MATLAB Doesn't Love Me: An Essay

Tijs van der Storm
storm@cwi.nl
Centrum Wiskunde & Informatica (CWI), Amsterdam
University of Groningen, Groningen
The Netherlands

Geor Bakker
Geor.Bakker@soseiheptares.com
Sosei Heptares, Cambridge
United Kingdom
Amsterdam University Medical Centres (AUMC)
The Netherlands

ABSTRACT

Programming is everywhere, and is becoming an increasingly essential component of knowledge work outside the realms of traditional software development. Examples include data journalism, scientific computing, machine control, machine learning, financial management, and others. A key aspect of this trend is that users have to use programming tools, but typically lack programming education, let alone a computer science background. In this short paper we revisit potential assumptions and preconceptions underlying traditional programming system design, from the perspective of practicing scientists using tools like MATLAB, R, Bash, Python, C++, and others. Specifically, we aim to peel off some ingrained assumptions that have informed programming language and system design for decades. Without giving a lot of answers, we hope some of our contrarian observations may turn out to be controversial, and stimulate a meaningful discussion towards a better programmer experience in the domain of science.

CCS CONCEPTS

• **Software and its engineering** → **General programming languages**; • **Human-centered computing**;

KEYWORDS

End-user programming, scientific programming, language design, programmer experience

ACM Reference Format:
Tijs van der Storm and Geor Bakker. 2020. MATLAB Doesn't Love Me: An Essay. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3397537.3397557

1 INTRODUCTION

Programming is everywhere, but not everyone is a professional software developer. Fields such as data journalism, medical science, financial auditing, machine control, and many others, are increasingly dependent on computing, yet their practitioners often lack formal programming education, let alone a computer science background. In this essay we take a personal look at one such domain, namely medical imaging research in the area of neuropharmacology. We sketch a typical scientific workflow in this domain to uncover which aspects of programming are at stake and how they are (mis)aligned with what end-users actually need [8]. Specifically, we aim to contrast the way that practitioners experience dealing with common programming systems (such as MATLAB, R, Python, Bash, etc.), with the expectations such tools have of their users.

The core problem that we aim to isolate and circumscribe is epitomized by the title: "MATLAB doesn't love me", a phrase of frustration to provocatively capture the experience of neuroscientist programmers and their discontent. The subtext of this phrase is threefold:

- *The Other*. Programming systems are anthropomorphized as The Other, the ultimate stranger, whose language we do not speak and who is unresponsive to our advances.
- *Self-blame* Even though the subject of the quote is "MATLAB" it is clear that it presupposes that MATLAB is interpreted as a given, and that the failure of love is due to the user: it is our own fault that MATLAB does not love us; the tool is above reproach.
- *Despair* The quote expresses an experience of one being left alone, to one's own devices.

Instead of computers as "bicycles for the mind" [9], it seems programming systems in particular are experienced as loveless entities, necessary evils, that, rather than work for their users, actually work against them. In the area of neuroscience this state of affairs has real and serious consequences: it stifles scientific progress because current programming processes and tools take inordinate amounts of time, are error-prone, and prevent scientific results to be easily reproduced. Finally, domain-experts (MRI physicists, neuro-imaging experts, pharmacologists, etc.) waste valuable brain cycles in fighting programming systems, which could be better spent on the science itself. One could thus say that current state of programming tools are "brakes on the mind" for this group of users.

In remainder of this essay, we first look at a specific case in the domain of neuro-imaging. Although we do not want to stress or promote any essential difference between "real programmers" and knowledge workers who happen to have a need for programming (let alone provide any value judgment on either category), we will refer to professional software developers as "programmers" and the latter category as "end-users". The distinction then serves as a tool to contrast how programmers might think about programming and programming systems, with how end-users are experiencing programming and its tools. Given this polarized spectrum of preconceptions and experiences, we look at some (recent) developments

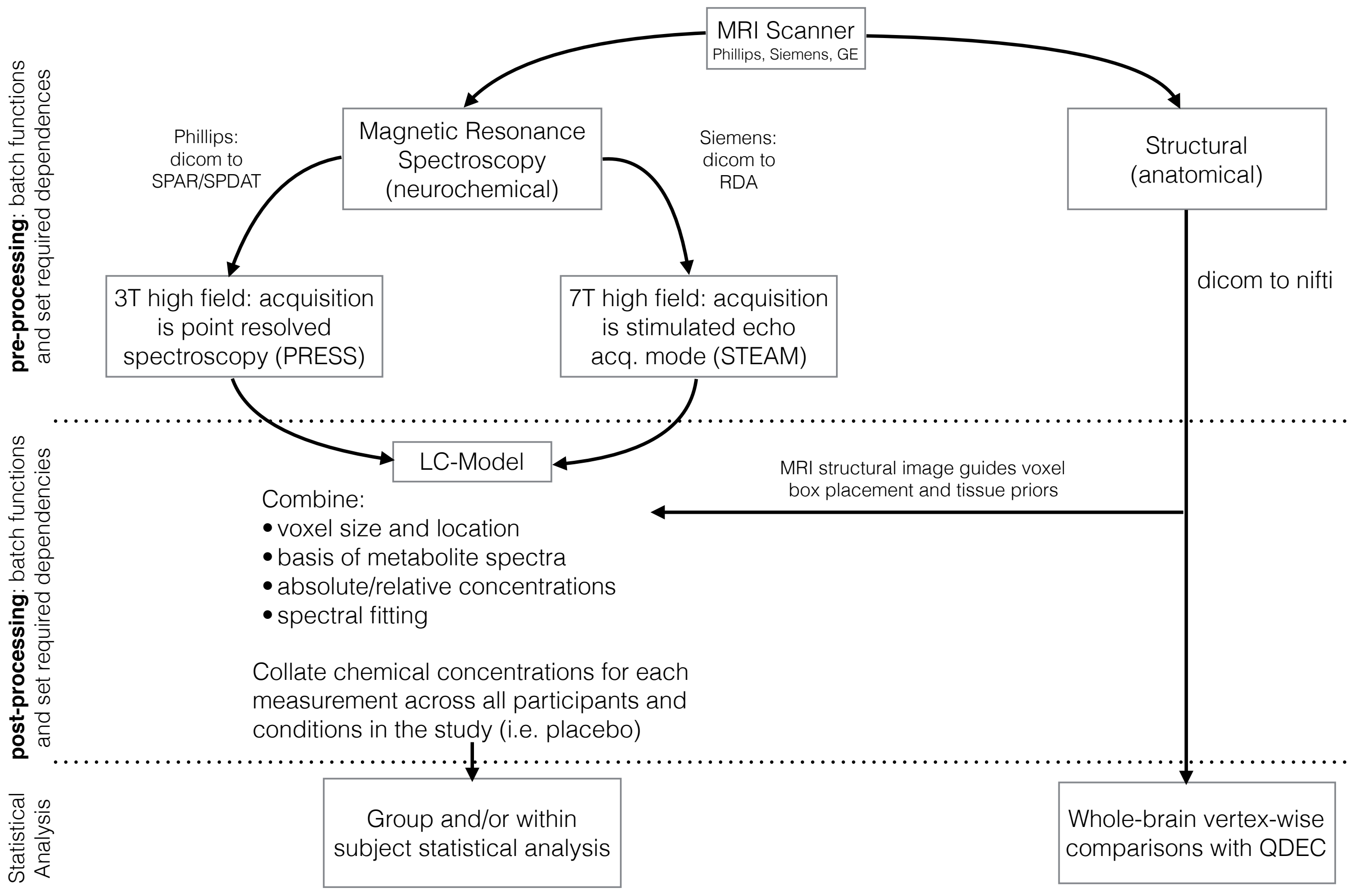
MRI Scanner
Phillips, Siemens, GE
pre-processing: batch functions and set required dependences
Magnetic Resonance Spectroscopy (neurochemical)
Phillips: dicom to SPAR/SPDAT
Siemens: dicom to RDA
Structural (anatomical)
3T high field: acquisition is point resolved spectroscopy (PRESS)
7T high field: acquisition is stimulated echo acq. mode (STEAM)
dicom to nifti
post-processing: batch functions and set required dependencies
LC-Model
MRI structural image guides voxel box placement and tissue priors
Combine:
• voxel size and location
• basis of metabolite spectra
• absolute/relative concentrations
• spectral fitting
Collate chemical concentrations for each measurement across all participants and conditions in the study (i.e. placebo)
Statistical Analysis
Group and/or within subject statistical analysis
Whole-brain vertex-wise comparisons with QDEC

Figure 1: Typical neuro-imaging pipeline (simplified).

to bridge the distance between them, and provide tentative directions for further research and design of more lovable programming systems.

2 THE PROGRAMMING THAT NEUROSCIENTISTS DO

Figure 1 shows a simplified example of a magnetic resonance spectroscopy (MRS) work flow commonly used to examine neurochemical mechanisms and drug efficacy in neuropsychiatric and neurological disorders (e.g., [2, 11]). Dealing with multi-vendor data is common due to use of historical data, multi-site scanning, consortium data, vendor switching within hospitals, research dedicated versus clinical scanners, PET-MRI combined machines, and level of investment from vendors to advance new sequences. In line with other fields there is an impetus in medical imaging to acquire and analyse larger data sets to improve statistical power and generalisability.

Workflows unfortunately are *sequence specific*, which means that the data processing pipeline that needs to be implemented depends on the sequence of steps the MRI scanner was programmed to perform. An MRS workflow can not be applied to diffusion imaging (microstructural changes), echo-planar imaging (regional activation), or positron/single photon emission tomography (drug binding to receptors).

In other words, for such tasks, there are markedly different pipelines like the one displayed in the figure. Sometimes these processes are even more involved, because different tasks require additional steps in the pipeline to, for instance, deal with temporal dimension, to apply motion correction and corrections for changes in field inhomogeneity.

The figure shows different stages of the process (image acquisition, pre-processing, post-processing, and statistical analysis). Each of these stages requires a variety of technologies: conversion tools, scripts, MATLAB libraries, Linux-based imaging processing libraries, R programs, and variety of file formats. Depending on varying expertise of the scientists involved, different stages may have to be performed by different persons. In the context of the figure, we can already distinguish the following technologies and tools:

- File formats: vendor-specific image formats (SPAR, DAT, RDA), generic file formats (DICOM, NIfTI), CSV.
- MATLAB: most of image processing in the pre-processing phase is done in MATLAB, combining the SPAR/DAT and/or

RDA files to obtain a model that is ready for statistical analysis
- R: most statistical analysis is performed in the R environment.
- Bash scripting: although implicit in the figure, various stages of the pipeline are meticulously glued together using command-line scripts, for instance to iterate over sets of patient data and aggregate results.
- Third-party tools: e.g., dcm2niix[1], QDEC[2], Anatomical Processing Script[3], etc.

In addition, the figure does not show the hardware and vendor specific programming needed to instruct a physical MRI scanner to perform an imaging sequence. This highly specialised form of programming is called *pulse programming*. Pulse programs are specific for machine models, machine vendors, versions of the hardware etc. These factors can influence their execution. Unfortunately this only becomes readily apparent while scanning. This leads to data loss and can affect the reproducibility of the outcome measure.

The software components (tools, scripts, libraries, etc.) exist in multiple versions, and have specific or outdated dependencies or platform requirements. The MATLAB functions used in the pre-processing stage of Figure 1 are being replaced by Python scripts. In different hospitals or research institutes, SPSS is preferred over R. More generally, Julia [4] is gradually becoming more popular in the domain. Nevertheless, many of the components are third-party black-boxes, of which the actual quality and/or reliability is hard to estimate; they are simply often "the tools that everybody uses".

In the end, programmers might object, "but, but, you could solve these problems if you would use ⟨*insert favorite programming language here*⟩...", but, professional software developers, end-user programmers typically do not care about programming languages, and experience them as a nuisance that distracts them from their goals. Compared to what programmers may think then, what could be the end-user experience in the domain of neuroscience?

3 WHAT PROGRAMMERS THINK VERSUS WHAT WE'VE SEEN

In the previous section we have shown a typical example of the programming, scripting, glueing tasks that neuroscientists typically have to perform. Specifically, we have characterised how this situation involves a complex product along various dimenions of goals, technologies, and machines. How does this compare to what programmers, and in particular, designers of programming systems, think are solutions to such problems?

Table 1 presents an exaggerated and polarized comparison between preconceptions of programmers and what end-users may experience. For effect, the contrast in the table is exaggerated to drive home the point that perhaps "we" as programmers might have it all wrong in terms of how to solve the problems sketched earlier. The rows in the tables are not the result concrete research, but mainly serve as a rhetorical device to raise the stakes of the discussion. Let's briefly discuss each of them in turn.

[1] https://github.com/rordenlab/dcm2niix
[2] https://surfer.nmr.mgh.harvard.edu/fswiki/FsTutorial/QdecGroupAnalysis_freeview
[3] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/fsl_anat

Table 1: Programmer preconceptions versus end-user experience (a hyperbole)

Programmer preconception	End-user experience
Early correctness	Human checking
Consistency	Repair
Abstractions	Invocations
Graph-structured	Linear processing
Control-flow-oriented	Transformation-oriented
Data structures	Files
Long-term	One-off
Homogeneous	Multifarious
Hierarchy	Rhizome
Creative design	Menial chores
Epic/Homer	Dystopian/Kafka
Mastery	Subjection

Programmers love early correctness: correct-by-construction, type safety, and code that is "obviously without deficiencies" [7] are important tenets of programming and programming language design. End-users, however, are more inclined to rely on human checking: inspecting intermediate results, error diagnosis performed during the processing. Related is the programmer's focus on consistency (data validity, well-formedness etc.) versus a focus on data repair if something happens to be broken.

At the heart of programming is the notion of abstraction: reusable pieces of encapsulated functionality. A carefully designed abstraction can be instantiated and reused many times over. End-users however, do not seem to be interested in this at all. As highlighted in Figure 1 the main meat of the programming tasks consists of invoking pre-packaged utilities or abstractions, designed by someone else (a "real programmer"?).

Whereas most programs and software systems resulting from traditional software development are complex graph-structured artifacts, with many dependencies, hierarchies, links and references (factoring for non-functionals), such complexity is mostly outside of the end-user's interest, instead desiring much simpler (linear) pipelines, where the only notions of complexity are control-flow and input/output relations. However, even though there is some control-flow involved, most tasks are transformation-oriented: data in one form or shape is transformed into another form or shape, either to convert its structure or shape, project out subsets, aggregate multiple sources, or cleanse it etc. Furthermore, programmers think in terms of data as shaped by data structures, whereas end-users tend to think at the level of files with coarse-grained semantics, like images, tables, or lines.

Programmers develop for the long-term: a product that is released, deployed many times, maintained for some period of time, etc. In contrast, work-flows as depicted in Figure 1 tend to be one-off. After the "science has been done", or "the drug has been tested", the "code" that was instrumental to obtain the result becomes often stale and irrelevant. It is code to be thrown away, a ladder to climb, but one that one would rather not think about anymore.

Programmers think in terms of clean hierarchies (directories, packages, subsystems, etc.) and explicit dependencies. End-users, unfortunately, cannot afford this luxury; their world is a like the rhizome, it goes in all directions, without hierarchy, without a clear distinction between purposeful or accidental, whether an aspect of software is relevant or irrelevant, and where valid combinations can only be tested by trial-and-error.

The programming world is still under the influence of the notion of "personal computing", where the individual controls and masters the world of computing by theirself. One aspect of this is the illusion of a homegeneous computing environment. The strongest manifestation of this idea is still Smalltalk [6] where "everything is an object", and the IDE, the program, and the OS are one and the same thing. This pipe dream is absent in the world of end-user programming sketched in the previous section. Instead, the end-user is confronted with a multifarious set of tools, components, libraries, languages, systems, machines, platforms, OSes, scripts, file formats etc. which somehow all have to be connected to obtain a meaningful result.

Programming is a creative activity, at least according to programmers. End-users dread the programming tasks they have to do, menial chores that take up too much time. Related to this is the hero narrative that many programmers submit to: it is a quest of mastery, of achieving a high goal: working, reliable, performant software realised with beautiful code. On the other hand, end-users may experience programming as being subjected to a Kafkaesque ruler in a dark, sadistic universe, designed to annoy instead of support.

The consequences of this split between how programmers think and what end-users experience, is that it reinforces the already existing "otherness" of the dominant way programming systems expect to be operated, in their formal, unforgiving, and non-cooperative ways. Above and beyond this, it has an effect on science itself. Scientific processes are time consuming, their realisation is error-prone, and hard to reproduce. A lot of time and brain cycles are wasted on menial tasks at the cost of scientific progress.

4 DEVELOPMENTS

Programming and computational support in the science domain is not new, and many developments have taken place to increase productivity, reproducibility, and scalability of the scientific endeavor. Scientific workflow systems have been one of the most influential concepts in dealing with this complexity [3, 5]. They allow groups of scientists to coordinate diverse data sources and processing algorithms using (often graphical) workflow languages. Such systems have focused on the increased volume and diversity of data, globalisation and decentralisation of scientific collaboration, while at the same time aiming to increase productivity, experimentation, and reproducibility. Research into the programmer experience side of these systems, however, is understudied. This is corroborated by a recent editorial [1], which highlights usability as a key criterion for advancement needed in the 10 years following its publication in 2017.

Notebook interfaces (such as supported by Jupyter or Rstudio) are another recent trend that is becoming popular in scientific programming, data journalism, machine learning etc. A supposed benefit of notebooks is that they support a kind of literate programming where input, output, code, and prose documentation are interleaved in a single, linear narrative. It is, however, as of

Table 2: Features of relevant programming system styles

Style	Wiring	End-user	Input	Output	Live	Diversity
Spreadsheets	○	●	●	●	●	○
Notebooks	○	○	◐	●	◐	◐
Scripting	●	○	○	○	○	●
Workflows	●	●	○	○	○	●

yet unclear how notebooks will scale up to scientific workflows involving multiple decentralised parties (cf. [13], however), and whether they in fact increase reproducibility [12].

Another development could be called "best-of-breed configurations", where experts in some area of computation (e.g., statistics) package the best libraries or tools in an easy to deploy and use bundle. One example is Tidyverse[4], which combines a carefully chosen selection of R libraries that are known to work well and work well together. Another example is the online service QMenta[5], which provides tailored solutions to customize certain automated imaging pipelines. An extreme example of such packaging is Julia[6], which aims to provide a whole best-of-breed language for the scientific domain.

Such efforts, however, either only partially address the situation (e.g., Tidyverse), or assume all participants in a scientific project use all the same language (e.g., Julia). However, scientific workflow solutions need to support at least four categories of experts [1]: domain scientists (e.g., neuro-scientists), research developers (e.g., MRI pulse programmers), data scientists (e.g., statistical analysts), and system engineers (e.g., dealing with deployment, clusters, etc.).

One way to see why a system like Julia might not be the solution is to frame it as a policy versus mechanism distinction. The domain scientists are the ones who decide *what* should be done, whereas research developers, data scientists, and system engineers determine *how* this can be realised. Julia addresses the latter category, but what about the programmer experience of the domain scientists?

In a sense the situation is similar to the separation of roles in domain-specific language-based software engineering, where there is a distinction between domain engineers and application engineers: domain engineers develop the (technical) building blocks from which application engineers assemble end-products using a custom, tailor-made notation [10].

5 INSTEAD OF A CONCLUSION

We end this short paper with a number of observations based on the neuroimaging case of Section 2 and the preconceptions versus experience analysis of Section 3, and hypothetical design directions for a better programming system that could improve the experience of scientists in the field.

First, we acknowledge the link to what Turkle and Papert call "bricolage" [14]: a style of engaging with programming that favors direct manipulation, concrete material, and soft-skilled negotation, over the hard and formal, systematic, abstraction-oriented style promoted by programmers. Failure to acknowledge this different style, causes users of programming systems to see their tools "just as

[4]https://www.tidyverse.org/
[5]https://www.qmenta.com/whitepapers/
[6]https://julialang.org/

tools", rather than as a creative medium to augment one's intellect. Many of the points that Turkle and Papert raise are apropos of this essay: the experience of computing as a chore, the desillusionment with technology, wasted time and talent, etc.

In line of the bricolage style of programming, we make the following further observations based on the case of Section 2:

- The pipeline of Figure 1 shows a high variety in technologies, but is itself low on domain content. The domain knowledge is encapsulated in libraries and third-party tools. In other words technical knowledge required for implementing the pipeline is disproportional to the user's goals.
- The "programming content" is limited to mainly glueing together of coarse-grained imperative steps, with the occasional conditional, or loop to iterate over elements of data sets. This is further acknowledged in the fact that scientific workflow systems often use visual languages.
- In line with the previous item, there seems to be little employ for abstraction mechanisms in implementing such pipe-lines. Scripts are written, but they are one-off, and not meant for reuse outside the current case.
- The lowest common denominator for data is the file (image file, CSV file, ...). The structure of these files is largely implicit; formal meta definition (e.g., a schema, data type, ontology) of their structure is often missing, and if it is present, it is weakly enforced.
- The high variability in technology and lack of unified data represents prevents a uniform way to inspect and repair input and (intermediate) output results. Checking the result of computation involves context switching and disparate technical knowledge.

To lead out this essay, let us lay down the observations from above along-side different styles of programming systems that have received attention recently to address the end-user programming problem in the context of practicing scientist.

Table 2 summarises how such systems support (1) wiring (as in "glueing components together", (2) end-user notation (distinct from "code"), inspection of (3) input and (4) output, (4) *live* inspection, and (5) heterogeneous forms of data. A full circle means that a certain system supports the feature fully, whereas a half-circle indicates partial support (or only in specific instances of the style).

Informed by these observations, here are provisional research directions that could provide a better experience for the programming scientist.

- Integrate data visualisation (a la notebooks) and editing (a la spreadsheets), but keep code and data strictly separated under the hood (to avoid the Smalltalk image trap and promote reproducibility).
- Keep the single notebook/document metaphor as a personal computing environment to avoid context- and language switching.
- Present the illusion of a single document (a la notebooks), but allow plugins for heterogeneous third-party domain-specific components (e.g., importers, visualisers, editors) to create an environment for "scientific mashups".
- Live inspection of all intermediate results in any relevant form (e.g., as a plot, image, diagram, table, or text), especially in the case of inconsistencies or dynamic errors.
- Downplay syntax and abstraction, promote the wiring and glueing metaphor in the user interface.

We might not ever be able to design a programming system that will love its users, but let's at least strive for systems that do not give the cold shoulder, for a more productive and enjoyable science.

ACKNOWLEDGMENTS

We would like to thank Alex Godwood for his helpful commentary, and the anonymous reviewers for their useful suggestions.

REFERENCES

[1] Malcolm Atkinson, Sandra Gesing, Johan Montagnat, and Ian Taylor. 2017. Scientific workflows: Past, present and future. *Future Generation Computer Systems* 75 (2017), 216 – 227. https://doi.org/10.1016/j.future.2017.05.041

[2] Geor Bakker, Claudia Vingerhoets, Daphne Boucherie, Matthan Caan, Oswald Bloemen, Jos Eersels, Jan Booij, and Thérèse van Amelsvoort. 2018. Relationship between muscarinic M1 receptor binding and cognition in medication-free subjects with psychosis. *NeuroImage: Clinical* 18 (2018), 713 – 719. https://doi.org/10.1016/j.nicl.2018.02.030

[3] Adam Barker and Jano van Hemert. 2008. Scientific Workflow: A Survey and Research Directions. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 746–753.

[4] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *CoRR* abs/1209.5145 (2012). arXiv:1209.5145 http://arxiv.org/abs/1209.5145

[5] V. Curcin and M. Ghanem. 2008. Scientific workflow systems - can one size fit all?. In *2008 Cairo International Biomedical Engineering Conference*. 1–9. https://doi.org/10.1109/CIBEC.2008.4786077

[6] Adele Goldberg and David Robson. 1983. *Smalltalk-80 – The Language and its Implementation*. Addison Wesley.

[7] Charles Antony Richard Hoare. 1981. The Emperor's Old Clothes. *Commun. ACM* 24, 2 (Feb. 1981), 75–83. https://doi.org/10.1145/358549.358561

[8] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, and et al. 2011. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* 43, 3, Article Article 21 (April 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[9] Michael R. Lawrence. 1990. Memory & Imagination – New Paths to the Library of Congress. Online. https://www.youtube.com/watch?v=ob_GX50Za6c (excerpt).

[10] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344. https://doi.org/10.1145/1118890.1118892

[11] Pradeep J. Nathan and Geor Bakker. 2020. Lessons learned from using fMRI in the early clinical development of a mu-opioid receptor antagonist for disorders of compulsive consumption. *Psychopharmacology* (2020). https://doi.org/10.1007/s00213-019-05427-5

[12] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of Jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. 507–517. https://doi.org/10.1109/MSR.2019.00077

[13] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley, and Carol Willing. 2018. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. In *Proceedings of the 17th Python in Science Conference*, Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer (Eds.). 113 – 120. https://doi.org/10.25080/Majora-4af1f417-011

[14] Sherry Turkle and Seymour Papert. 1992. Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior* 11 (March 1992), 3–33.