter what language you speak, there is some sort of basic core – perhaps what you guys call “kernel” in computer speech. This may be a universal grammar, abstract binary structures, a language acquisition device in the brain or certain semantic primes that are part of every culture. The assumption is that you can reduce all communicative behaviour to certain universals which allow us to translate and break everything down into these core elements.’

‘Which is an OK assumption, I suppose?’

‘It’s mostly how things have operated so far, yes. However, I think, you know, linguistic anthropology and other sciences have been trying to propose a different model. What if the exercise was not to assume that we need to break things down into “universal components”? What if we could use linguistic diversity in and of itself?’

Human languages – the so-called ‘natural’ languages – are not just made of grammar and syntax. The way we use language, the context in which we communicate, the nuances of our linguistic performances and the effects of our speech acts all have considerable significance. If that much is clear to most people when thinking about natural languages, my big question was how to transpose that to the context of programming languages. My chat with Jan showed me at least two ways to pursue that question. Firstly, look at the ways in which English dominates this scene and the resistance to it that may arise. Secondly, explore the varied ways in which programmers talk about programming languages by investigating code comments, code that writes code (meta-programming) and the history of software libraries. As Jan suggested, we may not need the grammar and the syntax to start understanding the structures that emerge when programmers talk about programming.

‘I’m curious to know something,’ Jan said. ‘Imagine a world in which there was no English. What would computer programs look like? What if we didn’t have English at all? What if we couldn’t use ‘do’ or ‘end’ and all those constructs? What if we had to find a different way of doing things?’

‘There’s a programmer called Ramsey Nasser who created a programming language in Arabic called قلب (Alb),’ I remarked. ‘You can write all your programs in Arabic, but the problem is this: your software, written in Alb, is never going to be commercially (or more widely) used because the entire infrastructure of the internet is in English. You must find … you’ll have to find ways to interface with that infrastructure – and to do that your programming language is going to have to succumb to English at the edges or at certain points, right? That was Nasser’s conclusion, anyway. His work is interesting, though, because it shows that there is a material or infrastructural aspect to this presence of English in computing. Whether or not this prevalence of English is intentional or coincidental is beside the point. It’s baked in.’

‘People use language to differentiate. In all communities there are multiple ways of saying things in the form of different varieties, or registers, or styles. But usually there’s some sort of hierarchy built into the relationship between language A and language B or variety A and variety B; one has more prestige than the other. However, that hierarchy doesn't always have to stay the same. Sometimes it shifts. In Paraguay, for example, there is Spanish and there is also Guaraní – but then there are moments in which Guaraní becomes, in certain situations, more prestigious. It's a dynamic hierarchy, and it really may vary from situation to situation or from place to place.'

'I never thought of it like that, but if you write a programming language in something other than English, for example, there might be different angles of looking at this. On the one hand, you might reach a conclusion that it's impossible to counter English in computing, it's just too dominant – just as Nasser did. On the other hand, the effect of creating a programming language in Portuguese, for example, puts that language in another position as well.'

'If you write in English, then OK, everybody will understand. But there's something special when you write a programming language in Arabic or in Portuguese. Again, you need a community of speakers who share your code and who share a particular understanding of it. This creates a space in which you can add something on top of the generic code of English or Spanish. It's generic code, but it's just assumed that everybody speaks English anyway, and everybody sort of defaults to it. However, you can have something special on top of that.'

Our conversation touched on programming and language ideologies, then moved back to Guaraní and finally returned to programming again. I told Jan about things such as 'Chinese-Python', in which a layer of Chinese characters is added on top of the Python programming language. It makes the program slower, because there is an added layer of translation, but it makes it easier for more people to code in. I told him about other linguistic mixes created by programmers: whereas in Japan programmers code mostly in English (without Japanese characters), in Brazil people tend to mix English and Portuguese when writing Ruby. The result is a mixed register in which the language keywords ('do', 'end', etc.) stay in English because they can't be changed, but the variables, class names and constant names are in Portuguese.

'There is something called meta-programming, which I don't understand very well, but people in the Ruby community seem to think it's very important,' I told Jan.

'OK …'

'It's a technique. You write code that generates code.'

'Very meta, OK …'

'Okay, listen. Imagine you're a virtual gardener,' I proposed.

‘Unlikely, but I guess you never know, with all the metaverse stuff.’

‘Listen! As a gardener, you have loads of things to do. You have tasks. You need to rake the leaves, then dig the ground, then plant some plants, then water everything. One depends on the other: there is a logical order in which you need to do them.’

It was hard to get this across to Jan without writing something down. I shared my screen with him and typed some stuff on a blank document. ‘Watering the plants depends on planting them. Planting them depends on digging, which depends on raking the leaves. Something like this,’ I said, hovering over the documents on the screen.

```
task :watering => :planting
task :planting => :digging
task :digging  => :raking
task :raking
```

‘What actually happens during each task?’

‘Oh, I don’t know … maybe you shout what you’re doing?’

‘OK.’

He wasn’t convinced.

‘So, it would look something like this, I think …’

```
task  :raking do
  shout "Raking the leaves!"
end

task  :digging do
  shout "Digging the ground"
end

task  :planting do
  shout "Planting the plants"
end

task  :watering do
  shout "Watering the plants"
end

def method_missing(name, *args)
  print "#{args[0]}\n"
end
```

‘I confess that I’m lost. I don’t really know what is what. Is this written in Ruby?’

‘Yes, but it’s using Rake.’

‘What about “shout”?’

‘That is our little bit of meta-programming. Ruby doesn’t know anything called “shout” and we haven’t defined it anywhere. If we simply ran the program without the ‘method_missing’ bit at the end, it would just crash, but we told Ruby: ‘if there is anything here that you don’t understand, just print it to the screen. Back in our little virtual gardening job, we could come to a new garden, run our program, shout our tasks to the screen and it’s all done.’

‘I think I’m totally lost now,’ Jan concluded.

‘What I’m trying to say, I guess, is that we have ways of using language to talk about language, right? It’s the same with meta-programming. We’re using code to program code.’

Jan and I finished our conversation about language after a couple of hours, both slightly exhausted. I’d failed to explain meta-programming to him, but he’d succeeded in telling me about the social aspects of language, especially the role of language ideologies in shaping society’s political and economic underpinnings. In the days that followed I kept thinking about this conversation, trying to match it up with other chats I had about meta-programming. Chris Seaton, a brilliant Ruby programmer, taught me many things. He told me that the most crucial bits of Ruby’s ecosystem are based on meta-programming, things such as Rails and RSpec.

‘Rails showed that meta-programming could scale and could work well,’ Chris told me. ‘It showed what it could do in terms of power. And I think it’s a big part of what makes developers happy to be working with Ruby.’

I was already struggling to understand meta-programming, let alone how it would connect to happiness. Chris went on to explain.

‘I think Ruby is low friction, low ceremony,’ he continued. ‘People can come and start working with it, no problem. This works because they don’t have to worry a lot about importing functions, and dealing with different parts of the system, like “This is a run time thing, this is a compile time thing”. So I think Ruby provides the illusion of an integrated system through this meta-programming. This means programmers don’t have to think too much about it; they can just go on to be happy.’

Notes

[1] Austin. *How to Do Things With Words*.
[2] Kroskrity. ‘Language ideologies’.
[3] Irvine and Gal. *Signs of Difference*.
[4] Silverstein. ‘Shifters, linguistic categories, and cultural description’.

8
Happy programmers

There is a sentiment among programmers that coding should be fun. Or that you should do it for fun, that it should be a hobby as well as a job. This sentiment puts a bit of pressure on people whose job is to program but who don't necessarily program 'for fun' nor as a 'hobby'. They do it, it's a job, pays the bills – but they might feel that unless you are tinkering with unusual programming hacks, you are not a proper developer. On the other hand, there is something about fun, weirdness and tinkering that makes the relationship between humans and computers visible in strange ways. Programmers often find happiness in things that serve no immediate purpose; programs that make you stop and think about how we relate to computers. A program that generates itself. Or one that translates itself into other languages. A language that is unable to hold any data for a long period of time, in which data decays every time you run it. A program that makes PDFs speak. A language that is not based in English and finds no way of working in the real world because software infrastructure only speaks that language. Things that make you pause, scratch your head and wonder. Random things.

Happiness is a big topic in the Ruby community. Matz often argues that the language itself is 'optimized for happiness'. Does that mean that Ruby has incorporated strange features to allow for joy in programming? Instead of immediately dismissing that pursuit of happiness as some sort of Silicon Valley fad, my research led me seriously to explore what Matz's idea might mean. How could a programming language influence happiness? It all began at Upstream, at a moment when things were not 'happy' at all.

'Hey, have you finished writing that rake task yet or are you still working on it?' Charles asked me, waking me up from my daydreams.

'Yes!' I replied.

Charles smiled and went away. I'd beaten him in his own game of always answering an either/or question with a yes. As for the task, I'd been writing a script to update the values of all products and variants, but this script would fail every time I tried running it. I was getting a bit behind and Charles knew it. He didn't usually put much pressure on anyone to get things done by a certain date, but this was something that Ivan, the big boss man, was pressuring him to do, so he was asking me to get on with it. Fair enough, chain of command.

I wasn't very happy to be writing this script. It was something that could be done manually by people in the company's product team. It would probably take less time to do it manually in fact, considering I had to write the code, test it, wait for the next deployment of the real app and then run it there again. It would take at least three or four days for a task that could be done manually in a few hours. A task that would probably never be used again. I wasn't happy about this at all. I could feel the code dying, never to be used again, even as I wrote it.

'Ivan, hi, sorry, hi,' I approached him like an idiot.

'Yes?'

'You know this script you asked me to write, you know, for the product team.'

'Oh, yes, have you done it?'

'Well, no, not yet. You do realise that this code would never be used again and that I'm kinda of wasting my time. People at the product team could just update these values themselves.'

'Yes, I know. Can you just do it though?'

'Really?'

'Yes.'

I walked back to my desk, slowly, like a donkey refusing to go up a hill. I sat down to write the task, but immediately got distracted. I was writing a type of script that Rubyists call a 'rake task'. It's something that you write to be used every now and again, something that changes or does something to the data or the app. It's not something that you want inside your app as a normal function because you don't want other parts of the codebase using it. You want to keep it separate, often in a folder called 'tasks'. As I browsed that folder, looking for a sub-folder to place my task in, I found a file called 'unfuck.rake'.

The file had a series of smaller tasks, grouped together under the heading ':unfuck'. The tasks had descriptions like 'Get rid of the table in the production database that shouldn't be there' or 'Unfuck orders that have the wrong adjustment amount'. Tasks usually show their status when you run them. They'll tell you at what stage they are at and let you know if things fail. If they do fail, they usually have no dangerous consequences, which is why they are so useful. By reading the code, I could see that some of them would tell you their current status by showing on your screen things like

> 'Unfucking the adjustments for #{adjustment.adjustable.number}'
> and
> 'Unfucking the totals for #{order.number}'.

What the fuck was this?

As I scrolled through the file, I realised that someone had already been in the same head space that I was in now. Someone else had already felt that they were writing stupid code and decided to have some fun while they were at it. Why call a useless task a beautiful name? Why not call it by its real, shitty name? The more I read the code, the more I was assured that this programmer and I shared similar feelings of being stuck with a terrible job to do. I came across a comment that would confirm my suspicions:

```
# This is the same logic that is in
# app/services/refunds/line_item_refund_creator.rb
# but its not in a reusable state,
# and I'm not refactoring the code
# for the sake of this rake task.
```

The line 'I'm not refactoring the code for the sake of this rake task' says it all. The person writing this is pissed off with the state of things; something has made them quite annoyed about having to write this task. And what they are doing is not worth the extra work of 'refactoring' some other code. Who was it? I opened the codebase on Gitlab, where the code was hosted, and searched through the history of changes in that file. I was looking for the person who'd made the changes to that specific portion of that file, the one that'd unfucked it all. What a surprise to find, in the history left behind, Ivan's name. Of course: he'd been here before. It wasn't just that he wanted me to do it, he wanted me to feel it, to learn from it. I thought about our conversation.

'Can you just do it though?'

Yes, Ivan, I sure can just write this rake task for you.

Jim Weirich created Rake in 2009 and it has since become one of the staples of Ruby programming. It's always there; you use it all the time.

I'd wager good money – I'll eat my own hat, as my old boss used to say – that 'rake db:migrate' is the most written set of characters in the history of Ruby. Rake is part of every Ruby application's ecosystem, it's always there. Boring, nothing new, furniture, part of the landscape. That's why it's special; rake is one of the things we take for granted.

Jim writes that he never intended to write the program in the first place. He didn't think it was particularly useful and probably wasn't going to be interesting to anyone. At the time, he was writing programs that help you put together a software library. These programs are called 'build tools' because they contain the steps to build a large chunk of software in a safe way. Why? To create software without having to resort to human memory to remember which bits go first and which ones go second. You can't build a house roof first; you must lay the foundations. Build tools contain the instructions that tell the computer what to do to build a software library. Believe it or not, programs have loads of things that they depend on and you must stack them together in a nice way. To do that, Jim used what everyone was using at the time: Make. Make was created at Bell Labs in 1970 and shipped during the first version of UNIX.[1] It uses Makefiles to describe how each part of a program should be compiled and in what order. When you run Make, it follows those instructions and packages your software.

Jim created Rake in 2009, just five years before he died.[2] With more than 500 million downloads, Rake is the 10th most downloaded Ruby gem. Rails is number 40.[3] The problem with Make and other build tools is that their uniqueness is so unique they don't know how to talk to any other programming language. Jim had the idea of creating Rake because he kept running into the same problems with Make. He had to use a Ruby script to work around that, then thought it would be nice to be able just to use Ruby inside a Makefile. He gave it a go. 'I showed the code to my co-worker, and we had a good laugh,' Jim wrote. 'It was just about a page worth of code that reproduced an amazing amount of the functionality of Make. We were both truly stunned with the power of Ruby.'[4] Rake is also tailored to a specific problem, but it is cosmopolitan and well-travelled. Rake has been to Japan and it now speaks Ruby. Rake is made of Ruby: it's like a little dialect of Ruby, a dialect that you mumble as you write the code that will execute the useless task that Ivan asked you to do.

If Rake is basically just Ruby, what makes it so special?

It is a well-established tradition in the Ruby community that whenever you are about to start writing a rake program you must clap three times and mumble to yourself the word 'task'. A task is 'the basic unit of work in a Rakefile. A task has a name, a set of prerequisites and a list of actions to be performed'.[5] Every time a task is 'invoked' – the term used by many programmers to give their actions a hint of magic – it checks which other tasks it depends on. A typical Rakefile will have tasks that depend on one or more dependencies. 'Task' means nothing in Ruby. It won't do anything – not even if you clap profusely and shout 'task, task task!!!' But move into a Rakefile – anything that ends with .rake – and you're golden. Task will do anything you want; it will unfuck all your problems.

Many months after I'd finish writing the task that Ivan wanted me to write, I was looking through the actual code that Jim wrote for Rake. I was prompted to do this by a comment made by one of the first Ruby programmers that I reached out to outside of Upstream. I came upon a list of projects on machine learning written in Ruby,[6] and found his project Rley.[7] As it sometimes happens with publicly available source code, the owner of the project didn't reveal who they were; they simply went by the name of 'FamishedTiger'.

I eventually managed to contact the secretive tiger. What followed was a long exchange of emails and interviews about natural language processing, and machine learning, but also about Ruby and what it means to create beautiful ruby code. We exchanged long emails about the history of programming languages; I even sent him a draft of one of my academic papers on the anthropology of programming.[8] He didn't like my paper particularly, but our conversation went on. One day he wrote about which Ruby programs or libraries he found the most beautiful.

> 'I was always seduced by gems / libraries that provided a DSL (Domain-Specific Language) that helped programmers to express their intents in Ruby in a fluent way.'

He pointed out the simplicity of something like Sinatra (which we saw in Chapter 1), which he admired – but what he really loved was Rake.

> I cannot but consider it as a masterpiece. The much regretted Jim Weirich created a tool that surpassed the original C Make by making rakefiles just Ruby files. As with C (or C++) we can express dependencies in our projects and still use the full power of the Ruby language. He could achieve this, thanks to the malleable Ruby syntax.

I understood what he was saying, but I couldn't feel it. I couldn't really explain why Rubyists loved it so much. Ruby allows you to create your own dialects within the language: like Jan and I did while using 'method missing'. It wasn't just because it was useful, I thought; there had to be something else. What makes Rubyists so happy about this? It must be something in this idea of the 'task', I thought. 'Task' looks like a lot of other things in Ruby: something important that opens a block. A keyword that unlocks (unfucks?) the door to another dimension. We could, for instance, insert our code block to sort a card_deck into a Rake task.

```
task sort_cards: do
  card_deck.each do |card|
    hearts.push(card) if card.suit == "Hearts"
    clubs.push(card)   if card.suit == "Clubs"
  end
end
```

Does that make us happier?

I rummaged through Rake's code and found a file with something called 'Rake::DSL'. If I am to feel what Rake is all about, I thought, I might find it here.[9] The file describes what Rake is, what its main components are, how to write a Rakefile and how to run them. The file is long, with loads of comments, but the key is right at the end, at the very end of the file. There Jim wrote a few comments, explaining how part of the code would 'Extend the main object with the DSL commands' and allow 'top-level calls' to task.

This is the code:

```
self.extend Rake::DSL
```

A simple and concise line. It literally says please extend (your)self with the language of Rake. I repeat: self.extend Rake::DSL. It takes Ruby by the hand and asks, 'could you please add my dialect to your programs? Would you allow us to say "task" every now and then?' The expression 'self.extend Rake::DSL' cements Jim's cleverness. More than cleverness, perhaps. This is the type of thing that makes Ruby code look so powerful; most of all, this is the kind of thing that makes Ruby programmers happy. It allowed Jim to add his seasoning to Ruby – something that very few programming languages allow you to do. Language designers often think this is a bad idea: it gives programmers too much freedom. Matz disagreed: he felt that you could go and make Ruby your own. Add your quirks to it, no problem, whatever makes you happy.

The beginning of Jim Weirich's announcement of Rake got stuck in my head for a while:

> OK, let me state from the beginning that I never intended to write this code. I'm not convinced it is useful and I'm not convinced anyone would even be interested in it. All I can say is that why's onion truck must by (*sic*) been passing through the Ohio valley.

What was he on about? What does a truck full of onions belonging to someone named 'why' have anything to do with him writing software? To find some answers I did what I never do, which is to go to Twitter.

'Would any Rubyists know what "why's onion truck" was about?' I tagged a few people and Brittany quickly replied, saying that Nick Schwareder would be the person to ask. He came back quite quickly, in a short burst of tweets.

> ooh now this is an interesting curiosity
> why's famous ruby guide included an onion. the idea was that the poignant ruby should make you weep, if not the onion would help.
> […]
> my gut says that it is in line with some whimsical phrasing of the time, and why was very active on ruby talk and message boards, and maybe he got swept up in the excitement, and this is how he phrased it
> (also Jim lived in the Ohio valley I believe)[10]

I have a very hard time using social media, the anxiety of it all, but this time it worked. Jim lived in the Ohio valley, and maybe why had been messaging on RubyTalk about trucks and onions. Nick put it all together: why's truck drove through the Ohio valley, the onions made Jim cry, tears fell on his keyboard and Rake was written.

Onion tears, coding tears, tears of joy.

I guess that is what we do when we are happy: we write a beautifully concise program such as Rake.

When Charles and I talked about his story as a developer, he told me I should speak to someone called Chris. They'd gone to university together and Chris was quite involved in the Ruby community – he was writing a compiler for it. I hesitated to contact Chris straight away because I wanted to build my connections in the community one-by-one. Charles didn't have Chris's contact, they hadn't kept in touch, so I would need to send him a cold call/email. I hate sending cold emails even more than I hate receiving them. This time I did it, however, and Chris's reply was hilarious.

'Developers get tons of these survey invites,' he wrote. 'I'll say yes since you look like a legitimate researcher and I'm very interested in human aspects of Ruby, but if you're wondering why you get low uptake it's because we get spammed with these things all the time.' I replied saying that it wasn't a survey at all; I wanted to have a chat about his life as a Ruby dev. That didn't help much and I considered giving up. I'm so glad that I didn't. Not only did Chris and I have a great chat, but he ended up becoming a great supporter of my research, encouraging me every step of the way. But that was later – the beginning of our first chat was tough. I introduced myself, explained the research, as I always do, and tried to break the ice by referring to Charles and mentioning that they both had gone to university together.

'I don't remember him,' Chris said, his face staring at me through our Zoom call. I asked something else and again, quick reply, face stare. Oh boy, this would be a hard interview. Chris and I talked about many things, including his trajectory as a developer and in the army, and how he was pulled back from his career in the armed forces because his work on compilers got some attention. He decided to focus on programming languages that, for various reasons, were harder to optimise, to make them work more efficiently. This was one of many conversations with him, this Ruby legend. One of them was about meta-programming, as I described in the previous chapter, but this one is from the first time we ever met.

'The problem with Ruby,' Chris said. 'is that it has a lot of "what ifs". What if a function hasn't been used yet during this run time? What if it has, but some other method has changed its meaning since the program started?'

'And those "what ifs" are costly?' I asked.

'In terms of performance, they are. They take a while, and these performance penalties come directly from Ruby's idea of "optimizing for happiness".'

In 2008, a year before the release of Rake, David Flanagan and Matz published *The Ruby Programming Language*. It came at a particularly crucial point in the development of Ruby, the troublesome crossing from version 1.8 to version 1.9. To this day, some 15 years later, Matz still refers to this epoch as the moment he was most afraid that the community would break into two: the ones using the 1.8 version and the ones using the 1.9 one. In their book Flanagan and Matz describe how domain-specific-languages (DSLs) work in Ruby. It reads like a description of what Rake would be when it was released a year later. DSLs use blocks 'as if they were keywords'[11] – they have first-grade properties – to extend the Ruby syntax 'in ways that make programming easier,' they write. This is what Rake does: it turns a 'task' into something you can use anywhere. By extending the main object of a program to include the methods defined in it, Rake makes it easier for programmers to write tasks. Rake and other DSLs makes the lives of programmers easier. They make programmers happy. They don't necessarily chuckle or sing when they use Rake, but I'd like to think that most Rubyists at least smile when they read Rake's code for the first time.

However, if reflection and meta-programming are tools to make programmers happy, they don't make the computer happy at all. As Matz often says, optimising for happiness is opposed to optimising for the compiler. He means that the more DSL-like techniques you inject into your programs, the more things need to be checked when your program is running. The more you create special syntax such as Rake's 'task', the less efficient your computational process will be. And compilers hate that. Compilers want to be as efficient as is machinically possible. Well … maybe not the compilers themselves, but certainly the people who design compilers and work to make Ruby as efficient as it can possibly be.

It seemed that the fact that things are malleable in Ruby, that each Rubyist does things a little bit differently, that there are a lot of domain-specific-language and loads of meta-programming – all of these things had a real impact on the speed of Ruby.

'Does that bother you, Chris, as someone trying to optimse things?'

'What?'

'That the Ruby community likes those things that make the life of a compiler harder?'

'Not at all. That's just idiomatic Ruby, it's the way people write it. I don't feel the community should change its ways. It all springs from the fact that Ruby embraces the chaos.'

Ruby embraces the chaos. I don't think I could find a better description of this community. A community in which (almost) everything goes in terms of programming. Things that make up the chaos and complexity of Ruby. For a programmer writing compilers, like Chris, instead of telling the community how they should write code to make it easier to optimise it, 'we said we would optimise Ruby as humans use it, as humans understand the Ruby language'. If people want cake, let them eat it. We'll deal with the crumbs at compiler level.

The useless task that Ivan asked me to do led me to understand the diversity of the Ruby language. It had a technical side to it, this idea of meta-programming, which is one of the ways in which Ruby allows programmers to turn their flavours of the language, their ‘dialects’, into real working code. In a way, Ruby is designed to accommodate these very human tendencies: to explore, to diversify, to make something our own. To rake it in. A human trait that is more than an individual desire because it only makes sense socially, operating within a community. That is how people create and shape meaning in social life.

Notes

[1] Kernighan. *UNIX*, 90.
[2] Rubygems. ‘Rake-versions’.?
[3] Rubygems. ‘Stats’.
[4] Weirich. ‘Rational’.
[5] Weirich. ‘Glossary’.
[6] Arbox. ‘NLP-with-Ruby.’
[7] Famished-tiger. ‘Rley.’
[8] Heurich. ‘Language Automata.’
[9] Weirich. ‘DSL Definition.’
[10] Schwaderer. Tweet.
[11] Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*.

9

Chunky bacon

In January 2020 I attended the monthly meeting of the London Ruby User Group (LRUG). Murray Steel, LRUG's organiser, would be one of the speakers there, giving a talk that had this curious description:

> Some time ago I stumbled across the header description for WAV files and wondered, what if I took a file and calculated the appropriate WAV file header for it, could I hear my data? Yes, you can. You probably don't want to, but you can.[1]

When I read the description of Murray's talk and the idea of trying to transform text files into 'music' (WAV files), it immediately pulled me in. It was my first encounter with another side of computer programmers; one in which logic and rationality are not the primary aim. There was something very interesting in that project. This was its jokey aspect, but it was more than that. It was something else. It was the fact that it didn't sound particularly useful. A project that turns a PDF into a file you can listen to? Listen to a PDF?

Computing is often framed as either a rational and mathematical pursuit or a purely capitalistic endeavour. However, this was something different. It was not only humorous but also useless, devoid of any commercial or scientific value. In his presentation Murray took the whole of Ruby's source code and successively transformed its files into sounds and images. Did anyone learn anything about Ruby by 'listening' or 'seeing' it? Probably not, but it was still amazing to watch.

'How did you come up with that, Murray?' I asked him almost a year after the talk. We were chatting about his life as a developer, his role in LRUG, the generational changes in the Ruby community and, of course, about the most whimsical character in the community: _why the lucky stiff. Murray told me the project was a few years old and had been inspired by people who were trying to visualise Microsoft Word's executable file.[2] At first, he said, he played around with turning any file into a WAV file by changing the file header. Next he explored changing files into images (bitmaps), which required changes in the header and data manipulation as well. Finally he ventured into MIDI, for which both data and headers had to be messed with quite a lot.

'There's something about being able to ask a computer to do something that's sort of pointless but is fun; there's like a playfulness.' Murray explained. 'And historically, I don't know if it's necessarily true any more, the Ruby community cared about that kind of thing, and characters like why embraced that.'

'Do you think it's changed more recently?'

'It was definitely something that Ruby kind of cared about … It is a bit more pragmatic these days as a community, I think. But that [i.e., the fun] was what drew me, I think, to Ruby in the first place. And there's no one creating this stuff that why used to create, and I think that's a real shame.'

Understanding the cultural aspects of a community of programmers requires many things. You need to grasp the technical nuances. You need to work as part of a team, building software. You need to hear their stories. You need to investigate the history of the community and understand the characters that shaped it. When I found out about why, it made sense to look into him. He showed me a fun and quirky side to programming that rarely extends beyond programming communities. This playful and humorous side to coding communities has been written about,[3] but is not, somehow, the image that most people have of programmers. In my case, I wanted to see how these ingenious and devious ways of programming were reflected in the Ruby community. What's more, I wanted to see how they become technical pieces embedded in code. In other words, how does quirkiness become a concrete part of software?

Why is a mythical character in the Ruby community. A token of a previous era, when everything in the Ruby world was a bit less corporate, when performance and speed were not so much on the horizon. A time when playing around with code was important; when libraries had names like 'bloobsaphone'. Back then, of course, Ruby was already paying the bills of a lot of people. You made a living writing Ruby, but you also had a laugh with it.

Why wrote many things. He wrote Ruby code; he also wrote C code. He devised, experimented and tinkered with code. He tried to keep his personal identity as secret as possible. He never revealed his name – at least not in any public forums. why gave talks in conferences, sometimes alongside his band, combining music and live coding with a technical presentation on the internal logic of Ruby. He wrote posts and articles, interacting with people on the mailing list. In his introductory book to Ruby, why created a guide to the language that was simultaneously technical and artistic. Throughout the book, he uses cartoons to create a parallel narrative in which foxes and other characters develop a relationship with the book's author – himself, presumably. It's all very meta. At one point the foxes need to get his attention, and they think that shouting 'chunky bacon' at him might help.

The book has a long section in which a rabbit battles an assortment of creatures. It is structured as a role-playing game, in which monsters are generated through meta-programming. Despite being built on a fantasy of sorts, it is the clearest chapter of his guide: very focused on delivering a message, on teaching something. It's less 'fever dreamy' than the rest of the book, where many characters and cartoons come together in strange ways. Oddly, it's the only chapter in which why quotes academics – like Terry Eagleton, a famous literary scholar. This is quite strange for him,

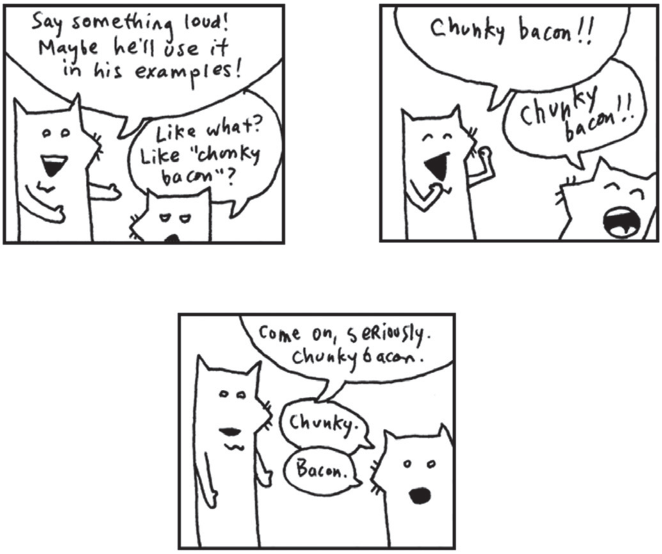

Say something loud! Maybe he'll use it in his examples!
Like what? Like "chunky bacon"?
Chunky bacon!!
Chunky bacon!!
Come on, seriously. Chunky bacon.
Chunky.
Bacon.

Figure 9.1 why's famous cartoon foxes. _why. 'why's (poignant) guide to Ruby'. CC BY-SA 2.5.

and why knows it. He feels obliged to downplay the seriousness of what he writes. He was getting too high-brow and needed consciously to dumb it down a bit.

> You could say that programming itself is meta-language. All code speaks the language of action, of a plan which hasn't been played yet, but shortly will. Stage directions for the players inside your machine. I've waxed sentimental on this before.

When why's guide first came out – as a website, not a book – people used the Ruby mailing list to give him feedback. why was very polite and replied to most of it. He agreed with some of the criticism, but he also defended the chapter with the rabbit warrior RPG. 'The code is simple, useful, practical (a usable RPG game is hardly contrived) and fully explained,' he replied. The topic – meta-programming – was not an easy one, but he thought he managed to write something 'more revealing than anything else […] around'.[4] Let's see. After describing meta-programming as 'writing code that writes code', a common definition, why comes up with his own, alternative 'definition':

> Let's say [meta-programming] is more like a little orange pill you won at the circus. When you suck on it, the coating wears away and behind your teeth hatches a massive, floppy sponge brontosaurus. He slides down your tongue and leaps free, frolicking over the pastures, yelping, 'Papa!' And from then on, whenever he freaks out and attacks a van, well, that van is sparkling clean afterwards.

How does a van-licking-coated-brontosaurus reveal what meta-programming is? Because, he continues, 'meta-programming is packing code into pill-form, such that a slender drop of water could trigger it to expand'. A tiny sliver of code that, when faced with the right conditions, blooms into the beast it was always meant to be. The code was always destined to become a van-licking-brontosaurus, you see: it was just waiting for the right amount of saliva to expand into its true form: [self.extend Van::Licking::Bronto]. Is that clear yet? Do we need some cartoon foxes?

In a 2006 short video-collage titled 'Time.now.is_a? MagicTime', why ironically describes the future of Ruby.[5] The video shows a guy with a microphone asking several people to imagine what Ruby will be like 3 years from now. Unsatisfied with their unimaginative replies, the guy with the microphone urges them to go further:

> What if we had a chewing gum that had a virtual machine inside it, and while you're chewing, it's using the kinetic energy from your colliding jaws to compile a cure for rabies?

How did such nonsensical humour become so influential in a community of computer programmers? Aren't they supposed to be engineers?

why wrote 'serious' code too. Among many other things, he wrote a famous HTML parser that allowed you to slurp a whole webpage without having to visit it, extract the relevant HTML pieces and play around with the content. You can grab all the content of a website, search through the resulting document, find all links and display them all. Or you can just store it somewhere. It also fixed bad HTML. And bad HTML was everywhere, especially during the age of web 1.0, when a lot of pages were created in simple text files on editors with no plug-ins to check your code. A lot of that code is probably still out there.

There is nothing exactly funny about Hpricot – why's HTML parser – or its code. If anything the code is a bit weird, with a funny constant called Hpricot that does all the work, a mysterious syntax that was easy to read but unexpected. It's not exactly great code, but it is the link between why's comedy code and his more serious code. Hpricot was, back then, the resistance. Resistance to Ruby becoming more corporate, centred on speed and efficiency. Whimsical resistance.

When Hpricot got a lot of attention in the community, people started building similar things and claiming they had made their own parsers faster than his. 'You're missing the point!' why would shout. Hpricot was about a quirky and dirty way of gobbling down a webpage's content. It didn't have to be insanely fast: that was not the aim. It seemed that, over the years, the Ruby community had changed a bit. While looking at why's writings, there had been a clear shift, as Murray told me. A shift towards productive and efficient code. This hadn't happened just in the Ruby community, though. If the early 2000s saw the emergence of free software and a politics of community building, the 2010s saw the dominance of big tech in every aspect of society. Code was still play, but it had become money too.

It must have been a shock to most people in the community when why started deleting all his public code, shutting down his website, and then totally disappearing in August 2009. 'The popular Ruby message boards, listservs and blogs descended into a state of panic. Had he been hacked? Who had heard from him? Was he in physical danger?' wrote a journalist, vividly recalling the concern about why's disappearing act.[6] Many people questioned and wondered. Some even tried to find out what had happened to him; others feared him dead. A few years later he came back and resurfaced with a closing piece, only to disappear once more.

John Weald joined the community after why disappeared, but before he performed his final act. I'd found John's name in one of the IRC chat logs that ensued why's comeback. After quite a bit of internet archaeology, I managed to contact him for a conversation about Ruby, programming and why's brief resurgence. I asked John how he felt about it at the time. why was already a Ruby legend by that point, and his self-announced return put everyone on their toes. People flocked to IRC to wait for whatever the hell why was going to do. Suddenly a link to a printer interface appeared in one of the web domains that he owned. Like a ghost in a machine, why was sending messages. There another link would download and print pages of a book. People printed them out, and waited – waited for the next link to come along.

'It happened over the course of a couple of days,' John told me in a Zoom chat. 'The whole thing feels like a fever dream. I read over a good chunk of it when you contacted me, and some of it I do remember, but it's all quite hazy.'

'Do you remember the files he was sending?'

'People would print out the pages and put them together. I think that, in the end, someone combined everything and gave it a name. I don't think why named the book, or whatever it was he was sending us, himself.'

'It's currently on Github with the name Closure,' I told him. 'Steve Klabnik holds the repo, I think.'

'That makes sense. He was definitely involved in it. And the name fits as well.'

'But how did it make you feel?'

'As if I was part of something. A chance to be closer to someone that the community looked up to. An era that I wished I'd been a part of,' John told me.

'Maybe a glimpse of what it might have felt like to email why, to interact with him, see his talks live?' I suggested.

'I definitely remember spending some time with his writing, CLOSURE, and trying to make sense of it, and sort of getting the gist of what he was saying,' John said.

And then it was finished, gone, closed. why stopped sending his stuff, he found closure, and left everyone to put the pieces together. People tried to the make sense of it, but they failed. It wasn't clear, at that point, that this was in fact his final act. Tired, and exhausted of trying to figure it out, someone leaves what was to be the last message in the IRC logs. 'Maybe he just wants us all to go insane.'[7] 'Because when you read something he wrote,' John continued, 'you can't take it literally. You know there is a lot of hidden meaning, and a lot of it is just a stream of consciousness. He is just connecting his brain to whatever medium is there.'

'You can never fully comprehend what's there.'

'Yeah, exactly. It doesn't make sense outside of his thought processes, although there is a lot of coherent stuff there. And, you know, the little anecdotes, the fantastical elements.'

In the document that he left behind, why gave some hints as to why he decided to withdraw from the community and the programming world. 'It is strange – I felt a great relief in those days, to no longer be programming,' he wrote. One may wonder if that was an indirect comment to all the people trying to compete with his HTML parser – to make it faster and more efficient just for the sake of it. why noted that he was 'totally disillusioned', feeling 'betrayed by computers' and 'glad not to be fighting NULL'. NULL, the ultimate enemy, the result that you don't want, the value that will make your program break.[8] Yet perhaps why was also annoyed by the people who were trying to out him, to discover his real name and force him to come out of internet anonymity. In CLOSURE, the name given to his final document, his own name – why – features as the topic of an imaginary dialogue between him and someone else, written in third person.

> 'Danny Douglas,' he said, holding his hand out. 'What's yours?'
> We shook. 'why the lucky stiff.'
> 'Eh?' he said, turning his head, but keeping his eyes on me.
> 'why the lucky stiff,' I said.
> 'You gotta be who you are, mate. Now what's your name? Go on, just say it.'
> 'Nah,' I said. 'You don't need it.'
> Danny insists, grabs him, yells at him to tell his name. why runs off into a forest. There tired, exhausted by all the madness of Danny's bullying, he sits and thinks about his name.
> Well, I knew there were reasons I liked 'why the lucky stiff', but I couldn't think of what they were. […] The name 'why' is introspective. It lends itself to profundity.

As if revealing the nature of contemporary capitalism, Danny Douglas says to why: 'You gotta be who you are, mate.' You can't hide any more, not in the world of constant surveillance. There is no room for privacy and hidden identities any more. Platforms need to know who you are, what you're doing, monitor your heartbeat the entire time. Have you done your steps today, mate?

In 2020 why's 'poignant guide to Ruby' was published as a printed book for the first time since its creation, more than 15 years earlier. The limited edition was given as a token of gratitude to everyone who purchased a ticket for that year's Brighton Ruby conference, which had to be cancelled due to the Covid-19 pandemic. A couple of years later, before I travelled to Japan for the first time to interview Ruby developers, I managed to secure 10 copies of the printed book thanks to Andy – the Brighton Ruby organiser – and Emma – who organised the printing. I took 10 copies of why's guide with me and gave them as gifts to a few developers I met in Japan. They had heard of why, of course, and fondly remembered his interventions on the mailing list and his projects. To one of them, it didn't remind her of a forgotten era of the community, but of a more continuous feeling.

'It's about the meaning of programming, isn't it?' she told me when we met. I had run into her at a local Ruby meet-up and we went out for a coffee in a small shop in Kyoto. She is Taiwanese and has been in Japan for quite a few years. 'He made me think about code and about myself in a weird way,' she continued. why does indeed make us think about code in strange ways. Not only does he write about meta-programming, he is himself a meta-programmer: he is the random programmer generator.

> 'His meta-programming stuff, is just wild, but some of it is lost these days. There is too much burnout. Most programmers just want to quit and open a coffee shop in the countryside. They are sick of it, a bit like why was.'

And that may be why left to open a coffee shop somewhere. In it, cartoon foxes serve large slices of chunky bacon, sugar-coated pills who turn into van-licking brontosauruses and chewing virtual machine gum. The shop is probably called cafe-meta and the coffee doesn't come in a cup. It comes as a short snippet of code, engraved on a clay tablet, that expands into a Mount Fuji-sized frapuccino when you drop a single drop of water on a footprint left by the only penguin to have ever written a compiler.

Notes

1 LRUG. 'January 2020 meeting' announcement.
2 Steele. 'Stegosaurus'.
3 See Levy, *Hackers*; Torvalds, *Just For Fun*; Coleman, *Coding Freedom*; Himanen, *Hacker Ethic*; Graham, *Hackers & Painters*.
4 _why. 'why's (poignant) guide to Ruby', Chapter six: Downtown.
5 _why. 'Time.now.is_a? MagicTime'.
6 Lowrey. 'What happened when one of the world's most unusual …'.
7 IRC. '14 May 2013 logs'.
8 steveklabnik. 'CLOSURE'.

Part III

Beyond binaries

10
Learning to see

‘We’ve got a lot of things planned for the next few months,’ Ivan said at the beginning of our weekly Friday meeting. This was called ‘Retro’ – short for retrospective – and it was the place to discuss what had happened during the week.

‘One of them is the new sign-in with Apple feature, which Gui will be responsible for.’

Oh boy, here we go, first project on my own.

‘What’s that?’ Akira asked.

‘It’s something that Apple requires now, that you need to give the user the possibility of logging in with their own Apple id if they want to.’

After the meeting Ivan stopped by my desk.

‘About the Apple stuff. Have you decided if you want to go with a ready-made platform or if you want to build our own?’

‘I think we should build our own,’ I said, trying to sound as confident as possible. I had done some research on the possibilities, and building our own sounded as if I would learn far more from this project than just using some off-the-shelf solution. I didn’t know if I could do it, I’d only been a developer for six months, but I went for it anyway.

I swivelled over to Charles’s desk after Ivan had gone.

‘Charles, sorry, got a minute?’

‘Sure.’

‘Apple Sign-In. What’s the best way of implementing it, do you think?’

‘You could use a Rails engine. The code that handles Facebook log-ins is already there. You should build handlers for each, and an adapter on top of that. Just use the adapter pattern.’

'Brilliant, thanks.'

I had no idea what any of that was.

Programmers love books about patterns. Design patterns, they call them. Like other programming books, they have a particular style. They have an air of authority, they tell you how things should be done, they explain how these should work. It is a style that intersperses bits of code with some sort of lesson, chunks of programs that give you some take-away learning. If you look at programming shelves in bookshops, you'll find titles like *How to Implement Design Patterns* or *Ruby Cookbook* – filled with recipes on how to write code properly. I often wonder if these books are written in this style because programmers are trying to shape the (messy) reality of software development into something neat. Programming books are always trying to organise the world, to transform it into a patterned reality.

I struggled a bit to understand what Charles was talking about when he mentioned Rails engines and the adapter pattern. Eventually, I came to see the code we wrote for the Apple Sign-In as quite beautiful. Charles would probably chuckle at that idea – a simple engine, beautiful? – but aesthetics is a tricky thing. What is or isn't beautiful lies somewhere between our own taste and what we've been taught to like by our community. The desire to apply cultural patterns to the world is something that programmers share with every other social group on the planet. The reality around each group might be different – from a rainforest to a London office building – but the technique is similar. It's a way of imposing a way of being, of creating a different world, one shaped by the way we would like to see it. I look back at some of the code that we wrote and I like it, but why? Because I learned how to see it.

I met Akira when he came back to Upstream's tech team after a period of leave. He took the empty desk space between Charles and me, making my swivelling for help towards Charles's desk a bit more difficult. Over the next year of working together, I would resort to Akira instead of Charles for help with some of my doubts. I asked about the adapter pattern and Akira sent me some articles about it. He showed me how to make the code work.

'Basically it's like a selector,' he told me. 'The handler will decide which route the code should take. Is the user trying to log in with Facebook? Then go that way and run this bit of code. Is the user trying to login in with Apple? Left door on the right. Google? Not yet …' Akira was hilarious.

‘I get that, but why is it called the adapter pattern?’

‘No idea.’

Akira would often reminisce about the time that he joined the company. He called it the ‘golden age of the tech team’. It was 2017, a time when lots of people were being hired, which is always happier than the lukewarm hiring or even the waves of redundancies that we faced together. But it was also a good time for him, as his confidence was at its lowest when he joined. He thought he was a terrible developer because he couldn’t really work or communicate with anyone. He had felt distraught when leaving his last place – ‘I couldn’t write code, nothing’ – but the team at Upstream turned it around for him.

Akira had joined the team dealing with customers and started working on ‘the API’. It was difficult to get his head around a massive codebase with many applications, but everyone had been helpful. ‘Although I was very scared to ask for help because of my previous experience,’ he told me over Zoom when we chatted about his career trajectory, ‘they were there for me and helping me, and gave me time to start doing things, and it was good. It was a lovely thing.’

‘Was Charles there when you joined?’

‘Yep.’

‘Was he your manager?’

‘Not at first. It was funny. Although you were kind of scared to go ask him for help, even if you even mention something, he would give it to you. He would come to your computer to help you – because you can’t really work on his computer and see what he’s doing.’

‘I know, it’s bizarre. All those opened files, the tiny code,’ I agreed. How had Charles ever taught us how to see?

‘You know … you know him, you haven’t heard him speaking above 70 decibels ever in your life. I don’t know if it’s 70, but you know what I mean. Very calm. I really wonder if he has ever raised his voice for something. Most senior developers, I guess, they hate working with less experienced developers … And maybe he felt the same, but you would never know.’

A lot of tech companies have book clubs and Upstream was no exception. A few months after Akira joined, we started a book club. Everyone in the company was invited. Well … everyone from the tech team. Well … everyone from the tech team who didn’t work remotely. This was pre-pandemic times, so the sessions were all face to face. After everyone shifted to remote, there were some attempts to include some of the people working from the Ukraine, but none of them ever came. This ended up being important to how the book club evolved, the absence of outsourced developers creating a bond between the UK-based developers. This in turn transformed the book club's purpose – something that happened slowly over two years. From being only about reading books, it became a space and a moment in which we could share news, tips, comments and personal anxieties. It was crucial to sustain some sort of camaraderie when Covid-19 made us work from home and prevented us from seeing each other on a regular basis. It also became a source of solidarity in troubling times, when a few members were made redundant. The club provided us with a safe space to share our worries about future redundancies and to try to help, as much as we could, those who had been made to leave or were anxious that they would soon have to.

The way in which the book club changed to what it eventually became happened in winding ways – not perhaps how a book club should be. Maybe at the beginning there was an intent to make us all better programmers by reading books – but that got lost along the way. Instead, the club opened spaces for us to discuss personal grievances, job prospects, company culture and, of course, different styles of programming, philosophies behind software engineering practices, programming languages and operating systems. It was also a place where we learned to see code together.

Some of us in the book club were not very experienced. A couple of us had changed careers, for example Lindsay, who used to be a teacher. Disappointed by the state of primary school education, she decided to follow in her partner's footsteps. She heard of a coding boot camp in London and enlisted there. Lindsay didn't find learning to code very easy. It was an experience that filled her with anxiety, not only because of how hard it was but also because it was a massive change. It meant abandoning her career, giving up what she knew. It meant she had to learn as much as she could fast to make sure that she would be able to find a job and be OK. She did find a job quite quickly after finishing the course, and then felt a bit guilty because a lot of people in her boot camp's cohort were having trouble finding a job. 'It was like I cheated,' she told me, 'because I only applied for two jobs and got one, while other people were applying everywhere and not getting a job.'

Starting a new career in tech often comes as a packaged deal: we'll give you a new set of skills, but also a tiny bit of impostor syndrome. Sure, everyone has a bit of impostor syndrome from this age of generalised anxiety in which we live, but learning to code takes that to a whole new level – or so it feels. All of a sudden, after a few months of learning, you end up in a well-paid job in which you are asked to do things that you probably haven't encountered before. You run into problems that you didn't experience in your coding course, and have to solve them with tools that six months ago you didn't know existed. A lot of people pretend to know what they are supposed to do – and then google the hell out of that problem to try and solve it.

But not Lindsay.

She realised quite soon that she needed to rely on more experienced programmers in order to find her path. She refused to pretend she knew things and instead asked for help, trying to do more and more 'pairing'. Pair programming – also known as 'pairing' – is one of those buzzwords in the programming world. It is a technique in which two programmers work together on the same bit of code. Ideally, one should be coding while the other observes; every now and then they should change positions. Coding with someone else can be one of the most rewarding things you can do as a programmer. You can learn so much from a single thing that another programmer does, something that would probably take you a lot longer to figure out by yourself.

Even though boot camps focus on pair programming quite a lot, however, whether it actually happens at workplaces really depends on individual company culture. When Lindsay realised that pairing was a bit scarce in Upstream's culture, she knew that she would have to convince people to do it. 'The reality is that I still have a lot to learn,' she told me. 'And that for me has been the best way to learn: by pairing and watching them code.'

I often wondered why pairing wasn't more of a thing at Upstream. One senior developer explained it was a question of size. 'Our company is just not big enough … we can't waste resources by having two people work together on the same problem.' A lot of developers seem to share that view: working with someone else on a task will halve the productivity of a team. Sounds unlikely though, given that two people can probably accomplish tasks faster by working together rather than separately.

Despite this local anti-pairing culture, and despite a lot of anxiety, Lindsay didn't seem to care. Quite philosophically, she told me

> 'Life is a stream that's constantly carrying you through different places and you gotta know that we're not going to see a lot of these people like ten or fifteen years down the road.'

She pointed out that we probably wouldn't even talk to any of these senior devs in the future, so we might as well just take advantage of what they can offer, find something that we can take from their experience. Realising that she sounded quite wise, Lindsay laughed. ‘But I always forget that too,’ she added. ‘I just say “Oh, yeah, I can do that by myself” and then struggle on a task.’

The club at Upstream did start with reading books, though, and one of them was Sandi Metz’s *99 Bottles of OOP*. Object Oriented Programming (OOP) is a way of thinking about code that relies on creating objects. Its applications are built on relationships between objects and Metz’s book is about building those objects properly. In it she observes,

> ‘Those feelings you have about the rightness of code are likely correct, but the big super-computer of your unconscious mind can’t supply words to defend them. Sadly, advocating changes to code based on feelings you can’t explain is not likely to be convincing.’

Metz writes in a reflexive style, perhaps trying to escape from the normativity that abounds in programming literature. She uses the famous song ‘99 Bottles of Beer’ as the running code example in her book. The program she implements in the book generates the lyrics of the song in a programmatic way. Instead of writing the whole thing down – from 1 to 99 – she uses a Ruby program to generate it for the reader. One of the things she uses to generate the verses is a Ruby block – but instead of using the do…end syntax that we used to sort our card_deck, Metz replaces it with an equivalent syntax: two braces {}.

```
def verses(upper, lower)
 upper.downto(lower).collect {|i| verse(i)}.join("Än")
end
```

The nice thing about this function is that it counts from the highest number (99) down to the lowest (1). For each number it generates a verse, and the type of verse – the variations within it – are handled somewhere else in the code. Finally, the verses are all collected, that is put together one after the other. That’s it. Done. The song is in front of you:

> 99 bottles of beer on the wall, 99 bottles of beer …

But why does she use the braces-syntax {} instead of the do … end syntax to write the block? Metz explains.

> It's sometimes possible to imbue actual language syntax with additional meaning. For example, the late Jim Weirich took advantage of the fact that Ruby allows code blocks to be delimited either with curly braces {…} or with do…end. Because most folks use these variants interchangeably, he felt free to co-opt them to send signals. Weirich used do…end to warn that the enclosed block had side-effects, and {…} to assure that it did not. Readers of his code remain grateful for these signals.

Then she adds a short, sad note: 'We still miss you, Jim.'

Jim – the author of Rake – was making a very subtle difference, something that a lot of Rubyists would take for granted. It isn't even something that became a cultural norm in the community. Like Metz says, the opposite is true: most people use do…end and {…} interchangeably when building their Ruby blocks; it's not culturally important. What Jim does is to adapt Ruby to his own accord, 'to co-opt them to send signals'. By doing that, he makes us see something about the language, perhaps something about software itself. He shows that meaning, in software, is not something you should take for granted.

Media scholar Friedrich Kittler once wrote that 'there is no software'.[1] There is no software because, no matter what you write, the only meaning that can ever be given is what the silicon chips will do to it. Everything in computing is always a material process that happens inside transistors. But what about the things that other programmers teach you to see? What about the subtle differences that Jim makes? What about the neat objects that Sandi makes us see? What about Charles's adapter pattern?

Perhaps Kittler was missing the point. In his journey to learn how to code, he abandoned a high-level language that he was using to write machine code. He wanted to get 'closer to the machine'. But programming is not about trying to get as close as possible to machine code – there is a reason why people put so much effort in translating machine code into more palatable ways of coding. The point is not that you are trying to find meaning in software. There is nothing there: you need to add it. You need to chase senior programmers and learn from them what you can attach to it, as Lindsay did.

In other words, you don't just add meaning by yourself; instead you learn from others how to do it. You learn to see what the people who will use your software might want from it, as Akira often asked himself, and you realise that perhaps a little improvement here and there can be significant. Then you also realise that finding a little bit of meaning creeping in is just enough to give you the energy to keep coding for another day.

Note

[1] Kittler. 'There is no software'.

11
Beautiful code

To get the first element of a set, in Ruby, you can write 'card_deck.first'. If you want to get the first element every day of the week except Mondays, you can do this:

```
return card_deck.first unless Time.now.monday?
```

Give me the first card unless it's a Monday. It's that sort of code that Rubyists talk about when they describe Ruby as a simple and direct language that allows you to write beautiful code. It might not be Shakespeare, but it is quite readable.

Code and literature are both meant 'before all else, to be read and understood by human beings,' writes Matz, the creator of Ruby. For him, writing code is like writing an essay. Most people assume that programmers just tell computers what to do, but they don't realise that programs need to be constantly maintained and re-written. What programs do is only one part of it; the other part is that programs need to be readable. 'Computers can deal with complexity without complaint, but this is not the case for human beings,' he writes. In real life '[u]nreadable code will reduce most people's productivity significantly, […] easily understandable code will increase it. And we see beauty in such code'.[1]

In a poetic manner, Matz is suggesting that in the way that Ruby is designed, he and others have put in a lot of effort to make it easier to write programs in it. Because Ruby is concise, because it doesn't have a lot of repeated code, because there are very few brackets and braces, Ruby is easy to write in. It's also easy to read, so you can nicely follow what the code is doing. That will make you more productive and happier as a programmer. However, such usable code requires a lot of complexity under the bonnet.

In other words, Ruby is constructed in a complex way so we can read it easily. ‘Because Ruby is not simple, the programs that use it can be.’[2] The internal machinery of Ruby is complex – but it is this very complexity that allows Ruby programmers to enjoy the simplicity of the language.

Programming is a form of writing. You spend your days typing words on a text editor, words that have some sort of effect on the world. Type a few sentences and you can sort your card deck or figure out how many products you have in your warehouse. Some computer scientists view code as a form of art. Programming gathers knowledge and applies it to the world through a skilful practice that ‘produces objects of beauty,’ writes Donald Knuth. Programmers who view themselves as artists, he remarks, even if not in a conscious level, will enjoy what they do.[3] Like Matz, Knuth views the programmer as an essayist who ‘chooses the names of variables carefully’.[4] In *Clean Code*, one of the most influential programming books, Robert Martin observes that good programmers have a ‘code sense’ that enables them to write good, clean code. Acquiring that ‘sense’ requires looking at the techniques and tools that experienced programmers have used in the past, just as experienced artists have developed their own techniques throughout their lifetimes. But a book on code, like a book on art, can only give you so much. Just like ‘books on art don’t promise to make you an artist’, Martin can’t promise to make you a good programmer.

Thursday 20 April 2006. An email comes through the Ruby mailing list, exhorting the beauty of a few lines of code.

> Subject: Symbol#to_proc is just so beautiful
> From: Daniel Schierbeck <daniel.schierbeck@…
> Date: Thu, 20 Apr 2006 01:27:18 +0900
>
> When is this ever getting into Ruby Core?

```
class Symbol
  def to_proc
     proc{|obj| obj.send(self) }
  end
end
```

> Consider this:

```
class Numeric
  def positive?
     self > 0
  end
end

[1, 2, 3].all? &:positive? => true
[–1, 2, 3].all? &:positive? => false
```

> It’s just so damn beautiful!
> Daniel[5]

No explanation of the context, nothing about the origins of this bit of code. The only question is: when? When will the Ruby core team implement this? The request, in fact, is a bit too direct. After all, why would the core team care? One could assume they might, but maybe they don’t. Maybe they have already considered adding it ‘to the core’, but then decided against it. So, to soften the blow, perhaps, the email concludes: ‘It’s just so damn beautiful!’ Daniel emailed the list on 20 April and less than a month later, the feature had gotten into the Ruby core language.[6] The feature itself had only been in its original library for a little over six months.[7] I searched the archives of the main ruby lists, including the Japanese ones, and could find no reference to discussions on the process of including this in the language.

The code change is visible on GitHub and we know who implemented it. In the committer’s comment we can glimpse the origin of this beloved feature: ‘imported from Active Support’.[8] But ‘symbol to proc’ is not the most readable feature to have. It uses an ancient symbol, the ampersand (&) that means ‘AND’, but changes its meaning to ‘turn this into a procedure’. Turn this symbol into a procedure, transform :positive into { |x| x.positive? }. In a community that praises readable and intelligible code, how did this hazy and ugly feature get merged in? Did Ruby people find it pleasing in some unexpected way? Could not-very-readable code be considered an object of beauty by the Ruby community?

Consider this, Daniel asks. There is a set of numbers and I want to know if they are positive or not.

```
[1, 2, 3].all? &:positive? => true
[−1, 2, 3].all? &:positive? => false
```

The first set is, but not the second one. Simple. But what is that ampersand symbol (&) doing there? What is it doing between the ‘all?’ and the ‘positive?’

I had used this symbol many times before I stopped to think about it. Only after a conversation with Murray did I come to appreciate everything that was packed inside this method. Murray, one of the main organisers of the London Ruby User Group (LRUG), wrote the program that allows you to 'listen to a PDF' (Chapter 9).

'You know, I think, as a programmer, you should be wary of brevity,' Murray said. We were discussing the 'symbol to proc' method, the one that Daniel wrote about in his email.

'If you want to iterate over all of the elements in this array and call one method on them,' Murray continued, 'I feel like "symbol to proc" is a good micro-level optimisation.' His point was that code shouldn't be long and filled with boilerplate, but nor should it be so brief as to be unintelligible. The ampersand method – the 'symbol to proc' – is one of those things that can get away with being very terse. It replaces six characters with only one, an ampersand, to achieve the same result.

Somehow, I couldn't square how the community accepted this compromise. It certainly doesn't make your code more readable. The braces and little bars in the first example are not exactly an easy read, but the ampersand doesn't make it better. Murray acknowledged that the ampersand does add some obscurity to the code. Even Daniel, in a follow-up email, recognises that 'symbol to proc' added some level of obfuscation to the code. Replying to a response to his email, Daniel writes:

> This has got to be the most obfuscated way to write 2 * 3:
> [2, 3].inject(&:*) => 6
> Daniel

Ruby programmer and writer Sandi Metz takes a different route than 'programming is art' in her thinking about code. Instead she wants to know about the facts and the stats. 'Our sense of elegance, expressiveness and simplicity is an outgrowth of our experiences when reading and modifying code,' she observes. 'Although your opinions about code matter, you would be well served by facts,' she adds, were it only possible to 'measure these qualities'.[9] When measurements and numbers become the criteria for beauty, the focus shifts towards the functionality of code. Weird things suddenly start being described as 'beautiful code': algorithms, C pointers, regular expressions, performance enhancements and recursion. 'When an algorithm clocks in at a quarter of the execution time of some earlier code, then the only word that I find appropriate is beautiful,' writes Charles Petzold.[10]

Metz and others take us into another arena, one that sees beauty in functionality. We have left the sphere of art and moved firmly into that of mathematics. Here code is about function, about measuring beauty and elegance to see if they stand up. It is about finding ways to measure how good code is, for example by counting lines of code, by evaluating the complexity of certain functions or by seeing how fast a different algorithm can make a program be.

Much like art, however, mathematics is often thought of as a discipline in which beauty plays a huge part. The two concepts are opposite sides of the same coin. Programmers-cum-artists write code that is visually beautiful because it is compact and terse; programmers-cum-engineers conceive of code that is functionally beautiful because it is measurably better. Programmers can and will disagree which is better, and which programs fulfil each criteria, but the two are inextricably linked.

It is through this link that programmers come to appreciate code, how they come to judge it. You learn to appreciate code as you learn how to write it. It is an acquired skill. The best code allows you to say 'that of all the ways the line could have been written, this way is the most compact, the most effective, and hence the most elegant'.[11] As a result, that line of code is the optimal line of code. Optimisation, of course, is the art of making things perfect. It often refers to improvements in code execution time, but it can, as the example of happy Ruby programmers shows, also mean code that improves developer experience, code that makes you happy. Beauty as brevity and function as beauty don't necessarily go together – not every brief code is faster, not all great implementations are readable – but they can sometimes be found on the same bit of code. This is the case of 'symbol to proc'.

In other words, an otherwise cryptic symbol – the ampersand (&) – may be excused for its brevity because it optimises your code. Instead of writing …

[1, 2, 3].all? { |x| x.positive? }…you write: [1, 2, 3].all? &:positive?

Certain programming communities take this idea of the programmer as an artist to an extreme. A good example is programming poetry competitions. Perl poetry, for instance, is not about just re-arranging words cleverly or experimenting with scansion. No: in Perl poetry, the poem is the code and the code is the poem. The main criteria for Perl poetry competition is that the code must be executable. It has to run, otherwise it doesn't count. In other words, the poem needs to make sense to both humans and machines; it needs to be both human and machine-readable. In this, Perl poetry is tied to a practice that entices many a programmer: writing code that is not understandable at all, but that still does something. Cloudy, hazy, opaque and nebulous code. The opposite, perhaps, of what a lot of programmers would associate with 'beautiful code'.

The opposite of writing readable code, writing obfuscated code implies something that is barely understandable – if at all. In contrast with Matz's idea that writing beautiful code should be like writing an essay for a human reader, writing obfuscated code takes literature to its extreme. It bends the language as much as possible, creates a gap between what things mean and hopes for some aesthetic pleasure. Instead of trying to name things in a clear and convenient way, here we have things such as 'naming obfuscation', in which variables should be named in the most incomprehensible way.[12]

There are competitions of obfuscated code, of which the International Obfuscated C Code Contest (IOCCC) is the most famous. Code submitted to this competition is judged by several different criteria and receives prizes such as 'most explosive' and 'most head turning'. Yusuke Endoh, a Ruby programmer and part of the Ruby core team, often wins the IOCCC. He writes many obfuscated programs, including a mind-bending one described as

> a Ruby program that generates Rust program that generates Scala program that generates … (through 128 languages in total)… REXX program that generates the original Ruby code again.[13]

A whole genre of programming serves to bring obfuscation to the centre of language design. The so-called esoteric programming languages, or 'esolangs', take the practice of obfuscation to another level by designing programming languages that are, in themselves, obfuscated. INTERCAL, for instance, the 'Compiler language with no pronounceable acronym', was created in 1972 to make fun of the idea that programming is about telling a machine what to do. INTERCAL is not about commanding a machine. In fact, INTERCAL programs only work if you write 'please' the correct number of times. Piet, a language inspired by the artist Piet Mondrian, works by coding with blocks of colour, generating programs that have the potential to look like works of art.[14]

Esoteric languages are, in fact, programming languages. You could potentially write web applications in them – but that would be missing the point. The idea here, says Daniel Temkin, a programmer and esolang designer, is to challenge the mainstream ideas and the default aesthetics of what it means to design programming languages.[15] Obfuscated code is clearly not about writing readable code, but nor is it about writing functionally beautiful code. If Ruby code is often thought as a compromise that sacrifices speed to have more readable code, which in turn increases developer happiness, what would be the point of writing code that is barely understandable and doesn't even take performance into account? Isn't that the definition of programmer *un*happiness?

The crucial thing is balancing the aesthetic pleasure that derives from confusing patterns and unclear language with the need to make the code actually run and the language function. If it doesn't run, it doesn't count. More than anything, obfuscating code is a practice that suggests, by contrast, the supposed 'values' of what computing must be, of what beautiful code means and what programming is all about. It shows that programming is not something automatic and disconnected from the world; it is something that involves play and fun. As Nick Montfort has observed

> All obfuscations [...] explore the play in programming, the free space that is available to programmers. If something can only be done one way, it cannot be obfuscated.[16]

Obfuscated code plays around with the multiplicity of code, a bit like Ruby does, by emphasising the many ways in which something can be implemented. In strange ways Ruby echoes this practice of obfuscation, giving freedom to the programmer to do what they want. For all the emphasis on readable code, it is quite refreshing to see that something as obscure as a magic ampersand – the & in 'symbol to proc' – might make its way into the Ruby standard library, and that it is a favourite example of how beautiful Ruby code reads.

If 'esolangs' made me rethink what programming language can be, 'symbol to proc' made me rethink the gulf between what people say about Ruby – praising a language that is easy to read and understand – and some of the things that the community value in practice. People use 'symbol to proc' and its mysterious ampersand all the time; it has become an idiomatic way of writing Ruby. There is nothing wrong with that, of course. It simply shows the distance that exists between prescriptive discourses about beautiful code and what programmers do in practice. A distinction between 'proper code' and '(actual) coding practices'.

If you want to understand a group of people, you have to look not only at what they say, but also at what they do (as well, of course, as what they *say* they do). Few people would look at an obscure symbol like the ampersand (&) and suggest that it is a beautifully clear and elegant way of writing code. Yet the same people might use that symbol in their code all the time. The difference between what programmers say and what programmers do, in this case, shows how coding isn't simply a 'type of literature'. It is also engineering; it is also about making machines more efficient. Uncovering these contradictory aspects of programming was illuminating. It helped to make explicit the hidden histories of programming. As we will see in the next chapter, such histories are crucial in shaping the way programming works today.

Notes

[1] Matsumoto. 'Treating code as an essay', 478.
[2] Matsumoto. 'Treating code as an essay', 481.
[3] Knuth. 'Computer programming as an art'.
[4] Knuth. 'Literate programming', 97.
[5] Schierbeck. 'Symbol#to_proc is just so beautiful (a)'.
[6] Ruby. 'imported Symbol#to_proc from ActiveSupprot'.
[7] Rails 'Add Symbol#to_proc'.
[8] Rails. 'Add Symbol#to_proc'.
[9] Metz et al. *99 Bottles*, 28.
[10] Petzold. 'On-the-fly code generation for image processing', 127.
[11] Black. 'The art of computer programming', 126–7.
[12] Montfort. 'Obfuscated code'.
[13] Endoh. 'Quine Relay'.
[14] 'Hello world' in Piet: https://retas.de/thomas/computer/programs/useless/piet/explain.html.
[15] Temkin. 'Esoteric programming languages'.
[16] Montfort. 'Obfuscated code'.

12
Computing gender

During the Second World War women were strongly encouraged to ‘join the war effort’. One of the ways they could do this was by taking up positions in heavy industry, for example ammunitions factories, plane construction hangars and tank manufacturing. There was room for any woman to become a ‘factory girl’; characters such as ‘Rosie the Riveter’ or ‘Wendy the Welder’ were created to encourage women that they could do such work too.[1] But if the encouragement to take up these positions was strong during the war, the push to make them return home afterwards was even stronger. Women were now expected to go back to their unpaid jobs as housewives. From the ‘home front’ to, well, just the home. However, not all women went back to that traditional role. They didn’t stay in the factory either though. They moved into the office.

In *Programming Inequality*, historian of computing Mar Hicks traces the relationship between automation and feminisation. Hicks writes about the move of early automated machines into offices and explores how office work created the foundations of the push for technocracy that happened in the 1960s and 1970s – the time when computing would start to become masculinised. Hicks focuses on the British Civil Service and the role of government in the early days of computing, revealing how women and machines entered the modern office hand in hand during the 1940s and 1950s. In 1948, for instance, the British Civil Service created a new job class, that of machine operators, which covered all ‘work on calculating, punch card and accounting machines’. Although the Civil Service had long been held as a level playing field in which class was less important than skill, many of its job classes were divided by gender. Jobs were not just implicitly geared towards women; they were explicitly defined, by the British Civil Service, as being ‘for women’.

The creation of this new class of civil service workers to work on punch card machines, to be purely filled by women, formalised the relationship between women and automated systems. Women and the machines that processed the increasing amount of public data would go hand in hand. Any computing work would now be deemed women's work – labour that was functional and low-skilled from the manager's point of view, and valued less than other clerical work done in any office environment. A crucial element here is a change in the perception of what a 'machine operator' does. This is what Hicks calls 'labour feminisation' – something different from being simply work done by women. To feminise labour in a patriarchal society implies changing the perception of how skilled a certain task is. In other words, it means that women are good enough for this job because it doesn't require much skill.

> As a historical process, feminization makes work less valuable as women become the majority of those found in it, setting up a vicious cycle in which workers' status drops further as the work becomes devalued, sometimes out of all proportion to the work's actual content. Once this process has occurred, even men who return to these job categories are subject to the effects of labor feminization, meaning that feminization can affect workers of both genders negatively.[2]

Hicks's narrative about the work of women in the British Civil Service stresses yet another point: these positions had a clearly defined expiry date. Women were not expected to work as machine operators forever. Like other jobs designed to attract women workers, these jobs had a comparatively high initial salary but very slow progression. There was an expectation that these women would leave the service when they married and settled as housewives. If the starting salary was good but did not improve, it made no financial sense to stick around for long.

In the 1960s, Hicks tells us, the British conglomerate J. Lyons & Co. became the first company in the world to apply computing to its business interests. The company bought plans from Cambridge University's 'Electronic Delay Storage Automatic Calculator' (EDSAC) and successfully used them to create their own system called 'Lyons Electronic Office, known as LEO I. Lyons created a whole women-only team who would call each Lyons manager around the country to ask for inventory data. The women would then feed this information into LEO I, which would spit out how much bakery products each shop would need for the next morning.

Handling inventory was therefore one of the ways the first computers were used in business. It is interesting to think that, almost 60 years later, we too would be attempting to come up with a system – the new availability service – to handle inventory at Upstream. Soon the LEO I started being used for payroll, a development that eliminated the need for two dozen workers in accounting. Many other large companies followed, with the retailer Sainsbury's and the pharmacy Boots being two of them. However, it was the British government that played the biggest role in the early adoption of computers.

The effects of labour feminisation of machine workers lasted a long time, defining the relationship between workers and machines until the 1970s. Slowly, however, managers started to try and undo it. As computation grew, and therefore grew in importance, it was reasoned (by managers) that there needed to be a way for men to enter this space. Computer work had to become more attractive to them, with better prospects and, crucially, a better perception of what the work entailed. It needed to look like a man's job, not just something that any former 'factory girl' could do.

A crucial change happened during the 1960s, when the image of the relationship between women and computers changed. Advertisements by British computing companies 'became the object of a specific kind of managerial "male gaze"'.[3] Instead of a combined women + machine combo, the 1960s created ads that portrayed women with mini-skirts sitting on high office chairs. Such ads showed how the male gaze onto female machine operators started to foreground the women – sexy and dressed in office clothes – and push the machines themselves to the background. The relationship between women and machines was no longer as important as it had been in the 1940s and 1950s. Now it was reduced to that of a conduit: 'she transfers the order data to the computer, via the terminal'.[4] If there had been little sexual subtext before, the gaze had now become a heteronormative male gaze. The machine was still there, but it had been feminised. Computers were now given female names such as SUSIE (Stock Updating and Sales Invoicing Electronically), BETSIE or SADIE. It seemed that by extending a heteronormative male gaze towards the machine, retailers were trying to make them attractive to men.

One of Hicks's conclusions, after years of researching the role of women in the history of computing is that gender, not women, should always be the category to use. Only by thinking about gender, by looking at the relationship between men and women and non-binary and trans, will we start to understand how 'patterns of underachievement and perceptions of women as less technically competent persist within Anglo-American culture, business, and higher education' and how 'the professional identity of computer workers is tied to a history of structural discrimination that has nothing to do with skill'.[5]

Male dominance in computing today makes it hard to imagine that it was ever different. Quite often, when we look at the history of labour and professions, the space for women is something that has slowly been carved out and conquered, over decades, from male-dominance. Men establish themselves and are only slowly made to loosen their grip on an industry. Computing, however, is a different story. Initially quite open to women, it was only from the 1970s onwards that it would start to become masculinised. Nathan Esmenger, another historian of computing, has tried to understand things by focusing on how the 'computer boys' eventually took over a field that had, for decades, been a working space for women.

> It wasn't that women were uninterested in computing, or unprepared or constitutionally disinclined to participate, the historical evidence seemed to suggest, but rather that their participation had been systematically ignored or underreported.[6]

In the 1940s the first famous electronic computer, the ENIAC, was operated by the so-called 'ENIAC girls'. They were responsible for turning the cranks that would make it calculate whatever was needed. There were plug panels, just like telephone switchboards, which 'reinforced the notion that programmers were mere machine operators, that programming was more handicraft than science, more feminine than masculine, more mechanical than intellectual'.[7] According to Esmenger, famous early computer scientists such as Herman Goldstine and John Von Neumann took the distinction between manual and intellectual labour to heart. If men should be doing the brainy stuff, like planning and designing things, then women could be responsible for the implementation. But this distinction between analysis and implementation quickly broke down in practice. The actual implementation required a lot of adaptions, compromise and improvisation due to the machine's limitations. It wasn't just the case of implementing the design as it had been designed. Fine-tuning and last-minute adjustments were always required.

What's really puzzling is this: how did we go, in the space of two decades, from images of women sitting on office chairs operating the terminal to images of young men sitting alone at home behind a screen?

Slowly male engineers created the perception that programming was a complex form of analysis that required the talent of 'unique individuals' who loved mathematics, engineering and tinkering with things. A series of aptitude tests and personality profiling reinforced the idea that companies should look for innately talented individuals who were born to solve puzzles but were unable to handle interpersonal relationships. As Hicks explains, in Britain,

> gains made by women operators as a class would begin to disappear in the late 1960s as employers and policymakers placed more hope in the young men who were expected to become experts in the application of computer technology in order to save the British economy. A potent new role for computer workers – that of the technocratic expert – was emerging in society and government, and structural changes made in the public sector would begin to solidify this new identity.[8]

The technocrat would be the one to save the computing industry from decades of feminisation. A type of saviour who echoes what the artist Grayson Perry dubs 'Default Man'. In his exploration of masculinity, Perry discusses a series of characteristics of men's behaviour, style and attitudes.

> They dominate the upper echelons of our society imposing, unconsciously or otherwise, their values and preferences on the rest of the population […] They are, of course, white, middle-class, heterosexual men, usually middle-aged.

In his broad brushstroke, Perry focuses on the various machismos inherent in Default Man behaviour. The phallic symbols, the lycra bike riders, the fitness freaks. Above all, he writes, Default Man believes in being an individual.

> He prioritises 'rational' goals like profit, efficiency, self-determination and ambition over emotional rewards like social cohesion, quality of life, culture and happiness. Centuries of patriarchy have fashioned the world to reflect and favour the middle-class masculine viewpoint […] Default Men are, of course, full subscribers to that glorious capitalist project, they are individuals.[9]

Working as a programmer, I inhabited spaces that were overwhelmingly masculine. Offices, meet-ups, conferences, meetings. Much more masculine, in proportion, than the academic environments I'd mostly inhabited before. But geeks are not typical machos; they perform a different type of masculinity. They share, with Default Man, the belief in efficiency and individualism as the ultimate directions. But they are not out on their bikes wearing spandex or wearing a bushy beard in lumbersexual ways. Geeky masculinities draw on other symbols to create a narrative of what counts as 'normal' or 'default'. What is normality, though, when it comes to gender?

Imagine that gender is like the law. We accept the law and give it a certain authority; we behave in a certain way. We act how we think the law expects us to act. In other words, it's not that the law creates obedient subjects, but rather that people create themselves as law-abiding citizens by performing what is expected of them. 'I wondered whether we do not labor under a similar expectation concerning gender,' writes the philosopher Judith Butler. An expectation that 'ends up producing the very phenomenon that it anticipates'.[10] In this sense, gender is a performance – which doesn't mean that gender isn't real. It means that people perform their gender many times a day. It is a repetitive and ritualised act, a 'persistent impersonation that passes as the real'.[11]

In computing, the gendered expectations that inform how people act are defined by this peculiar past in which women were erased from the discipline's history. Through years of persistent acts, the history of computing created a 'natural order' in which geeky men now seem to have always dominated the field – contrary to what history tells us. It is against this natural order – this law – that programmers perform their gender today. Reflecting on this inaccuracy might help us to challenge certain assumptions of what counts as 'normal', 'correct' or 'proper' programming. It will help us to see how being a programmer is not only an individual performance; it is also a gendered one.

> Realness, as the men in Paris is Burning understood, is the pinnacle of heterosexual masculinity. Realness, authenticity, genuineness, legitimacy: all qualities that back up a man's feeling that masculinity is somehow the baseline from which all other identities are judged or attached. Which in turn implies that other identities, feminine or homosexual, are not real, not authentic, not legitimate. But sorry, guys, realness is an act too![12]

Writing about men of my generation, millennials, Perry says that we 'seem more at ease with gender fluidity' and are 'less afraid of appearing gay'. I'm not so sure. I've seen men being judged and joked about when sharing their stories of sexual fluidity and gender experimentation. Perry is hopeful of a new gender dawn and the emergence of a plethora of masculinities – but, at least in the case of tech, the history of computing seems to suggest that heteronormative masculinity is not shrinking or diversifying: it is growing. As new forms of automation and financial insecurity emerge, perhaps driven by waves of large language model machines, God knows what male tech geeky spaces will become. What seems to be characteristic of the history of programming is the increase in the maleness of its social space. Computer programming is not an area in which masculinity is threatened or 'in crisis' – on the contrary, it's thriving.

Notes

[1] Wikipedia. 'We can do it'.
[2] Hicks. *Programming Inequality*, 77.
[3] Hicks. *Programming Inequality*, 113.
[4] BARIC commercial leaflet *apud* Hicks *Programming Inequality*, 123.
[5] Hicks. *Programming Inequality*, 231 and 237.
[6] Esmenger. 'Making programming masculine', 120.
[7] Esmenger. 'Making programming masculine', 123.
[8] Hicks. *Programming Inequality*, 147.
[9] Perry. *Descent of Man*, 23.
[10] Butler. *Gender Trouble*, xv.
[11] Butler. *Gender Trouble*, xxviii.
[12] Perry. *Descent of Man*, 63.

13
Proper programmers

'One thing I was told when I started programming was that I could not be a good developer because I've never watched *Star Wars*,' said Marie. She was a witty and sharp-tongued developer at Upstream and another key member of our book club. Always ready to give her opinions, and with great knowledge of many things, including the new flavours of ice cream that come out each year, Marie trusted me one day with her opinion about what she called the gatekeepers of programming, the members of an 'inner circle' that was hard to access. If there is a lot of talk about increasing gender, racial and cultural diversity in tech, our chat about gatekeeping and geeky masculinity put things in perspective.

'What does *Star Wars* have to do with programming?'

'Well, nothing and everything,' she said. 'You won't learn much about Object-Oriented Programming or software architecture from the films.'

On the other hand, by being familiar with lightsabres, Darth Vaders and Jedi warriors you are partaking in the cultural references on which a lot of programming culture was built. If someone makes a Jedi joke in a meeting and you don't get it, or at least if people see that you don't get it and make a comment, you lose prestige and clout. A wise nerd, the programmer must be.

While Marie explained this to me, I suddenly remembered a conversation with another programmer from the team that had taken place more than a year before. We were in a lift together and I was trying to break an awkward silence by asking him about something that I assumed he'd be interested in. I asked if he'd seen the latest *Star Wars* film. Without turning his head towards me, and seeming quite embarrassed, he said

‘I’m not really into *Star Wars*. Please don’t tell anyone.’ As I brought my attention back to what Marie was saying, she had already moved on and was talking about job adverts. She was describing job adverts that declare ‘attractive bonuses’ such as after work role playing game (RPG) sessions. She explained how such job adverts discourage lots of people from applying because they think ‘oh, fuck, how boring’.

‘What about the latest Dungeons and Dragons campaign, ‘Dragon of Icespire Peak’? Isn’t it great?’ I thought about saying to her, but didn’t.

‘I’m going through a book about AR [Augmented Reality],’ she continued, ‘but all the examples are, like, there’s a mini tank, a mini car, etc. You know, it’s also kind of boy-centric in all the examples. For the big boys to be able to relate to the small boy inside them and for the girls to feel excluded.’

‘Do you think this is a conscious effort?’

‘I don’t think any of the guys actually, consciously, try to exclude women from this club. They just don’t understand that examples about displaying Arsenal logo or football scores are not actually appealing to everyone as a cool thing to build.’

It would take a while to unpack what Marie summed up in those sentences. Take the idea that men don’t consciously try to exclude women, for instance. The sort of unconscious bias that is quite tricky to discuss. ‘What’s wrong in building a project that generates football logos?’ Well quite a lot, as it turns out. There are infinite varieties of projects that you can build; by focusing on clearly male-oriented ones – the train-football axis of masculinity – you end up limiting the number of people who could enjoy the content of what you are creating and the process of creation itself. But the problem is, as Marie was saying, that most of this is unconscious. This is probably why many men get so angry when someone suggests that the examples they are using – such as the car and the Arsenal logo – are not very cool.

Because the team at Upstream wasn’t that big, I ended up getting involved in hiring new programmers. I’d only been there for a few months, but I think Charles and the other senior developers felt it would be good for me to be there. We analysed their code tests, interviewed candidates, made them do some live coding and decided who we should accept. We hired a couple of developers, both of them women. This was a conscious decision by a couple of the senior developers, who felt the team was excessively male dominated. There was only one female developer in the team when I joined. ‘At least there is one,’ another programmer told me. ‘It’s the first time I’ve worked in a team that has any women in it.’

‘Hey, welcome to the team. Let me know if you need anything,’ I said to Lindsay on her first day at Upstream. One of our senior managers who was walking past heard me saying that, and felt like he needed to jump in the conversation. Before Lindsay had even replied to what I’d said, he said ‘I just wanted to say that we hired you because you are the best person for the job, not because you are female.’ He said that and continued walking. Lindsay looked at me with raised eyebrows and open eyes.

Months later, I relayed that conversation to an American Ruby developer called Mae when we spoke about her career trajectory. Part of my interviews were deliberately with people from outside Upstream and Mae works at a big Ruby shop. I mentioned that I didn’t really understand why he had done that: no one had asked him anything, he wasn’t part of our conversation, he just felt that he need to say that. ‘That’s not the worst bit,’ Mae said. ‘The worst part was that he probably thought he was doing the right thing.’ I told her how the topic of gender often arose by itself when I talked to women, but that it almost never came up when I was talking to men.

‘Well, that’s pretty obvious,’ Mae said. ‘It’s usually people in the position of privilege who can ignore the issues. Some recent discussions within the (Ruby) community highlight this too. They highlight the difficulty in having wider social debates within the community.’

‘In light of the Black Lives Matter movement,’ I told Mae, ‘someone at Upstream suggested we changed all references to “blacklisting” and “whitelisting” when we control access to certain parts of the product. The suggestion was to switch to “block-list” and “allow-list” as non-racialised ways of dealing with access. There was quite a bit of pushback on that, though.’

Such pushback was not surprising, according to Mae. ‘You know, the tech industry’s motto of “moving fast and breaking things” works very well except when it comes to addressing sexism or racism.’

Mae told me that the problem of gender in programming has to do with the fact that women and non-binary people in the Ruby community produce very little of its technical content. There are exceptions such as Sandi Metz, whose books we read in our book club, but most blogs, technical posts, programming guides and books are written by men. Regarding this, Mae explained, you need to be very careful, as a woman in programming, with the style that you use to voice your opinion on things. Several people in the tech world – the Ruby community included – tend to write articles and books which are very forceful: ‘This is my strong opinion on this!’ But a woman could never write something in that style and survive the backlash. ‘Women can’t do this,’ she said. ‘If I want my writing to be read, I can’t go super strong.’ The expectations of what technical writing should look like are, therefore, defined through gendered lines. Women and non-binary Rubyists must write in accordance with these unwritten norms,[1] which have become part of this cultural space.

Having strong opinions does come naturally – or culturally, we should say – to some programming books. Organised by Robert Martin, known as ‘Uncle Bob’ in the programming world, *Clean Code* is a classic book in programming, and the first we read together at Upstream.[2] The text encapsulates a few traits of the programming world. It has an imperative style that tries to teach the ‘best practices’ a good programmer ‘absolutely needs to have’. It uses acronyms and catchphrases to maximise its influence – the principle of avoiding repetitive code, for instance, is called ‘Don’t repeat yourself’ and abbreviated as ‘DRY’. It has examples in Java, which are a bit dated and hard to read. And it only features knowledge from white male programmers. That’s a whole programming world and a half in a few hundred pages right there.

At the beginning of the chapter on ‘Comments’ there is a depiction of three women sitting on a desk drinking coffee and gossiping. Underneath the drawing is a quote that reads ‘Don’t comment bad code – rewrite it’. Code comments are lines of code prefixed by a special symbol that tells the computer ‘this is not code’. Programmers use comments for many things, but here the message here seems quite clear: comments are like gossip, which serve no real purpose, is trivial and can distract you from the proper job of rewriting bad code. Use comments only if there is no other option. Most people in the book club absolutely hated this chapter. Everyone agreed that the drawings were terrible. Lindsay said, with heavy irony, ‘They captured exactly what we [women] are like’.

When Marie joined Upstream, she was added to a Slack channel called ‘#badcode’. She was taking over the role from a developer who had just left. This was the chat that resulted in her being added to the #bad-code channel.

> <Marie> was added to #bad-code by Developer1
> <Developer1> @Marie To slate me after I leave
> <SeniorDeveloper1> its traditional to blame the last person that left for everything, especially if you’re the only developer working on something
> <SeniorManager1> Is it time to retire some traditions?

<SeniorDeveloper> no [point up emoji]
<SeniorDeveloper2> this is a big part of our identity as a company

Developer1 adds Marie to the channel so she can 'slate him' after he leaves, meaning that she can talk trash about his code and complain about the 'millions' of bugs he left. SeniorDev1 steps up to explain that such blaming and shaming is traditional and should be respected. A manager intervenes, but gets shut down – not only by the developer's 'no', but also by another developer who seconds the tradition. 'Blame culture', as it is often referred to in tech, is about banter – especially the type of banter in which you try to make others feel stupid through making witty comments. Some managers do try to combat 'blame culture', but programmers can be quite reluctant to accept that, especially because it relies a lot on making fun of people's code.

The conversation on Slack when Marie joined might look a bit harsh. It certainly exemplifies certain postures in tech, but we should take such defence of the company's 'identity' with a pinch of salt. Very little slating of colleagues actually occurred in this channel. Some certainly did in conversations and during stand-ups, but there was not much public 'blaming and shaming' of anyone. Most of the time the #bad-code channel was used to showcase random bits of terrible code, or even some programming gifs that people came across during the day. I'm not sure what to make of this contradiction between lauding blame culture and not really practising it.

I posted an example on #badcode once. I was working on a service that no one really wanted to work on. After working on that service for a couple of days, I felt there was so much truth in that comment; it described that code perfectly. There was no time to fix that function. It just had to be done like this. 'The proper use of comments,' writes the author of *Clean Code*, 'is to compensate for our failure to express ourselves in code.' He goes on to state that 'comments are always failures', arguing that we shouldn't celebrate them; we should just accept that only sometimes do we need comments in the code. As a programmer, however, I came across many instances in which comments were not exactly the failure of expressing oneself, but rather a way of expressing things that weren't computable – like the context for a piece of bad code that I posted on the channel.

```
# This is a fucking mess, but I'm not fixing it as there's
# no time to untangle the stuff on the producer portal end.
# Not sure there is a clean way to deal with that
# at the moment. Or that it matters.
```

‘I know this code is terrible, but I just won’t fix it,’ the programmer seemed to say. They left it in the comments, the only place where it could go. Maybe it was an expression of his team’s failure in fixing that code, but then perhaps there was no way around it. Like half-broken monoliths, half-broken code leaves behind comment jewels.

Contemporary code repository managers such as GitHub or Gitlab allow you to check which developer changed each portion of a file. This is a feature called ‘blame’. It is basically a nice and easy way of ‘blaming’ someone if the change they made ends up having unfortunate consequences, or if the code is particularly bad in style. One of the funniest comments I came across while working at Upstream wasn’t a failure either. It was a way of leaving something for posterity.

```
# Don't believe the blame. Ivan made me do it.
```

Below that particular comment was a function that updated all elements on a table in the database. There was not much there, the class was clean and readable. So why was this developer saying that someone made him do it? And why shouldn’t we believe the blame? The context seemed to be similar to the comment that declared ‘this is an absolute mess’: major forces of the world made me do this. In the first case, the code depended on another service; there was just too much mess that had to be sorted out on the other end before sorting this particular bit of messy code. Here the programmer knew someone would eventually come across that bit of code – like I had – and might think that the code didn’t make sense. They would look at the history of the changes and find out that the ‘blame’ pointed to that programmer. But it wasn’t him; Ivan had made him do it.

If there is one maxim that can be applied to most human cultural groups, it is that there will always be some people who believe they embody the original and proper spirit of the group. They do things in a way that separates them from the others and gives them a feeling of superiority. It is the group of the ‘select few’. Although the existence of such groups might be quite widespread, the shape that the group takes, the cultural form that its activities have and the way in which the select few perform vary significantly. In programming, they are the people who refuse to use any form of Graphical User Interface, known as GUI. These ‘proper developers’, as Marie described them to me, do everything from the terminal (or console), the ancient black screen that harks back to the days of big mainframes that occupied whole floors. A terminal or console from which you can type your instructions to the machine. No clicks, no mouse, just the keyboard. That's how true developers work. While Marie and I talked about these issues, she said that maybe they were both connected. Maybe the gatekeepers of the 'inner circle' and the hackers that only use the terminal were connected. Both made her feel excluded, but she couldn't work out why. Suddenly, as we talked, she realised what the link was.

'There's something in that,' she said. 'Maybe deciding who is in and who is out. Yes!' With a big smile on her face, Marie pursued her train of thought.

> It's the knowledge of certain things, like Star Wars and RPG, and the use of certain tools, like the terminal, that differentiate who is 'in' and who is 'out'. And if you don't meet those secret criteria then you're an outsider; you don't fit in and you're not a developer. Not a proper developer anyway, right? And that's an important distinction: to become a proper developer.

A proper developer with a proper gender, enjoying proper games and the proper football club, writing proper code with the proper comments. No need to blame anyone, no need to. Just bring the lightsabre in, speak the ways of the Force. The proper Jedi for the proper codebase, proper monolith. We'll get there soon enough. We'll get there. Communities may be welcoming, but they are also spaces that reproduce toxic masculinity. Proper developers who like things to be 'pure'.

'Purity,' reflected Marie. 'I generally don't understand this approach. Kind of difficult to say more. I think that, in a way, it excludes people who have a more common-sense approach to stuff.'

'What do you mean by purity? Is it because code should be written in a very specific way?'

'I mean, I'm opinionated about where the curly braces go, but certainly yeah, people are opinionated about stuff like that a lot. I'm dyslexic and I use git tower for all my commits because I can see what's going on. But a lot of people say they only use git from command line like it is another badge of honour.'

Proper programmers and the inner circle: two things that Marie felt blocked her and other people from feeling more comfortable and at home in tech. Git from the command line, not through some fancy graphical interface. She became thoughtful.

> There's no real value in what you choose to tackle the task with. I think it's easier to see those things if you come into this whole community from the outside. So, if you had a career beforehand that was completely different, then you look at everything with fresh eyes. I guess if you go to uni you absorb a lot of this as the 'normal' way you do things. But when you come to it, what's the value of this? I don't see it. Maybe there is a value of doing it from command line.

Her perception impressed me. 'Do you think it's been like this a long time, Marie?'

'Well, my mum was a developer. More than half of her team were women. And that was true maybe until the late 80s. I think what happens is when a job gets better paid than it used to be, you get a lot of men coming in. The same thing when you see a lot of women going into a profession, only then the salaries drop. It's just awful. Having more women signifies that a profession is less of a desirable career path.'

'I don't think a lot of people realise that there used to be way more women in tech.'

'I think people know that for sure. They just don't care. How many times have I said something in a meeting to be told it is absolute nonsense, then Paul says 'actually, it's not nonsense' and everyone agrees! It's only when it kind of gets that confirmation from the man that we can have a discussion. Which is … it is annoying and it's discouraging. As far as I know, most women don't make it past five years in software development; they just cannot handle it and they drop out. There's a lot of that, just not being listened to.'

In 2022 I sent the first draft of this book to a few research participants. Getting feedback was a part of a dialogic process that began before I even started the book when I published blog posts – some of which would become chapters in the book. People's perceptions of what they told to me were particularly interesting. Some programmers told me that it was weird to see their own words in print. It made things more permanent than our conversation. Most people were fine with the quotes I showed them, but some did have comments or things they wanted to change. One of them was concerned that it didn't represent his opinion. He offered some edits, which I accepted. He still wasn't happy, though, and offered more edits, then some more. It got to a point where I barely recognised our initial conversation in his edits. I suggested that perhaps I shouldn't quote him at all. He was OK with that and I cut that bit out.

I had a long, back and forth discussion with a participant who offered a lot of edits to her quotes. We must've sent a dozen emails to one another, each of them with rewritten bits. In the end, she wasn't worried about the content of what she'd said to me or about any perceptions that future readers would have about her. She was worried that she didn't 'sound like herself'. This was quite tricky, because our opinion of ourselves is probably the most subjective of all. I included most of her edits in, but there was one that I couldn't accept. 'I never would've said something like that,' she told me. 'It just doesn't sound like me.' I showed her the recording. The comment was there, word for word.

'I think some of the names you've used are too European,' Charles told me when we met to talk about the book. 'I spent years trying to build a diverse team, and the names you've chosen don't reflect that.'

'Which names in particular?' I asked.

'Martin and … well, Charles.'

I smiled. It was a fair criticism and I took it on board. I changed some of the names, including Martin. I'd grown too attached to Charles, though, so I kept it.

Marie gave me her feedback on the manuscript when we met for drinks in central London. I was anxious to get her opinion on it because she is often very direct and honest. I knew she wouldn't hold things back, which was important – if hard – at that stage for me.

'It reads very smoothly,' she started. 'But there are many things you could improve.' And that's how a five-hour feedback session started, through which I took pages of notes. She was very thorough and we went through each chapter together. She would point out the things that she felt were weird or unexpected, things that could be explained better.

'What about the bits where we talk about the inner circle and the gatekeepers?' I asked Marie.

'I felt very seen,' she replied.

The draft I sent Marie had a few chapters missing. Not because I didn't want her to see them, but because I simply hadn't written them yet. I was waiting to include the conversations I would hopefully have when I finally made it to Japan.

Notes

[1] Butler. *Gender Trouble.*

[2] Martin. *Clean Code.*

Part IV
Tokyo days

14
Not my type

'I was born before the beginning of time, you see?'

'What do you mean?' I asked Neil.

'Before UNIX was created.'

I laughed. 'Is UNIX the Jesus Christ of Computing, then?'

'Oh, God no!'

We both laughed a bit too hard. An awkward moment of silence between two people who barely knew each other. We each drank some coffee. I looked out of the window, Neil reached for his phone. It was June 2022 and I'd been in Brighton for a couple of days, attending Brighton Ruby. This was the first face-to-face conference I'd been to after Paris RubyConf more than two years before. I didn't know Neil until the day we were introduced to each other at a pub in The Lanes, one of Brighton's bohemian areas. Hachi, a Ruby developer that I knew, texted me to find out where I was, then suggested I joined him and Neil at the Druid's Head. The pub was super loud, but Neil and Hachi were deep in conversation about Ruby. I introduced myself to Neil and sat down. Hachi mentioned the research I was doing to Neil and added:

'Gui really wants to know about the Ruby community in Japan!'

'Well, yes …,' I mumbled, but Neil immediately jumped in.

'It's hard, over there, in Japan. Communication is very difficult.'

'Yes!' Hachi interrupted him. 'I've been trying to talk to one of the Ruby core maintainers and it's so hard.'

'Let me guess. He takes a while to reply?' Neil asked Hachi with a smile.

'Yes!' Hachi replied. 'Sometimes a whole week!'

Hachi was one of the top committers in an important Ruby tool, which was maintained by a Ruby core team member from Japan.

He'd been struggling to talk to the maintainer. He couldn't get access to the repository even though he was a big contributor. Neil listened to him, patiently, for a long time. Eventually he started sharing his experience as well.

> 'You know, Japan is very Japanese. When I moved to Japan, I had to assemble a whole team of Ruby devs. I didn't speak any Japanese; they spoke some English. I think what they did to me was a social experiment: throw a foreigner who doesn't speak Japanese into a very Japanese company. I very quickly set up a Tokyo meet-up. But I could never learn the language, it was too difficult for me. Maybe if I did it could've helped with the communication. You know, many people say that you have to learn English to work in this business, but I think that, in the case of Ruby, we should all be learning Japanese.'

Ruby was created in Japan in 1993. Although it is used by thousands of developers worldwide, a core part of the maintainers of the language is in Japan. Maintainers are responsible for, well, 'maintaining' a certain part of the Ruby ecosystem. For instance, perhaps they ensure a specific library is constantly being developed; or perhaps they are responsible for improving the language itself. The separation between Japan and the rest, however, is both real and not real. It feels real when an issue that is raised doesn't get a reply, as Hachi told me. But it also feels less real if you manage to get in touch with some of the Japanese developers. As with most things in the world of computing, it all boils down to cultural differences expressed through language – not only programming languages, but human ones too.

Neil and I left the pub and headed to a karaoke bar where a bunch of people from the conference had gathered. It was a squared room with grey leather sofas going around three walls. There was a big TV, two mics and a tablet to search for songs. Devs taking turns on the microphone, beers flowing around, no product managers in sight. When I sat down, Neil came close and shouted in my ear, 'is this an anthropological experience or what?' Andy, the organiser, was not only there but also the most talented singer in the room. He was always singing, all the songs. Even without a microphone, Andy was always ready to take over. After a few minutes, I felt the pressure to join in. I really wasn't sure I had it in me, but I knew I had to try. A song that I knew the lyrics to came up: Fleetwood Mac's 'Sands of Time'. I jumped out of the seat, gave my beer to Neil, grabbed the mic and started singing. ‘Nice!’ Andy shouted when the song finished.

Neil and I left the karaoke bar after an hour. It was already late, and we both had trains to catch the next morning. I threw a last cigarette on the floor and we shook hands. I promised to write to him so we could have a chat about the community and his trajectory as a developer. ‘Just let me know when it works for you,’ he said. ‘I’m sure I can find some time.’ About a month later we did find some time. On Zoom, of course.

‘I’m glad we’re doing this,’ Neil said.

‘Me too. Do you mind if I record it?’

‘I guess …,’ he replied, clearly surprised by my question. For the rest of the interview he would oscillate between moments in which he was comfortable and talkative and other moments when he was stiff and controlled. In these harder moments, Neil sounded as though he were reading from a script. He would say things like ‘My company has contributed …’ or ‘The important thing about the community is being nice to each other’. It sounded very rehearsed. I felt so bad for having asked to record it, but there was little I could do at this point. In research, it’s like that sometimes. Most people are OK with recording and trust you to use that data with care and respect. Occasionally, though, you get someone who is very wary of having their thoughts recorded, their words written down. Still, even when that happens, you might get lucky and find something to talk about.

‘I think it was the Ruby conference in Los Angeles, whenever that was, not long ago. I remember that Koichi was there, and there was Matz there, and some of the Stripe people were there, and they had a bit of a kind of a meeting. Someone was telling me about the ins and outs of it; they tried to figure out what the path was.’

‘What path?’

‘Oh, the path to add types to Ruby. You know, Ruby is a dynamic language, so it doesn’t have types to tell you which function returns what. It just figures it out during runtime. It’s different in other languages, in compiled languages. In those the compiler checks that functions are returning the correct types. Like, is this an Integer or not? Does this function return a String or something else? The compiler will do that for you.’

Typing is a way of assigning a data type to variables, functions, objects and other constructs of programs. A way of saying that the function ‘add two numbers’ will always result in a number, for example; or that another function will always give you a set of characters (and not numbers).

Loads of old (and new) programming languages have types, but recently there's been a surge in languages using them. Over the past decade pressure from the Ruby community has grown to add types to the language. This is a big issue; Ruby is a dynamic language and it doesn't have types. Famous Ruby programmers have publicly said they don't like types, including Matz himself. But the pressure continues to grow. As Neil observed,

> 'I think that Matz, I don't want to put words in his mouth, but I think he has a much more long-term view. He's my age, a little bit older. He's thinking, 'Ruby has been around for 25 years. But 25 years from now, are types going to be a thing?' I think he feels in his mind like that. It kind of pollutes Ruby as well. For example, you come to a code base in five years and types have been solved in another way by the computer. Suddenly, the Ruby interpreter can handle types, but now you've got a million lines of code and 400,000 or more type annotations. If you've got thousands of lines of code all over the world, millions of people, maintaining this code for something that doesn't apply any more. What do you do then?'

I saw his point. 'So, you think Matz is betting that the interest in types that many modern languages have might go away?'

'Exactly. It's kind of similar to introducing Rust instead of C to handle some stuff in Ruby. I'm not sure where they are with that, but Ruby uses GCC, which is 50 years old. GCC is not going anywhere. It's going to be here forever. I think Matz thinks that, you know, after he's done and gone, in another 50 years, will you still be able to build Ruby? Jumping on the latest bandwagon, which is what most technical people tend to do – and I used to do it as well – is complicated. 'Oh, wow, this shiny new thing, it's amazing.' But then, you know, five minutes later you're onto something else.'

At the European Ruby Conference in 2021, Matz said in his keynote speech that the landscape of typing in contemporary programming languages is changing: 'Now is the time of static typing'. Modern languages such as Go, Rust and Swift all have static types, while old ones such as Python, Php and JavaScript have added syntax for it. The way Matz was going, it almost seemed like he was going to announce that Ruby 4 would have fully-fledged static typing. But then he asked 'Shall we do that? I don't think so. We will have static types without adding new syntax.'

On Discord, Rubyists kicked off a conversation while Matz was talking. One person explained what static typing was. ‘A type checker,’ said one, ‘can check your program without running it and that’s the “static” part of “static typing”.’ Someone else emphasised how static typing reduces the cognitive load associated with programming by preventing silly mistakes and bugs that a type checker can easily catch.

Writing code without specifying types sounds dangerous and problematic to a lot of programmers. But there is a flip side. If most of the time you can predict or guess what type functions and objects have, then you can write code faster. If you don’t need to worry about types, you’ll have a better programming experience. Thinking about types is a terrible task. No one likes to think about types – you only do it if you have to. As one conference goer explained, ‘I felt liberated from having to think about types when I switched to Ruby’.

Other people on Discord focused on what could only be described as animal welfare concerns: they were worried that a type checker would end up killing duck typing.

‘What’s duck typing?’ I asked on Discord.

‘A way of performing the “duck test”: if it looks like a duck and quacks like a duck, then it probably is a duck. Duck typing is trying to infer what type a function or object has by looking at its behaviour,’ someone replied, very quickly.

Developers use duck testing a lot in Ruby because you often have to think about how better to interact with certain objects and functions. You need to think about what kind of object you’re getting from it. Is it an Integer? Is it a String? Is it a Hash? You also need to keep an eye out for rogue types, perhaps an Array that should only contain strings but that, somehow, has a funny Integer in there. You’ve got to make sure that they are all ducks. No geese, no swans. Although Ruby hasn’t totally accepted that types could be a thing in the language, there is a clear sense in the community that types is where things are going.

When I spoke to Aaron Patterson about his trajectory as a developer, we also talked about types. Aaron lives in Seattle and is a key link in the relationship between the American and Japanese developers who focus on open source. If Seattle was crucial to the development of open-source software, as we saw in Chapter 2, the Ruby community in Seattle was, and remains, crucial to the community and the language. Aaron speaks Japanese; he is a self-confessed computer nerd, hardware hacker and struggling baker. He currently works for Shopify, one of the biggest Ruby companies. I asked him if he felt that types are something that is growing in the community, something that people talk about more and more.

‘To me,’ I told Aaron on Zoom, ‘it feels like a big change from the way things were done before, a big change in the cultural landscape of the community. If we take Why as an example of that funnier or quirkier time of the community, it seems that the community is in an entirely different place now. Does that make sense to you, Aaron?’

‘Oh, for sure. It makes total sense. That change has to do with it not being a community of hobbyists any more, but of people writing a lot of company code.’

As a Rubyist, now, company code has become more important than personal projects. And company code needs standards, some sort of control, some way of making sure that things are safe. It all sounded as though Aaron was going to say he loved types – but I was wrong.

‘So what’s your take on it?’

‘Yeah, I’m not a fan,’ he said and laughed. ‘I’m not a fan of putting types into my code, so I will not do it. I mean, I understand, I understand why people do it. I get it. I just … I would never do that in my own projects. It seems like busy work, basically.’

‘Busy work?’

‘Yeah, like making the code very busy, having to think about things that are not essential. I already know what the types are and I don’t think that types actually matter. The other thing is, duck typing is important to me. I got to do these things, with the types, and it just seems like busy work. I just don’t want to do it. If the code works without it, so why do I need to do it? Why do I need to do this thing? I don’t use types for the same reason I don’t use Rubocop in my personal projects. I don’t want to make more work for myself.’

Aaron turned quite serious suddenly. Not in a bad way; I don’t think he could ever be like that. But he seemed to have a strong opinion about it. He actually reminded me of Charles. Not because Charles doesn’t like types – I think he might, actually – but because of the clear and consistent opinion expressed by someone who has been doing this for a long time. And someone who doesn’t want to create more work for himself. I explained to Aaron about Charles, then told him a story.

‘I remember going to ask Charles for help after I broke some code,’ I said. ‘I was trying to refactor something, but broke it and couldn’t fix it. I went to Charles and explained it to him. He looked at me and said, “Was it working before?” I nodded. “Then why did you touch it?” he asked.’

I laughed at the memory.

Aaron smiled, but he didn’t laugh at all. Serious.

I think he was still focused on the type thing.

> ‘My feelings on types are: I understand the usefulness of it. I get why it’s useful and it helps to find bugs. I think it’s a very good thing in enterprise Ruby applications. We’re currently adding it to all of Shopify. I mean, we have like thousands of developers working on the same code base. Being able to have those kind of guard-rails and support in an app this big, where you might not know different stuff, totally makes sense, I get it.’

The main person behind adding types to Shopify is Rafael França, the Brazilian programmer who we met in Chapter 3. His take on this was slightly different. It didn’t have to do with his personal projects. It had to do with what he described as a ‘deeper feeling’: accepting types in Ruby is like giving up on his freedom as a developer. Perhaps Aaron didn’t express it quite as Rafael did, but there is certainly an echo in what he was saying. If Aaron’s take sounded quite practical – ‘If I don’t have to do it, I won’t’ – it was perhaps the expression of a philosophical issue, something that Rafael expressed to me as a very personal matter. During our second Zoom chat about his life as a dev and his work at Shopify, he told me

> ‘I feel this as an internal conflict. I have within me this notion that freedom is paramount and that having this sort of paternalism is not something I want in my language. Ruby has no paternalism whatsoever. It doesn’t force you to do things in a specific way. Because of that freedom, there will be people who will write terrible code and you will get really angry at the pile of crap that these people do. But, at the same time, you are able to do amazing things if you know how to do the right thing. […] Ruby is not a paternalistic language like Python or Java.’

He tilted his head forward, as if considering another point of view, then continued.

> ‘However, working in the same company for six years, I can also see the other side of it; how a lack of paternalism can lead to bad outcomes. So I have this internal conflict today, in which I have to accept certain paternalism even though I hate to have them. I have to impose certain rules on everyone else, rules that I don’t want to follow. I end up following them, but I don’t want to. A clear example of this is types.’

There was a lot here to think about. ‘What do you mean by freedom, though? Is dynamic typing about freedom?’

‘That’s what Ruby is for me: it has always been about freedom, about freedom to express yourself in different forms. Dynamic typing is about giving the programmer the responsibility to handle things by themselves,’ Rafael explained. ‘Contrary to the paternalism inherent in static typing, free languages treat the programmer as an adult. Programmers should be allowed to make mistakes. Ruby is not a language that will tell you “Come here, my child, hold my hand, let’s cross the road and nothing will happen to you”.’

Ruby has come a long way since the days of whimsiness and fun that characterised the code written by people like ‘why the lucky stiff’. Ruby now runs in billion dollars’ worth of production code sustaining thousands of applications. The question, then, is this: is the move towards static typing related to the fact that Ruby now generates a lot of money and needs ‘safer’ production code? Are Rubyists getting scared of being free? I asked Brittany Martin about this. She is a Ruby and Rails dev, and she hosts the Ruby on Rails podcast – one of the most popular podcasts in the community. ‘Is weirdness gone?’ I asked her. ‘What has the Ruby weirdness turn into?’

‘Maturity,’ she replied. ‘We now have these massive companies that are running on Rails. We no longer have GitHub running three versions behind on Rails. Everywhere feels caught up and stable. I think there is still some weirdness in there, and I believe that’s what keeps some community members in, but overall I think weirdness has been supplanted by maturity. As we mature, I think that the weirdness changes. I don’t expect to see something like the ‘Keep Ruby Weird’ conferences any time soon, but I still expect to see some of that weirdness in the main conferences, but less and less.’

It seems that the community is evolving towards the desire to control and restrict its objects. Today, the latest version of the language adds the ability to declare the types of objects, classes and functions by adding additional files. That was a compromise according to Neil, who seemed to have a few insights into how the core team has been dealing with this. ‘Matz sees that typing annotations makes the code a mess,’ he told me. ‘He sees that it is redundant, in many cases. But other committers wanted a typing system. So creating an extra file – like RBS – was a compromise. But switching between a file and the extra file while we are coding, is awful.’ Maybe Ruby 4 will have syntax for types. If that happens, there will be no more ducks in the

Ruby ecosystem. Geese will no longer have the freedom to sneak into a flock of swans.

The conversation with Neil about types during the Brighton Ruby conference was an unusual moment during the research for this book. It took place during the first face-to-face events that I had attended since the beginning of the Covid-19 pandemic. Slowly, things were beginning to change in the UK; more meet-ups and conferences were starting to happen offline. Our encounter was also a prelude to other conversations that I'd been wanting to have for a long time. After a couple of years of learning about the Ruby community in London and chatting remotely with Rubyists around the world, the time had finally come to go to Japan. To speak to developers in the place that gave Ruby to the world.

15
After the rain

I was prepared for a few things before going to Japan. I'd been told that it would be very difficult to meet and connect with people who didn't know me at all. My experience in trying to contact people through Discord, Slack and Twitter before going corroborated those warnings – it was difficult. So I was prepared to go and find myself unable to chat with many developers about the Ruby community in Japan. It would've been a terrible experience, but I was ready for it. However, I was not ready for the humid heat, nor the rain. Nor was I expecting to feel so welcomed by everyone I met.

On my second day in Japan, still adapting to about 9 hours of jetlag, I couldn't sleep at all. After a whole night of trying I gave up, going instead for a 6.00 a.m. walk around Tsu City. Tsu is in Mie Prefecture, just south of Nagoya, and the main Japanese Ruby conference – the RubyKaigi – was being held there that year. I walked towards the river, passing through blocks of small residential buildings and houses. On my way back, I got caught in the rain without an umbrella. I turned a corner and found the gates of a Shinto shrine, through which I walked and took shelter under the canopy of a tree. There was a distance of probably 20 metres or so between the main shrine and the road. A large stone path leads you towards the main building, which was slightly raised (about a metre off the ground). Stone steps on both sides led to a balcony that overlooked the path from the road. On the back stood the entrance to the shrine. It looked closed.

I wasn't the only one caught in the rain. A woman came running through the gates and almost slipped on the worn-out stones. She walked up the steps leading to the main shrine and stopped for a moment to look at two enormous vases. She smiled as she gazed into them, then suddenly turned towards the shrine door. She took her shoes off and placed them facing outwards. She turned, bowed twice, opened the door and slipped inside.

My attention turned to the front gates again, where another person was running to safety and splashing in puddles. He had an umbrella, but the rain was just too heavy. He walked past the main shrine and along a side path that led to somewhere I couldn't really see. I forgot about him and turned my attention to the two large vases. They were about a metre in height and maybe 90 cm wide. Placed at the edge of the balcony, they were not totally protected from the rain. Inside them was a school of bright and tiny blue-silvery fish.

Enthralled by the rain and the shrine, I waited there until the same splashing noise released me from my early morning daydream. It was the man I'd noticed earlier, back from wherever he'd been to. He walked into the shrine's balcony and put his large bag on a stone bench. He sat down for a minute or two, then walked towards the edge to watch the rain. He was in his late fifties, I thought, and suddenly an old memory rushed into my mind: images of the most beautiful film I've ever seen. It featured a couple who take shelter in an inn because of a typhoon. Other people are there too, also taking shelter, and after a few days everyone starts to get hungry. The couple are poor and have run out of money to buy food at the inn. The man decides to go out and find some work, coming back after a few days with a load of cash. He gives it all to the innkeeper and asks for food to be served and music to be played continuously until the rain has stopped. They party, dance and eat until the sun returns.

The man in the shrine was standing not too far from me. Suddenly, he turned towards the door of the shrine, took his shoes off, placed them facing outwards and tried to open the door. It was locked. Very strange. How did that lady get in? The man gave up and turned towards the rain coming down on the main path. It was still going strong. I looked at him and then looked at the bag he'd left on the stone bench. It was a large tote bag, with a little red devil on it. Under the devil, it read 'FREEBSD CONF 2019' in big letters. 'No way,' I thought. 'This guy must be here for RubyKaigi. What a small world.' I immediately walked up to him. I knew that if I didn't ride the adrenaline of that moment, shyness would kick in and I would never have the guts to talk to him.

'Excuse me, did you come for the RubyKaigi?' I asked.

'Yes.'

'Ah, me too.'

'Are you a programmer?' he asked, a bit suspicious.

'Programmer and researcher. I'm Brazilian, but I live in England.' I'm not quite sure why I said that.

'Ah, I live in Tokyo.'

He told me something about FreeBSD and mruby, but I couldn't quite follow it. He kept going, convinced that I knew more Japanese than I actually do. He asked me about work and I managed to say I used to work as a Ruby back-end developer, but that now I'm doing research about the Ruby community. Before he starts talking again, I apologised and asked if we could switch to English. He nodded.

'I have no job now,' he tells me. I'm not sure why he is telling me this, but he continues. 'I only contribute to open source now. I work on FreeBSD and mruby. I started using FreeBSD in 1988. A lot of people used it back then, but now very few. All switched to Linux. I'm a very old man.'

'Oh, you are not old at all. Things change too fast.'

Suddenly he goes towards the stone bench and signals for me to come with him. He opens his bag and grabs a small device and a charger. It's a beige rectangular object, with a few LAN slots.

'Please, take it.'

'For me? Oh no, I couldn't, really.'

'Yes, take it. It has Ruby inside. If you have any trouble using it, just let me know. And if you are ever in Tokyo, send me a message.'

He looks out past me. He looks towards the main path and the rain. He gathers his stuff, says goodbye and leaves. I don't know his name; I don't know what it is exactly he just gave me. I see him walking towards the entrance gate and leaving the shrine. It is still raining and I don't have an umbrella.

RubyKaigi is a very technical conference. Most talks at RubyKaigi are about what people call Ruby 'internals', which to me means a lot of C and Assembly, garbage collection and memory leaks, compiling and transpiling, making Ruby faster and more efficient. There aren't any talks on communication skills, working together as a team or diversity in tech. Talks that one would expect to have at a programming conference these days. But the RubyKaigi wasn't always a very technical conference. It included talks on Ruby on Rails, real-world applications, case studies and talks on soft skills and diversity. It started changing about 10 years ago, when a new team started organising it. 'I wanted to see talks that I couldn't really understand,' explained Matsuda-san, RubyKaigi's chief organiser, when we met in a Tokyo *sukiyaki* restaurant about a week after the RubyKaigi 2022 conference. We met on a Friday at 1.00 p.m. in front of the Kaminarimon gate. This was the main entrance of the Senso-Ji shrine in Asakusa, a neighbourhood in Tokyo.

'Have you had *sukiyaki*?'

'I haven't, actually.'

'OK. We'll go to a restaurant that's been around for 100 years. It survived the 1923 earthquake and the American bombings during the Second World War.'

We walked through one of Asakusa's covered market streets called *shotengai*, which are closed off for cars. Through a sliding door, we stepped inside, took our shoes off and stepped up towards the raised floor of the restaurant. Matsuda-san ordered, and it didn't take long for it to arrive: a couple of plates of raw meat and vegetables, to be cooked by ourselves on an iron frying pan sitting on top of a low table. A network of gas tubes beneath the table and across the entire restaurant floor reached small stoves where the pans sat.

'I started programming in 2001. Ruby was not a thing, even in this country. I once had heard about Ruby in a comparison with Perl, but no one was using Ruby. I was a Java programmer when I first found Ruby myself. It was maybe 2004. I played with Ruby, but there were no Ruby jobs in Japan. I was doing Java and Microsoft something as a daytime job, but I started playing with Ruby as a hobby. Then DHH made Ruby on Rails.'

As I would hear from other Rubyists in Japan, Rails was the main drive for many of them to discover Ruby. It seems that the myth of Rails has a tight grip in Japan as well. Not only Rails has had a big impact in Japan, but also open-source software more generally. In a strange echo of my conversation with Luis Felipe about the free software movement (see Chapter 2), I would hear a lot about the relevance of Seattle in the development of the Ruby community in Japan. Matsuda-san continued

> 'I built my first Rails application in 2006. I think it was Rails 1.0 or Rails 1.1. After that I never went back to Java. I think I'm a pure Rubyist since then. I quit doing anything else as a job – because it was a language made in Japan. Soon I found the community. There was a mailing list, a Ruby developers' mailing list. That mailing list was the community. People could discuss about the language in Japanese. It was a refreshing experience for me, because we could never do that with Java. We could actually reach the developers and discuss, in Japanese.'

It’s easy to underestimate how significant it is to use your native language if your native language is English. Developers – and academics too – are obliged to use English if they want to be relatively successful. English is so pervasive in computing, academia and the business world that we forget that English is also a local language. Someone’s local language. As Oleksandr, the ‘embarrassed dev’ at Upstream, would say, what is it like for native English speakers to work in their native language all the time?

‘Then the community started a conference, RubyKaigi, in 2006,’ Matsuda-san continued. He spoke confidently and in bursts. Between servings of *sukiyaki* and a bit of rice, he would think for a long while before continuing his story. I had so many questions for him, but I tried very hard not to fill in the gaps.

> ‘For the first RubyKaigi, we had DHH as keynote speaker. And for the second one, we had Dave Thomas. His keynote was amazing. He said he came to Japan to say thank you to the Japanese community, thank you for inventing Ruby, that he feels at home when visiting Japan because he is a Rubyist. So Ruby was a small island until Rails prevailed. And then everyone started doing Rails, and the community became so big so drastically. And the atmosphere, the quality, I don’t know, the atmosphere was changing, and Dave said, let's welcome each other, let’s keep this feeling and this atmosphere, let’s be nice to each other. His keynote was something like that.’

I was impressed. ‘Has it worked, to keep it welcoming and nice?’

‘I think so,’ Matsuda-san replied. ‘At the time, we were running a local user group in Tokyo. Rails Benkyokai, literally Rails study meet-up. We had a physical meeting once a month. The main purpose of the group was to study. Because I was a veteran in that group, my main role was to teach the newcomers, so I made a lot of friends in that group. But I gradually started feeling that there wasn’t much else to do in that group. I wanted to do something more. And then I founded Asakusa.rb, and Asakusa.rb’s main purpose was, and still is, to publish software. Our role model was Seattle.rb.’

‘You mean not just study Ruby or talk about Ruby, but actually to release stuff?’

‘Yes,’ he answered, after a mouthful of meat.

‘When did it start?’

‘After RubyKaigi 2008. After the third RubyKaigi.’

‘Were you releasing things about Rails or just Ruby?’

‘We started with Rails. At that time there were a lot of Ruby core developers who lived in this area. Koichi lived here. Nobu, the patch monster, came by train. And I was just a Rails developer. My first goal was to make Ruby on Rails work on Ruby 1.9. It was not released at that time, but Rails was not working on Ruby 1.9. Since we had the VM author in the group, we could ask him about what was wrong.’

‘You could ask Koichi about it?’

‘Yes, although Koichi was not interested in Rails at all. But I told him “You know, Rails is a thing”.’

We both laughed. And I just had to ask if Koichi ever became interested in Rails after that.

‘Hmm, no.’

We laughed again. Rails certainly had its grip in Japan, but, like other Ruby communities, it is not everyone’s bowl of ramen. Or cup of tea.

‘After the first Asakusa meet-up, we made a patch for Rails,’ Matsuda-san told me. ‘That was 2008 and GitHub had just started. I had just learned git, but Ruby on Rails was not accepting pull requests. I made a patch, and I posted the patch on the issue tracker. That was my first open-source patch.’

I was surprised because this sounded like a really complex task.

‘Your first patch was to make Rails work on Ruby 1.9?’

‘Yes.’

‘Did it get merged?’

‘Yes, but we actually had to merge about 100 patches to make Rails work on Ruby 1.9. It took us about a year.’

Ruby meet-ups in Japan are very different from the meet-ups I’d been to in the UK. To start with, there isn’t just one Tokyo Ruby meet-up like there is a London Ruby User Group. There are lots. Someone told me 20 was an approximate number – ‘pre-Corona’, as they say in Japan. Shibuya.rb, Shinjuku.rb, Asakusa.rb, etc. When I went to the Asakusa.rb meet-up in Tokyo, I found it consisted of people getting together and eating, drinking and talking. There were no talks, no fixed schedule. It was the first face-to-face meet-up they had had since the beginning of the Covid pandemic, and I was very lucky to be able to attend it. All through the pandemic, they kept Asakusa.rb going online, meeting every week – unlike many other local Japanese meet-ups, which just stopped altogether. At some point during the meet-up, someone brought a board of Japanese chess, which people played for a couple of hours. There were about 20 of us.

Matsuda-san continued. ‘Asakusa.rb was actually the second “.rb” in Japan, but the first one was just a drinking party. We were the first serious one and now there are hundreds of “.rb” in Japan. We chose the name in respect of Seattle.rb.’

‘And did Asakusa continued making patches and releasing stuff over the years?’

‘Hmm, it changed a bit, but we are still a group of hackers. We copy the Seattle.rb style and we meet up every Ruby Tuesday. We don’t manage the meeting schedule. So, if you are available, just show up. That’s our style. We have 50 meet-ups per year and 500 in 10 years. So far we have had around 700.’

That structure has inspired many other Japanese Ruby meet-ups: no schedule, just show up, every Tuesday. As I write these lines, I’m sitting in a house in the suburbs of Osaka. It’s the offices of 6VOX, a local software company which is hosting Ruby Tuesday, one of the local meet-ups. Again there are no talks. The office is in a small annex of a house, but there would be room for many people. There are three people online and only two of us in the actual office. When I came in, the host introduced himself and said it was a ‘self-running’ meet-up, which means each person working on their own thing. ‘What will you study today?’ he asked me after I had introduced myself and my research. ‘I think I’ll just write, if that’s OK.’ ‘OK,’ he agreed. After about an hour, everyone got together for ‘share time’ and explained what they had done for the past hour. Some people had played around with Docker while others had explored Ruby internals. A couple of people had been trying to make a Scratch game to run in a Micro-bit processor.

It isn’t just Japanese meet-ups which are quite different to meet-ups in the UK. Ruby conferences in Japan are too, or at least have become so in the past few years. Conferences have become a place to praise Ruby committers – the people who develop the language itself and not just those who use it. As one Rubyist in Japan told me, the RubyKaigi has become a place to ‘worship the committers’.

‘RubyKaigi is a very technical conference,’ I told Matsuda-san halfway through our lunch. ‘There aren’t a lot of talks about how to work together as a team or mental health aspects. Some people told me that it didn’t use to be like that.’

‘I became the RubyKaigi chief organiser in 2014 or maybe 2013. I chose to make conferences super-technical by design. Many of them were changing, at the time, to become non-technical conferences. And I missed technical conferences. I had been attending so many conferences. I’ve been to maybe more than 30 conferences in the world. RubyConf, RailsConf, Regional conferences in the US, Asian conferences in Taiwan, Malaysia, India, Singapore. Some in Europe, of course, like Euruko, but also in Ukraine and Russia. Brazil as well.'

'You've been to Brazil?'

'Yes. I'm kind of a Ruby conference *otaku*.'

He laughed. I immediately felt the need to clarify what he meant by *otaku*.

'Lover, lover of Ruby conferences.'

I needed to think about this. Matsuda-san continued to describe his move to change how RubyKaigi works.

> 'I'm a conference otaku, but there was no perfect one, there was no 100 per cent perfect conference for me. What was missing were talks I did not understand. I want to listen to talks I know nothing about. That makes me really excited. I want to feel that experience more. Conferences like RubyConf used to have that part in the past, but they are no longer like that. I knew these people who were working in something difficult regarding Ruby issues. I wanted to put the spotlight on these people. People who are working on open-source software. And conferences in the US are changing. These people are no longer focused, which is … I need to choose the words. For me, I want to preserve the old values of a conference style that I liked.'

I found this a bit confusing. 'Why are RailsConf and RubyConf changing?'

'It's a social change. If they do not change, maybe people start attacking the conference team, forcing the team to change the event. That's America, I think, but Japan is not like that. We can have a different event talking about diversity or hiring, but RubyKaigi is not for that.'

'You said you wanted to keep some values?'

'Yes, good old Ruby values.'

'But what are those values that you want to keep?'

Matsuda-san thought about it, taking his time to reply. "Having a good cycle," he said at last. 'A good cycle between the conference and open-source development. We make a change for the speakers to bring their own code, bring their own new technology, and that drives development. It's called "RubyKaigi-driven development" and if that cycle stops it will slow down open-source development.'

'So people come to RubyKaigi and present, and people at the conference value what they do, and they feel appreciation. Then they keep working on that, and come back to present again …,' I suggested, perhaps explaining it to myself rather than him.

'Yes, yes, yes,' confirmed Matsuda-san. 'We require the speakers to be open-source developers, not just good developers. We ask them to publish their software and talk about their own software, not their company's product. That is our speciality.'

'And about that cycle of RubyKaigi. Did you get the inspiration from somewhere? Was there an inspiration to design it like that?'

'RailsConf was initially like that. It used to have more speakers from the Rails team. When they made bundler, the Ruby package manager, DHH himself, (came). He used to talk about new technologies, like when he made the asset pipeline, etc. But now RailsConf is more of a user's conference. And RubyKaigi, from the first season it was the engine, it was for Ruby language developers.'

I was still trying to clarify this. 'When you say users, you mean people who are using the language, but who are not working in developing or changing the language?'

'Yes.'

'So users on one side, and developers on the other?'

'Yes.'

'Was it a decision at RubyKaigi to not have many talks about Rails?'

'Yes.'

'Did it use to have more talks about Rails?'

'Yes. RubyKaigi didn't used to be so focused. I think I changed RubyKaigi to be more focused on Ruby core. That was my taste.'

The *sukiyaki* pan sizzled with the remains of some noodles and beef. 'Do you mind if I take this bit?' I asked. Matsuda-san just nodded. I picked up the last piece of beef and dipped it in the bowl of raw egg.

'And speaking of Ruby core team, how does one get a commit bit? How do people become committers?'

'You write a patch. That's the first thing you do: patch first.'

Matsuda-san's approach to what a conference should be surprised me. I understood the need to focus on open-source and on the work done by the core team. However, I didn't quite understand why such focus couldn't exist alongside the social changes that have made the Ruby community increasingly aware of the need to discuss diversity in tech, mental health problems and ways of working.

When I spoke to an American programmer who had been to RubyKaigi several times, he said it was 'a proper hacker conference'. He went on to explain.

> ‘A lot of the Western conferences are more people focused, maybe a mix of people focused and technology, whereas RubyKaigi is strictly technology. I really love those people-focused talks, but also I’m a huge computer nerd, so I’m very happy to watch very technical talks.’

I was interested. ‘A couple of years ago, I watched a conference on programming and mental health, on the issue of burnout, during Paris RubyConf.’

‘That’s what I mean by “people-focused”.’

‘I see.’

‘But RubyKaigi is a conference for hackers, for people who code.’

It’s easy to be an Orientalist when it comes to Japan, to think that everything there is naturally ‘different’. But in fact Matsuda-san’s thoughts on RubyKaigi express how connected and attuned to the wider Ruby world Japanese developers are. What he and others told me is very different from what I’d heard from a few non-Japanese Rubyists. For instance, the idea that Rails was not important proved not to be exactly true. Perhaps what is happening is a gradual move away from Rails, with all due respect. A move towards Ruby itself, to a language created and cherished by the Japanese developers’ community. That has also meant a shift away from so-called ‘non-technical’ talks – which makes sense when we look at the history of computing and the emergence of the idea that skill and technique are separate from relationships with people.

One of the most striking bits of my chat with Matsuda-san was to see how important using his own language was to him. When he found Ruby, he abandoned Java. When he found the mailing list, from which he could speak to other hackers about programming, *in Japanese*, he found the community. As he says, the mailing list was the community: a community of words, filled with Japanese language characters. If you don’t speak Japanese, you might feel disconnected from that community, but the reverse is true in a much bigger way. If you don’t speak English, as a programmer, you’ll miss out on a lot. You’ll never be invited to speak at conferences, no matter how good your coding is. Isn’t it OK, then, to have communal spaces in which English is not the only language spoken? Given that a lot of conferences and meet-ups in the Ruby world are run in English, doesn’t that seem fair?

I left our conversation with those questions in my mind. My perception of what Ruby might mean in the Japanese context was slowly shifting, and there were many other shifts to occur. The more I spoke to developers in Japan, the more it changed my assumptions of what a programming community might be. The conversation with Matsuda-san was only the start.

16
Patch first

In a community, opinions and misapprehensions run like wildfire. From the very beginning of my research, it became clear that a lot of developers in the Western world had a few beefs and tensions with the Ruby core team in Japan. They often described it as ‘the Ruby community in Japan’, but I’m pretty sure they meant the core team, the people who work on developing the language. There were a lot of things that only started to click when I got to Japan. The expectations that non-Japanese developers had about the governance of Ruby and its community started to make less sense to me.

One of the expectations that fell apart was a linguistic one. I realised that while a lot of people simply accepted English as the dominant computing language, Japanese developers were clearly indicating that although many things could be done in English, perhaps not all of it should be. The second expectation that slowly crumbled was the notion that open-source development should be the same ‘everywhere’ in the world. More and more, ‘everywhere’ became equal to North America. And if people in Japan had been clearly and manifestly inspired by the free and open-source movement in America – they followed the example of Seattle, as Matsuda-san told me – they also did things differently. Not everything was wide open to the public, for example, and the language core team remained a bit of a secretive group.

Trying to understand these miscommunications and tensions was crucial to me. It was a way of combining the social and the technical aspects of programming; a way of making them part of the same network of socio-technical relationships.

‘What’s the name of the conference that they have in Japan?’ a prominent American developer asked me.

We met over Zoom in early 2020 to chat about his story as a programmer, as well as to consider the relationship between the Japanese Ruby community and the American Ruby community. A few Ruby devs had pointed out to me the tension that existed between these two communities. One, in Japan, largely focused on Ruby; the other largely focused on Rails.

'Ruby Kaigi?' I replied to his question about the conference.

'Thank you. They absolutely refuse to have any sort of Rails talks. Like that's forbidden. I think it's the purist mindset. I think that it's pride too. They're the ones who are in charge of Ruby and so they want to keep it in its purest form. We don't really hear from them about how they're *using* Ruby. We just hear from them about how they're improving Ruby. On the flip side, in the States, we only hear about how they're improving Rails based on using it in production. It feels like with Ruby it's very closed.'

This was not an unusual complaint. But the American had not finished.

'It's weird too that all of the Rails development is on GitHub – it's a very public group. On the other hand, Ruby has its own forum: I can't even recall off the top of my head what it's called. It just doesn't feel as accessible. In some ways Rails has to pay a lot of kudos to Ruby because Rails wouldn't exist without Ruby – but then in some ways Ruby doesn't pay any kudos to Rails because they think the language would be extremely popular if it weren't for Rails.'

I can't imagine many Western developers reaching out to random Ruby developers in Japan; they would have very little reason to do so. On the other hand, Ruby developers are quite involved in the open-source community, so it is only natural to try and interact with Ruby committers – i.e. the people on the core team. In any other open-source project, you would normally go to GitHub to contribute and reach out to code maintainers. However, the Ruby language has never really been on GitHub. During our lunch in Asakusa, Matsuda-san told me that 'actually, there was no issue tracker until 2006 or 2007, so we started using Redmine'. This is the name of a free and open-source platform for project management, created with Ruby on Rails in 2006. Redmine hosts different language servers, one of which is the Ruby issue tracker.

'Ruby 1.9 release manager, Yugui-san, she was the first female Ruby committer,' Matsuda-san continued. 'She proposed to use some system for tracking issues, and then she started the Ruby server on Redmine. Before that there were only patches on the mailing-list. If Matz liked it, he merged it. If the patch was incomplete, it was just abandoned.'

‘Was there ever a discussion to move it to GitHub?’ I queried.

‘Yes, but some people didn’t like the commercial side of GitHub. Even if they claim GitHub is not evil, some people think it’s a problem that it belongs to a company.’

‘It’s a bit weird, right?’ I said. ‘GitHub hosts the majority of open-source projects in the world, but GitHub itself is not open source. What if GitHub decides to shut its services down?’

‘Exactly. Or what if the company is acquired by Microsoft? What would happen then?’ He laughed and said, ‘I’m just glad that’s never happened!’

We laughed. Microsoft acquired GitHub in 2021.

I had heard that story before, that time from Aaron Patterson when we first chatted about his life as a developer in 2021. On that occasion I also wanted to get his point of view on the relationship between non-Japanese developers and the core team.

‘The reason I started studying Japanese,’ Aaron told me, ‘was because at the time, in 2006, there wasn’t much English documentation for Ruby online – but I still wanted to read the Japanese docs. I could read the code, but I couldn’t read the documentation, couldn’t read the blog posts, so I started studying.’

‘And is it still the case that a lot of discussions are all in Japanese?’ I asked him.

‘That used to be the case. It’s not true these days though. Every language feature and conversation is done in English. All decisions are made in English. Sometimes people will propose in Japanese and then they’ll essentially translate it into English. The language barrier is still a problem, though, because I’d say 80 per cent of the people on the Ruby core team are based in Japan. And, I don’t know man, they just don’t use English.’

‘How do you see that?’

‘For me it doesn’t matter, it’s fine. I speak Japanese.’ Aaron laughed, then continued a bit more seriously. ‘I think there’s a few different things here. One, there’s a language barrier which I think can be overcome; it’s not as big a deal as people make it out to be because everybody speaks a little bit. Everybody in Japan must learn English at school. The other issue, though, is that we don’t use GitHub. I mean, we do, kind of. We take pull requests on there, but any feature discussion happens on our own thing, Redmine.’

‘Those two things together, though, can be an issue.’

‘Oh, totally, yeah. The combination is the problem. I suspect that Westerners are afraid that they won’t be heard, because they’ve all heard of this language barrier. But I think it's more an issue of perception. The other thing is that you got to register for this weird Redmine website when everybody already has a GitHub account. And people are like "I don't want to do this, I don't want to do this weird thing". I think it's a combination of those things that makes a higher barrier to entry for folks than exists with normal open-source projects.'

'But is there a reason why Ruby is not fully on GitHub?'

'Yeah, there is. The reason is that GitHub is not an open-source project. That is the reason. We have some members of the Ruby core team who are very enthusiastic about only using open-source software. So they basically said, "If it's on GitHub we won't participate. We don't want to be on the core team". And Matz doesn't want to lose anybody from the core team. He thinks it's acceptable to stay on Redmine as long as we keep contributors.'

There was something else I wanted to ask him, but I didn't quite know how to say it. So I did what any experienced researcher would do in this situation and started babbling and talking nonsense. Aaron looked a bit worried. Finally, I managed to put my thoughts into words.

'About the language barrier and the difficulty in communicating with the core team. Quite a few people have told me these stories of very difficult and slow exchanges about an issue or a feature. And these people have, let's say, quite strong opinions about this relationship, how it is very hard to get anything across. They feel like the core team is ignoring the community, that they don't care about the rest of the community.'

Aaron's response was thoughtful. 'I don't think that's fair,' he said. 'Probably there is less discussion than people are used to. To flip this around: if there's an issue that's in Japanese, I can read it, but I am definitely less likely to add any comments because it's not my native language. I always have this thing in my head, "Do I really want to spend time and effort in writing this comment? Does my comment actually matter that much?" So I put myself off and I'm less likely to do it. I can absolutely see that, on the flip side, if you know just a bit of English, you might think the same. So it could be that this lack of comments is perceived as not caring.'

'But there is another thing, Aaron. I feel that there is this expectation that everything needs to be in English – as if no other language in the world existed.'

'Some people do say that.'

'Say what?'

'Say that everyone in programming should learn English. I really don't understand why people say that. Yeah, English is so pervasive or whatever, like it's so ubiquitous that people say "Oh, you should just do it, you should just learn English". But that's like … You can't tell people what language to speak. That's not … it's not a thing.'

No, you can't tell people what language to speak. And yet it's not easy to create non-English spaces. At Upstream there were several developers who worked out of the Ukraine. Once they had a meeting among themselves and didn't invite any of the non-Ukrainian developers. Later they shared what they discussed in the meeting with everyone else, explaining that the only reason they had a separate meeting was because they wanted to discuss something very technical and complex, which they would have a hard time doing in English. 'But they work for an English company!' a British developer grumbled to me after that meeting. 'How can they think that it's OK to have a separate meeting and not let anyone know? I'm sorry if you are offended by this, Gui, but I just don't think it's acceptable to have a meeting in anything other than English if you are working for an English company.' But what's the big problem about having a meeting in Ukrainian if some people feel that it would help to solve a technical problem the company is facing?

A few months after my chat with Aaron, I met a developer called Stacey in central London for a chat. We'd had a few conversations online already, but this was the first time that we were meeting up face to face. It was a good feeling; things were starting to open up and to get back to the new/old (ab)normal. Stacey had been contributing to a part of Ruby that was managed by a Ruby core team member. She was furious about the difficulty in communicating and in getting things across.

'It's very hard to work with the maintainer. He doesn't reply to most emails, doesn't share any control of the project. I'm the second highest committer and I don't even have committer status to the project.'

'How long have you been working on this?'

'About a year, I think. I need to send every pull request as a general contributor. I have no privileges, even though I've been working on this for a long time. Then I need to beg for him to review it. It always takes a few weeks.'

'I can see you are very frustrated by this.'

'Oh, hell, yes. I see this as a big problem of governance. The Ruby core team is very closed; most developers on the team have full-time jobs and only work on Ruby when they can. Matz is going to retire in two years and there is no process in place for this. What are they going to do then? Who is going to be in charge?'

Stacey kept using an expression, ‘there is no procedure on the procedure’, to explain why it is very hard for non-Japanese developers to become Ruby committers. For her it was clearly a big issue.

> ‘It’s not clear how people can become contributors and it’s not clear how this can become clear. There is no procedure on the procedure to change how the core team includes new people. This is going to kill the language one day. Eventually, people outside of Japan will get fed up with this and start moving towards a different language. The pace of change in the language is just too slow, too slow. The core team needs to start including more things that people say, more suggestions on how to improve the language, but there just isn’t enough preoccupation on the core team on how this can be done.’

The sands of time: how quickly should they move?

Stacey expressed expectations that developers have in the open-source community: to be able to contribute to change and to do it in a public forum; to see a relatively transparent process as to how things can be modified. ‘The procedure of the procedure’, as she said. Yet there are other things that perhaps set the Ruby Core Team slightly at odds with some normal expectations in the open-source community. One of them is the ‘once a committer, always a committer’ principle. ‘Now we have about 100 Ruby committers all over the world,’ explains one Ruby committer, ‘but the number of active members is much smaller. If you become a Ruby committer, you can’t throw away the title of “Ruby committer”.’[1]

Compare this to the way in which being a member of the Rails Core Team works. There are always 20 people in the team: no more, no less. A list contains the names of all current and past members, as well as describing a way of contributing. Everything is on GitHub, as expected. On the other hand, in the Ruby core team, there are more than 100 people on the team – and yet there is no list. No one really knows who they are. Being a committer and being a member of the core team is a different thing, but the difference is not explicit and it’s not procedural. There is no formal distinction.

‘Other languages, like Go or Python, focus on only one way of doing things,’ I told Stacey. ‘I mean, it’s a key cultural aspect of the Python community, right?’

‘Yes. There is one preferred way of doing things.’

'And that's different to what people in the Ruby community emphasise, right? People always tell me that in Ruby you are free to do it your own way.'

'It is different, of course, but I think that after so many years the community has found one way of doing certain things. And the core team should take that on board! They should say "This is the way to do it and we should prefer this way", or something like that.'

When we talked, Stacey mentioned that she finds it very frustrating that the Ruby core team doesn't want to create tools to make Ruby developers do things in one way. Very few tools like that are incorporated in the standard library, and there isn't a 'correct' way of doing things. She feels that if we had that, Ruby developers could focus on other things instead of having to create yet another way of doing the same thing. Although Stacey was talking about 'technical' things, it seemed to me that what she and others really want is for the core team to create a space in which members of the community can steer the course of the language a lot more. The path to become a committer, for instance, is not clear at all. 'There is no written rule,' one of the oldest Ruby core team members told me. And that infuriates some developers, who expect procedural transparency and a roadmap to lead them there.

'The core team's lack of will to sponsor or to choose to make certain things official is damaging to the community,' Stacey continued, 'because people who worked on those projects don't feel recognised – and also because it creates strange situations. For example, we now have two tools dealing with type checking: RBS and Sorbet. Instead of having only one tool, we now have two. That's very frustrating because now Rubyists don't know which one to use.'

When Stacey mentioned the problem of having too many tools to choose from when using types in Ruby, I kept thinking that this was probably what Matz intended. A way of being able to say 'Yes, we've dealt with the issue of static typing, like you asked', but also of not endorsing one particular solution and keeping the Ruby ecosystem multiple. In this context, it makes sense that many procedural things remain murky; it's a way of avoiding the prescriptive culture of 'this is the right thing'.

What seems to be a lack of attention, or a lack of care is, to me, created by design. Some Ruby core team members do feel that the community grew too much too quickly when Rails boomed; they are quite happy to see it shrink back to a more manageable size. Unlike many devs in the US, they don't really worry about the Ruby community dying. The multiple meet-ups attest to that. The fact that many programmers in Japan keep going to meet-ups and to RubyKaigi even after they've stopped writing Ruby also attests to that. They may have left the language, but the community hasn't left them. All of this became clearer when I finally had a chat with Matz.

Note

[1] Sasada. 'Introduction of MRI development culture'.

17
Supreme beings

There are no bullet trains that come near Matsue, the capital of Shimane Prefecture in the island of Honshu. ‘When people from Tokyo come here, they feel they are in another country,’ Hasumi-san, the organiser of the local Ruby meet-up, told me. ‘They think they've gone abroad,’ he concluded. Around 200,000 people live in Matsue. There are a couple of airports nearby, one of which sits very close to Izumo-Taisha, the most important Shinto shrine in Japan. It is set in the mountains that surround the town of Izumo. Inside a special place within the temple itself resides Okuninushi, a mythical god who helps people with their relationships. In other Shinto shrines you bow twice and clap twice to pray for the gods, but at Izumo-Taisha you must do this four times each. Two for you, two for your partner.

The connection between Okuninushi and personal relationships explains why the grounds of Izumo-Taisha are populated with incredibly sweet statues of rabbits. As the story goes, Okuninushi helped a rabbit to heal his wounds after a shark bite; the rabbit, in turn, predicted that Okuninushi would be the one to marry a princess. Okuninushi does indeed marry her, establishing the connection between the couple, rabbits and marriages for eternity. At Izumo-Taisha the connection is also set in stone, as the dozens of pairs of rabbit statues give the solemn shrine an air of cuteness. Sacredness and cuteness: a unique combination, perhaps, but one that would echo my conversation with Matsumoto-San, the creator of the Ruby language.

Matsumoto-San and I met at the hotel where I was staying and walked to a nearby cafe. The cafe was full in the early afternoon. The coffee machine didn’t stop for a second and we waited for our coffees to arrive. I didn’t want this chat to be any different than other conversations

I'd had with other Rubyists, so I asked him about his story and how programming had started for him. First, however, I needed to clarify something about his name. Given that Matsumoto is a very common family name in Japan, I wanted to check whether I should be calling him Yukihiro. Like many other things in the history of the Ruby community, his name was also constructed by design.

'Matsumoto-San, how did people start calling you Matz?'

'Oh, I invented that. People outside of Japan don't normally use people's last names like we do. They usually call each other by their first name. And that would've been a problem.'

'Why?'

'Only my mother calls me Yukihiro.'

Matz's parents converted to Christianity before he was born. Their family is Mormon. In a country where most people don't consider themselves religious, being not only a Christian but also a Mormon definitely singles you out. 'I'm used to being a minority,' he told me. Matz started programming when he was 15. He was in the third grade of junior high school. His father bought a computer from a company called SHARP. It wasn't very big, as he shows me with his hands: 'about this big, 20 cm wide'. It had BASIC on it, one of the languages that wanted to make programming closer to human languages. Matz started programming in it, following the examples of a book that came with the computer. As he told me with a slight grin,

> 'Father bought the computer for himself, but I took the computer away from him, and I played a lot. At the start, I felt both positive and negative. From the positive side, I was very interested in the computer because other kids' toys are controllable, you know what I mean? I used to play hoops, so I throw the hoops by myself and it spun, but the computer moves by itself. It kind of makes a decision based on what I taught him. I felt these computers are adorable, like pets. That was my primary motivation as a programmer: those computers are adorable.'

I wasn't sure that most people would describe computers as adorable. Much like the sacred rabbits at Izumo-Taisha, it seems that it is also possible to find cuteness in technology.

'At the same time, this computer was pretty limited,' Matz continued. 'It has only 400 steps, 400 lines of code. Every variable has length of one: $a, $b. All global. No local variables, no local functions. I got kind of frustrated, but I never knew other languages. In a bookshop

I found Pascal. I read Pascal in the book, but I had no computer to run Pascal. Back then, the compilers were so expensive. Probably 2,000 US dollars, or even more expensive. You know, it was 40 years ago.'

Matz went on to learn Lisp, C and SmallTalk. He was fascinated by the world of programming languages. Languages which were 'designed by humans, you know? Not like Japanese, or English, or Portuguese. We don't know who invented those languages, but BASIC, for example, was designed by the professors at Dartmouth College. Pascal was designed by Niklaus Wirth from Switzerland. Those languages were designed by a specific person. If those languages are designed by people, with intention, why not create my own programming language?'

'It was my high school dream,' Matz revealed. 'But back then it was the 80s, the early 80s. We didn't have any internet; we didn't have any good materials. The only materials were for programming language design and programming implementation, textbooks for university lectures. Back then it was difficult for me, as a high school student from a very small town. So I took my notebook and took down some ideas for my programming language.'

'You wrote it all down in a notebook?'

'Yes.'

'Do you still have that notebook?'

'Actually, a few years ago, I went to my parents' home and looked in old boxes, but I couldn't find it.'

I bet that book would've been adorable.

'Then I went to university, majored in computer science, learned a lot about computers and the programming. After I graduated from university, I got hired as a computer programmer and spent a few years there, learned a lot, and the skills. Finally I decided to create my own programming language. For 10 years I had a dream of creating my own programming language, and that small prototype gradually became Ruby.'

'Were you already here in Matsue?'

'No, it was Hamamatsu City, Kanazawa prefecture, halfway between Tokyo and Nagoya.'

'When was this?'

'It was 1993. A bit earlier that year, in 1991, 92, we had a very big depression in Japanese economy and my project was cancelled. The members of my team were scattered in many places, assigned to new tasks. Only two were left behind to maintain the existing software. I was one of them. But actually I had very few jobs assigned. Sometimes the users of the software called me to say, "OK, I have an issue with your application". I would reply, "OK, reboot your PC". That was all I did.'

Turn it OFF and ON again. Even Matz had his days of IT crowd.

'I had time,' Matz explained. 'I had loads of time because my manager was assigned a task that was very important. He looked in on us less often. I had my high school dream of creating my programming language and it was probably time to start something. So, you know, the bubble economy created Ruby.'

A significant amount of open-source projects get done because managers are not constantly checking in on their employees. There are several techniques one can develop, over time, to keep managers away and, slowly but surely, to integrate open-source development as part of your daily tasks. Technique number one might be: never ask for more work. Let them think you're still working on whatever it is you are working on. Technique number two: be generous with the amount of time you think a certain task might take. Two weeks is the least anything should take. The bubble economy created Ruby, but also an absent, laid-back manager.

'Whenever I'm programming Ruby,' I said to Matz, 'one of things that I like the most are blocks. They feel kind of like home. Does that make sense to you? Where did the idea to use blocks came from?'

Matz's answer was quite involved; I had to concentrate on his reply.

> 'Blocks came from the Lisp high-order functions. I like LISP very much. In my university time LISP has some functions, like the map functions, that apply the function to the list by which that function is called, with each element of the list. This was my first inspiration. The second inspiration was the language called CLU, from MIT, which was invented in the 70s. It has the feature of iterators. In CLU you can call some special function and iterate from the full statement. In iterator, you can call the statement named "yield", which is inherited from CLU to Ruby. That element is assigned to the loop variable, then the block is executed for more than one time, or something like that. That pattern is sort of like Ruby. I combined those two, the high-order function in LISP and the iterator in CLU, and came up with this idea of the block. The block is rather like a high-order function, but it works like the CLU iterator. However, the block is not limited to the loop function. The blocks can be used in anything, such as specifying callback or the scope of a DSL. Relaxing the limitations of the iterator opened a broader application to that kind of block abstraction.'

A Ruby block is a pattern. It's what you used to sort your card deck.

```
card_deck.each do |card|
  hearts.push(card) if card.suit == 'Hearts'
  clubs.push(card) if card.suit == 'Clubs'
end
```

The Ruby block is everywhere: it's the quintessential structure of the language. It's even hidden inside Ruby's obfuscated 'symbol to proc' function, which translates the block structure into the ancient and enigmatic ampersand (&). In their minds Rubyists know that whenever they need it, the block will be there. As a pattern, the block is very appealing. It helps you to focus your attention; it also gives you a sense of tranquillity. Whatever you do, you know that between those two words, from the do to the end, nothing else matters. The block is the programmable bridge that translates your thoughts into code.

Matz has highlighted, time and again, how blocks became the master tool in every Rubyist tool set. It is the tool that rules them all; the tool that allows you to do, essentially, whatever you want. It created within the community this sense of being able to adapt and to change Ruby, to make it bespoke to their needs. In 2019 Matz and two other Ruby core team members went to a Japanese company's office in Bristol. A hackathon followed, in which developers tried to patch Ruby, and people submitted a PR at the end of the day. TenderLove, of course, submitted a joke PR. In a blog post about that event, Noah Gibbs writes that 'Matz feels that blocks are the greatest invention of Ruby (I agree.)'.

Our coffees finally arrived, but Matz's phone rang suddenly. He answered, quickly hung up after a minute or two and apologised.

'What was I saying?' he asked.

'I think you were about to tell a story,' I replied.

'It's a kind of silly story. I was Ruby's release manager until 10 or 15 years ago. But I'm not really good at releasing or managing things. Soon after I made a release, we found out that I had forgotten some important files. Files that needed to be in that package, or something like that.' Matz constantly adds 'or something like that' at the end of his sentences. It makes him sound as though he does't know what he is talking about. Which in this case might be true, given that other developers admit that he was a terrible release manager.

'A few days later we have to release an even newer version or something like that,' Matz went on. 'But meanwhile I had been fired by the other core members as a release manager because I was doing a bad job. And then a few years later, I wanted to commit something to Ruby, but I forgot to check if the committed code had compiled well and it had some errors. So some core team members complained to me and said "OK, if you do these things, I will remove the commit bit from you". That's a kind of silly story, but, you know, we make decisions that are committed as a core team or even a creator – but it is a decision. We have reasons before each decision and the core team values each decision and each reason. So I can't just say "OK, I am the creator, this is my language, I will do whatever I want". Even I can be removed from the community.'

'Yes but, on the other hand, you did create the language. So what happens when someone changes something in a way that you don't like?'

'That could happen. But first they must persuade me. In the past, they have succeeded several times. But it's OK because I'm a person, and I don't feel sad.'

'But what about the recent changes about types? I know you don't like types, you've said it publicly before. You didn't create Ruby to be statically typed. How does that make you feel?'

'Oh difficult, you know.' Matz embarked on what would be a long explanation, with three separate points, on how he's not very happy about the whole thing. Types were not his type. 'I love seeing creative things for Ruby, but at the same time I must look at programming for the long time future. I'm a long time user of static typing.'

I was startled. 'What do you mean?'

'Well, I've been writing C my whole life. I understand the benefits of static typing. For me, it's OK to talk about static typing in Ruby. I understand the desire to get the benefit from static typing for Ruby. But at the same time, adding type declarations like other languages, PHP and Python, could change the feel of the language, you know, the feeling of programming, so I refuse to add type declarations'.

'The first reason is that it would change the feeling of programming in Ruby. The second reason is more about the community. In other programming languages, we see the "typing police". We would start to see the typing police in the community: "OK, your gem does not have type declarations. You have to have type declarations as a gem." And that's kind of forceful, you know. It's kind of bad for the community, I think'.

'The third reason is that, in the history of programming, programming evolves in decades. Maybe 10, 15 years ago, dynamic programming, such as PHP, Python, Ruby and Perl, was very popular. And then 20 years before that, the most popular programming was C, C++ and Pascal, you know, static type programming. Then even more years ago, SmallTalk was popular and had no static typing.'

Matz took a sip of coffee and went on. 'The dynamic and static, it goes like this, like a pendulum. Thinking about the future, maybe in 10 or 20 years, we might have the languages without type declaration. The compilers are very smart and they guess the intention of the programmers and they have code completion; maybe they even have error detection without type declaration. If this kind of future comes, we would not be able to go into that camp because we already have type declarations,' he explained. 'We want to keep Ruby for that distant future, not for the, you know, the present time benefit. It's like a longer-term view based on what's happened in the past. And how it kind of fluctuates. Maybe 20 years later we will have better compilers without type declaration because the pendulum goes the opposite way. And I want to keep Ruby for that distant future.'

What Matz was saying was exactly what Neil had told me in Brighton a few months before. Matz is, basically, a C programmer and he knows about types. For the sake of future-proofing Ruby, he is not very happy about the request to add types. On top of that, however, there was also something else, perhaps something more important, which he described as 'the feel of Ruby'.

'But what is the feel of the language that you want to keep for the future?' I asked.

'Being concise. Ruby programs can be concise and as small as possible. That makes you productive; it makes you feel like a strong programmer. That is the feel, I think. For example, in 2004, creating web applications with a website was a major task. It took days or even weeks, but DHH showed, you know, it's a matter of 15 minutes. That feeling is Ruby, I think.'

The myth of the blog all over again. Rails had had a major impact in Japan as well. DHH's talk was an event that changed the Ruby community forever. When Matz described his relationship with his first computer, he compared it to his other toys. He had some control over the computer, as he did with other toys, but not full control. Something similar happened with the community. He had some control over it, but not full control.

> 'Soon after I released Ruby, 200 members joined the mailing list and some people start talking about Ruby, even outside of Japan. People were asking me some questions about documentation in English because, at the early stages, most of the documentation was in Japanese so it kind of got out of control. And then there's

> the book writing and organising conferences and that's kind of out of my control too. And then people are forming the community, it's getting bigger and bigger, and then some years later Ruby on Rails came along. Then, you know, the size of the communities kind of exploded. That's far beyond my expectations.'

'I wanted to ask you something a little bit more personal, Matz, if I may. It's about religion.'

'It's OK, we can talk about that. My parents converted to Christianity some 50 years ago. I was born into a Christian family. Almost all my life I've been a Christian, a minority,' he said.

'I can imagine that might not have been easy.'

'I was probably the only Christian in my school. Most people in Japan visit temples or shrines for sightseeing. They want to experience a religious atmosphere, but they probably don't have the belief in their heart. If you ask most people, they don't believe in religion, they don't believe in God. Most Japanese people don't deny the existence of the Supreme Being, but they don't strongly believe in this either. That's a big difference. Being a believer changes a lot.'

Technology and religion are not topics that usually go together, and I was surprised to hear a major computer scientist talk about his beliefs in a Supreme Being. Matz was frank and honest. It was clear to me, at that moment, that religion might be one of the reasons why Rubyists describe Matz as 'nice'. I wondered how much of his religious background influenced his view of the community.

'My religion has its own community,' he told me. 'We have congregations and church members, all with very different backgrounds. Like the Ruby community, we sometimes see conflicts and misunderstandings. Both have complex relationships, even though people try to be good. That's a similarity between them, and I've learned a lot from the religious community on how to handle that.'

'I spend most of my time as a programmer,' Matz went on. 'I talk to technology people, I work with computers, and with the people behind the computer. In the Ruby community, I'm the kind of leader. I have a position that I can say whatever I want to. I'm in a strong position in the community. But in the religious community, they don't care about that. They don't care about me being the leader of the other community. No. I'm just a normal member. And that humbles me. Sometimes a member of the Ruby community admires me, and I understand that I'm a very important person in the Ruby community. However, I'm also a mere human. Being a mere human is a kind of an important balance to me.

That kind of balance is pretty important, to act as a good member of the community, so that I can avoid being a dictator.'

Matz explained to me that his congregation had influenced him to try and be a good person, a good leader, a good member of the community. And that behaviour results from the difference in how he is positioned within each community. He is the leader of one; he is nothing but a member of the other. So he is not only a benevolent dictator, but a humble and religious one. A tamed dictator who has been put in place by the Supreme Being. A believer. And a believer in technology as well. One who finds cuteness in a computer. One who is OK with Ruby being a small community.

'Japan is not a country where most people are religious,' Matz acknowledged. 'The people are inspired by Buddhism a little bit, maybe they have a membership in the Buddhist temple, but they're not very enthusiastic about religion. And for a Christian, especially the Christian Mormon – we prefer to call it Vida, but anyway – we are a minority. In Japan we are very, very, very much a minority.'

'So, you really understand what it feels like to be a minority?'

'I've been a minority for all my life. And I've learned not to be afraid of being a minority. In the beginning, probably no one knows about Ruby. No one knew about Ruby until Rails. Very few people used Ruby for their jobs. But then there was the Rails boom in 2012 or something. And the community grew a lot. And now people come to say that Ruby community is decreasing. That Ruby is dead. But we are not dead. Ruby is just going back to being a minority again. For me, that is quite OK. The minority is kind of, you know, part of myself.'

18
The end

'I am tired of hearing that Ruby is dead,' someone posted on Reddit in 2020.[1] The post mentioned an article that pinpointed the beginning of Ruby's decline to between 2015 and 2016. If you were around the community at that time, you know that that was when Node came round. Node, the JavaScript back-end, was the straw that almost broke Ruby's back. It was the moment when a lot of people left the community. A Ruby developer told me that

> When JavaScript developers really started trusting Node, that's when we started losing people. Suddenly people didn't need to know JavaScript and a back-end language like Ruby: they could just use JavaScript for both.

By then JavaScript had already dominated front-end technologies; now it was conquering back-end ones too. Suddenly, you didn't need to know two languages to work on a web application. It was a stark reminder that code doesn't last forever.

I was on holiday in August 2020 when my phone started to beep.

'Hi everyone,' started a message from Amina, an engineer at Upstream. 'This is a group to talk about the redundancies at Upstream.'

'Oh, God,' I thought.

'As most of you know,' the message continued, 'a few of us have been let go. You might not have received any communication yet. Redundancies, again!!!'

I panicked and started checking my email, but there was nothing there. Maybe they were waiting for me to come back? Surely that would be a bit cruel? God knows what people might think. I was only a junior developer, so definitely in the firing line. Trying to move away from the panic, I reached out to Charles via WhatsApp to see what was going on.

'Hey Charles, I just heard about redundancies.'

'Hey Gui. Yeah, it's crap, but I'll be all right.'

'Wait, you've been let go?'

'Yes.'

'Shit, sorry to hear that.'

'Oh, it's fine. I'll be fine. I'm worried about other people.'

'Like?'

'I'm worried about Shazia really. She's only been with us for six months. And it was her first job. It will be hard for her to get another job so soon.'

'You think?'

'I think so. The second job is harder than the first one. Especially if you need to explain why you were made redundant so soon.'

'I hope she'll be OK. Are you OK?'

'I wasn't expecting it, but I'm going to be OK. I'll find something else.'

Charles had worked at Upstream for four years. He was the principal engineer. He knew everything about the codebase, he knew everyone in the company. I honestly thought he would be devastated.

'Charles, do you know if I have been made redundant?' I texted him.

'Have you been contacted by anyone?'

'No.'

'I wouldn't worry about it, then.'

'But what if they are waiting for me to come back from holidays?'

We stopped texting for a while. I went to sleep without any certainty. Not a great night. The next morning I found Ivan on Slack and asked him to call me when he had a chance. He was the CTO – he had to know something.

'Hey Gui,' Ivan sounded a bit too cheerful on the phone.

'Hi … how are you?'

'Good. You?'

Well …'

'So, you've heard, hm?'

'I have.'

'Who from?'

What a weird question. I mumbled something but didn't say anything about the WhatsApp group.

'Ivan, have I been made redundant as well?'

'No. Don't worry, you're good for now.'

'For now?!?!?' I thought – but again didn't say anything.

'The fact is,' Ivan continued, 'you are, how can I put it, you are very cheap.'

What was that supposed to mean?

'You are quite efficient, so you are very cheap for what you do.'

Weird, again. Very cheap for what I do? Is that supposed to be a compliment? Should I be asking for a pay rise?

'Yeah, that's it. You're still here. Have a good rest of your holidays and I'll see you when you get back. Bye!'

After the call, I followed the chat on the WhatsApp redundancy group. It turned out that some 20 per cent of the company had been let go. Proportionally, the tech team was the most affected. The higher salaries of the team, compared to the rest of the company, made it the most logical place to cut. The redundancies also included some of the outsourced devs from Ukraine. The general feeling in the WhatsApp group was one of doubt about the company's future. People wondered if periodic redundancies such as this would become a feature, not a bug. After all, this was the third round of redundancies in three years, always in August, after the (now) traditional slump in sales during the summer.

Upstream's mission was to tackle climate change in the food retail sector. The majority of the developers in the company not only believed in this mission, but they had also joined the company because of it. People like Charles, Akira and Shazia. People like me.

'I mean, so much for trying to change the system, right?' Marie told me on a call a few weeks after the redundancies. She was also 'still there', but was getting more and more pissed off with the way things were going. 'It's a bad omen,' she continued, 'when you start making people redundant regularly. There's clearly something going wrong.'

'It sounds like we're veering towards the end,' I added.

'All of that code, just dead!' Marie cried.

A cry that echoed the Ruby community's doubts about the longevity of their language. I never thought about code that way, about the possibilities of code no longer being used, of code eventually dying. There are old and recent articles on Quora, posts on dev.to and even an article on Forbes wondering if Ruby is dead or dying. In 2010 one of these questions got a reply saying that 'Ruby on Rails was a Hype. That means a lot of people jumped on the bandwagon because that is what they do: jumping on bandwagons (for a living).'[2] This response went on to explain the influence of Rails and Ruby in other frameworks and languages, before concluding on a happier note: ‘Rails is not just hype. It is a fantastic framework. With a still very active community around it’.

Reflecting on this question years later, the author of the answer writes that large software companies that chose Ruby are all from the 2000s.[3] Companies like Upstream. A historian of computing wrote that ‘people seem to have an inordinate amount of interest in whether or not Ruby (and Ruby on Rails) are dying’. Certainly ten years is a long time to be dying. Ruby (and Rails) peaked around 2012 and has been in slow but steady decline ever since.

What does it really mean to say that a programming language is ‘dying’? FORTRAN, created in 1958, is definitely not a popular language any more. You wouldn’t recommend it to a programmer today. For a newbie, FORTRAN died a long time ago. Still, there are jobs in the aviation and nuclear industries that still hire FORTRAN developers. As John, a FORTRAN programmer I met at University College London (UCL), said to me: ‘Just remember, every time you watch the weather forecast in the news, it’s FORTRAN code running them.’ In other words, FORTRAN may have ‘died’ a long time ago, but it’s still around.

More than a question of ‘should I learn this tech or not?’, tales about dying languages and frameworks are just stories that programmers like to tell. Horror stories. Stories they tell around the campfire. Moments in which they share passages from their favourite books: ‘The death of a programmer’ by Donald Knuth or ‘Haskellstein’ by Sandi Metz. The sort of books programmers should be writing. To be tired of hearing that some technology is dead, like that Reddit thread, might just mean that the community has moved on to something else. ‘Ruby isn’t dying,’ wrote someone on that thread. ‘It’s maturing. It’s just not the poster child for web development any more, and that’s OK.’ Maturity is a big word in the Ruby community these days. Depending on who you talk too, it means things like ‘we like types now’ or ‘we should focus on speed’ or ‘it’s not a community of hobbyists any more’. Whatever the specific point, it certainly signals that Ruby has reached a plateau. A plateau on which Ruby and Rails coalesced to create a community that settled down. It found a nice house in the computational suburbs and had a couple of children processors.

‘The Ruby community was not destined to be enormous, really,’ Noah Gibbs told me during one of our chats. He had long hair and beard, a dark grey velvet robe and a podcaster microphone. It looked as if Noah was adapting quite well to a life with three kids and a partner in a country that he recently moved to. ‘The Ruby community was never going to be huge,' he continued. 'Truly, too many things that it does are too weird – not just in the sense of not being like other programming languages, but in the sense that there are good reasons other programming languages don't do it that way.'

'What do you mean?'

'It's so weird to me talking to people who have really only written Ruby. Ruby is a mature enough, large enough community at this point that I can talk to people that have never programmed in anything else.'

'Why is it so weird?'

'I'm an old guy. I've been through a lot of different languages. I did a lot of C. And I love Ruby, it's wonderful. But it will always feel a bit weird, because I remember all these other things.'

'And Ruby is not like those other languages …'

'Oh not, not at all. I mean, programming is programming is programming. It's all the same. But that said, Ruby is very weird.'

'Give me an example, Noah.'

'When I'm talking to these young Ruby people and I say, "What we do around here, where we dynamically generate a whole set of classes shaped like your database by reading your database schema and then just dumping it into your program, there are lots of Java people that would tell you that's insane". And people who have never done anything but Ruby go, "What, really? Doesn't everybody do that?" No, my sweet summer child, everyone does not do that.'

I left Upstream in July 2021. It was time to end the research fieldwork period. It was also a month before the next predicted round of redundancies, which fortunately didn't happen that year. It was hard to leave. I immediately missed coding with the people who I'd been coding with for such a long time. But I now had to focus on writing. I kept in touch with a few of the developers who had left Upstream in the past year or so, and also with the ones who stayed. A few of them were part of the book club, which eventually transitioned from a solidarity group among Upstream developers into a brilliant group of friends.

A little more than a year after that round of redundancies in the summer of 2020, Upstream closed shop. Jean and Lindsay were still working there and were a bit shocked. They'd stuck around until the end.

'It's not great,' Jean told me, 'to see all that code just never being used again.'

'Well, we did get paid to write it,' I replied.

‘Sure we did. But it’s still a bit strange. To think that suddenly the machines have been turned off and it just doesn’t run any more.’

‘I still have the app on my phone though.’

‘Can you buy anything with it?’

‘Nope.’

‘Precisely my point. Code is just code if it keeps running, otherwise it’s just … it’s just, I don’t know.’

Weird or not weird, many companies have chosen Ruby as their language. A language that started out as a hobbyist language, that developers used on their side projects. Then Rails came and used all the meta-programming possible to make the community grow, expand, to inspire companies to use it. Slowly the weird and useless side of hobbyist Ruby started to die down. It matured, evolved into a language community that needs to be aware of efficiency. Many companies did, and still do, rely on Ruby to function. Many of them are massive, important companies in the web’s ecosystem. Companies that use Ruby, every day. And that code still needs maintaining – at least until it doesn’t.

Is coderspeak still a language if no one is there to speak it?

Notes

[1] veravash. ‘I am tired of hearing that Ruby is dead’.

[2] iljkj. ‘Is ruby on rails (or at least the community) dying?’

[3] berkes. ‘The waning of Ruby and Rails’.

Glossary

Active-support ‘Active Support is a collection of utility classes and standard library extensions that were found useful for the Rails framework. These additions reside in this package so they can be loaded as needed in Ruby projects outside of Rails.’[1]

back-end The data layer of an application, where the logic and the operational aspects reside. Always hidden from the user, it manipulates data, makes calls to external applications and packages all the information needed for the presentation layer, the front-end. Sometimes spelled as back end or backend.

BASIC Beginner’s All-purpose Symbolic Instruction Code, created at Dartmouth College in 1964. Very influential and extremely important in popularising microcomputers in the 1970s. The British Broadcasting Corporation (BBC) had its own version of it.

C Perhaps the most influential programming language of all time. Denis Ritchie created it in 1972 while working at Bell Labs. *The C Programming Language*, a book written by Ritchie and Brian Kernighan in 1978, set the standards on how to write user manuals for computing languages.

C++ C with classes.

COBOL Computer-Business Orientated Language, created in 1959, partly based on FLOW-MATIC.

Elixir The most famous language to emerge from the Ruby community, Elixir is the brainchild of José Valim. After making massive contributions to Rails, José created a new language on top of Erlang – the programming language devised by the telephone company Ericsson. It’s a concurrent, fast and reliable modern programming language.

FLOW-MATIC A language to process data. It was directly inspired by the English language and designed by Grace Hopper in 1955.

FORTRAN Also known as ‘Formula Translating System’, FORTRAN appeared in 1957 at the offices of the International Business Machines Corporation (IBM). FORTRAN is said to be the language that started the whole business of thinking about programming as some sort of magic or devious art.

front-end User-facing, presentation layer of an application; the actual interface that a user interacts with by clicking, touching, typing or dragging. Usually connected to a back-end. Sometimes spelled as front end or frontend.

Go Programming language created by Robert Griesemer, Rob Pike and Ken Thompson in 2009 while working at Google. Looks like the C programming language, but deals better with memory management. Often referred to as Golang.

Java A programming language, consistently ranked as one of the most popular for the past 20 years. A nightmare to write in, according to many Rubyists who abandoned it in the early 2000s. High performance.

JavaScript It is said that JavaScript is the most ubiquitous programming language on the internet. It first appeared in 1995 as a way of injecting programming logic into webpages that had previously mostly used HTML and CSS. Designed by Brendan Eich while working at Netscape – one of the earliest internet browsers.

keyword Words that a programming language reserves for itself, for instance ‘true’, ‘false’, ‘class’ or ‘fun’. A program cannot use those words for anything other than their intended original meaning.

Kotlin A language that wishes to be the new Java. Chosen by Google as the preferred language for the development of Android applications, it has consistently gained a lot of traction.

library, software library A bundle of computer code that can be used by other computer programs. It is composed in a versatile way so that many different and unrelated programs can use it. For instance, a library that has code to manipulate images can be used by various social media applications. A library is often bound to a specific programming language, such as Ruby, C, Swift, etc.

Lisp LIS(t) P(rocessing) Not just a language, but a family of them. First appeared in 1960 and has influenced everything in computing ever since. Loads of brackets everywhere. People say that once you've learned L, the world makes more sense.

make Created by Stuart Feldman in 1976 at Bell Labs, make is a tool to build an executable program from source code.

Perl High-level, general purpose programming language. Loved by hackers for its capacity to be very synthetical. First released in 1987. Larry Wall, the language's designer and developer, has been a huge influence in how to steer programming communities.

Php As of July 2023, 'php is used by 77.5 per cent of all the websites whose server-side programming language we know'.[2]

proc A proc is short for 'Procedure' – meaning a record that stores a function together with an environment. This record can be sent to other functions, where it can be executed. It originated in the Lisp language, based on Alonzo Church's lambda calculus.

Python Ruby's archrival programming language, created by Guido Van Rossum. He calls himself 'Python's Benevolent Dictator for Life' – an influential 'style' of community governance in which a big man makes most decisions or has veto power on everything but doesn't often exercise it. A big chasm almost split its community in two when version 3 of the language came out: it almost killed the language. Like Ruby, Python has an approachable syntax.

Rails Ruby on Rails (usually shortened to just 'Rails') is a web framework built with Ruby. It provides tools and commands to help you build applications that will have their main interface on the internet.

React A collection of software libraries used to create user interfaces. It is written in JavaScript and it is, currently (2023), one of the most common web frameworks used to build applications on the internet. Sponsored and developed by Meta (the Facebook company).

Slack A modern messaging app, fruit of Silicon Valley. It should've replaced emails, a bit like CDs should've replaced vinyl. CDs are not here any more, so there you go.

Smalltalk Created by a team of visionaries working at the Xerox Palo Alto Research Centre in 1972. Truly object-oriented.

SQL Structured Query Language, the main language used for querying databases. SQL was invented in the early 1970s.

Swift Designed by Apple and released in 2014, Swift is one of the youngest languages to have dominated the world of apps. It replaced Objective-C in the development of applications for the family of iOS devices such as the iPhone and the iPad.

web framework A collection of software libraries that provides an easy to use and systematised way of building an application to run on a browser. It allows programmers to get something up and running very quickly.

Notes

[1] Active-Support. 'README.rdoc'.
[2] W3Usage. 'Php Usage'.

Bibliography

_why the lucky stiff. 'why's (poignant) guide to Ruby'. Brighton: Consonance Press, 2020.

_why the lucky stiff. 'Time.now.is_a? MagicTime'. Accessed on 31 October 2022. https://www.youtube.com/watch?v=HNsQxI2PdAI.

_why the lucky stiff. 'why's (poignant) guide to Ruby, chapter Six: Downtown'. 18 May 2005. Accessed on 31 October 2022. https://web.archive.org/web/20160625111758/http://rubyforge.org/pipermail/poignant-stiffs/2005-May.txt.

/DATA. Developer Nation Report 2020. 11 October 2020. Accessed 31 October 2022. https://www.developernation.net/developer-reports/de20.

Active-Support. 'README.rdoc'. Accessed on 31 October 2022. https://github.com/rails/rails/tree/main/activesupport.

Amrute, Sareeta. *Encoding Race, Encoding Class: Indian IT workers in Berlin*. Durham, NC: Duke University Press, 2016.

Arbox. 'NLP-with-Ruby'. Accessed on 31 October 2022. https://github.com/arbox/nlp-with-ruby.

Austin, J. L. *How to Do Things with Words*. Oxford: Clarendon Press, 1962.

berkes. 'The waning of Ruby and Rails'. Accessed on 31 October 2022. https://berk.es/2022/03/08/the-waning-of-ruby-and-rails/.

Black, Maurice. 'The art of computer programming'. PhD thesis. University of Michigan, 2001.

Butler, Judith. *Gender Trouble: Feminism and the subversion of identity*. New York: Routledge, 1990.

Ceruzzi, Paul. *Computing: A concise history*. Cambridge, MA: MIT Press, 2012.

Coleman, Gabriella. *Coding Freedom: The ethics and aesthetics of hacking*. Princeton, NJ: Princeton University Press, 2013.

Coleman, Gabriella and Alex Golub. 'Hacker practice: Moral genres and the cultural articulation of liberalism', *Anthropological Theory* 8 (3) (2008): 255–77. Accessed on 31 October 2022. doi:10.1177/1463499608093814.

Conway, Melvin. 'How do committees invent?', *Datamation* 14 (5) (1968): 28–31.

Elmendorf, Dirk. 'RubyGems', *Linux Journal*, 27 May 2006. Accessed on 31 October 2022. https://www.linuxjournal.com/article/8967.

Endoh, Yusuke (mame). 'Quine Relay'. Accessed on 31 October 2022. https://github.com/mame/quine-relay.

Esmenger, Nathan. 'Making programming masculine'. In *Gender Codes: Why women are leaving computing*, edited by Thomas J. Misa, 115–41. Hoboken, NJ: The IEEE Computer Society, 2010.

Esolangs. 'Piet'. Accessed on 31 October 2022. https://esolangs.org/wiki/Piet.

Famished-Tiger. 'Rley'. Accessed on 31 October 2022. https://github.com/famished-tiger/Rley.

Flanagan, David and Yukihiro Matsumoto. *The Ruby Programming Language*. Sebastopol, CA: O'Reilly, 2008.

Fowler, Martin. 'Microservices. A definition of this new architectural term', 25 March 2014. Accessed on 31 October 2022. https://martinfowler.com/articles/microservices.html.

Fowler, Martin. 'Rake', 25 March 2014. Accessed on 31 October 2022. https://martinfowler.com/articles/rake.html.

Fowler, Martin. *Refactoring: Improving the design of existing code* (2nd edition). Boston, MA: Addison-Wesley, 2018.

Graham, Paul. *Hackers & Painters: Big ideas from the computer age*. Sebastopol, CA: O'Reilly Media, 2010.

Gregory, Chris. *Gifts and Commodities*. Chicago, IL: Hau Books, 2015.

Grimm, Avdi. 'Confident Ruby'. Author's edition, 2013.

Heinemeier-Henson, David. 'Ruby on Rails demo', 8 November 2005. Accessed on 31 October 2022. https://www.youtube.com/watch?v=Gzj723LkRJY.

Heurich, Guilherme Orlandini. 'Language Automata: The pervasiveness of English in computer programming', unpublished.

Heurich, Guilherme Orlandini. 'Your random Chomsky', Accessed on 31 October 2022. https://your-random-chomsky.herokuapp.com/.

Hicks, Mar. *Programming Inequality: How Britain discarded women technologists and lost its edge in computing*. Cambridge, MA: MIT Press, 2018.

Himanen, Peka. *The Hacker Ethic and the Spirit of the Information Age*. New York: Penguin, 2001.

Hoff, Todd. 'LinkedIn Moved From Rails to Node', 4 October 2012. Accessed on 31 October 2022. http://highscalability.com/blog/2012/10/4/linkedin-moved-from-rails-to-node-27-servers-cut-and-up-to-2.html.

Hopper, Grace. 'Keynote address at the Association for Computing Machinery SIG PLAN History of Programming Languages (HOPL) conference (1978)'. In *History of Programming Languages*, edited by Richard L. Wexelblat. New York: Academic Press, 1981.

iljkj. 'Is ruby on rails (or at least the community) dying?' Accessed on 31 October 2022. https://stackoverflow.com/questions/3794270/is-ruby-on-rails-or-at-least-the-community-dying/3794316#3794316.

INTERCAL. Accessed on 31 October 2022. https://en.wikipedia.org/wiki/INTERCAL.

IRC. '14 May 2013 logs'. Accessed on 31 October 2022. https://viewsourcecode.org/why/CLOSURE/ircLog.html.

Irvine, Judith and Susan Gal. *Signs of Difference: Language and ideology in social life*. Cambridge: Cambridge University Press, 2019.

Ishitsuka, Keiju. '[ruby-dev:5173] Re: to_i,to_s の素朴な疑問', 12 February 1999. Accessed on 31 October 2022. https://web.archive.org/web/20220516231050/http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-dev/5173.

Jargon File. Accessed on 31 October 2022. https://www.catb.org/jargon/.

Kernighan, Brian. *UNIX: A history and a memoir*. Kindle Direct Publishing, 2020.

Kittler, Friedrich. 'There is no software'. In *The Truth of the Technological World: Essays on the genealogy of presence*, edited by Friedrich Kittler, 219–29. Stanford, CA: Stanford University Press, 1992.

Knuth, Donald. 'Computer programming as an art'. 1974 Turing Award Lecture, *Communications of the ACM* 17 (12) (1974), 667–73.

Knuth, Donald. 'Literate programming', *The Computer Journal* 27, 1984.

Kroskrity, Paul. 'Language ideologies'. In *A Companion to Linguistic Anthropology*, edited by Alessandro Duranti, 496–517. Malden, MA & Oxford: Blackwell, 2004.

Kurtz, Thomas. 'BASIC'. In *History of Programming Languages*, edited by Richard L. Wexelblat. New York: Academic Press, 1981.

Lawrence, Halcyon. 'Siri disciplines'. In *Your Computer Is On Fire*, edited by Thomas S. Mullaney, Benjamin Peters, Mar Hicks and Kavita Phillip. Cambridge, MA: MIT Press, 2021.

Levy, Steven. *Hackers: Heroes of the computer revolution*. Penguin: London, 1998.

Lowrey, Annie. 'What happened when one of the world's most unusual, and beloved, computer programmers disappeared?', *Slate*, 15 March 2012. Accessed on 31 October 2022. http://www.slate.com/articles/technology/technology/2012/03/ruby_ruby_on_rails_and__why_the_disappearance_of_one_of_the_world_s_most_beloved_computer_programmers_.html?via=gdpr-consent&Via=gdpr-consent#return.

LRUG. 'January 2020 meeting'. Accessed on 31 October 2022. https://lrug.org/meetings/2020/#january-2020-meeting.

Marino, Mark. *FLOW-MATIC*. Critical Code Studies. Cambridge, MA: MIT Press, 2020.

Martin, Robert. *Clean Code*. Upper Saddle River, NJ: O'Reilly, 2009.

Matz, Yukihiro Matsumoto. '[ruby-talk:00382] Re: history of ruby', 4 June 1999. Accessed 31 October 2022. https://web.archive.org/web/20220516220808/http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/382.

Matz, Yukihiro Matsumoto. 'The man who gave us Ruby', interview, June 2002. Accessed 31 October 2022. https://www.japaninc.com/article.php?articleID=828.

Matsumoto, Yukihiro. 'Treating code as an essay'. In *Beautiful Code: Leading programmers explain how they think*, edited by Andy Oram and Greg Wilson, 477–81. Sebastopol, CA: O'Reilly, 2008.

Mauss, Marcel. *The Gift: The form and reason for exchange in archaic societies*. Routledge: London, 1990.

McPherson, Tara. 'U.S. operating systems at mid-century'. In *Race after the Internet*, edited by Lisa Nakamura and Peter Chow-White. London: Routledge, 2012.

Metz, Sandi, Katrina Owen and T. J. Stankus. *99 Bottles of OOP: A practical guide to object-oriented design*. Durham, NC: Potato Canyon Software, 2020.

Montfort, Nick. ‘Obfuscated code’. In *Software Studies: A lexicon*, edited by Matthew Fuller. Cambridge, MA: MIT Press, 2008.

Nasser, Ramsey. ‘A personal computer for children of all cultures’, *Decolonising the Digital: Technology as Cultural Practice*, 21–36. Sydney: Tactical Space Lab, 2018.

Nasser, Ramsey. ‘The قلب Programming Language’. Accessed on 31 October 2022. https://github.com/nasser/—.

O’Reilly, Tim. *What Is Web 2.0*. Accessed on 31 October 2022.

Perry, Grayson. *The Descent of Man*. London: Penguin, 2012.

Petzold, Charles. *Code: The hidden language of computer hardware and software*. Washington, DC: Microsoft, 2000.

Petzold, Charles. *The Annotated Turing*. London: Wiley, 2008.

Petzold, Charles. ‘On-the-fly code generation for image processing’. In *Beautiful Code: Leading programmers explain how they think*, edited by Andy Oram and Greg Wilson. Sebastopol, CA: O’Reilly, 2008.

Quetteville, Harry de. ‘Teaching tech: How coding moved from the bedroom to the classroom’, *The Telegraph*, 20 March 2018. Accessed 31 October 2022. https://www.telegraph.co.uk/technology/teaching-tech/.

Rails. ‘Add Symbol#to_proc, which allows for, e.g. [:foo, :bar].map(&:to_s)’. Accessed on 31 October 2022. https://github.com/rails/rails/commit/69bf71f5e9b537f88acc0d4492a057336e7305d1.

Raymond, Eric S. *The Cathedral and the Bazaar: Musings on Linux and open source by an accidental revolutionary*. Sebastopol, CA: O’Reilly Media, 1999.

Remington Rand Univac Corporation. ‘Introducing a new language for automatic programming: UNIVAC FLOW-MATIC’, brochure, 1957. Accessed on 31 October 2022. http://archive.computerhistory.org/resources/text/Remington_Rand/Univac.Flowmatic.1957.102646140.pdf.

Ruby World Conference. RWC2020 基調講演 2, David Heinemeier Hanson 英語. 9 February 2021. Accessed on 31 October 2022. https://www.youtube.com/watch?v=QycKZT0Spfg.

Ruby. ‘object.c (sym_to_proc): imported Symbol#to_proc from ActiveSupprot’. Accessed on 31 October 2022. https://github.com/ruby/ruby/commit/ac4d6ddfa3219c212d2865ed600a0ab568d5f0b5.

Rubygems. ‘Stats’. Accessed on 31 October 2022. https://rubygems.org/stats.

Rubygems. ‘Rake-versions’. Accessed on 31 October 2022. https://rubygems.org/gems/rake/versions.

Sammet, Jean. ‘General views on COBOL’, *Annual Review in Automatic Programming* 2 (1961), 345–9.

Sammet, Jean. ‘History of IBM’s technical contributions to high level programming languages’, IBM Journal of Research and Development, 25 (5) (1981), 520–34.

Sasada, Koichi. ‘Introduction of MRI development culture’, Ruby Hack Challenge. Accessed on 31 October 2022. https://github.com/ko1/rubyhackchallenge/blob/master/EN/1_culture.md.

Schierbeck, Daniel. ‘Symbol#to_proc is just so beautiful’, (a) Ruby-talk mailing list, 20 April 2006. Accessed on 31 October 2022. https://web.archive.org/web/20220516104938/http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/189423.

Schierbeck, Daniel. ‘Symbol#to_proc is just so beautiful’, (b) Ruby-talk mailing list, 20 April 2006. Accessed on 31 October 2022. https://web.archive.org/web/20200814190729/http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/189468.

Schneem, Richard. ‘The room where it happens: How Rails gets made’, blog post. Accessed on 31 October 2022. https://schneems.com/2021/05/12/the-room-where-it-happens-how-rails-gets-made/.

Schultz, Eric. ‘The effect of last week on Ruby on Rails’, Ruby on Rails discussion. Accessed on 31 October 2022. https://discuss.rubyonrails.org/t/effect-of-the-last-week-on-ruby-on-rails/77702.

Schwaderer, Nick (@schwad_rb). Accessed on 31 October 2022. https://twitter.com/schwad_rb/status/1520021704847855617.

Shepelev, Victor (@zverok). ‘A long rant about Ruby, its value, its innovations and its (sad) fate’, 25 February 2021. Accessed on 31 October 2022. https://twitter.com/zverok/status/1365014133180141578.

Silverstein, Michael. ‘Shifters, linguistic categories and cultural description’. In *Meaning in Anthropology*, edited by Keith H. Basso and Henry A. Selby, 11–55. Albuquerque, NM: University of New Mexico Press, 1976.

Stanton, Andrea. ‘Broken Is Word’. In *Your Computer Is On Fire*, edited by Thomas S. Mullaney, Benjamin Peters, Mar Hicks and Kavita Phillip. Cambridge, MA: MIT Press, 2021.

Star, Susan Leigh. ‘The structure of ill-structured solutions: Boundary objects and heterogeneous distributed problem solving’. In *Distributed Artificial Intelligence*, edited by Les Gasser and Michael N. Huhns. London: Pitman Publishing, 1998.

Steele, Murray. ‘Stegosaurus’. Accessed on 31 October 2022. https://github.com/h-lame/stegosaurus.

steveklabnik. ‘CLOSURE’. Accessed on 31 October 2022. https://github.com/steveklabnik/CLOSURE/tree/master/PDF.

Strathern, Marilyn. *The Gender of the Gift: Problems with women and problems with society in Melanesia*. London: University of California Press, 1990.

Susser, John. ‘Symbol to Proc shorthand’. Has Many Through Blog. Accessed on 31 October 2022. https://web.archive.org/web/20061222213756/http://blog.hasmanythrough.com:80/articles/read/8.

Takhteyev, Yuri. ‘Open source, open world: Where free software came from – and where it’s going’, *Foreign Affairs*, 13 September 2022. Accessed on 31 October 2022. https://www.foreignaffairs.com/world/open-source-open-world.

Temkin, Daniel. ‘Esoteric programming languages’, UCL Centre for Digital Anthropology Workshop on the Limits of Programming Languages. https://youtu.be/JUX1NoTyqI0.

Tomasello, Michael. *The Cultural Origins of Human Cognition*. Cambridge, MA: Harvard University Press, 2001.

Torvalds, Linus. *Just for Fun: The story of an accidental revolutionary*. New York: HarperBus, 2011.

Triplett, Kevin. ‘Why The Lucky Stiff Documentary’. Accessed on 31 October 2022. https://www.youtube.com/watch?v=64anPPVUw5U.

Turing, Alan. ‘On computable numbers, with an application to the Entscheidungsproblem’, *Proceedings of the London Mathematical Society* 2, 42 (1) (1937): 230–65.

Turing, Alan. ‘Computing machinery and intelligence’, *Mind* 49 (236) (1950): 433–60.

Turkle, Sherry. *The Second Self: Computers and the human spirit*. Cambridge, MA: MIT Press, 1984.

Venners, Bill. ‘A conversation with Yukihiro Matsumoto’, Part III. 22 December 2003. Accessed on 31 October 2022. https://www.artima.com/articles/blocks-and-closures-in-ruby.

veravash. ‘I am tired of hearing that Ruby is dead’. Accessed on 31 October 2022. https://www.reddit.com/r/ruby/comments/hp3yar/i_am_tired_of_hearing_that_ruby_is_dead/.

Visser, Sander. ‘What tutorials would make a non-developer think you’re a magician?’, 3 November 2017. Accessed on 31 October 2022. https://dev.to/sanderfish/what-tutorial-would-make-a-non-developer-think-youre-a-magician-4lh.

Wall, Larry. ‘Larry Wall: Why Perl is like a human language’. Accessed on 31 October 2022. https://www.youtube.com/watch?v=ju1IMxGSuNE.

Weirich, Jim. ‘DSL Definition’, *Rake*. Accessed on 31 October 2022. https://github.com/ruby/rake/blob/master/lib/rake/dsl_definition.rb.

Weirich, Jim. ‘Glossary’, *Rake*. Accessed on 31 October 2022. https://github.com/ruby/rake/blob/master/doc/glossary.rdoc.

Weirich, Jim. ‘Rational’, *Rake*. Accessed on 31 October 2022. https://github.com/ruby/rake/blob/master/doc/rational.rdoc.

Wikipedia. “Universally unique identifier’. Accessed on 31 October 2022. https://en.wikipedia.org/wiki/Universally_unique_identifier.

Wikipedia. ‘We can do it!’. Accessed on 31 October 2022. https://en.wikipedia.org/wiki/We_Can_Do_It!.

W3Usage. ‘Php usage’. Accessed on 17 July 2023. https://w3techs.com/technologies/details/pl-php.

XKCD. ‘How standards proliferate’. Accessed on 31 October 2022. https://xkcd.com/927/.

Index

‘Heurich perfectly captures the generous camaraderie, quirky spirit and intellectual curiosity at the heart of the Ruby world. Packed with tidbits of Ruby history, code snippets and fascinating conversations, this book has something to teach every Rubyist.’
Jemma Issroff, Ruby Core Team

‘This delightful book provides a fascinating window into the world and words of computer programmers. Beautifully written, engaging and insightful, Heurich’s ethnography takes us on a journey through the history and life of Ruby programming, showing the joy, humour and poetry needed to bring code to life.’
Hannah Knox, UCL

Software applications have taken over our lives. We use and are used by software many times a day. Nevertheless, we know very little about the invisibly ubiquitous workers who write software. Who are they and how do they perceive their own practice? How does that shape the ways in which they collaborate to build the myriad of apps that we use every day?

Coderspeak provides a critical approach to the digital transformation of our world through an engaging and thoughtful analysis of the people who write software. It is a focused and in-depth look at one programming language and its community – Ruby – based on ethnographic research at a London company and conversations with members of the wider Ruby community in Europe, the Americas and Japan. This book shows that the place people write code, the language they write it in and the stories shared by that community are crucial in questioning and unpacking what it means to be a ‘coder’. Understanding this social group is essential if we are to grasp a future (and a present) in which computer programming increasingly dominates our lives.

Guilherme Orlandini Heurich is Honorary Research Fellow at the Department of Anthropology, UCL and Software Engineer at the BBC.

Free open access
version available from
www.uclpress.co.uk

UCLPRESS

Cover design:
www.hayesdesign.co.uk

ISBN 978-1-80008-599-2
90000
9 781800 085992