Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Traces of Accommodating Neurofriction in APIs

Nina Matthias
Matrikelnummer: 4854056
nina.matthias@inf.fu-berlin.de

Betreuer/in: Prof. Dr. Lutz Prechelt
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter/in: Prof. Dr. Volker Roth

Berlin, November 21, 2024

Abstract

Context: API Usability, Inclusion in Software Engineering
Objective: Analyze, if the Perl core modules seem more open to different styles of problem solving and information processing, and with that being more neurodiversity friendly, than Python's standard library and if so how.
Method: Comparision of Python's standard library with Perl's core modules by using GTM.
Results: The results are mainly about the reference style API documentation. Perl documents some things differently than Python, which might reach a broader audience by assuming less background knowledge, providing more conceptual knowledge, supporting an explorative approach better, and helping more with problem solving.
Conclusion: It might be useful to further investigate the Neurofriction hypothesis. Different modes of information processing and problem solving should be considered when designing APIs and their documentation on many levels. It might be helpful to diverge from strict reference style documentation.

Freie Universität Berlin

Fachbereich Mathematik, Informatik und Physik

SELBSTSTÄNDIGKEITSERKLÄRUNG

Name: MATTHIAS	(BITTE nur Block- oder Maschinenschrift verwenden.)
Vorname(n): NINA	
Studiengang: BSC INFORMATIK	
Matr. Nr.: 4854056	

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende BACHELORARBEIT selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: 21. NOV. 24 Unterschrift: N. Matthias

Contents

1 Introduction

For a few years some tech companies make dedicated efforts to hire and integrate neurodivergent employees, recognizing their unique strengths and the advantages of a diverse workforce [1].

There is some research about neurodivergent software developers [16], but this focuses mostly on social and workplace challenges.

The "Neurofriction in Programming Tools" theory predicts that there also is a mismatch between the needs different people have when using programming tools and what the actual tools provide [11]. The authors named this mismatch "Neurofriction" (NF) and theorize that this happens, because people's approaches vary depending on their learning, information processing, and attention management style, but so does the design of such tools and sometimes certain styles and approaches are even deemed desirable. A better understanding of NF could help with designing more universal tools [11].

To further investigate the NF theory, we aimed to explore with Grounded Theory Methodology how an Application Programming Interface (API) could cause or accommodate neurofriction on the example of Python's standard library [24] and Perl's core modules [19].

Perl and Python seem like good candidates because they are both scripting languages, thus having similar use cases and abstraction levels. Their mantras however, are antitheses. While Perl's mantra is "There is more than one way to do it" [25], Python's mantra is "There should be one – and preferably only one – obvious way to do it." [18]. This lets us conjecture that Perl might better accommodate neurofriction than Python, by acknowledging that programmers work differently. Perl's style guide mentions "Larry [the creator of Perl] has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does." [22]. This basically describes neurodiversity, which we define in section 2.1, further supporting the conjuncture.

Researching APIs as a first example for a programming tool that might create neurofriction, is a good starting place, because they often have a relatively small scope, which makes them easier to compare. Also using APIs is an integral task in software development, making it a good object of research.

In this thesis we aim to answer the following research questions:

- Can we find indicators that the APIs, their documentations, and communities around Perl seem more open to diverse information processing styles and problem solving strategies than Python?
- If so, what are the indicators for that?

We will explicitly only consider neurodiversity in our analysis, not the intersection with other facets of diversity, like gender.

In Section 2 we introduce important to understand the methodology and neurodi-

versity. We also discuss some of the related research. Section 4 describes how the analysis was conducted and in section 5 we present our findings about how diverging from a strict reference API seems to be a theme in the Perl core module documentation. These are discussed in section 6, together with the limitations and an outlook. A summary of our results can be found in section 7.

2 Background

In this section we introduce the concepts necessary to follow the thesis. We define the neurodiversity paradigm in section 2.1 and explain the methodology used in section 2.2.

2.1 Neurodiversity Paradigm

In this paragraph we define the neurodiversity paradigm using [26]. The variety of neurocognitive functioning is called Neurodiversity. Individuals who diverge from the neurocognitive standard defined by society are neurodivergent (ND), as opposed to neurotypical (NT). Examples for neurodivergences are autism, ADHD, and dyslexia. The neurodiversity paradigm interprets neurodiversity as a valuable form of diversity and rejects the idea of a "normal" brain.

The neurocognitive functionings influence the learning, information processing, and attention management style that the authors of [11] used for their NF theory.

2.2 Grounded Theory Methodology

In this section we will give a brief introduction into Grounded Theory Methodology (GTM) after the school of Strauss and Corbin [4].

GTM is a qualitative research method for developing new theories grounded in data. "Grounded" means the concepts from which the theory is built are derived from data that is collected during research. The resulting theory should explain a phenomenon conceptually, rather than delivering a description for it.

The analysis of data is done in iterations, where the newly collected data is broken into chunks, that are constantly compared, and similar chunks are grouped in concepts. For the following iterations the goal is, besides identifying new concepts, to develop properties (characteristics of the concepts) and dimensions (variations of the properties) for concepts, and relate concepts to one another. If concepts have properties and dimensions, the authors speak of a categories. The different categories are later arranged around a core category, which describes the major theme in the concepts. Together, these categories and their relationships form the theory.

The results of the previous iteration drive the data collection of the next iteration with the goal to move the analysis forward and develop concepts and their relationships. This is called theoretical sampling. Memos are used during the whole process to document thoughts and questions about concepts and their possible relationships, which helps with theory building. To make progress with the analysis and better

sampling, it is important to have insight in what might be relevant. This is called theoretical sensitivity.

At some point, no new concepts will emerge or evolve ("saturation"), so the analysis iterations end and only finishing the theory is left to do.

3 Related Work

In this section we present related research to APIs and their usability, neurodiversity in software engineering, and the programming languages Perl and Python.

3.1 API Usability

[3] describes strategies developers use when working with APIs as personas. While the systematic persona writes code defensively and strives for deep understanding, the opportunistic persona writes code exploratory and only builds enough understanding for the task at hand [3].

[2] describes how they adapted the Cognitive Dimensions Framework from [8] to describe and measure API usability at Microsoft. Amongst others, they evaluate APIs on the dimensions of abstraction level, learning style, and consistency. They used these dimensions to create profiles for the previously mentioned personas from [3].

[15] conducted an observation study on developers who solved programming tasks with an API new to them. They observed differences in development activities and how the documentation was used supporting the opportunistic and systematic programmer personas proposed by [3]. For accommodating these differences, the authors of [15] propose API documentation guidelines to support the efficient access of content for task oriented working, facilitating entry points into the API, and supporting the different strategies adopted when working with a new API.

In conclusion, there is research about different information processing preferences when using an API [3], [8], [15], but none of the research considered neurodiversity explicitly. In the next section we will look at research about neurodiversity in SE.

3.2 Neurodiversity in Software Engineering

[16] conducted the first interview and survey study on neurodivergent software developers. They find that diagnoses are often received in adulthood, implying that there could be many undiagnosed employees, and that neurodivergent employees often do not disclose their diagnosis at work. Therefore, accommodating neurofriction in programming tools might help software developers, whom do not get any other accommodations at work. Although their results were mainly about social and workplace challenges, [16] identified rigid rule following, difficulties with focusing on mundane tasks, and being confrontational in code reviews as challenges specific to software development, that some interviewed participants experienced. They found noticing patterns, visualizing information, writing clean code, and achieving high states of focus, to be strengths of some of their participants.

3.3 Comparison of Perl and Python

[27] conducted an explorative analysis of cultural differences in the Perl, Ruby and Python community. Amongst others, the concept of “One way vs. many” in a programming language and “Idiomatic Code” emerged. Especially the concept of “one vs. many” lays the foundation for our analysis, but our focus is on API documentation with their related discussion and whether we can find traces of accommodating neurofriction there. Both are influenced by the communities around the languages and APIs, since they are implementing and writing the API (documentation) and answers in discussion forums.

4 Method

Since we aimed to explore a theory, a qualitative method like GTM seems like a good starting point to get a first understanding of what could be relevant. Analyzing textual data, in the form of API documentation, needs less resources than doing interviews to sample data, for example. For conducting the data analysis we used MAXQDA [12], a software for qualitative research.

4.1 Sampling

In this section we describe how we sampled five API pairs from both programming languages and some of their related Stack Overflow discussions.

4.1.1 Sampling of API pairs

We selected the API pairs in multiple iterations. This process was not much informed by results of previous analysis iterations, how it would be done with theoretical sampling. Instead we optimized the pairs for covering different domains of programming, because the scope of standard library APIs is not very large. As a result, we had a small data set, so the sampling must be done carefully to avoid introducing too much bias. In this case, analyzing APIs that are too similar could lead to analyzing a style of a specific group in the programming language community or play in the specific strengths or weaknesses of one languages. E.g. Perl is known for its text processing abilities [6].

An API pair is composed of the intersection of one or multiple APIs from the Python standard library and the Perl core modules each. We selected them, such that they have a similar feature set, abstraction degree, and scope for better comparability. At the same time, we paid attention to them not being "too similar", i.e. just a really thin wrapper around another library or a 1:1 implementation of a protocol, because then we cannot expect to find any structural differences. In the process of choosing the API pairs we had to exclude many candidates, because there simply did not exist a counterpart in the other language or the abstraction levels differed too much. Also Perl’s modules were oftentimes smaller than the matching Python library, because Perl divides its modules further into sub modules. In order to analyze a larger feature

set per API pair, we considered intersections of multiple modules or libraries for the API pairs, as opposed to one API per language.

We decided to only consider the standard library for Python and core modules for Perl, because these come with the installation of the respective programming language and are officially maintained by them. Therefore we can be sure that the chosen APIs are truly representative for the respective programming community, thus reflecting their mindset about one vs. many ways. The API documentations were all in "reference style" mode, which the documentation style guides of both programming languages [21], [23] define as descriptive documentation with minimal information.

The API pairs we analyzed are:

Hashing APIs: We analyzed hashlib for Python without the sections about key derivation and BLAKE2 and Digest with the submodule Digest::SHA for Perl.

HTTP Requests APIs: From Perl we analyzed HTTP::Tiny completely and from Python we analyzed the introduction, the section about Request objects, and the examples section from urllib.Request.

JSON APIs: We analyzed the whole json API from Python and the whole JSONPP API from Perl.

Math APIs: We analyzed cmath for Python and Math::Complex from Perl.

Testing APIs: From Python we analyzed the unittest documentation and from Perl we analyzed Test with the sub modules Test::Simple, Test::More and Test2 with the submodules Test2::Suite, Test2::BundleMore, Test2::BundleSimple, and Test2::ToolsBasic

4.1.2 Sampling of Stack Overflow Posts

For each API we analyzed, we imported some related Stack Overflow posts. Similar to the sampling of APIs in section 4.1.1, we did this in iterations. In this section we describe how we sampled these posts.

We used the query "`tag:<programming-language> <API-Name>`", that filters for posts tagged with the respective Programming Language, searches for the name of the API and sorts the results by Stack Overflow's relevance metric, to find posts related to the API. We selected five to seven posts per API from the first page of results with help from our theoretical sensitivity. Besides insights from our previous analysis iteration, selection criteria were that the posts were not about technical difficulties like installation issues or closed because of irrelevance (e.g. error was cause by a typo). If there were more than one page of answers, we only analyzed the first page, because answers that seemed irrelevant to the question already appeared before the end of the first page. For theoretical sampling we prioritized posts where the questioner or answerer delivered extensive explanations and posts where it looked like discussions about good practice emerged. We included the comments under answers, but chose to ignore additional questions asked and answered in the comment section, because these have very little context due to their length.

We chose Stack Overflow [5] as Q&A platform, because it is a popular platform for questions related to software engineering. The Perl community seems to be more active on their own platforms and mailing lists though, but sampling from different platforms for both languages would have made the data less comparable.

4.2 Analysis

In this section we describe our analysis, roughly divided into three larger phases. Each phase consisted of multiple small analysis iterations, for which we summarize the results for better overview.

4.2.1 Phase I: Analysis of the first APIs

At the start of the analysis, the main research interest was whether we could find differences in the structure of the APIs and their reference style documentations that might cause or accommodate neurofriction. We oriented our theoretical sensitivity to the mantras of the languages, the systematic vs. opportunistic approach when using an API (as extremes in problem solving approaches), and general needs and strengths that come with certain neurodivergences.

In order to get started and get a better overview, lots of concepts in the initial stage were more descriptive and later refined to analytical codes or dropped completely.

We started with the Hashing APIs, then introduced APIs related to HTTP Requests, and later math APIs.

What quickly emerged was that Perl seemed to have less expectations on prior knowledge of the users, but this was heavily fluctuating over the different APIs. In order to get a deeper understanding we searched for where background knowledge were implicitly or explicitly assumed and where it was provided. It quickly turned out that it is easier to look out for where background knowledge is actively delivered, because the assumption of it is harder to classify. For the provision of background knowledge we differentiated, if the reader is redirected to another resource to look it up, or an explanation is given. However, linking to further resources seems common in online documentation, so we assumed that the in-documentation explanations are more relevant and focused on that, because it made the constant comparison of concepts easier.

Other first impressions were that the structure of the Perl documentation is helpful to opportunistic programmers, because it usually starts with a recipe style code example that gives an overview of the API and the Perl documentation tries to foresee potential errors and misunderstandings of the reader.

At the end of this phase, we researched the documentation style guides [21], [23], which confirmed that Perl considers missing prior knowledge even in the reference style documentation mode and Python does not.

4.2.2 Phase 2: Analyzing the first Stack Overflow Posts

For theoretical sensitivity we dived deeper into API usability research, especially the adapted cognitive dimensions framework [2] discussed in section 3.2. This helped the next concepts related to the structural differences to emerge. However, besides Perl offering aliases for function names more often than Python, which was also observed by [27], we could not identify much structural differences between the APIs. This might be, because both languages are too similar in abstraction level and typical use cases. When working with the cognitive dimensions, we also noticed that both languages share features that are more helpful to the opportunistic programming persona. This makes sense because both are scripting languages. So we adapted our theoretical sensitivity and looked broader for signs of multiple approaches in our data.

We started our Stack Overflow analysis with the intention to confirm our findings from the previous phase. To achieve this, we classified how much additional info was provided besides a simple answer or solution and in which form (example, link, how to understand error, or come to the same solution as the answerer). It turned out that links were almost always provided, which is encouraged by the Stack Overflow guideline for answers [10], so we omitted it.

Later we added the APIs for Testing and JSON. What really stuck out in some of Perl's Testing APIs were the use of mnemonics to help with the different unit test functionalities and also the number of concrete examples. We started to look into the other Perl and Python APIs for hints on whether and how much they actively help to learn or understand the API. This helped us to identify multiple expressions of that and also supported that Perl seems to do this more often. Also Perl tends to explain more how functions work internally or why they designed the API in a certain way.

In order to look for hints on the phenomena of prior knowledge and help with learning and understanding, we categorized the answers on Stack Overflow on how supportive they are: answers that explain or solve barely enough to help, answers that give additional support or explanation (like help for language beginners, an in-depth explanation to understand the problem or the way to the solution), and the answers somewhere in the middle. Then we tried to connect the in-depth answers to our API concepts and also went over the answers in the medium category, since it was hard to draw a line. However, we could not find that many connections. Most of the Stack Overflow answers seemed to stick out, because they differentiated between different possible use cases or solution approaches. Sometimes they tried to connect API concepts. The in-depth answers on the problem or way to solution were another hint to look at where help with problem solving is given.

4.2.3 Phase 3: Building the Theory

For the last phase we took a closer look into the concepts that are related to helping with problem solving. We were interested in how differentiated the solution approaches and explanations on Stack Overflow are. For both data sources, APIs and Stack Overflow posts, we looked into how they proposed good practices, if they somehow explained why this is, and how they handled potential problems, errors,

and security risks a user could encounter.

For the Stack Overflow posts we analyzed the different ways in which answers and explanations where differentiated. E.g., if use cases or approaches are differentiated or if the author provides advantages of their solution. We also connected the segments to the problem solving and good practice concepts. For the discussion we were interested in whether the community added to or criticized the solution and about the mindset respective "one vs. many ways." For that, we categorized the mindset behind the answers, based on much they aligned with one of the mantras.

Against our expectations, by sheer numbers the Python posts were a lot more differentiated. Under the Python posts were also a lot more answers and the Python users generally seemed to care more about the reputation point system on Stack Overflow, thus not directly answering the question and instead providing answer that might be interesting to people that get to the post via a search engine. This is why we considered concepts per post and not per answer for our analysis, but it still did not seem representative for Perl.

During all three phases we constantly adapted our concepts and wrote concept and theoretical memos. Due to time constraints we had to stop the analysis before reaching theoretical saturation. We used these memos and the analysis tools from MAXQDA to find a theme among our concepts, and noticed that they mainly were about the API documentation. Interestingly, all our concepts are not something one would expect to see a lot in a classical reference documentation. Since we only analyzed reference style documentations from both languages "diverge from classical reference documentation" became our core category.

5 Results

In the following section we present the results from our analysis, starting with the core category. We print concepts **bold** and direct citations from our data *emphasized*. The resources referenced by the citations can be found in the associated MAXQDA database [7].

5.1 Core category

The core category of our theory is **diverge from strict reference documentation**, which conceptualizes reference style API documentations that do not strictly follow the respective style by providing additional conceptual or procedural information or examples that would be more fitting for a tutorial.

In the following we present the concepts arranged around the core category, which we divided into three groups: many backgrounds, many ways of learning and understanding an API, and many ways of problem solving. Each group has its own section, in which the related concepts are explained, the occurrences in the different data sources are discussed, and, if applicable, it is classified how this applies to the background of the respective programming language.

5.2 Many Backgrounds

Expectations on reader's prior knowledge are expectations on the background knowledge, which the reader needs in order to understand the text. We exclude basic programming knowledge in the respective language from that. Assuming more than that can include (implicitly) assuming a CS degree, the reader being a professional software developer, or knowledge in specific programming domains like cryptography or web development. **Prior knowledge** is only about conceptual knowledge and terminology, not about problem solving skills, experiences, or similar which we we will discuss in section 5.4.

We focused the analysis on places where such background knowledge is explicitly provided: **Providing Background Knowledge** includes explaining the conceptual background knowledge or jargon/notation. With provision we mean an in-text explanation and not just **linking to another resource with minimal context**. These explanations can be further differentiated into which domain knowledge is needed and how much or little prior knowledge is needed to follow the explanation.

For example in Python's math API documentation, it is assumed that someone who wants to program with complex numbers has studied higher mathematics. The provided links lead to a paper and a math textbook. Perl, on the other hand, provides a simple tutorial that invites to explore the concept of complex numbers.

An example for how Perl explains cryptographic hashing is: *"An important property of the digest algorithms is that the digest is likely to change if the message change in some way. Another property is that digest functions are one-way functions, that is it should be hard to find a message that correspond to some given digest. Algorithms differ in how 'likely' and how 'hard', as well as how efficient they are to compute."* (pl-api: Digest - Perldoc Brows, p. 1)

For Perl this could be observed the most in the Hashing APIs, but not at all in the HTTP Requests API. Generally, Perl gives lots of background information in more than half of the analyzed APIs. Especially for the more academic topics, whereas Python constantly assumes more prior knowledge by not providing explanations or only referencing somewhere else.

Perl's documentation style guide [21] requires to specify the intended audience and state the assumptions on expected prior knowledge at the beginning. According to the style guide, at least the core documentation should make as few assumptions about prior knowledge as possible: "Assume readers' intelligence, but not their knowledge" and domain-specific jargon should be defined. The style guide also acknowledges, that Perl is no longer just a community of people with a background in C and UNIX, but rather "a person learning Perl might come from any social or technological background, with a range of possible motivations stretching far beyond system administration."[21] However, the style guide was started in 2020 and was written as a starting place to review Perl's existing documentation, while adapting the style guide to Perl [21], so most of the documentation was written before the style guide existed.

Python's style guide [23] does not say anything about the background of the audience, except that the tutorial should be suitable for newcomers.

The style guides might explain why Perl assumes less prior knowledge than Python and also why HTTP::Tiny is not in line with this. Similar to the C/UNIX background, Perl was also popular for web development [20], so it probably made sense to assume the reader had knowledge about web requests when the documentation was written.

5.3 Many Ways of Learning

This section is divided into two parts and discusses how different ways of learning can be supported in an API documentation or Q&A forum. The first part is about conceptual understanding of the API and the second part is about how the documentation can support readers who do not read the documentation linearly.

5.3.1 Learning and Understanding API concepts

The concepts in this sections are about helping to learn and understand the API. As opposed to the **background knowledge** of section 5.2, this is about knowledge directly related to the usage of the API. We start with the concepts that help with a more general understanding of the API and move to a more detailed degree with concepts that help the understanding of classes and functions.

Giving an overview of the API is about explaining the general structure or core idea of the API and its functionalities. This includes giving an overview of interfaces, classes, or constructors and their use case, flags, mappings, exit codes, as well as listing the most important or most commonly used functions. Reading hints, like where to start depending on the experience level help navigate the API.

This can also include helping to decide, if this API is even the right choice for someone by **Clarifying what the API is (not) about** - informing the user about the features, scope and typical use cases of the API. An expression of that is **Reference to other (maybe more suitable) APIs/sub modules**, which is a way to inform about alternatives and compare features, scope, and typical use cases of the respective API with other APIs or sub modules.

"This is a very simple HTTP/1.1 client, designed for doing simple requests without the overhead of a large framework like LWP::UserAgent. It is more correct and more complete than HTTP::Lite. It supports proxies and redirection. It also correctly resumes after EINTR. If IO::Socket::IP 0.25 or later is installed, HTTP::Tiny will use it instead of IO::Socket::INET for transparent support for both IPv4 and IPv6. Cookie support requires HTTP::CookieJar or an equivalent class" (URL > Pl-HTTPTiny, p. 1)

In the analyzed Perl APIs, users have not much need look for alternative APIs by themselves, because the documentation oftentimes makes it clear what the API does and does not do, what the alternatives are, and in which cases it makes sense to use them. Python does this too, but not in that extent, especially in regards to the direct comparison to other APIs.

Help understand API concept helps with deeper understanding of smaller API components, like classes or functions, by explaining the core idea.

For example, this is Perl's explanation of what completely resetting the incremental parser in JSON means. *"This completely resets the incremental parser, that is, after this call, it will be as if the parser had never parsed anything."* (Pl-JSONPP, p. 8)

Another aspect that can help with understanding is giving **Behind the scenes information:** Information about API internals that are not really relevant for the usage of the API, but that might help facilitate a better understanding, like explaining an API design decision or how a certain feature works.

The following is an example for the design decision on how the so called spaceship operator in Perl works on complex numbers. *"In order to ensure its restriction to real numbers is conform to what you would expect, the comparison is run on the real part of the complex number first, and imaginary parts are compared only when the real parts match."* (Pl-MathComplex, p. 3)

The following is an example for how serialization in JSON works internally: *"This works by invoking the FREEZE method on the object, with the first argument being the object to serialise, and the second argument being the constant string JSON to distinguish it from other serialisers."* (Pl-JSONPP, p. 10)

Connect or contrast API concepts and features sets the prior concepts in relationship, by pointing out similarities between API concepts and/or explaining where they differ.

Like Perl's explanations of the difference between the functions `incr_skip` and `incr_reset` in JSONPP: *"The difference to* `incr_reset` *is that only text until the parse error occurred is removed."* (Pl-JSONPP, p. 8)

As opposed to "Help Understand API concept", a **Mnemonic** is about helping to remember how or when to use a certain feature by providing memorizable explanations.

For example, Perl makes extensive usage of mnemonics in its Test documentation. Here it helps with how to think of the skip-function: *"The first argument should evaluate to true (think "yes, please skip") if the required feature is not available."* (Pl-Test, p. 3)

There are many examples in the Perl documentation on method and API level for these concepts, oftentimes either more focused on understanding or memorizing, but fewer in Python, which in many cases require more prior knowledge.

5.3.2 Many ways to read a documentation

This section is about how people read and work with a documentation. The section 5.3.1 made no assumptions about whether the user reads linearly and/or completely through the API. In this section we will handle the case that they do not.

The concept of **Repeating information** describes how information is repeated across the API documentation in different forms and abstraction levels, for example information covered in section 5.3.1 that help the conceptual understanding of the API.

Ideally this redundancy is **Supporting different ways of understanding:** Repeating

information with alternative explanations and examples, including visual explanations, code examples, repeating an explanation in code or math notation. For example, in the module Test, the Perl documentation provides a verbal description of the effect of skip method, followed by a code snippet that describes the basic idea of the effect:

"This is used for tests that under some conditions can be skipped. It's basically equivalent to:

```
if( $skip_if_true ) {
    ok(1);
} else {
    ok( args... );
}
```

...except that the `ok(1)` *emits not just '*`ok testnum`*' but actually '*
`ok testnum # skip_if_true_value`*'."* (Pl-Test, p. 2)

The concept of **Concretizing** describes how the abstraction level is lowered and information is repeated more concrete, like giving a concrete example, explaining a code example or stating functionally equivalent function calls.

For examples, in Perl's JSONPP documentation it explains for the call of `decode_json`:

" This function call is functionally identical to:

```
$perl_scalar = JSON::PP->new->utf8->decode($json_text)
```

Except being faster. "(Pl-JSONPP, p. 2)

The concept of **Explorability** describes how usable the API is for programmers, who work in the style of the opportunistic programming persona proposed by [3] or prefer to learn exploratively. Concepts that help with explorability are:

- Documentation-wise:
  - Because opportunistic programmers make heavy use of examples.
    - **Recipe style example:** A code snippet that demonstrates how to solve a multi-step problem. In API documentations this usually showcases a range of API functionalities, sometimes it's only for a specific method or constructor. It can show case a concrete example problem, which is more tutorial-like.

      For example, Perl has a synopsis at the beginning of each module, which is a recipe style example for the whole API.
    - **Standalone code example:** One can understand from the example alone what this paragraph of reference documentation is about. **Repeating conceptual information in code comments** is a option to support that.
  - **Motivating the reader to explore:** Addressing the reader directly, helping them getting started solving a problem or exploring an example problem.

- Structural: **Alias/shorthand:** An alternative name to call a function, method or constructor.

The perl API seems to be more explorer-friendly in general. They use aliases and lots of illustrating examples. Recipe style examples at the beginning of each module documentation are part of their synopsis.

An example is Perl's math API documentation: *"If you wonder what complex numbers are (Math > Pl-MathComplex, p. 1)"* followed by an tutorial-like introduction to complex numbers with the Perl Math::Complex package.

5.4 Many Ways of Problem Solving

This section is about various problem solving approaches.

Imply something as good practice: Imply that a strategy, technique, or feature usage is good practice. This can happen two different ways: Implicitly, via an example or anti-example, or just by starting with the best approach, like it is described in the Perl documentation style guide [21]. Explicitly, by directly declaring how something should be done and with varying in the firmness of tone. Optionally there can be an explanation included about why this is good practice.

In TestMore, the author gives an anti-example, a positive example and lets the reader first think for themselves before explaining why the latter is more helpful. The author also emphasizes that it's optional.

" By convention, each test is assigned a number in order. This is largely done automatically for you. However, it's often very useful to assign a name to each test. Which would you rather see:

```
ok 4
not ok 5
ok 6
```

or

```
ok 4 - basic multi-variable
not ok 5 - simple exponential
ok 6 - force == mass * acceleration
```

The later gives you some idea of what failed. It also makes it easier to find the test in your script, simply search for "simple exponential". All test functions take a name argument. It's optional, but highly suggested that you use it. "(TestMore - yet another framework for writing test scripts - Per, p. 2)

Help problem solve: Documentation or answerer tries to equip the reader to solve (part of) their problem (in the future) by themselves. This can be a detailed explanation of how to get to the solution or explaining why something was chosen the way it is.

"For the time being, any necessary padding must be done by the user. Fortunately, this is a simple operation: if the length of a Base64-encoded digest isn't a multiple of 4, simply append "=" characters to the end of the digest until it is." (Pl-DigestSHA, p. 2)

Unexpected behaviors/errors etc.: Hint on potential errors and exceptions that might occur, what to consider before calling a certain method or constructor, or to deflect a certain expectation someone might have. Sometimes there are examples provided of how such a scenario looks or advice on what to do then or how to still reach their goal.

For example, Perl's HTTP documentation warns the user that the `keep_alive` parameter of an request constructor works for only one connection, but also explicitly mentions that one can solve this issue with multiple objects.

"The `keep_alive` *parameter enables a persistent connection, but only to a single destination scheme, host and port. If any connection relevant attributes are modified via accessor, or if the process ID or thread ID change, the persistent connection will be dropped. If you want persistent connections across multiple destinations, use multiple HTTP::Tiny objects."* (Pl-HTTPTiny, p. 2)

Perl seems to give more hints on how to recognize errors, what could be misunderstood, and how one can solve these situations. When they provide hints on good practice it is usually explained or illustrated with examples to give the users the chance to understand and decide for themselves, whereas Python seems to have a more authoritative approach and more often just states that something is done in a certain way, which is in line with [27].

6 Discussion

In the following sections we will discuss our results, name the limitations to our analysis, and give an outlook for the neurofriction hypothesis.

6.1 Discussion of Results

In the following two sections we discuss how our findings connect to existing literature on API usability and why some of our findings might help to accommodate neurofriction.

6.1.1 API usability

In this section we discuss which of our concepts can be connected to existing research on API usability.

[15] conclude that API documentation should not be structured to signal the type of information provided, like a strict reference API, and instead use categories reflecting the functionalities or content domain. This is in line with with our core category.

The same authors find that **background knowledge** needed to work with an API should be provided and important information should be provided redundantly, so

that opportunistic developers do not miss these information, which is in line with **repeating information**.

One of [15] guidelines is that it should be made easier to start actually using the API as early as possible. A strategy they propose for that is to provide code examples that can be used to generate sample API calls, which corresponds to **recipe style examples**.

According to [14], documentation should provide an overview of main features and what to use these features for, which can be connected to **Giving an overview of the API**.

[17] finds that being able to explore an API with the autocomplete popup from the integrated development environment, is important to developers. **Alias/shorthands** might help with that, because it is easier to guess a function name, if there are multiple options. This also demonstrates, that even though the naming of functions is a structural API aspect, it is still connected to the documentation.

6.1.2 Neurofriction

In this section we discuss three aspects that might be influenced by someone's neurotype and for which the identified concepts might help.

Support information processing style The neurotype might influence the needs for a specific information processing style or way to learn, e.g. because of attention management or impulsivity.

Repeating conceptual information, giving multiple solution approaches, and entry points into the documentation help to give everyone a chance to find a connection to the API.

Concepts from section 5.3.1 could help learners who want to deeply understand, while concepts from section 5.3.2 help users who cannot or do not want to read much of the documentation, e.g. because of dyslexia.

Helping Attention Management and Memory Providing information inside the reference documentation like background knowledge, jargon explanations, or comparisons with related APIs helps users who have difficulties recalling such information and users who would get "lost in tabs", if they needed to research themselves.

Repeating information and being "explorer friendly" means that it is not an issue to just skim the docs, which also helps users who have difficulties with reading, e.g. because of a short attention span or dyslexia.

Perceived Self-efficacy Concepts related to problem solving could help with perceived self-efficacy, because users have pointers that they are doing it right, which might reduced overwhelm for some users.

However, some users might still get overwhelmed, if confronted with multiple options and prefer to have simple rules. Adhering strictly to rules related to

programming was also challenge [16] found in their interviews. On the other hand, differentiated approaches and explanations might help exactly these users to not rely too much on rules.

We have to keep in mind though, that Perl was not designed explicitly with having neurofriction in mind and neurodivergence is a huge umbrella term spanning lots of different neurodivergences, whereby even a single neurodivergence can present in wildly varying ways [26]. Some things in Perl documentation might actually cause neurofriction for neurodivergent and/or neurotypical programmers. Perl has lots of metaphors in its language design, which can be an issue for someone who has difficulties understanding language figures. Also Perl's library seems to be in varying degrees of maturity and being designed with prototyping in mind, which might be too much uncertainty for some users.

6.2 Limitations

Our results only apply to the analyzed data sources and might not generalize to other Perl and Python libraries. We never reached full theoretical saturation, so the theory is not fully formed yet and there might be additional properties and dimensions not explored yet or completely new concepts that should be part of the theory.

Due to the small scope of a bachelor thesis our sample size was quite small, which might have introduced bias. For example the choice of data sources could be an issue:

- Because the APIs are all relatively small and encapsulate common concepts in programming, naturally there is mainly reference style documentation. The neurotype however, shapes how someone solves problems, processes information, and manages attention, which might influence a decision for or against using the reference style documentation or if they use official or community provided material.
- Community maintained APIs that are not in the standard library or the core modules play an important role for software development, but were not analyzed for sampling reasons. However, these APIs are often larger and more complex, thus we might find different concepts or structural differences there.
- Not everyone likes to ask or answer questions on Stack Overflow. This could be due to the questions and answer guidelines [10], [9] or the reputation system, both could be due to preferences influenced by someone's neurotype.

We noticed that the Perl modules seemed to differ much more in quality and style than the Python APIs and that the Perl community seemed underrepresented on Stack Overflow.

Also we do not know anything about the background of the authors of the analyzed API modules and the users participating in the analyzed Stack Overflow discussions. This might introduce a bias based on their neurotypes, how sensitized they are for neurodiversity, or their programming background and experience level.

More variation in data sources could potentially uncover new concepts, making it

easier to dismantle structural differences in the APIs, and enlarge the scope of the theory.

6.3 Outlook

By further studying examples of neurofriction and learning how to accommodate for it, we could learn how to design programming tools for larger audiences, thus making software engineering more inclusive.

To get a realistic overview what might cause and accommodate neurofriction in programming tools, an interview and survey study with neurodivergent software developers might be useful. The prior results on the neurofriction hypothesis could be used to prepare the questions and forms. Since neurodiversity is such a big umbrella term, finding enough participants might only be feasible with more results on the neurofriction hypothesis.

To reach that, it might be worthwhile to adapt the framework proposed by [13] to create programming personas with different neurotypes, similar to the programming personas from [3]. [17] discusses how Human Computer Interaction evaluation methods can be adapted for API evaluation. With that, the APIs could then be evaluated for how much neurofriction they create for the neurodivergent personas.

7 Conclusion

The results of this thesis show that it might be worthwhile to be open minded about neurofriction from the beginning of developing an API, since it is something that can be observed in the API documentation, and possibly also in the structure of APIs and the discussions inside the community about it. Thus catering to a more diverse audience would help to reduce neurofriction. One has to keep in mind though, that something can accommodate neurofriction for parts of the community, while it is creating neurofriction for others.

In conclusion, there are hints that Perls approach of allowing multiple ways also is reflected in its core modules documentation and helps to reduce neurofriction. It does so by considering that readers might have different backgrounds and prior knowledge, different needs on how to learn an API, and varying problem solving skills and styles. This results in not being set on a strict interpretation of reference style documentation.

References

[1] "10 Companies Leading the Neurodiversity Movement in Tech | BestColleges." (), [Online]. Available: https://www.bestcolleges.com/resources/companies-leading-neurodiversity-movement-tech/ (visited on 11/21/2024).

[2] S. Clarke and Steven, "Describing and Measuring API Usability with the Cognitive Dimensions," Jan. 1, 2006.

[3] S. Clarke, "WHAT IS AN END- USER SOFTWARE ENGINEER?," 2007.

[4] J. Corbin and A. Strauss, *Basics of Qualitative Research*. SAGE, 2015, 457 pp., ISBN: 978-1-4129-9746-1.

[5] "Empowering the world to develop technology through collective knowledge – Stack Overflow." (), [Online]. Available: https://stackoverflow.co/ (visited on 11/21/2024).

[6] "Essential Perl." (), [Online]. Available: http://cslibrary.stanford.edu/108/EssentialPerl.html (visited on 11/21/2024).

[7] Github, *NinaMatthias/Traces-of-Accommodating-Neurofriction-in-APIs*, Commit Hash: edcca1fa483ac5850f1f6d7801ec746d866674e9, Nov. 21, 2024. [Online]. Available: https://github.com/NinaMatthias/Traces-of-Accommodating-Neurofriction-in-APIs (visited on 11/21/2024).

[8] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, Jun. 1, 1996, ISSN: 1045-926X. DOI: 10.1006/jvlc.1996.0009.

[9] "How do I ask a good question? - Help Center," Stack Overflow. (), [Online]. Available: https://stackoverflow.com/help/how-to-ask (visited on 11/21/2024).

[10] "How do I write a good answer? - Help Center," Stack Overflow. (), [Online]. Available: https://stackoverflow.com/help/how-to-answer (visited on 11/21/2024).

[11] A. Ko, L. Prechelt, and J. Stylos, "Neurofriction in programming tools," in *Theories of Programming (Dagstuhl Seminar 22231)*, T. D. LaToza, A. Ko, D. C. Shepherd, D. Sjøberg, and B. Xie, Eds., Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: 10.4230/DagRep.12.6.1.

[12] "MAXQDA | Offizielle Webseite | Die #1 Software für Qualitative Datenanalyse." (), [Online]. Available: https://www.maxqda.com/de/ (visited on 11/21/2024).

[13] C. Mendez, L. Letaw, M. Burnett, S. Stumpf, A. Sarma, and C. Hilderbrand, "From GenderMag to InclusiveMag: An Inclusive Design Meta-Method," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2019, pp. 97–106. DOI: 10.1109/VLHCC.2019.8818889.

[14] M. Meng, S. Steinhardt, and A. Schubert, "Application Programming Interface Documentation: What Do Software Developers Want?" *Journal of Technical Writing and Communication*, vol. 48, pp. 295–330, Jul. 26, 2018. DOI: 10.1177/0047281617721853.

[15] M. Meng, S. Steinhardt, and A. Schubert, "How developers use API documentation: An observation study," *Communication Design Quarterly*, vol. 7, no. 2, pp. 40–49, Aug. 26, 2019. DOI: 10.1145/3358931.3358937.

[16] M. R. Morris, A. Begel, and B. Wiedermann, "Understanding the Challenges Faced by Neurodiverse Software Engineering Employees: Towards a More Inclusive and Productive Technical Workforce," in *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility*, ser. ASSETS '15, New York, NY, USA: Association for Computing Machinery, Oct. 26, 2015, pp. 173–184, ISBN: 978-1-4503-3400-6. DOI: 10.1145/2700648.2809841.

[17] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, May 23, 2016, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2896587.

[18] "PEP 20 – The Zen of Python | peps.python.org," Python Enhancement Proposals (PEPs). (), [Online]. Available: https://peps.python.org/pep-0020/ (visited on 11/21/2024).

[19] *Perl core modules - Perldoc Browser*, https://perldoc.perl.org/modules. (visited on 11/21/2024).

[20] "Perl for CGI and Web Programming - The Perl Beginners' Site." (), [Online]. Available: https://perl-begin.org/uses/web/ (visited on 11/21/2024).

[21] "Perldocstyle - A style guide for writing Perl's documentation - Perldoc Browser." (), [Online]. Available: https://perldoc.perl.org/perldocstyle (visited on 11/21/2024).

[22] "Perlstyle - Perl style guide - Perldoc Browser." (), [Online]. Available: https://perldoc.perl.org/perlstyle (visited on 11/21/2024).

[23] "Style guide," Python Developer's Guide. (), [Online]. Available: https://devguide.python.org/documentation/style-guide/ (visited on 11/21/2024).

[24] "The Python Standard Library," Python documentation. (), [Online]. Available: https://docs.python.org/3/library/index.html (visited on 11/21/2024).

[25] "There Is More Than One Way To Do It." (), [Online]. Available: https://wiki.c2.com/?ThereIsMoreThanOneWayToDoIt (visited on 11/21/2024).

[26] N. Walker, *Neuroqueer Heresies: Notes on the Neurodiversity Paradigm, Autistic Empowerment, and Postnormal Possibilities*. Fort Worth: Autonomous Press, 2021, 193 pp., ISBN: 978-1-945955-26-6.

[27] A. R. Wille, Bachelor's Thesis, 2024. [Online]. Available: https://www.mi.fu-berlin.de/w/SE/ThesisProgrammingLanguageCommunites.