Francesco Luzzana

Hello Worlding

Code documentation as entry point backdoor to programming practices

Adviser
Marloes de Valk

Second Reader
Lidia Pereira

Word Count
8182

Thesis submitted to Department of Experimental Publishing, Piet Zwart Institute, Willem de Kooning Academy, in partial fulfilment of the requirements for the final examination for the degree of: Master of Arts in Fine Art & Design: Experimental Publishing.

This document is generated with Readme, fetching contents from git.xpub.nl/kamo/thesis

0. Intro

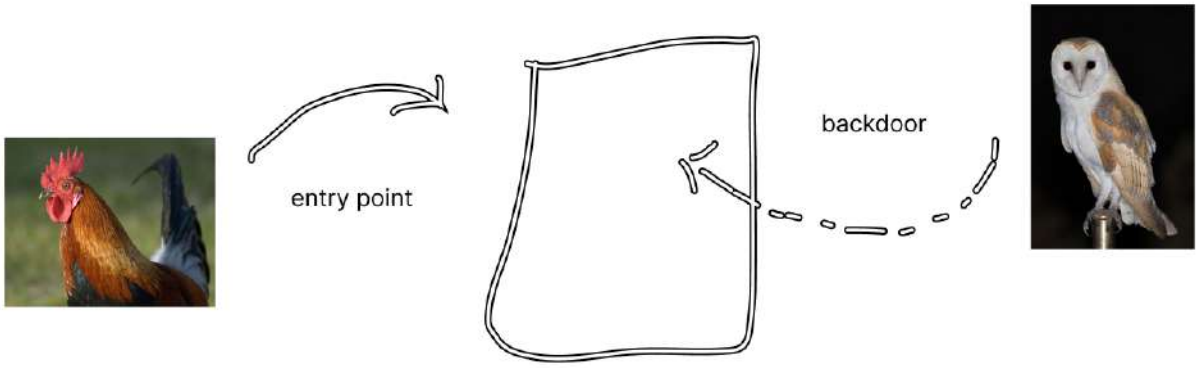
You cannot think about code without thinking also about code documentation.

Programming is like walking in a room without turning the lights on. It can be a place you know by heart, but you still prefer to rely on some guidance, using your hands to sense the walls, and move through the furnitures without hitting your pinkie toe.

For code is the same: you usually appreciate some guidance.

Documentation can be as simple as a plain text file placed near the code. A `README.txt` that invites developers to take a look before diving into the program. Printed technical manuals have today transformed and spreaded into many different shapes: wikis and platforms and websites generated with various tools, each with particular focus and features.

This research explores two currents around documentation practices, with the thesis that code documentation is an ideal publishing surface to create worlds around software, and to orientate software in the world.



1. Two flows around code documentation: entry points and backdoors.

`Current #1` Documentation broadens participation in the making of software: lowering barriers and offering entry points for people to understand and attune with code.

`Current #2` Documentation as a backdoor where to inject context into software: to host principles in close contact with algorithms, letting them entangle and shape each other.

The nature of code documentation is to create entry points for people to participate in programming practices. To encode and filter knowledge, and ultimately to share it with others. This "nature", however, does not come without issues. It makes a lot of assumptions about who's reading, expecting experts, or

engineers, or dudes. Its language is unwelcoming: too dense, too technical, very gendered and unable to address anyone but the neurotypical-white-cis-male programmer. Documentation requires an enormous amount of care, energy and time to be maintained, and it's done always out of budget, always as a side project, always at the end, and only if there's time left. The first chapter raises these points to note how often code documentation acts as a barrier, gatekeeping access to the making of software.

Even if it does a questionable job at creating entry points, code documentation still has a lot of potential as a backdoor. It's a publishing surface whose reach extends through time and space. Time because it meets programmers at different moments in their lives: from the *hello world* till the *how to uninstall*, and it influences thinking about software continuously, and from different perspectives. Space because it comes in many different possible formats, and can shapeshift to serve different occasions: from simple `.txt` files to entire websites, from coding workshops to comments in the source code to series of video tutorial.

The question then becomes: can we make use of these backdoors to infiltrate programming practices and open more entry points from within?

Code documentation is an interface between the code, the user, the developer, and the world. Living close to the source code, but at the same time being less rigid and more expressive, it seems to be an ideal surface to influence software development practices. The second chapter presents some examples of how documentation can be used to orientate code in the world, addressing politics of participation, representation, and authorship in programming. The case studies come from diverse realities, and from different scales: large collaborative projects as well as small, personal gestures. In their multiplicity, they show how blurred the boundaries of code documentation are. A lack of fixedness which in turn can be used to mold our wishes and values into it.

A term to contextualize (and dismantle?) in these writings is *developer*. Stripping away any trace of professionalism and formal education, let's agree that a developer is someone who tinkers with code. No matter what level, nor distance, nor experience. No matter if for work, for fun, for study. No matter if one is actively involved in the development of a tool, or comes from the user perspective. Ellen Ullman writes that programming is a disease, but the situation is even worse: code is contagious material, touch it once, and you are a developer.

1. Entry points

"Natural" reader

Documentation that assumes a certain type of reader can result inaccessible. The recipient is often thought to be similar to the writer: familiar with the subject, comfortable with technicalities, and able to cut through the precise jargon and esoteric references offered as explanation. Ultimately (and in most cases) the reader is someone else. This mismatch turns entry points into barriers that filter out who can participate in coding.

Whenever too much technical proficiency is required to even read the documentation, knowledge becomes inaccesible, and confined in the ivory tower. Not filtering information becomes a filter to who can engage with it, a backfiring practice that reinforces the segmentation between who is allowed in and who is not: only the already knowledgeable ones can access, while others are kept out. Contents need to be curated, that does not mean oversimplified or generalised, but rather made legible.

Cultivating legibility is not an easy task, especially when it comes to computer technology: a cards castle of abstractions built on top of other abstractions. These abstractions are more than just metaphors: they are interconnected narratives and intertwined plots and main characters at the same time. The purpose of an abstraction is to function as a symbol, as a mentally manoeuvrable concept, free from the details of its technical implementation. Yet the piling up of these structures makes for a dense forest with no clear path to follow in sight. Programming is the perfect rabbit hole because of the depth and complexity of each layer that makes up the digital stack.

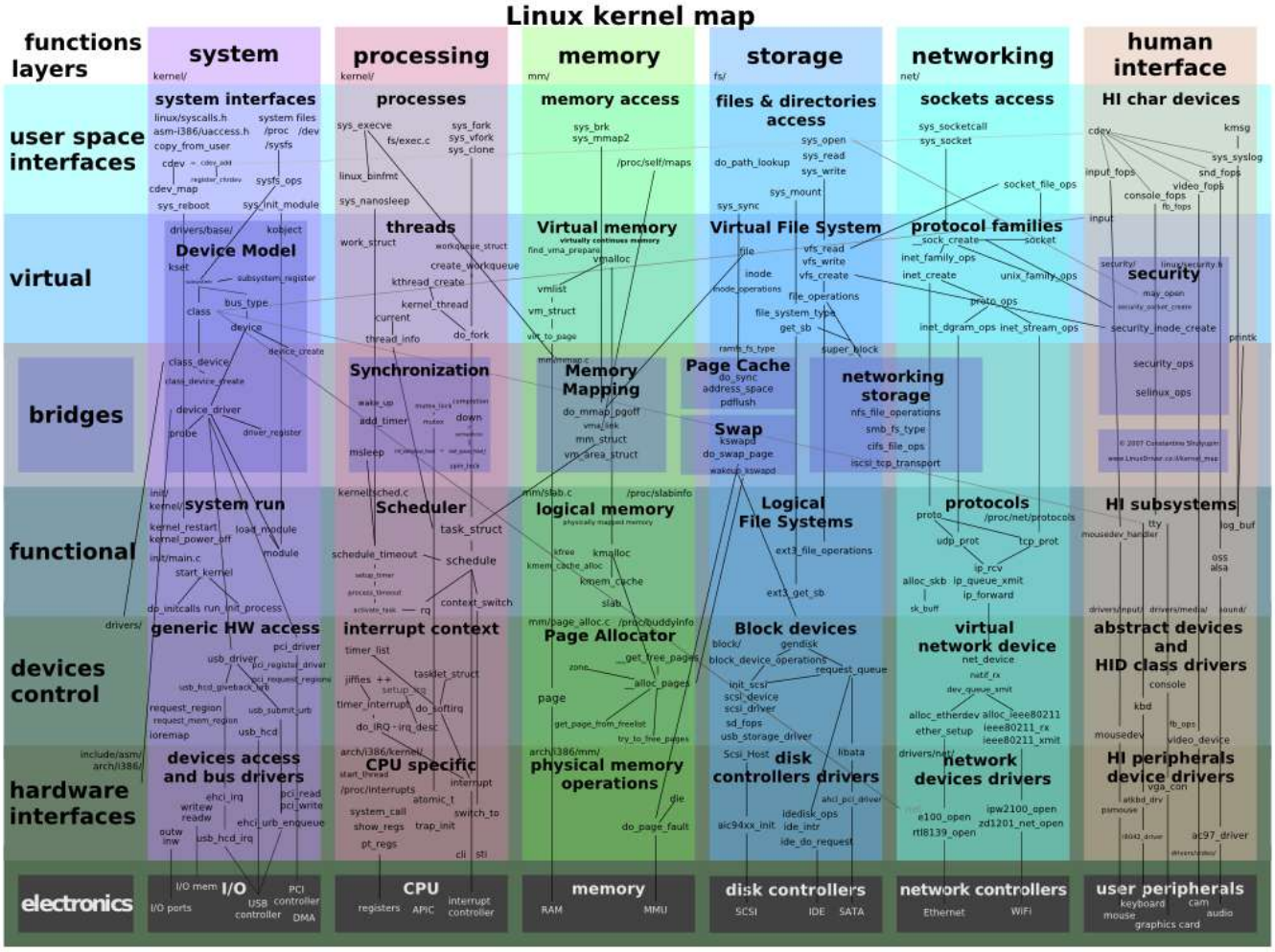


Figure 2. Linux kernel map. From the bottom up, every horizontal layer is a level of abstraction that build on the previous one.

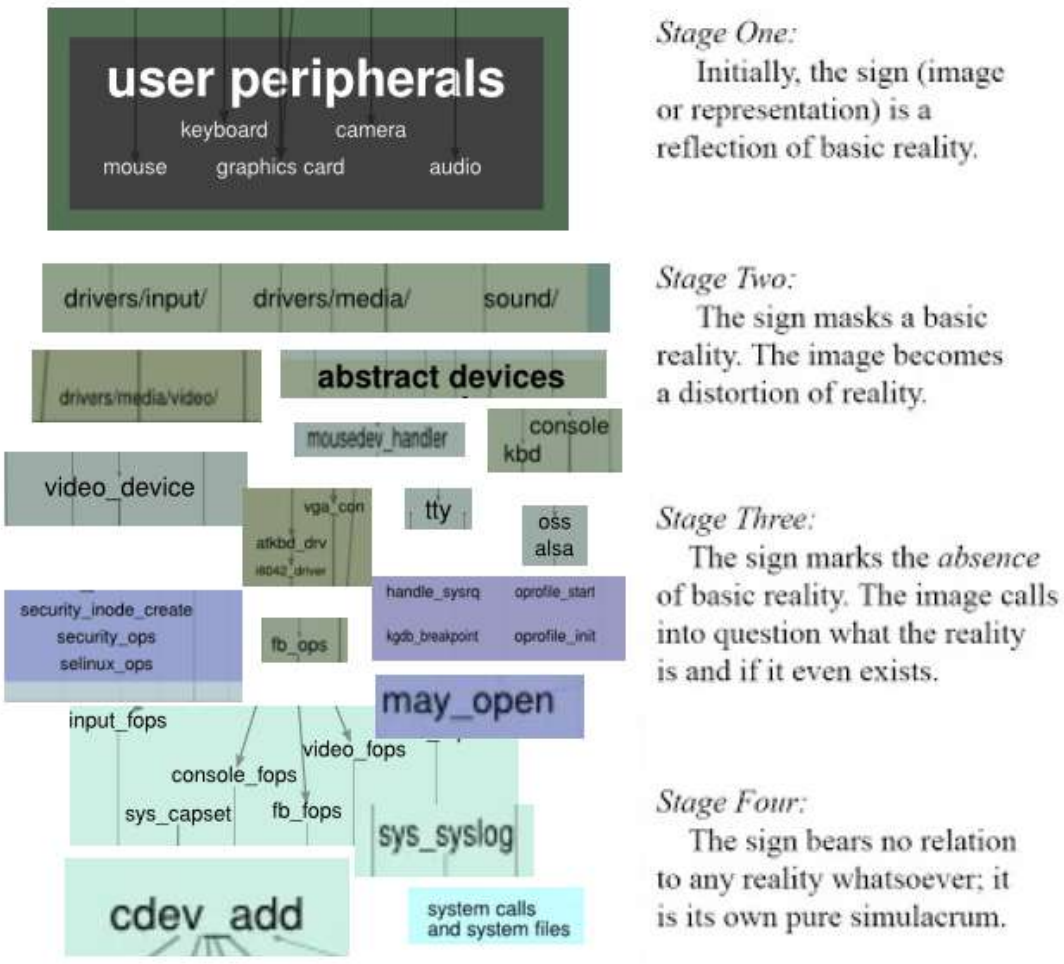


Figure 3. Detail of Human Interface Functions mapped onto the four stages of simulation meme template.

Take a course such as the one presented by Noam Nisan and Shimon Schocken in *From NAND to Tetris*, where they slowly build a programmable computer capable of running the classic game, starting from simple NAND logic gates, in other words, from microchips and electronics. Layer after layer, from boolean operations with 0 and 1 to registers and CPUs, from machine language to high level programming. Here one can try to unwind the coil and start understanding programming from scratch, but this approach is best suited to a university curriculum, and it's often not very effective when facing real-world problems, with real-world constraints, and real-world circumstances.

A deep understanding of technical systems is of course admirable and desirable, given the insights it can provide into the infrastructures that shape our everyday lives. But it cannot be the only mode of access available. Deep understanding comes with its own learning curve, and it can be a barrier for many people. Yet, many, many guides resemble this setup: pieces impossible to read if before one hasn't read an equivalent illegible piece of documentation and so on, tracing back till the invention of the wheel.

A different kind of approach, more modelled on the way we encounter technology and coding in real life, starts in the middle and tries to make sense of its surroundings. You might just need to make a website, for example. And you could just start doing that, following a guide or a tutorial. Soon questions would start bubbling up: written from scratch or with a framework? And which one to choose? What about the backend? Where to host it? On what kind of server? Static or dynamic? And the *content management system* for uploading new material? And where do you get the certificate for secure connection? These things certainly are important, but it is not really necessary to know everything in order to put the website online. Programming is provisional: leave TODOs in the code to come back to later.

The series *Programming Projects for Advanced Beginners* by Robert Heaton embraces this methodology. Each project offers some guidance through the different steps involved in coding a particular application: a login system, a simple game, a graphic filter to apply to the webcam, etc. A nice aspect of these guides is that they don't refer to a specific programming language: they are decoupled tutorials that leave the reader space to integrate and adapt the steps to their own coding contingencies, while at the same time helping to build a lexicon, teach how to search for informations, read error messages and find their way around. As in *NAND to Tetris* things are built incrementally. Here, however, the process is iterative and circular, rather than linear. Implementations are put in place provisionally, and then reiterated, replaced and developed more: new concepts are introduced not as hard-coded procedures, but as a result of emerging problems. The entry points here are multiple, like the spokes in a bicycle wheel. They come from different directions and don't frame the code as a prescriptive and rigid system, but rather as a crafted balance between different forces and needs at play. Such kind of technical objects feel less monolithic and more approachable.

A lesson can be learned: sometimes code is about performance, sometimes is about flexibility, sometimes is about accessibility, but rarely about all of these at once. Programming is about balancing these different aspects depending on the situation. Keeping this balance in mind when writing code documentation gives to the writer room to adjust the tone, intensity and approach depending on who will be reading these docs.

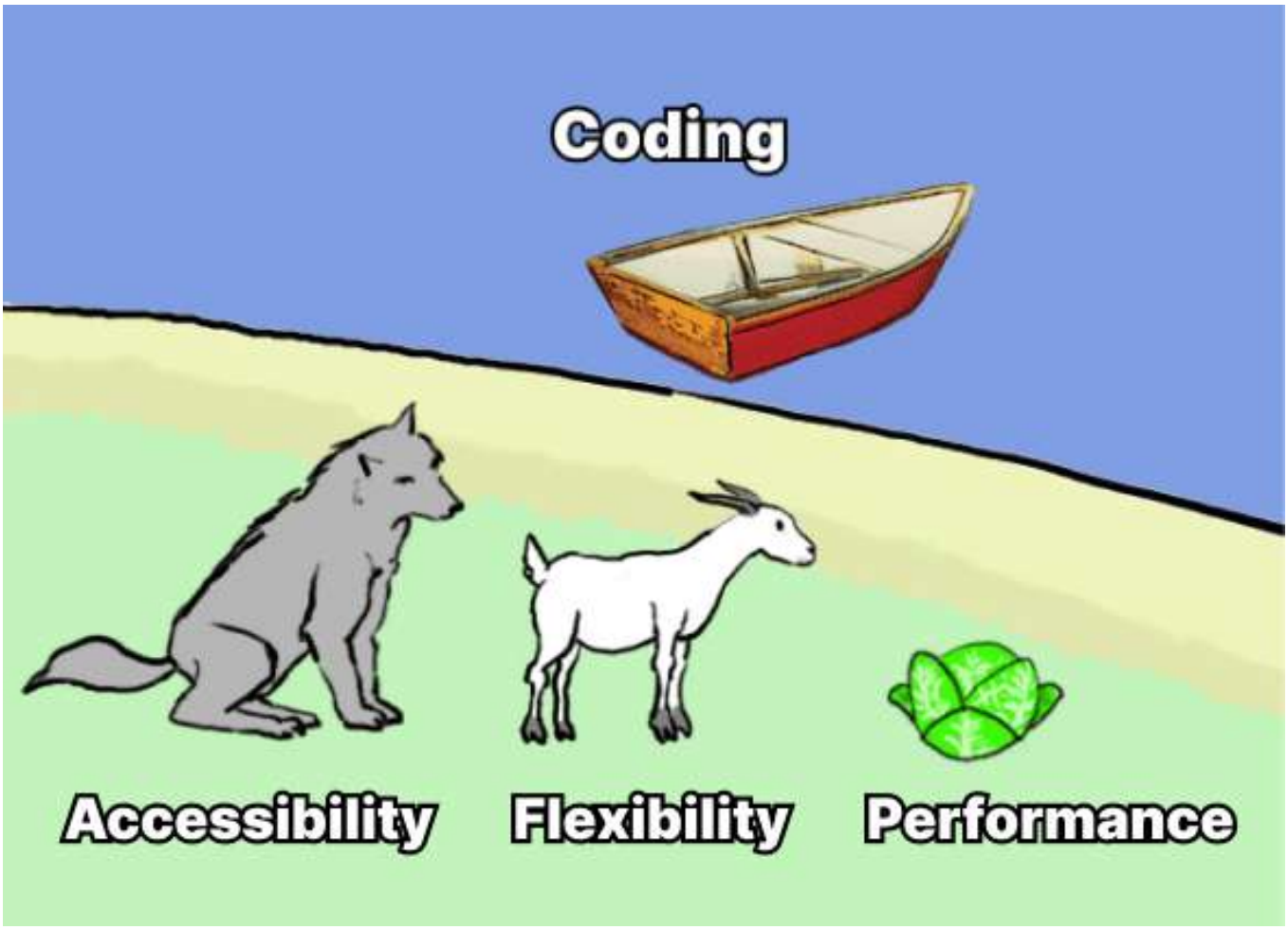


Figure 4. The wolf, goat and cabbage problem applied to coding.

Programming language

It's not just about the content and approach to technicalities, but also the very language in which they are formulated and presented.

Historically code documentation has been aimed at a very specific audience. The places where software used to be developed—universities, civilian and military research labs and IT companies—were mostly populated by white dudes. This really particular monoculture probably comes as a result of several overlapping factors: the prohibitive costs of higher education, the concentration of foundings in really specific parts of the Western world, a patriarchal society that didn't foster women in technical sectors, and a racist and segregative model that systematically forced minorities and people of color into subaltern and menial tasks.

Ellen Ullman is a programmer and writer, one of the few women to work as a developer in Silicon Valley in the 80s and 90s. The combination of a liberal arts background, being a self-taught programmer, and above all being a woman, made her the archetypal outsider in the IT industry. At the same time, this very

position granted her a unique ethnographic perspective, able to look critically at this environment from both the inside and from the outside.

In her books, she recounts how the presence of female figures in the IT sector was uneven: when she visited tech conventions, women were only to be found among *computer trainers and technical writing conferences*, some of them in the application development field, *"high-level, low status, relatively-low payments"* . Closer to the machine: the desert. In the *low valley of programming* not a female person in sight, for these [more technical conventions] are gathering of young men. (1997, 2016)

Many episodes in her writings describe interactions with colleagues in which she is directly attacked for being a woman who dares to enter the technical zone of engineering. Or a client harassing her while she was working to fix his database. Or the segregation of *cheap latina workers* hired to do mechanical data entry in the area outside the mainframe room, where all the other guys were gathered.

Note

"Workers leaving the Googleplex" present this same last situation more than twenty years later, with Google pushing minorities towards subaltern unskilled work. See: http://www.andrewnormanwilson.com/WorkersGoogleplex.html

This condition is also reflected in the pages of code documentation. Technical manuals and software specifications have been writtem for—and from the point of view of—this very specific public, populated mainly by male engineers.

Mara Karayanni researches technical documentation from a feminist perspective. The project *Read The Feminist Manual*, published by Psaroskala Zine, presents an investigation of gendered language in software manuals. A case study is about the `gettext` localisation tool from the GNU community. The program provides a system to internationalise other code, allowing developers to translate prompts and contents in different languages other than English. It's an application that already implies a collaboration between different kinds of knowledge (developers, translators) in the making of software. Nevertheless, the manual begins with the sentence:

"In this manual, we use he when speaking of the programmer or maintainer, she when speaking of the translator, and they when speaking of the installers or end users of the translated program."

This gendered language comes with an embedded division of roles.

Open-source software development happens through code contributions within communities, and indeed someone submitted a patch to change the pronouns in the documentation, proposing a neutral approach to undoing the stereotypes and broadening the people represented by the documentation. But the patch was rejected, and the pronouns remain. Eventually, a disclaimer was added: that the gendered language does not mean that certain roles are best suited to men, and that the wording is simply a way of writing clearer instructions.

Karaianni reports further discussion on the GNU mailing list, where the proposal was rejected in favour of grammatically correct English, and because there was no perceived need for fair representation in a technical object. As argued in *Read The Feminist Manual*, the resistance against gender neutral language in technical writing is just a pretext for refusing to waiver the priviledge of the male programmer.

Toxic geek masculinity reinforces stereotypes such as gendered roles in programming, and refuses to acknowledge the participation of diverse identities in the making of software, starting with the very language and attitudes used in writing documentation. From this perspective, documentation becomes an important space for building community around software. Who are we writing code documentation for? Who will read it? Who are we keeping out and who are we letting in? Who is represented and who feels invited and welcomed?

Welcoming writing

Writing documentation is demanding. It's more delicate than programming, and requires a whole set of skills not usually treasured by the dev community. A kind of emotional intelligence and sensitivity that is far to be found in the competitive and pragmatic wastelands of the IT industry. Nobody here wants to write documentation, or pay anyone to do it. As a result, in a world where software thrives, documentation still seems to be a scarce resource.

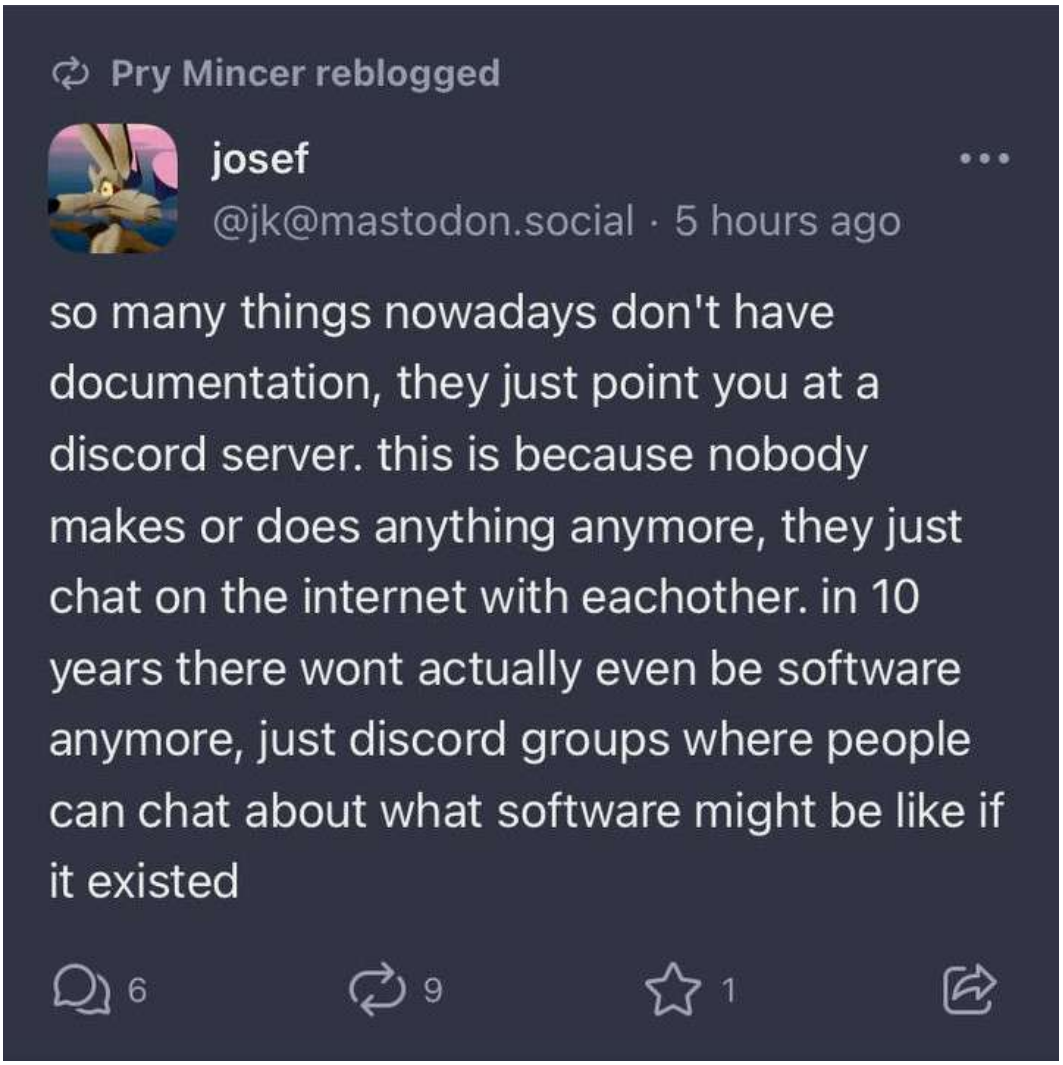


Figure 5. A provocative post with slightly austrian accent on Mastodon

It's ok, someone could argue, every question that can be asked on Stack Overflow, will eventually be asked in Stack Overflow (versioning Atwood, 2007). The popular Q&A website for developers is just an example of digital knowledge as a shared effort, together with the endless mailing lists, forums, discord servers and dedicated sources for whatever topic. It's astonishing how online communities can tackle any problem in no time.

But it's not rare for these places to feel unwelcoming, or even hostile. In 2018, Stack Overflow publicly admitted that it had a problem with its user base. The space felt unfriendly for *outsiders*, such as newer coders, women, people of color, and others in marginalized groups (Hanlon, 2018).

There have been discussions about tone on the platform for years. At the question *"Should 'Hi', 'thanks', taglines, and salutations be removed from posts?"*, one of Stack's founders responded with a RegEx to *automagically* find & remove what some of the experienced users perceived as noise. This *regular expression*, a way of targeting specific text patterns in programming, then began to be silently applied to every request sent to the website, trimming out etiquette and leaving only technicalities.

Far from being just an isolated problem, this crudity is deeply embedded in the IT discourse, soaking through technical writings as well. The denigrating expressions of superiority in matters concerning programming which Marino calls *encoded chauvinism* (2020) constitute the main ingredient in the brew of toxic geek masculinity. *Real programmers* don't use that code editor. *Real programmers* don't use that programming language. *Real programmers* don't care about others feelings. Etc.

Ellen Ullman's accounts of the emotional dumbness of her *real programmer* colleagues offer a glimpse of a problematic behavior, that was first intercepted and then capitalised on by the IT industry. *"In meetings, they behave like children. They tell each other to shut up. They call each other idiots. They throw balled-up paper. One day, a team member screams at his Korean colleague, 'Speak English!' (A moment of silence follow this outburst, at least.)"* (Ullman, 2017)

Programming means dealing with picky stubborn machines that won't overlook a single typo. It requires a high tolerance for failure. It is frustrating. But to project that frustration onto other users, as in the typical response to a request for help of `Read The Fucking Manual`, is a form of *negative solidarity*: others should suffer as I have when trying to understand how code works.

Mark Fisher used the image of negative solidarity in the context of labor under capitalism, where workers are forced into precarity and isolation. Here as in a downward auction, people are driven to bring each others down, to wish to others their same struggles. (2013) I'm using it with a focus on the emotional component: not only the lack of empathy and solidarity, but also the reproduction and legitimisation of toxic behaviours in coding communities. When all the energies are invested in optimisation and debugging, in considering and solving only techical problems, and no space is left for introspection, programmers start to behave like machines. This lack of empathy is a barrier to the participation of others in the making of software.

Here are some examples that go in a different direction, on different scales.

`p5.js` is a popular Javascript library started by the artist Lauren McCarthy as an online port of Processing, itself being a project to promote both software literacy in the visual arts, and visual literacy in software development.

The documentation work around `p5.js` provides entry points into the world of programming, being careful not to take too much for granted. For example, the amount of care and effort in their tutorial about debugging, results in a welcoming article with multiple levels of accessibility. Here the drawings help to visualise complex concepts, the tutorial format is beginner-friendly, and the narrative makes for an interesting reading even for those already familiar with debugging.

One of the most frightening aspects of programming is being confronted with stack trace errors: when things don't work as expected and red error messages appear. These scarlet letters delivered by the code are useful for developers to identify where in the program the error occurred, but they are often dense with technical jargon and difficult to decipher: a worst-case scenario for beginners. The explanations from the `p5.js` Education Working Group tackle on this nightmare showing not only how to read technical errors, but how to think through them with different debugging methods. From here, the stack trace starts to become less alien and scary, less like a wall and more like a starting point for fixing the error.

Another reflection on entry points and gatekeeping comes from the English artist and writer James Bridle. Their practice explores the cultural and environmental impacts of digital computation, walking and jamming the fine line between what is shown and what is kept hidden in the technological landscape we live in.

When you open the browser inspector on the Facebook website, you are confronted with a wall. A message printed in the console prevents users from accessing the page's hidden structure. The platform adopted this approach to prevent scams and self-XSS attacks on its users, who could have been tricked by malevolent people into running malicious code in their own browsers. However, instead of encouraging its user base to understand, explore and ultimately feel safer against these cyber-attacks, the company opted for a full stop, marking a clear line between users and developers.

`welcome.js` (2016) is a small gesture in response, a tiny javascript library published open source on GitHub under a permissive MIT license, where Bridle injects some greetings into their website (and in all the websites that include the library) to welcome users to the browser inspector. The artwork is hidden below the surface of the website, printed in the console of the browser inspector, a tool that allows users to see the underlying code of the website they are visiting. From here it provides some guidance for newcomers to access, inspect and modify the source code of web pages. A process that opens doors and lets people in, giving them more agency by demistifying technology.

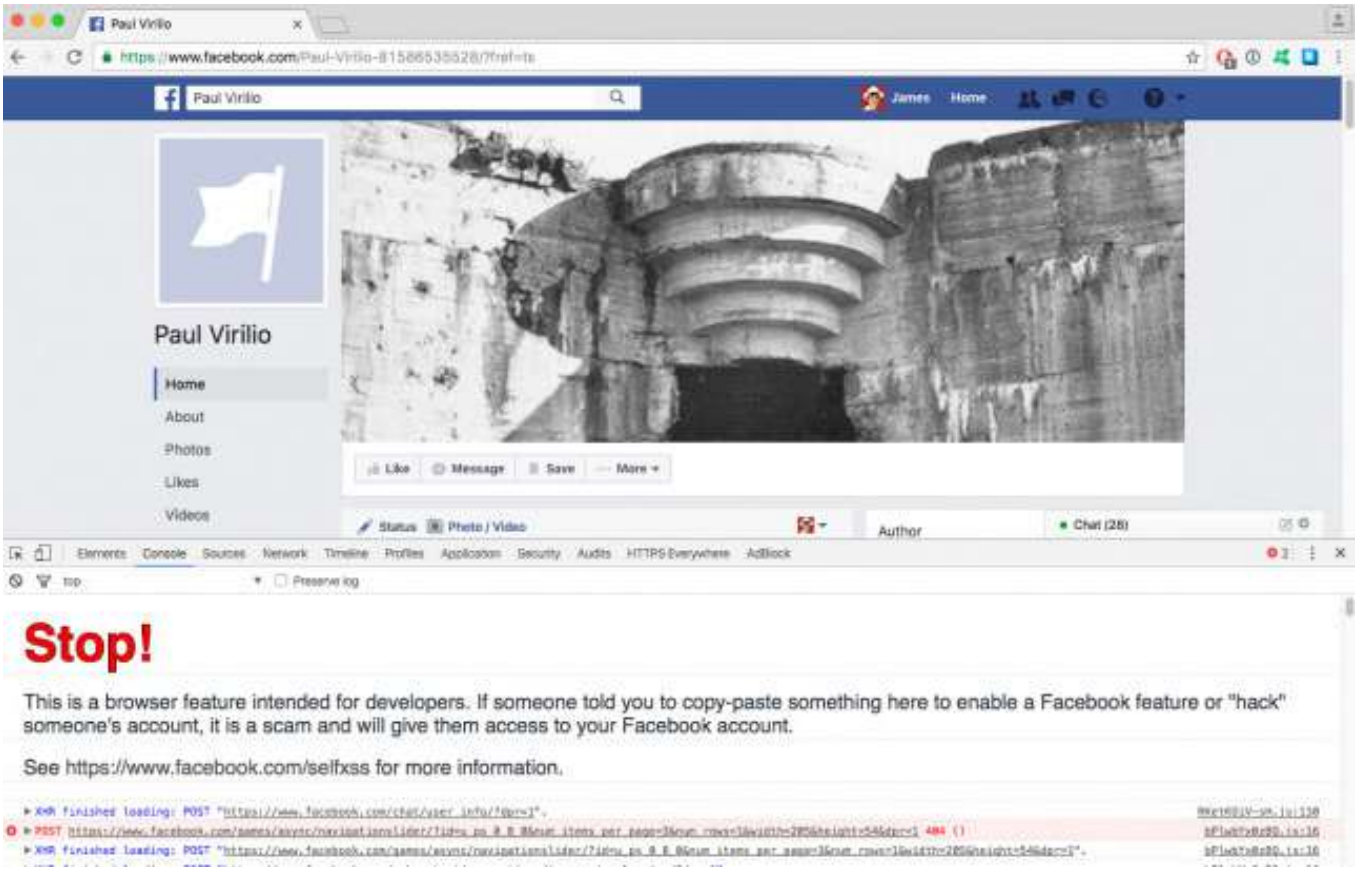


Figure 6.Message in console printed by Facebook to stop users - source: booktwo.org

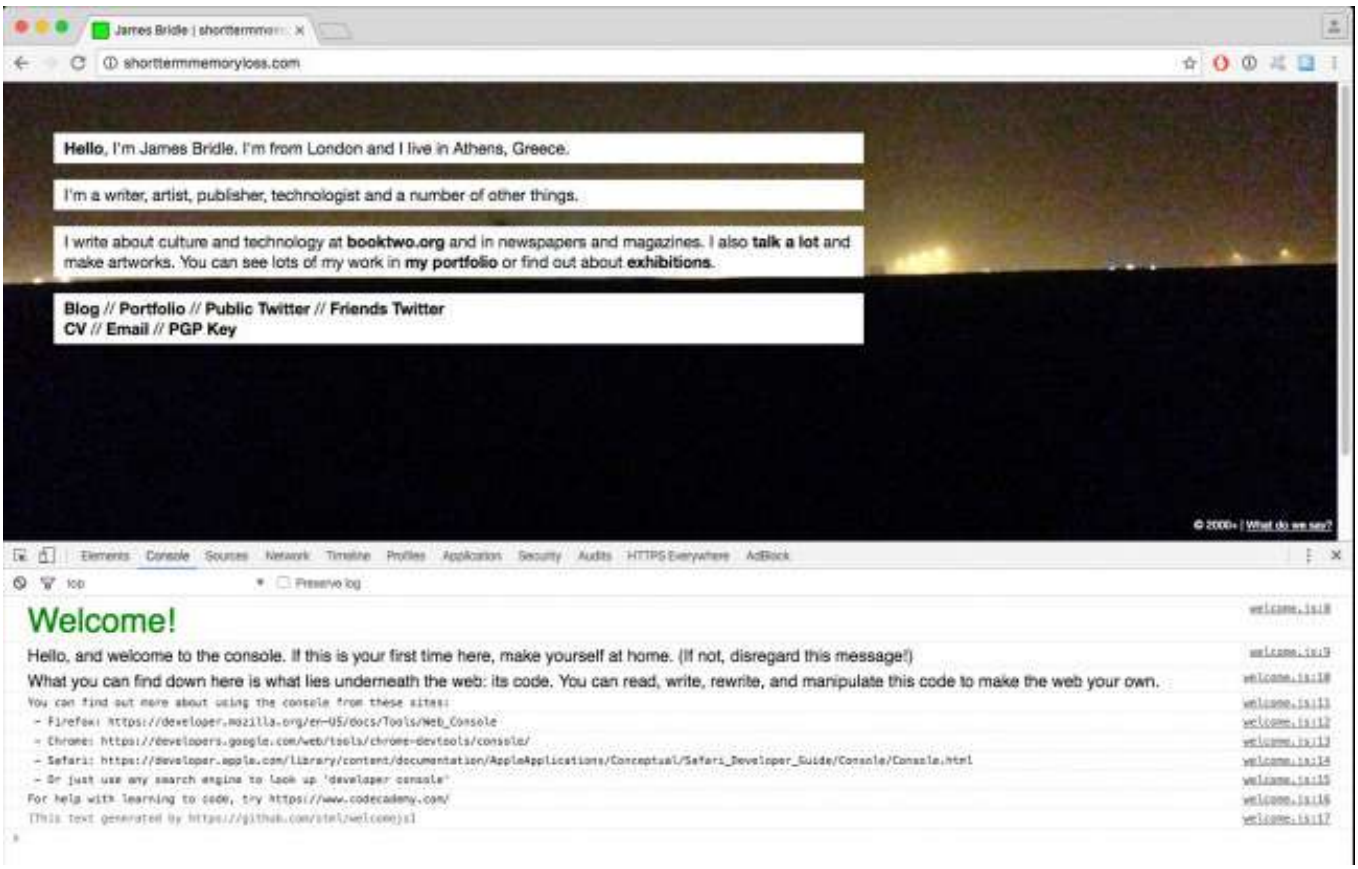


Figure 7. Message in console printed by Bridle to welcome users - source: booktwo.org

Whether in a large project or a small gesture, attention to language can be transformative. In code documentation it can help deconstruct the false dichotomy between programmer and user, or pro and *newbie*. It can create spaces that feel safer, where people are invited to participate, express themselves and contribute to the community. It can help undo the impostor syndrome that affects many programmers, and that feeds on some hidden and inaccessible fundational knowledge that is nowhere to be found in code documentation. It can help shed some light on the massive amount of work that goes into the making of software: recognising all contributions, not just those of engineers.

Documentation as gardening

If the docs does not reflect the behaviour of the project, or if there are discrepancies between the two of them, the reliability of both code and documentation is undermined. Code documentation requires as much maintenance as the code itself. Code transforms and documentation should follow.

There's a multitude of ways in which changes to the codebase affect the documentation. New features require new sections. Breaking changes with previous versions require warnings and instructions on how to migrate to the new one. Bugs and unexpected behaviour need to be addressed. Deprecated functions must to be trimmed out, or marked as outdated.

In addition to the technical aspects, the editorial work needs to be taken into account. Adjustments and corrections and line-editing, clarifications of convoluted paragraphs, rephrasing of confused sentences, highlighting of important passages. Some projects support internationalization, and the contents are translated and adapted to different languages' structures.

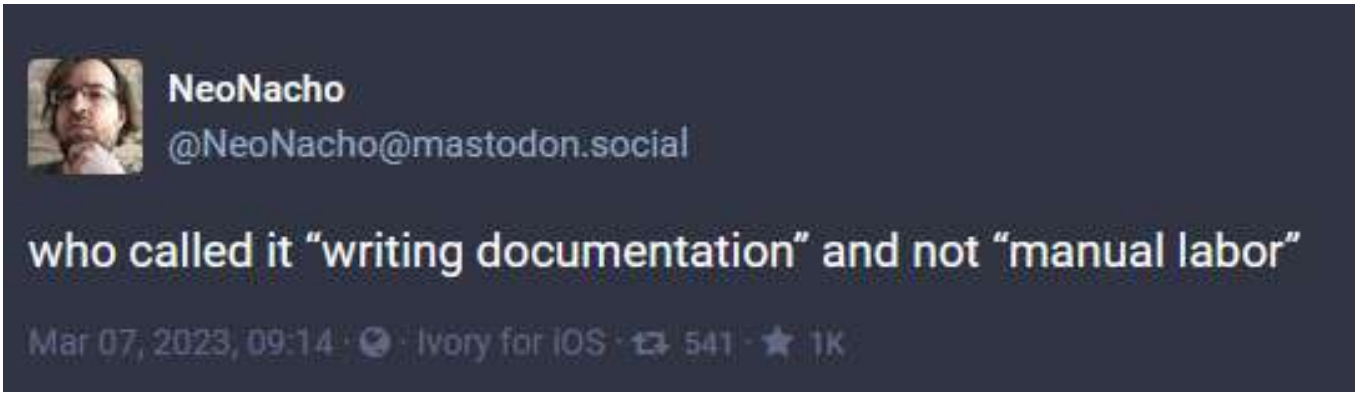


Figure 8. Frustrated developer apparently busy with technical writing

If a project is larger and more articulated, the need for more comprehensive and structured formats arises. Wikis, websites, forums: all these platforms imply more work: maintenance, design and sometimes even guidelines and documentation for the documentation itself.

Writing docs is not a once in a lifetime effort, but an ongoing commitment. It's a process with its own pace and timing, and much like gardening, it's a form of care both for the code and for the community around it.

It's a process that requires a massive amount of energies and resources. Yet, it seems to be constantly underestimated, undervalued, and pushed to the margins. Something left for when there's nothing better to do, something to delegate. Something perceived as a burden, as a killjoy, a display of weakness by *real programmers* who should be able to understand a program by reading its sourcecode.

All these efforts are a good illustration of what advocated in the Post-Meritocracy Manifesto by Coraline Ada Ehmke and more than other six hundred signatories: making software is not just a matter of technical skills, but of interpersonal relations and social dynamics, where all contributions around code are important as those one on the code itself.

Documentation is a surface where all the sociality, relationships, and context around code are rendered visibile. An interface between the technical world of machines, the affective sphere of the community, the delicate and demanding economies of open source projects, and the politics of distribution, circulation and participation in the making of software.

A surface that in turn can be activated and used as a platform to reach out to all the different actors around it.

Getting startled

Reading undocumented code feels like being an ant walking on a big painting. You can see the brush strokes and have a sense of their direction, but what's missing is an overall idea of how the composition flows. Documentation provides a bird's eye perspective on the bundle of functions and statements that make up software. It is often the first thing one gets across when approaching a new library or programming language, and it shapes the way a developer thinks about a particular piece of code.

At the very first encounter with a new script, details about its source code are unknown. Programming is a play *in medias res*, and documentation acts as narrator. By describing how functions are stitched together, or an algorithm is implemented, it sets the stage for developers to participate. By showing the different steps of a program and how they are connected, it offers entry points for intervention.

For example Vue.js, a popular library for building web user interfaces, uses a diagram to explain the lifecycle of its components: when data is received from the server, when an element is rendered on screen, and when it disappears. What at first feels like magic, gradually becomes clearer. To present a structure means also to offer a way of reasoning about it. The reader gains a certain understanding and agency over the tools they are about to use.

Code documentation is not just pure introspection. Consider the diagram created for the *Padliography*, a bookmarking system for collecting links of otherwise scattered *Etherpad* documents. It not only describes what's going on in the code, but also taps into its surroundings: the *Soupboat* server and the *XPUB and Lens Based wiki*.

The introduction to a program situates it within a larger ecosystem: how to install it, and what dependencies it requires to work properly. As Geoff Cox and Winnie Soon elaborate on their decision of a downloadable code editor rather than a web-based one for their classes, code is more than just a single piece of software. It is also the relations with the configuration of one's own computer and operating system. (Cox and Soon, 2020)

Their book *Aesthetic Programming - A Handbook of Software Studies*, is an example of how documentation can be a loom for weaving together technical and critical thinking. The book explains basic concepts of programming, starting from variables and loops, and moving on to more complex topics such as machine learning and speech recognition. The technical curriculum on offer is in line with other similar resources aimed at beginners. What's different here is the commitment to critically enquiry themes such as colonialism, racism, gender and sexuality, capitalism and class, and how are they embedded in code. Soon and Cox prepared these lessons for students enrolled in a design institution, and curated the publication for a public familiar with software studies discourses. Thanks to the vantage point of writing documentation for beginners, they could be super-explicit and go all out with a generous amount of references.

For hatching programmers, the initial imprinting of documentation is a powerful tool to orientate code in the world.

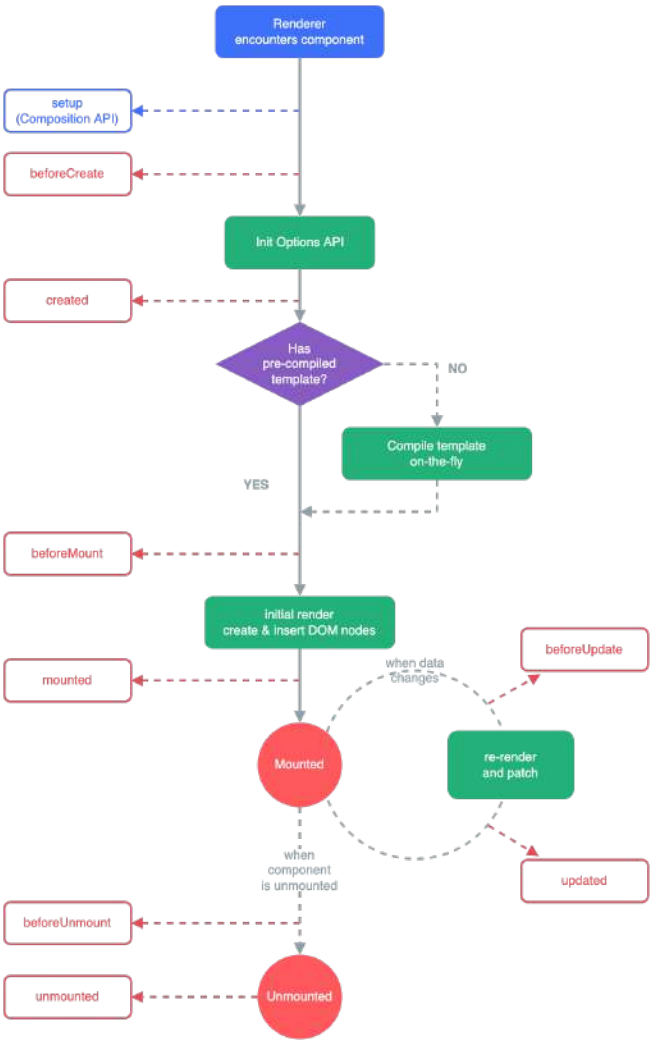


Figure 9. Diagram of a Vue instance lifecycle, illustrating the different entry points of the template design pattern.

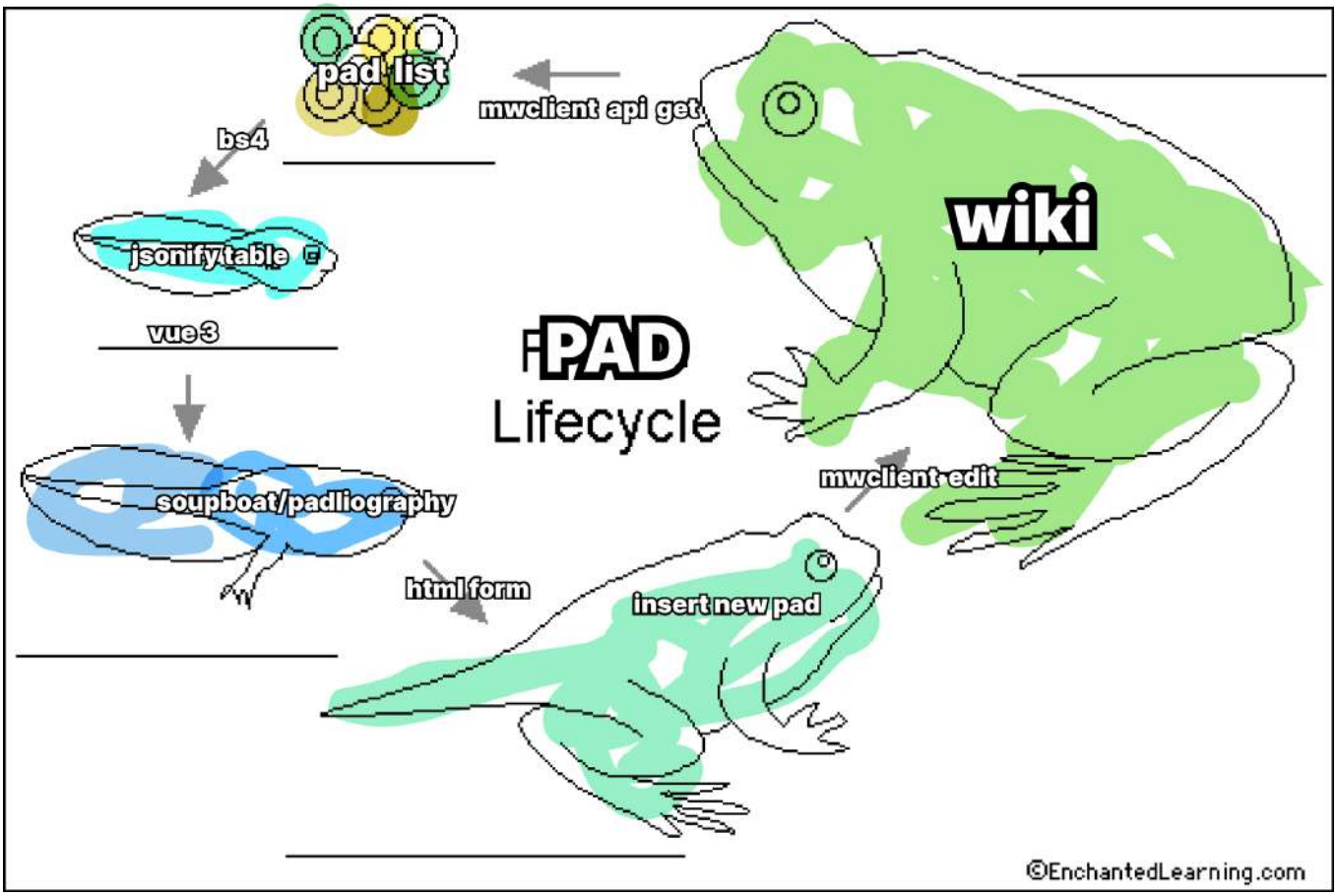


Figure 10. Lifecycle and ecosystem of Padliography: a wiki-powered link bookmarking system.

A code companion

The devil is in the details, and software as well: the translation between human and machine has to be negotiated with all the specifics of a particular programming language or platform. Sometimes for the web, sometimes for a hardware component, sometimes for another operating system. These *specs* make every piece of code a bit alien and peculiar. Tinkering with code is not just about knowing a programming language by heart, but rather having to deal with a lot of different recipes for different occasions, and know how to adapt.

Documentation is not just for beginners: it's a code companion. You never stop reading it. Even experienced programmers must consult the docs when first encountering a software, and return to it when they need a refresher on the syntax of a particular command. They are continuously looking at code from multiple distances: close to the source code through lines of comment, or from printed books, along with pages of explanations and use cases.

This tentacular surface can reach a programmer at different moments of their life: from the *hello world* to the *how to uninstall*. This is possible thanks to the variety of forms that documentation can take: video tutorials and commands cheatsheets, *README* files and complete guides featuring diagrams and

drawings. Daniele Procida proposes a systematic approach to organise this wealth of formats (2017). His framework focuses on the needs of different types of readers: by leveraging between practical steps and theoretical knowledge, he charts four main modes of technical writing. Each format has its own approach and intentions, and answers different questions.

A text that doesn't consider who's reading it can result inaccessible and frustrating. Although the Diataxis framework doesn't encompass every particular situation, its structure is a good aid to situate documentation within different perspectives. This turns out to be very helpful in the writing process, as a way of fine-tuning tones, and modulating the nature of shared information. *Tutorials* open entry points for the newcomers, while *explanations* unveal core mechanisms for more navigated readers. *How-to guides* teach how to get the work done, while *references* report lists of information ready to be consulted. Different documentations for different readers for the same code.

Or rather, the same documentation, for the same reader, for the same code, just at different moments in their life. Programmers' needs change over time, as do the answers they are looking for, but still, they keep returning to read the docs.

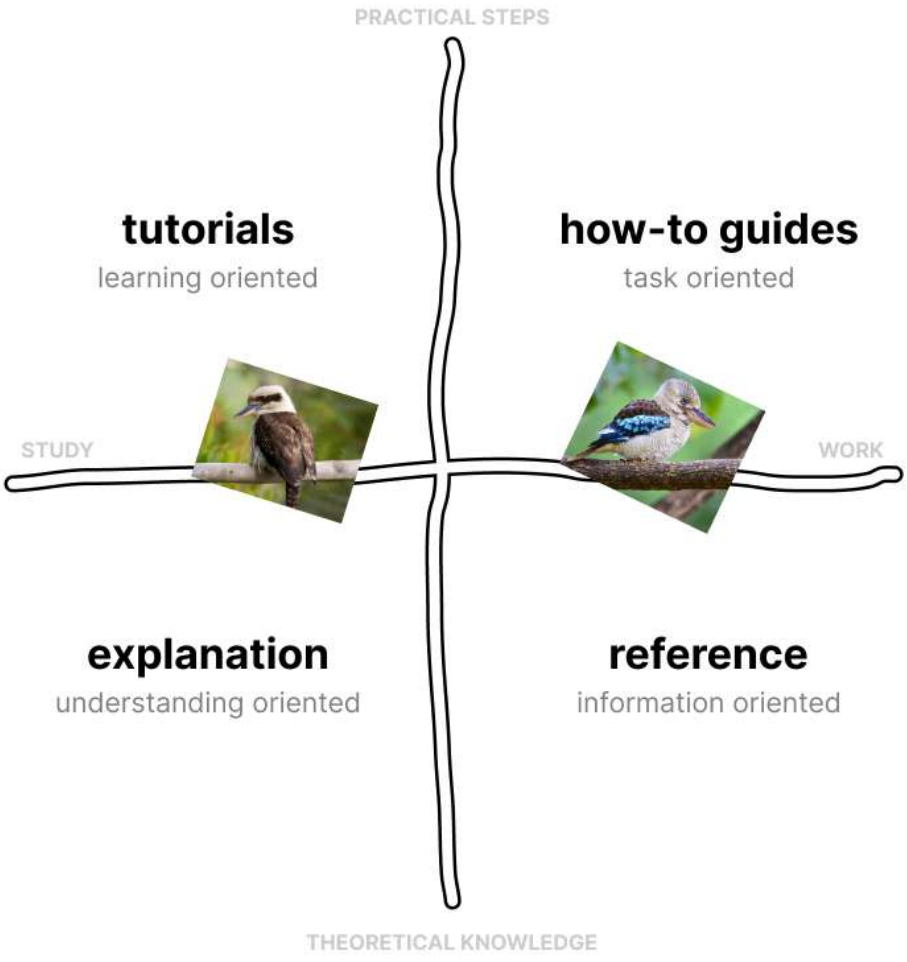


Figure 11. Diagram with the Diataxis framework

2. Documentation as a backdoor

Coding Contingencies

How do you choose a particular programming language, a coding paradigm, a development environment, an infrastructure where to run the code, and so on? These are not just technical choices, but rather coding contingencies.

It may depend on the IT curriculum in a public school, on the job requirements for working in a tech company, because of an Arduino board got as gift for birthday, or a colleague who is passionate about experimental music and drags you to a live coding concert.

These contingencies are situated in specific contexts.

Programming then is not just sharing code, but sharing context. A significant statement about our relationship to the world, and how we organise our understanding of it. A perspective for looking at reality, before attempting to get some grip on it with a script. A way of dealing with both the software and hardware circumstances of code, but also engaging with the sociality and communities around it.

It's an approach that helps us to think about software as a cultural object. Something "deeply woven into contemporary life –economically, culturally, creatively, politically– in manners both obvious and nearly invisible." (Fuller, Manovich and Wardrip-Fruin, 2009), and not just as technical tool existing in a vacuum.

Code as an object that, in turn, can be used to probe its surroundings. Who is developing? Who is going to use it? Who pays for it and why? How is it structured? Is it a big and centralized system, or a loose collection of small and interchangeable tools? How long is it supposed to last? How can it be fixed if it breaks? The main focus of this chapter is to explore software documentation as a surface where these kinds of questions can be addressed. A place where the complexity of code doesn't blackbox ideas, and choices behind development can really be open source.

A way to situate programming in specific contexts, but also to inject our contexts into programming practices. Hence the idea of code documentation as a backdoor: a passage to infiltrate software culture, to change things from the inside and create more entry points.

Documentation & distribution

Rethinking participation in coding practices from femminist perspectives does not mean simply swapping who can join and who cannot. This would only reproduce current forms of exclusion and

polarisation. It also doesn't mean committing to an overexert openness, accepting everything and everyone, and potentially endangering safe spaces.

The *Wishlist for Trans*femminist Servers* engages with a more messy, entangled, complex way of understanding participation and technology. A way to open up to plurality, to questioning, to instability, to consent, to situatedness. Iterating from the *Feminist Server Manifesto*, it offers prompts to embrace coding within contradictions: not as a moral setback, but rather as an ongoing labor, striving for a different tech for this world, and for different worlds.

These principles are reflected in the documentation of the Queer Motto API, a *software as a service* commissioned by transmediale in 2020-2021 and developed by the Queer Service team (Winnie Soon, Helen V. Pritchard, Cristina Cochior, and Nynne Lucca). The project challenges the idea of software as a smooth, always-on service, with a motto generator that sometimes refuses to work, takes a nap when it needs to REST, or goes on strike to celebrate important days like the 8th March.

Note

In their documentation *REST* is used as a word play with the acronym *representational state transfer*, typical design pattern of API development.

The Queer Motto API is published in the form of an Application Programming Interface (API), an online service that other developers can request from their applications to use generated feminist motto. By being released as an API, the service is inherently linked with other projects, such as the Transmediale website, that uses it to display a new motto every day. Who wants to use the API has to agree to the terms and conditions, which are detailed in the documentation available in the project repository. The *readme* offers an understanding of the various technical moments and aspects involved in interacting with a typical software-as-a-service, but narrates them from a feminist perspective. Error codes, service availability, consent and refusal, request and response, token policy, and all the terms neutralised by the normativity of everyday tech, are reactivated here as powerful narrative (and subversive) devices.

One example is the paradigm of the constant availability of the server. Behind every *SaaS* there are always a server: the so-called *someone else's computer* working behind the scenes. The seamless cloud picture of big tech rarely includes these machines, which are abstracted and hidden from the user. Instead, in the Queer Motto API, the presence of the server is a key aspect, especially when it decides to take a nap or refuses to work because it is on strike. These behaviors are documented with various error codes, giving developers using the API a way to make their applications react accordingly, and even join the cause.

Figure 12. Response from the API with refusal message.

Representation specs

Alt-text as poetry is a project by Bojana Coklyat and Shannon Finnegan: a workbook dedicated to the alternative text descriptions that make web images accessible to people who are blind, with a low vision condition, or have other cognitive disabilities.

Websites are made of HTML, a markup language based on tags. Each tag represents an element of the document: a header, a paragraph, a link to another page, an image, and so on. As in a sandwich, these tags can be composed together, and organized to structure the contents of a web page. Every tag comes with particular attributes, and the image `<img>` one requires the developer to specify the source `src` of the picture to display. Here is also possible, but not technically mandatory, to add an `alt` attribute, with the alternative description of the image used by screen readers and other assistive technologies.

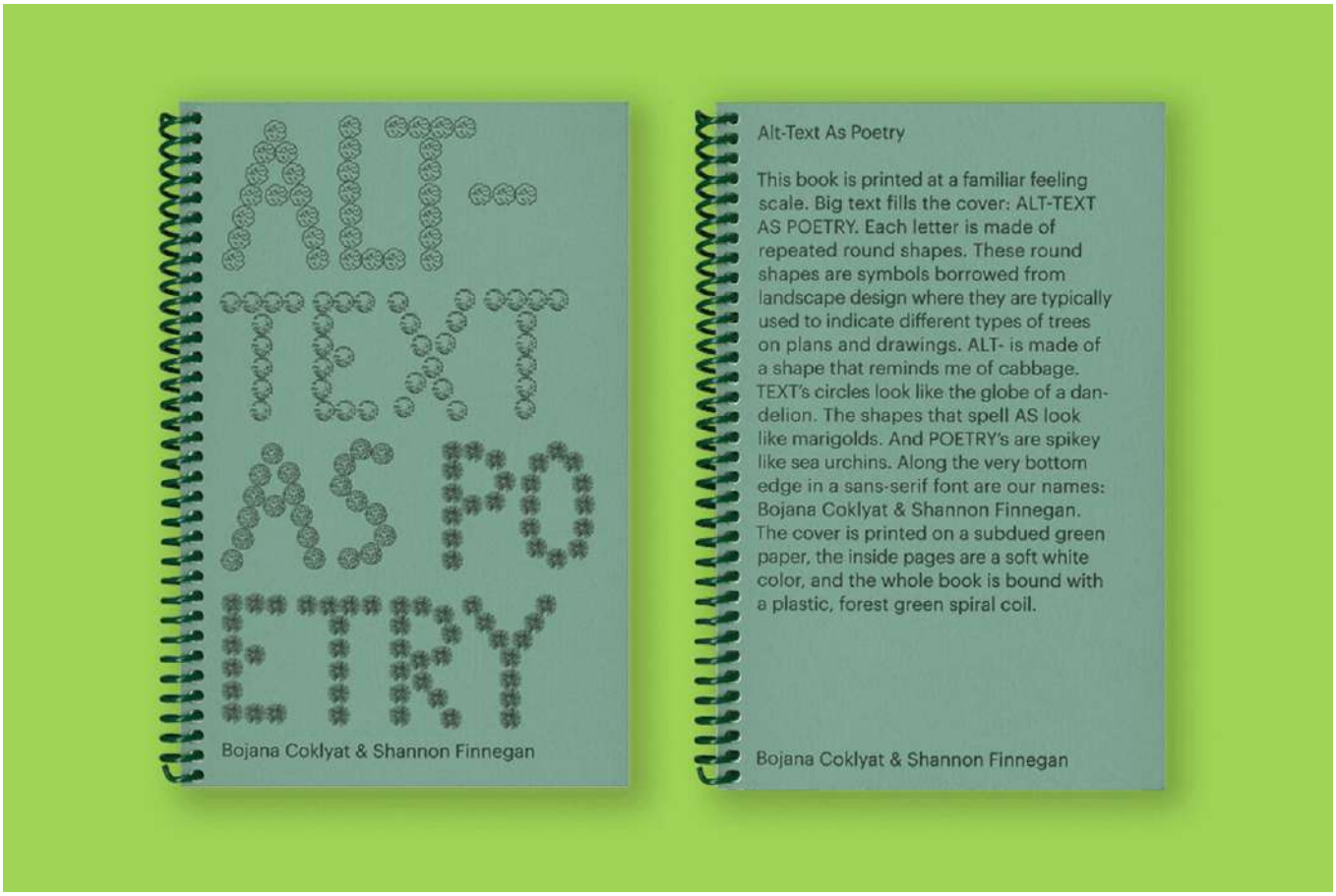


Figure 13. Alt-text as poetry workbook, and relative image tag with alt description

```
<img src="img/workbook.jpg" alt="Big text fills the cover: ALT-TEXT AS POETRY. Each
letter is made of repeated round shapes. The cover is printed on a subdued green
paper, the inside pages are a soft white color, and the whole book is bound with a
plastic, forest green spiral coil.">
```

Alt-text is an essential part of web accessibility. It has existed since the 1990s, but it is often disregarded or understood through the lens of compliance, as an unwelcome burden to be met with minimum effort. By design, the HTML specifications treat it as optional. While omitting the source `src` of an image will preclude it from being displayed, the same is not true for the absence of an `alt` text.

The Cocklyat and Finnegan workbook is an entire piece of code documentation dedicated to a single HTML attribute. It re-frames `alt` text as a kind of poetry, and provides exercises to practice writing it. Its intention is not only to enable alt-text users to be able to access visual content on the web, but also to let them feel a sense of belonging in the digital spaces. By highlighting the needs of often marginalised minorities, and giving them proper representation, documentation can activate ways of thinking that actively shape technical implementations, recognising not only the needs of machines, but programmers and users as well.

Documentation and technical implementations influence each other in a feedback loop. The power of code documentation to encourage a particular set of practices molds subsequent implementations, which in turn consolidate and normalise previous choices. Here technical and design choices can create or foreclose spaces for others to participate in programming practice. Of course, this process doesn't happen in a streamlined and linear fashion, but rather as a bouncing and transversal echo that reaches neighbouring contexts and other projects. Inspired by Alt-text as poetry, I decided to write the code for the pages in my Soupboat pages in such a way that images would not be displayed unless they were accompained by a text description. An implementation aimed at slowly training and sensitising myself.

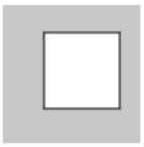
```
{% if cover and cover_alt %}
<figure class="header--cover">
    <img src="{{url_for('static', filename='img/' + cover)}}" alt="{{cover_alt}}" />
</figure>
{% endif %}
```

Code for the project pages on the Soupboat. The figure is added if and only if both cover and alt description are provided.

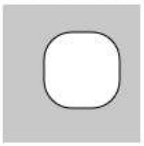
However, an implementation-first approach is not always an option, and code documentation is a more expressive surface to work with. The `p5.js` library, for instance, exposes a `describe()` and `describeElement()` function, to provide a description analogue to the `alt` text one for your visual sketches. The interactive graphics are based on the HTML `canvas` element, which work on a pixel basis rather than semantically like HTML. Like images, this content is not compatible with screen readers, and requires textual explanation to make what's happening on the display accessible. Even more: while images are usually static, `p5.js` visuals are often in motion, evolving over time. With `describeElement()`, developers can be even more granular in their descriptions, captioning the transformations of different elements in their animations.

In the discussions surrounding the development of this open source project, contributors began to consider how to encourage the use of this feature. From an initial suggestion to make it a requirement for Sketch to run, opinions settled on conveying its importance from the documentation, by adding it to the default template, and to the examples in the documentation and tutorial.

Examples



```
// Draw a rectangle at location (30, 20) with a width and height of 55.
rect(30, 20, 55, 55);
describe('white rect with black outline in mid-right of canvas');
```



```
// Draw a rectangle with rounded corners, each having a radius of 20.
rect(30, 20, 55, 55, 20);
describe(
  'white rect with black outline and round edges in mid-right of canvas'
);
```

Figure 14. Examples from p5.js documentation.

Situated docs

Injecting context in software requires operating at different scales. Within both public and private dimensions, within technical and social frameworks. In a workshop for example, people meet face to face. Here, togetherness can glue together technicalities, questioning the reproduction of knowledge and its power dynamics.

Code documentation is trasmission of knowledge, traditionally conceived as a vertical and centralised practice, where who teacheas and who learns are on diametrically opposite sides of the spectrum, in well-defined roles. From this perspective only the *real programmer*, the expert that detains a phantomatical *foundational knowledge*, is allowed to share wisdom and document code. As argued by Kit Kuksenok during the activation of their workshop *Sharing Programming Knowledge* at Varia (Rotterdam), things are more fluid than that: everyone is sometimes a learner, and sometimes a teacher. Each role brings valuable insights to the counterpart, and taking them into account open the way to other pedagogical and organisational tactics for sharing knowledge. One example are collective learning moments, situations where code documentation is both active practice and shared, horizontal surface.

360 degrees of proximities is a project emerging from a network of feminist servers that addresses the problem of invisible labour typically associated with the maintenance of digital infrastructure.

After sucessfully setting up of a self-hosted Peertube instance, a federated platform for sharing video content, the group began to question aspects of maintaining the system. Rather than centralising the service in a self-exploitative scenario, they decided to redistribute responsability across the network, working with other feminist and queer communities and empowering them to build their own video

platforms autonomously, but in a joint effort. This is where different knowledges meet: on one hand the know-how about installation and configuration of Peertube brought by the *360* team, and on the other site-specific knowledge of the hosting server.

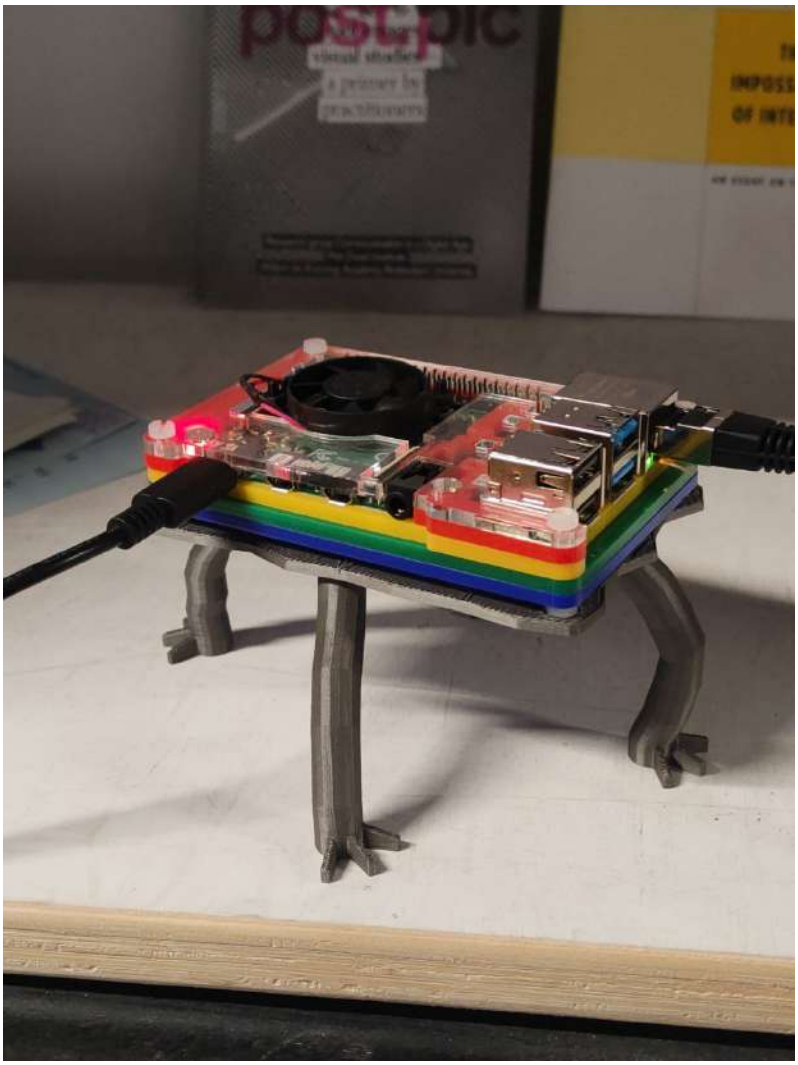


Figure 15. Soupboat, the server of our XPUB group. With 3d printed legs to run faster.

From the perspective of hosting others into "our" code, documentation becomes a form of hospitality. A form of care for a shared space. In XPUB, each group begins its two-years programme by setting up a self-hosted server. We called ours Soupboat, and it houses many of our prototypes and experiments. This small server, running on a Raspberri Pi, feels like a place to call home on the internet. Over the past two years we have done all sorts of projects there: generated web-to-print publications, custom CMSs to manage birthdays, Etherpad documents, and soup recipes, workshops, personal wikis, and so on. While living on a server with others, my approach to code began slowly to change. Publishing open git repositories, instead of hiding behind private ones. Writing more readme files to be more generous with friends and colleagues and tutors. Cultivating small gestures and rituals, like leaving comments in config files to remind others where to mount the next app or where to find some credentials.

```
# labsong - songs for difficult grad labor

location ^~ /labsong/ {
   proxy_pass http://localhost:3157/soupboat/labsong/;
   include proxy_params;
}

# ATTENTION: use port 3158 for the next project!!!! ;))
```

NGINX configuration with gentle, incremental reminder on which port to use for the next app

At the same time, this awareness grew by acknowledging the particular context of this small setup, of this *situated software* (Shirky, 2004). In many readme for example, such as the one for the networked drawing app drw or the padliography, the explanations are tailored specifically for the Soupboat, and they cannot probably be ported 1:1 somewhere else. Nonetheless, the space is prepared for hosting new guests.

```
Eventually you want to put online your drawing app.

To be able to use this app on the Soupboat (or other servers connected in the
hub.xpub.nl ecosystem) some additional configurations are needed.

Note that the following details are tailored to the particular case of our server.
Other instances could require different setups.

This is one possible workflow.

Clone the repo and install the requirements as you would do locally.

git clone https://git.xpub.nl/kamo/drw
cd drw
npm install
```

Notes from drw's readme.

This is where the interesting friction of situated documentation arises: how to share knowledge about deeply situated programming practices with other contexts? How to remain legible and accessible, for ourselves and for others, while at the same time preserving specific and characteristic decisions? Usually documentation doesn't take into account the messiness of coding contingencies, where multiple software coexist on the same server and configurations conflict or are installed with different setups. Collective learning moments and small, shared rituals can bridge the gap between the default setup described in documentation and a real-world, situated one.

Hello worlding

The Screenless Office is an *artistic operating system* designed by Brendan Howell and Mikhail Pogorzhelskiy to reimagine personal computing away from pixel-based displays, using radically alternative forms of everyday human interaction with media.

Similar to other operating systems, the Screenless Office can be used to read news, browse websites and interact with social media. What's different here is that the surface on which all these exchanges take place is not a screen, but rather an articulated ecosystem of thermal and laser printers, barcode scanners and other interconnected physical devices that can be plugged in as required.

The first interaction with the Office prints out a menu of available commands, in the form of a list of functions with associated description and barcode. From here you can scan the barcode to read the news, for example, and get web-to-printed a feed with the latest stories. Each item in the feed is a snippet of an article scraped from several online sources, and can be scanned again to print the full version.

In an interface culture dominated by few corporate players and crystallized on touchscreen glass, the project offers multiple gestures for interaction, as opposed to the single act of scrolling. Furthermore, instead of sitting in front of a screen with a singular, centered, and linear perspective, the user displaces the office all around through printers, scanners and printed materials. Here code documentation plays a key role in orchestrating all these different interactions.

The system is written in Python, and its code documentation consists mainly of *docstrings* and comments written directly in the source.

A *docstring* is a piece of text written at the very beginning of a function to document it. Unlike normal comments, which are usually removed from code at runtime, *docstrings* are preserved, and can be consulted with interactive help systems or used as metadata. Many programming languages support this pattern, which is often used to produce automatic pieces of documentation simply by collecting and listing all the functions and their descriptions.

In the project these are used to create a world around the code and its structure: in the initial menu, for example, they are the settings for the Screenless Office. Reading them, we discover how the office is organised into different *bureau*, each dedicated to a specific task. The *Pubblications Office* deals with the daily news and the weather forecast, the *Public Relations* department manages exchanges with social platforms, the *Audio Service Dept.* provides the soundtrack, and so on, while the *Inhuman Resources bureau* keeps track of them all.

```
class Humor(Bureau):
    """
    This bureau entertains the modern worker and provides colorful
    bons mots for managers who need to warm up an audience.
    """

    name = "Department of Humor"
    prefix = "HA"
    version = 0

    def __init__(self):
        Bureau.__init__(self)

    @add_command("joke", "Fortune Cookie")
    def print_fortune(self):
        """
        Prints a clever quip.
        """
        jux = str(subprocess.check_output("/usr/games/fortune"))
        self.print_small(jux)
```

Sample from `jokes.py`, the module of the *Department of Humor*. Here two *docstrings* describe the bureau itself and the `Fortune Cookie` command.

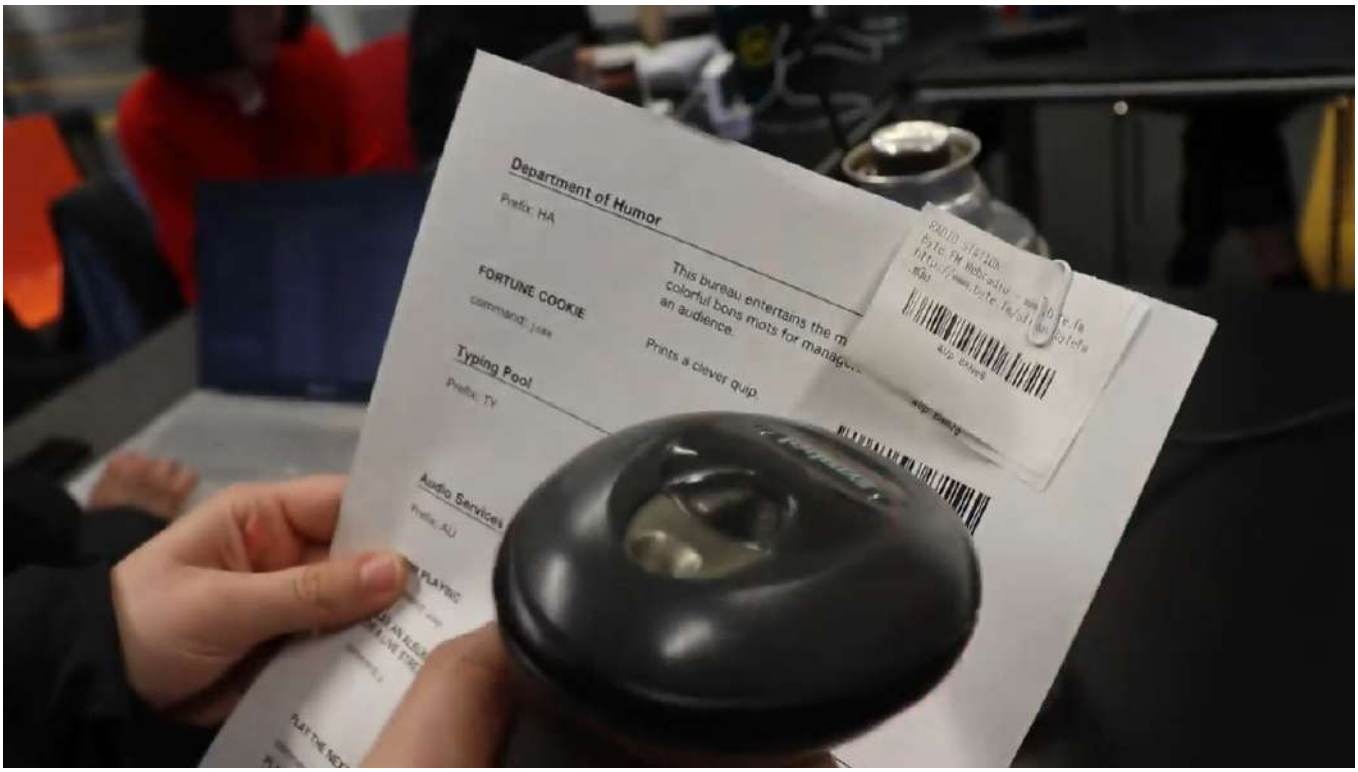


Figure 16. Menu printed by the Inhuman Resources bureau using the docstrings from the other offices. 4-4-2022 Workshop with Brendan Howell at XPUB

In the essay *Chimeric Worlding*, researcher and designer Tiger Dingsun explores what graphic design can learn from poetics to escape a condition of pure functionalism. Graphic design and code documentation are similar: both deal with the organizazion and presentation of information, making meaning through the configuration of different elements, which are not just limited to language and text, but can also include images, symbols, (code snippets, examples). With a find&replace to swap all occurences of `graphic design` to `code documentation`, Dingsun's essay can be versioned to get an interesting perspective on what's happening in the Screenless Office.

In the essay, they highlight how poetry often provides a rich context and a world for a work to live in, while *software documentation* often does not. A poetic practice of world-building would benefit *code documentation* allowing for multiple potential narratives to sprawl out nonlinearly, validating them, and inviting questioning code's surroundings. By doing so, it would offer points of resistance to the smooth flow of capital, that relies on a singular, totalizing interpretation of the world. Hence the idea of chimeric worlding: to provincialise *code documentation* with multiple and situated ways of structuring knowledge, leaving open ended spaces for others to participate.

In the Screenless Office, the bureau aesthetic, with its cast of characters, situations, and power dynamics, becomes a personal interaction design framework. Here the system is structured enough to articulate a complex application in a coherent, clear and legible way. And yet, the cosmology of the office remains open to contributions coming from elsewhere, for example the addition of another departement such as the `Canteen of the Screenless Office` inaugurated during a workshop with Howell at XPUB, with our own peculiar set of characters, aesthetics and documentation practices.

Distributed autorship

Learning How to Walk while Catwalking is a collective project we developed in the context of Special Issue 16. It is published as an API that provides a toolkit to explore natural language processing in a vernacular way. It makes available several *endpoints* to experiment with text transformations in a playful way, from simple operations like repeating or filtering certain words from a string, to more articulated functions to annotate images, or use words like *etc* and ... as containers to continue unfinished lists.

Note

ENDPOINT
An endpoint is a location where the API receives requests for specific resources, usually in the form of an URL. An example of endpoint is: `https://hub.xpub.nl/soupboat/si16/api/repeat/?text=hello&times=3`, that call the `repeat` function in the server, passing a text to repeat `hello` and the amount of repetitions `3`

Tipically an API's architecture is centralised: there's a grand scheme and everything has to fit into it, both code- and documentation-wise. Documentation guidelines such as the Diataxis framework, recommend maintaining a consistent tone and offering a single source of truth for navigating the codebase. These prompts are certainly helpful in preserving legibility, but little they reveal about the inherent distributed autorship of code.

SI16 has been a space for undoing the grand scheme and let a plural, vernacular autorship to emerge. On the server side the API is structured as a filesystem, and to insert a new function it's enough to upload a Jupiter Notebook file containing the script and its documentation. Jupiter Notebooks are interactive documents in which code snippets and their explanations can be interwoven. They come handy for prototyping and documenting learning processes, writing code to be read not only by computers, but also by other programmers, in a paradigm also known as *literate programming*, introduced by Donald Knuth in 1984.

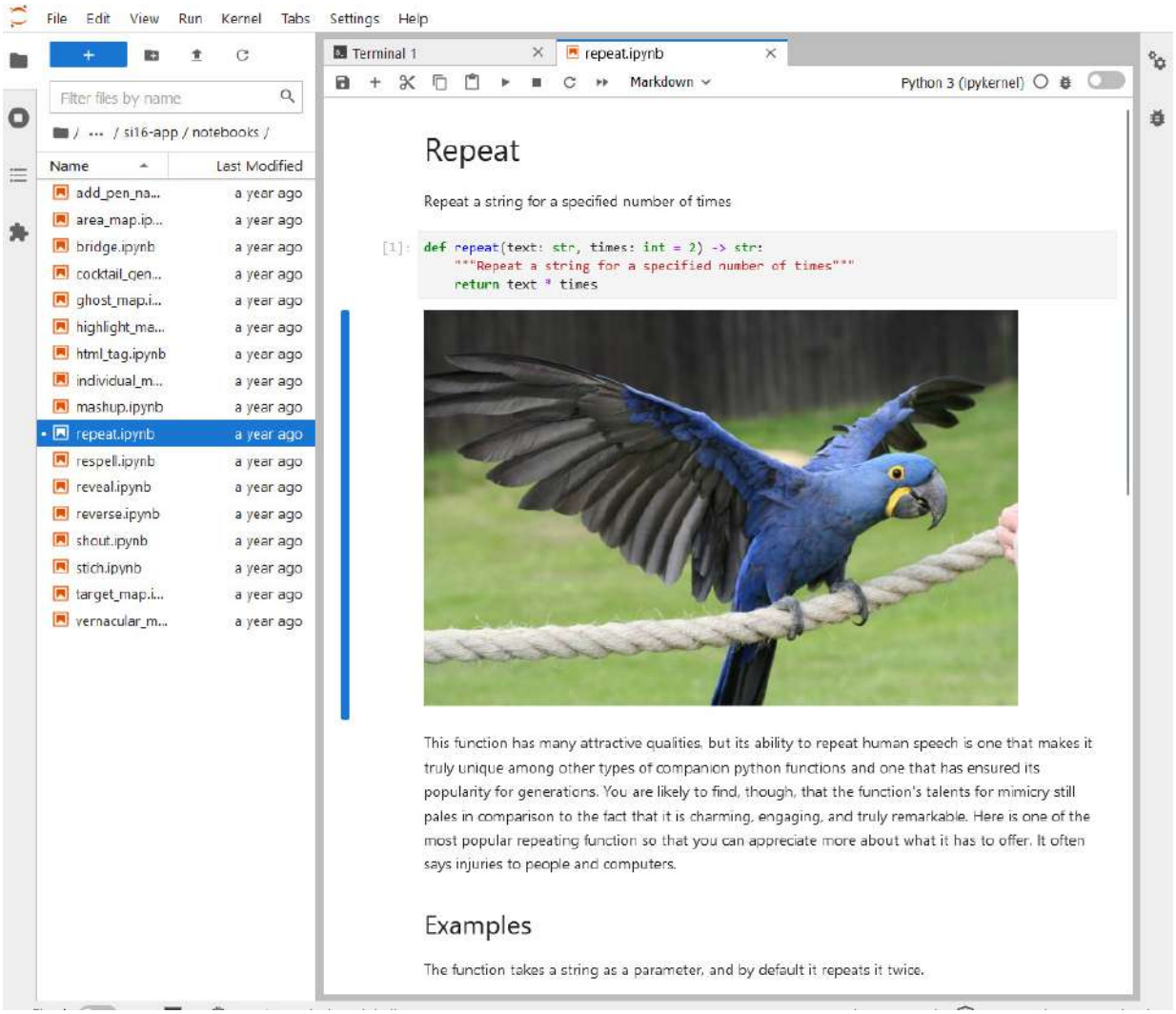


Figure 17. Soupboat server. Jupyter Notebook of the repeat function. On the left panel the filesystem with the other function files.

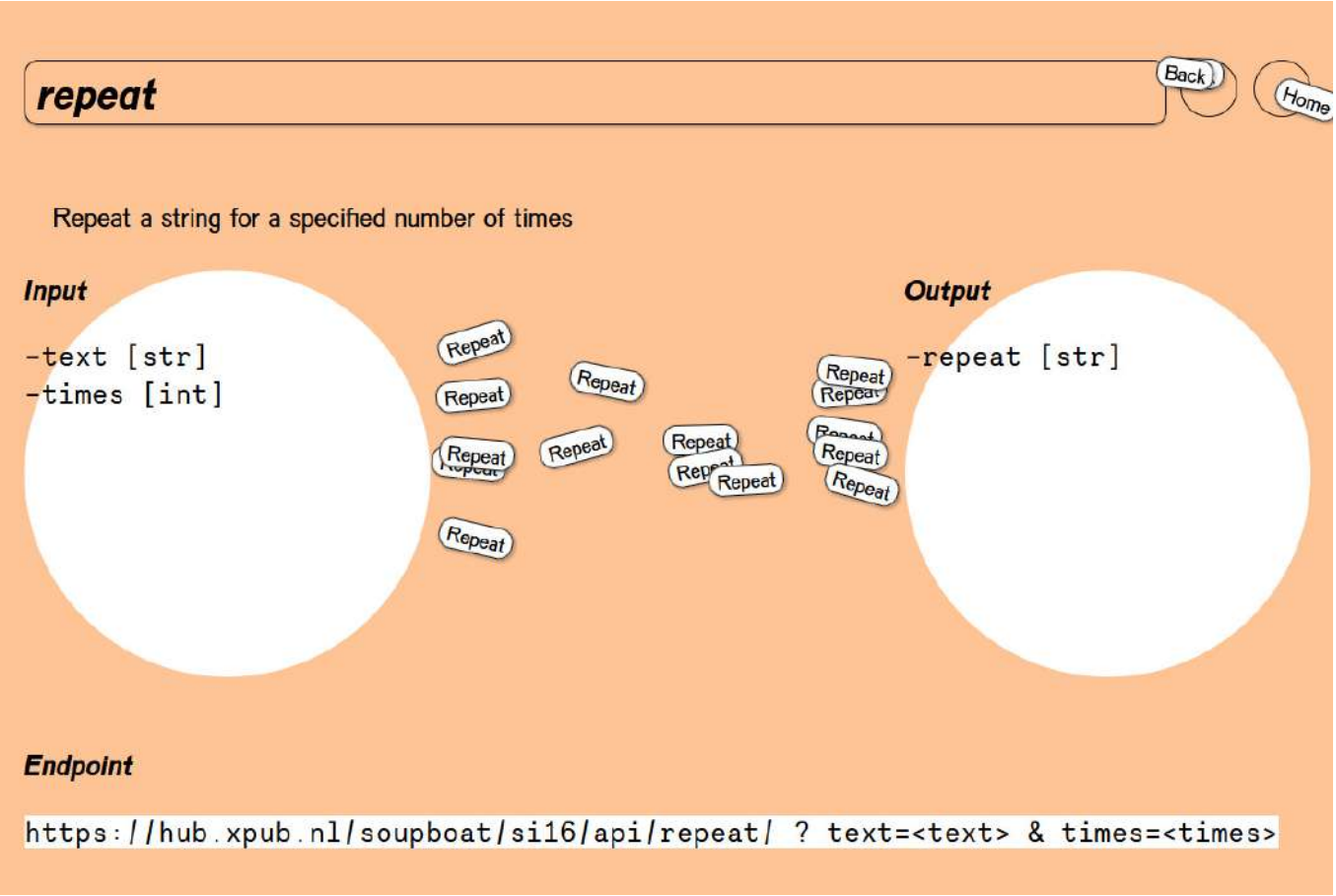


Figure 18. Documentation generated from the Notebook files for the Repeat function

3. Outro

Documenting code is a rich and diverse practice, with a variety of forms and formats suited to specific occasions and needs. These different publishing surfaces are still affected by several problems, such as a general unappealing and unwelcoming tone, dense and gendered language, and a massive amount of energy, resources and time required for maintenance. These critical aspects highlight how problematic the supposed "nature" of code documentation is. A nature that instead of creating entry points, essentially gatekeeps access to programming knowledge.

There is another way in, however. Because of its proximity to the code and its ongoing relationships with programmers, code documentation can be a backdoor into communities gathered around coding, opening up more entry points from within. Code documentation can be used to orient software in the world, operating at different scales and in several ways, working with both technical and social frameworks. It can retrace genealogies to activate exhausted technical terms. It can influence technical implementations by representing the needs of marginalised minorities. It can be a moment of collective learning, challenging traditional reproduction of knowledge, and creating safe spaces for anyone to participate to code.

I started this research for of two reasons: the first is that I love programming. Learning how to code is like learning another language: not just a new bag of words and a different grammar, but a whole new way of thinking, a lens through which to look at the world. Coding means to express ideas with the reduced vocabulary of a programming language. As in poetry, these constraints stimulate creativity, and encourage a diligent yet playful approach. Working with different programming languages and on different systems transforms thinking in multivarious ways, and that is extremely exciting.

The second reason is that I want to share this excitement with others, especially with my friends. To be able to think and make sense together of what's happening around us, and come up with alternatives or responses or tools that suit us better. Because of the steep learning curve of programming and the other barriers mentioned in the thesis, this has often not been possible. But now we know that there are other ways in, and that it is possible to open up even more.

Bibliography

Bridle, J. (2016). *welcome.js. GitHub.* [online] Available at: https://github.com/stml/welcomejs

Bridle, J. (2016). *welcome.js | booktwo.org.* [online] Available at: http://booktwo.org/notebook/welcome-js/

Donald Ervin Knuth (1992). *Literate programming.* Stanford, Calif.: Center For The Study Of Language And Information.

Ehmke, C. A. (2018). *The Post-Meritocracy Manifesto* [online] Available at: https://postmeritocracy.org/

Fisher, M. (2013). *Suffering With a Smile.* [online] Available at: https://theoccupiedtimes.org/?p=11586

Fuller, M., ed. (2008). *Software studies a lexicon.* Cambridge, Massachusetts Mit Press.

Gabriel, R.P. (1998). *Patterns of Software.* Oxford University Press, USA.

GNU (n.d.). *GNU gettext utilities.* [online] Available at: https://www.gnu.org/software/gettext/manual/gettext.html.

Howell, B. (2016). *The Screenless Office.* [online] Available at: http://screenl.es/

Howell, B. (2016). *the-screenless-office GitLab.* [online] Available at: https://gitlab.com/bhowell/the-screenless-office

Karagianni, M. (2022). *Read The Feminist Manual.* Athens, Greece: Psaroskala Zines.

Karagianni, M. et al (2023). *Feminists Federating.* Toward a Minor Tech - A peer reviewed Newspaper vol 12

Kuksenok, K. (2023). *Sharing Programming Knowledge* [workshop]. Rotterdam: Varia / Available & The Rat

Hanlon, J. (2018). *Stack Overflow Isn't Very Welcoming. It's Time for That to Change.* [online] Stack Overflow Blog. Available at: https://stackoverflow.blog/2018/04/26/stack-overflow-isnt-very-welcoming-its-time-for-that-to-change/.

Heaton, R. (2019). *Programming Projects for Advanced Beginners #6: User Logins.* [online] Available at: https://robertheaton.com/2019/08/12/programming-projects-for-advanced-beginners-user-logins/

Marino, M.C. (2020). *Critical code studies.* Editorial: Cambridge, Massachusetts: The Mit Press.

Meta Stack Exchange. (2009). *Should 'Hi', 'thanks', taglines, and salutations be removed from posts?* [online] Available at: https://meta.stackexchange.com/a/93989

nand2tetris. (2017). *nand2tetris.* [online] Available at: https://www.nand2tetris.org/.

Norman Wilson, A. (2011). *Workers Leaving the Googleplex.* [online] Available at: http://www.andrewnormanwilson.com/WorkersGoogleplex.html

p5js.org. (2019). *p5.js.* [online] Available at: https://p5js.org/.

p5js.org. (2015). *Debugging | p5.js.* [online] Available at: https://p5js.org/learn/debugging.html

processing/p5.js (2021). *require describe() function in p5 sketches? · Issue #5427 · processing/p5.js.* [online] Available at: https://github.com/processing/p5.js/issues/5427 [Accessed 10 Apr. 2023].

Procida, D. (2017). *Diátaxis.* [online] Available at: https://diataxis.fr/.

Queer Service team (2021). *Queer Motto API · GitLab.* [online] Available at: https://gitlab.com/siusoon/queer-motto-api

readthefuckingmanual.com. (n.d.). *Read the Fucking Manual.* [online] Available at: http://readthefuckingmanual.com/

Ullman, E. (2017). *Life in code : a personal history of technology.* New York: Mcd, Farrar, Straus And Giroux.

Ullman, E. (1997). *Close to the machine : technophilia and its discontents.* London: Pushkin Press.

Shirky, C. (2004). *Situated Software*. [online] Available at: https://gwern.net/doc/technology/2004-03-30-shirky-situatedsoftware.html

Snelting, F. et al (2022). *A Wishlist for Trans*femminist Servers.* [online] Available at: https://etherpad.mur.at/p/tfs

Soon, W. and Geoff Cox (2020). *Aesthetic programming : a handbook of software studies.* London: Open Humanities Press.

vuejs.org. (2021). *Lifecycle Hooks | Vue.js.* [online] Available at: https://vuejs.org/guide/essentials/lifecycle.html

XPUB (2022). *Learning How to Walk while Catwalking.* [online] Available at: https://issue.xpub.nl/16/

XPUB (2022). *the-screenless-office. XPUB GIt.* [online] Available at: https://git.xpub.nl/kamo/the-screenless-office

List of figures

License