Needs of API Designers: An Interview Study

Oluwatosin Alliyu
Comp. Science Dept.
Haverford College
Phil, PA 19041
alliyut@gmail.com

Lauren Murphy
School of Info.
Univ. Michigan
Ann Arbor, MI 48109
laumurph@umich.edu

Andrew Macvean
Google, Inc.
Seattle, WA
amacvean@google.com

Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu

ABSTRACT
Programmers spend a significant amount of their time evaluating, choosing, learning and using Application Programming Interfaces (APIs), which are the user interfaces between the programmer and the code they want to reuse. Although the usability of the APIs is both crucial to programmer productivity and the correctness of the resulting code, previous research has shown that many APIs suffer from substantial usability issues. We performed extensive interviews with 14 API designers from 4 companies, and workshops with about 50 more designers, to identify their training, design and evaluation process, barriers and needs. Results include that the designers learned on the job from their own and others' experience, the key process for ensuring API usability is code reviews by others, there are a few helpful API analysis tools but more are needed, and designers would like more help with the early "sketching" phase of API design.

Author Keywords
API Usability; Empirical Study of Programmers; Developer Experience (DevX, DX); Web Services.

ACM Classification Keywords
H.1.2 User/Machine Systems: Software psychology; D.2.2 [Programming Languages]: Design Tools and Techniques – Software libraries.

INTRODUCTION
Application Programming Interfaces (APIs), are the user interface between programmers and the libraries of code, SDKs, and frameworks that they use [18]. APIs are central to the software development process. Given the growth of web-services, microservice architectures, and core language frameworks (jQuery for Javascript, the .Net Framework, etc.), it is very likely that almost every line of code a programmer writes will include an API call. As of September, 2017, programmableweb.com lists over 18,000 APIs for Web services, with hundreds more being added each year. The study of the APIs Usability [18] has often focused on the programmers who *use* the APIs, known as *API Users*, neglecting to focus on those who *design* the APIs, who we call the *API Designers*. With an increasing number of companies creating APIs, it is important to have an understanding of the questions that API Designers have and challenges that they face. These are likely to be different than those of the API Users, due to their different interactions with the API and their different goals and constraints. In order to better understand the needs of the API designer, we conducted interviews with 14 programmers experienced in API design across 4 software companies. We also moderated two workshops with about 50 additional API Designers from one of the companies.

Even though there are a vast array of available sources that could potentially help API designers, including company-specific design guidelines, blogs, and books [4][2], adherence to the many processes and rules recommended in these sources is not always easy to accomplish. APIs come in different forms and sizes, including both public and private APIs, as well as APIs for web services (REST), Java, .NET and also for new languages such as Dart. This diversity of APIs introduces an underlying complexity to the research that we are conducting since API Designers' needs and constraints vary widely across the different platforms. Our research identified commonalities and differences across the designers.

The information gathered from these interviews can be helpful to companies and teams who are designing APIs and want to identify best practices, to API designers of all levels of experience to understand what others designers do, and to tool builders and researchers wondering what would most help designers. The interviews provide recommendations from different companies about their company-wide processes, tools, concepts and guidelines that others could implement to enhance the developer experience (called DevX or DX by analogy with UX) of their API designers. Similarly, veteran API designers can discover new methods and conventions that they may not know. Novice designers can see the concerns that experts have, and what resources they recommend for learning API design. Finally, this paper discusses missing knowledge and tools that the API designers would like to have, which may help researchers focus on what would be most helpful to investigate.

Some of the specific contributions of this paper include the following observations:

- Designers learn on the job; there are no classes in school about API design, even though it is an important and difficult task.
- The design processes for APIs was generally fairly informal. There were often required reviews of the finished design, but no prescribed process *during* design and development.
- Design reviews, following guidelines, and beta testing were largely felt to be adequate for ensuring API quality, and those interviewed did not report using usability studies of their APIs or API documentation.
- Most designers were very wary of making changes that would break users' code, whereas a few designers who had the full code base that was using their API were much less worried about backward incompatible changes, since they were responsible for updating the code, often using "refactoring" tools.
- A few of the companies had automated tools to help in the API design process. For example, one reported using a "Linter" tool and Error Prone (errorprone.info) to check for common errors.
- Though good documentation was explicitly mentioned as a positive quality for an API, many designers had found that they had challenges with writing documentation with high discoverability.
- We identified a list of future tools that designers would like to have, which primarily focus on design time issues, especially at the early stages.

RELATED WORK

API Usability

As APIs are a form of user interface, considering the usability of that interface is important [17]. Indeed, the usability of an API has been shown to impact the productivity of the programmer, the adoption of the API [18] and the quality of the code being written. If used incorrectly, research shows that the resultant code is likely to contain bugs and security problems [7].

However, APIs are often hard for programmers to learn and use [18], with prior work identifying many causes, including the semantic design of the API, the level of abstraction, the quality of the documentation, error handling, and unclear preconditions and dependencies, all impacting the usability [24][14][21][12][19][29]. Additionally, changing an API after it has been deployed is difficult, due to its potential to break the software that depends on it [15]. All of this combines to make the API design process critically important.

API Design

There are a number of decisions API designers must make to create an API [27], and quality attributes that must be evaluated [18]. This makes the task of designing and building a usable API challenging. The impact of some API design decisions have been explored, providing API designers with empirically based guidelines to follow. For example, the use of the factory pattern [6] and required constructor parameters [26] were shown to be detrimental, and method placement was shown to be crucial [28]. Additionally, recent work has looked at defining metrics for encapsulating and measuring API usability, allowing for larger scale quantitative assessment of an APIs design [20][25]. As much as 25% of the variance in the proportion of erroneous calls made to an API could be predicted by looking only at 7 structural factors of the API's design, including the number of required parameters in the API call, and the overall size of the API [14].

In addition to the recommendations from the aforementioned evaluations, comprehensive API design guidelines exist in the form of books [4][2], and API Style Guides, e.g. those from Google [9] and Microsoft [16]. A recent review of 32 publically released REST API Style guidelines identified core design principles, consistencies and inconsistencies in advice given to API designers [13]. Frameworks, such as the Cognitive Dimensions of Notation, can be applied as a way of interpreting and understanding an APIs design [3].

Understanding The API Design Process

There are a number of examples of the positive impact that can result from using traditional HCI methods during the API design process, including usability studies, design reviews, and heuristic evaluations to aid in understanding and improving the APIs' usability [6][26][10][15][28][8]. However, more broadly understanding the needs of the API designers, and the process they go through while they design, implement, release, and maintain an API, is less well explored. Macvean et al., discuss part of the web-API design process at Google, which includes an expert review of a proposed API design. The goal is to ensure consistency and quality while providing API designers with the ability to consult with API design experts [15]. Henning stresses the importance of education for successful API design, arguing that good API design can be taught [11].

While there has been much work done in understanding the software engineering process in general, e.g., the role of knowledge sharing within organizations [22], the impact of distance between engineers [1], and the importance of well defined tasks [5], understanding the specifics of the API design process, and the barriers facing API designers, have not been studied.

METHODOLOGY

To better understand the actual needs and constraints of API designers, we undertook an interview study of practicing API designers from industry. We first created an initial draft of a questionnaire, which was then shared with some of our current and former collaborators for comments and additions. The result was about 35 questions, many with follow-up subparts, that touched on many topics of concern, including what a good API is, the different design processes, difficulties faced when designing, and how to improve this process in the future. (The full questionnaire is available at <<*anonymized--see suppl. doc*>>) Using this questionnaire, we interviewed 14 different API designers from 4 large software companies and from various positions within the companies. During each interview, which lasted from an hour and a half up to two hours, we took extensive notes on what each interviewee said. The interviews were all performed through video conferencing, since the interviewees were remote (with participants from USA, Asia, and Europe). The interviews were recorded so we could go back and review as necessary. All the interviews took place during working hours, and there was no compensation offered. After reading a standard introduction that was approved by our university's Institutional Review Board (IRB) and by appropriate people at the companies, we asked the questions on the questionnaire. Separately, we participated in two "workshops" at one company during an "API Summit" in June, 2017 to validate the questions and increase the number of respondents. About 50 people participated across the 2 sessions. We then organized all the responses into an Excel spreadsheet to better analyze and identify any interesting patterns, statements and contentions. Participants were promised anonymity for their APIs, companies and themselves.

PARTICIPANTS

We used our personal contacts at many companies to invite participants, and they further asked appropriate people at the companies who were involved in API Design. This resulted in 14 participants who were interviewed between July and September of 2017 from 4 large software companies that will be called Companies A, B, C, and D to protect their anonymity. All participants were experienced programmers with a range of 8 to 28 (mean=16.1) years of programming experience and a range of 1 to 18 (mean = 9.1) years of experience designing APIs. Most participants had obtained a bachelors degree, but the range of education levels spanned from high school diploma to PhD. When asked to describe their level of expertise in API design, most were reticent to label themselves as "experts" but they clearly were. The APIs they described spanned both internal and external use and were various sizes. Though it was difficult to measure size, most used the number of methods as a proxy and this ranged from 20 to 300 for their largest API. All had designed APIs that were in use by many programmers, ranging from hundreds to millions.

RESULTS

Learning to do API Design

We first asked the API designers how they learned about API design, and they all said that their knowledge and skills came from firsthand experience with actually designing APIs on the job. Interviewees felt there were no university classes or up-to-date "API design 101" books that new designers can look to for the basics of API design. Only one person mentioned any relevant training they received in school, with API design generally not being a topic covered in any classes, and is notably missing from Software Engineering classes. Some mentioned working with more senior designers to learn through apprenticeship but most identified learning through seeing what happened when their own designs were used. Even though the companies had in place API design processes, official and unofficial API design guidelines [13] and tools built and used by smaller groups or the whole company to help with designing and evaluating APIs, API design continues to be a skill that one can only learn on the job. With the growing popularity and requirement that APIs need to be designed by more and more companies, it is important that appropriate resources be created to help train novice API designers so that they can learn the proper skills to build well-designed APIs, make the right decisions when designing their APIs, and can better understand the different properties that constitute a well-designed API.

Key Decisions

API designers have key decisions to make while designing APIs. Some of the ones mentioned by the participants include which methods that should be made available, resources and the relationships between resources, abstraction and naming, future proofing, overall flow of an API, structure, and what specification language to use, such as Swagger (https://swagger.io/) or RAML (https://raml.org/). In order to make decisions regarding these issues, the API designers mentioned questions and information they need or want answered. For example, a key requirement is to better understand the API user's needs. The roadmap of the overall product was also mentioned as important to know to guide decisions regarding the level of abstraction. Additionally, as with any software, the time available before the launch date can help decide functionality that designers may need to consider including. Information on how a product will be used, such as calling conventions, naming, and tools, assists in constructing maintenance procedures. Other information reported that would be helpful early on in the design process includes what an API is trying to do, what is considered out of scope, who will be using it and what their expectations are, what the resources and actions are, if it can be changed later, and emerging industry standards. Not all designs start from scratch, since many designers work on new versions and updates, so existing usage data are informative when available. One designer mentioned that feedback on prototypes is important for design iterations. Once an API is implemented, an interviewee mentioned wanting to know where users are being slowed down, what parts are frustrating, and what parts are difficult to learn or understand.

What is considered a well-designed API

Two questions asked, "Are there some APIs that you think are particularly well designed? What about them do you admire?" One characteristic that was frequently mentioned by the API designers is simplicity. A well-designed API is recognized for its learnability and ease of use.

Similarly, an attribute that was continually mentioned was the idea that the smaller the API the better. However, this comes with a caveat where smaller size should be coupled with intuitiveness, where all the necessary things are properly included and the unnecessary features are left out. A related idea is that less clever is better. API designers should attempt to design APIs so that the cognitive burden on API users is as small as possible. As an example, one designer mentioned that the best design would be if the user can correctly understand how the API works just from using the names shown by the code editor's auto-complete. Adding more things into an API in an attempt to make an API featureful as a proxy to show off the designer's cleverness is never the way to go and just introduces unnecessary complexity to the API.

The Stripe API (stripe.com/docs/api) was mentioned in two different occasions by API designers from different companies as a paragon of a well-designed API for the web. The particular attributes of the Stripe API that these interviewees admire are its 3-panel style documentation, simplicity, straightforwardness, SDKs and "RESTfulness". One of the interviewees mentioned that this API has inspired half of their REST API documentations, where they use the design elements of the Stripe documentation to model their own documentation. The key non-web API mentioned as a model of excellence was Guava (google.github.io/guava/releases/18.0/api/docs/). The attributes of this API that designers admire are its intuitiveness, consistent and predictable design, that it tackles immutability well, and has a strong rule of not breaking users' code. While speaking about this API, one of the API designers revealed another indication of a well-designed API, which is the designer's *intent* -- it is a requirement for a good and usable API for designers to *intend* for it be, so they spend the appropriate amount of research and work on this. In addition, interviewees mentioned that having good documentation with examples as another attribute of a well-designed API, which is consistent with results of studies of API users [22]. Several other attributes were mentioned such as expressiveness, consistency, discoverability, being able to evolve well and having understandable abstractions.

The overarching theme that emerged when considering *poorly* designed APIs was a mismatch between user expectation and the reality of the API design. Interviewees discussed examples where the conceptual model of the API was at a lower level of abstraction than that expected by the user, providing more control and flexibility, but a steeper learning curve. Designers also discussed the impact of inconsistency in the design.

Best practices

We asked designers to identify best practices, and also the opposite - mistakes they have made or encountered. Some recommended finding out who customers are, getting them involved in the development of an API, and even knowing what client code would look like. Getting feedback, and getting it early, was brought up by many designers. One mentioned that they gather feedback using quick "sketching" and prototyping, where early drafts of the API design are reviewed by stakeholders.

As mentioned above, the designers valued simplicity. Keeping an API as simple as possible and not having two ways to do the same action were recommendations, but, as one designer mentioned, simplicity needed to be balanced with usability and "future proofing" - which means to make something unlikely to need to be changed in the future.

Advice about versioning warned to implement it in the right way in the beginning; for example, by using good structure. Excessive, too tight, or inflexible versioning was an issue that some designers had had in the past. One designer did note that designers might have to weigh the cost of breaking existing users code against the cost of producing a better product, but to keep in mind that users have varying sensitivity to the tradeoffs between improvements in a new version vs. having to change existing code. Designers also discussed the benefits of getting the API released quickly and learning from usage / evolving the design, versus the desire for stable and predictable API surfaces.

Consistency as a best practice, especially in naming, was mentioned by many participants (and was also identified by a designer as a place that they had previously made mistakes). Consistency also was identified as a key requirement in API Style Guides [13]. Additional best practices recommended for designing an API included keeping as low of a burden on a reader as possible, using case studies to demonstrate what did and did not work well, and keeping in mind the scope of an API -- both who would write code for it as well as the size of the project.

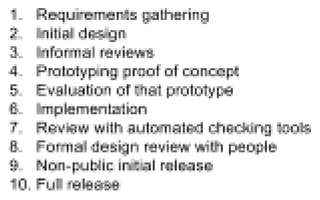
Other issues to avoid include underspecification of API behavior, such as what happens when something goes wrong, as one designer mentions:

"In Swagger or in other API definitions there's often no way of even specifying to the extent that you really need to be able to, all these edge cases. So what happens is that these edge cases become emergent behavior that then we're afraid to change because then if we change that behavior it breaks people."

One designer brought up their mistake of making assumptions about use cases, and not testing said assumptions. Another designer mentioned that they had incorrectly focused on a narrow use case for one of their functions, which made it too easy to implement incorrectly. Also mentioned were a warning against violating expectations, over coupling, using too much boilerplate code in common use cases, and creating new types or class templates.

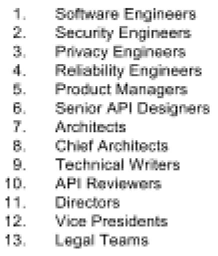
API design and evaluation processes

To achieve a "well-designed" API, as defined by the API designers interviewed and the companies that they work for, each API designer described the design and evaluation processes that they follow. These processes vary between companies, departments, teams and even between individual API designers.



1. Requirements gathering
2. Initial design
3. Informal reviews
4. Prototyping proof of concept
5. Evaluation of that prototype
6. Implementation
7. Review with automated checking tools
8. Formal design review with people
9. Non-public initial release
10. Full release

Figure 1: Maximum steps in API design process for Company A.



1. Software Engineers
2. Security Engineers
3. Privacy Engineers
4. Reliability Engineers
5. Product Managers
6. Senior API Designers
7. Architects
8. Chief Architects
9. Technical Writers
10. API Reviewers
11. Directors
12. Vice Presidents
13. Legal Teams

Figure 2: Roles involved in API design across all companies.

In Company A, there are many steps which might be used for API design (see Figure 1), with the maximal number of steps being used for large public APIs. There are also many roles who may be involved (Figure 2). In general, product managers are responsible for requirements gathering, while software engineers are responsible for the implementation and design. However, there is a blurred line between these roles, where software engineers will at times take on the role of product managers either because there are not enough product managers or the software engineers have the background and training to act in that capacity, They begin by outlining the specifications and requirements for the API, sometimes creating one or more documents describing the basic model and definitions of the API. This is then sent out and reviewed, where the review comments are then used to make the necessary changes. Documentation is also written and sent out for review. Within Company

A, there is a standard design review as well as approval process that all departments and teams have to go through before releasing an API. All code has to be reviewed by another engineer or some review committee before it is released. This review usually incorporates many human eyes and automated tools, which check for API quality, adherence to company style guidelines, fit with other APIs in the company and in some cases, but not all, the API's usability. There are many people involved in this review process including managers, directors and VPs who focus on the business aspect of the API, to security and privacy engineers, legal teams and even reliability engineers, who check for scalability and monitoring of the metrics of the API. The automated tools within Company A's design process include tools such as a "Linting" tool, that check for a variety of common errors. Explicit usability testing of an API is *not* a standard prescribed step within the review process (beyond the expert evaluation of other designers), so APIs do not necessarily go through usability studies. Reasons identified for this include that they do not think it is necessary, they do not have enough time to complete it, they have used the API internally themselves, or the "beta" testing after initial release is sufficient. The level of intensity of adherence to the full process of Figure 1 varies within the company depending on what type of API is being designed (i.e., internal or external facing, completely new API or a new version of existing API, etc.) and the department designing the API.

Similar to Company A, the review process of Company B is the most formal part of the entire design process. An API designer from this company described the process as follows:

(1) An offering manager states some new requirements for an existing or new API.
(2) A development manager delegates the implementation of the requirements to a developer.
(3) The developer creates an initial idea of what this requirement would look like.
(4) The initial API design gets reviewed while developers also begin to implement the API at the same time.

The review process is formally carried out by a designated senior API designer and a group of other stakeholders such as the development manager, the offering manager and other leaders within the company/team. In Company B, the offering managers, which are basically product managers, also set forth the requirements of the API as in Company A. However, in Company B, they report through marketing and are the key decision makers around the functionality of the APIs. While the developers deal more with low level implementation issues and designs, they do not deal with high level decisions such as service naming. However, it is usually the case that individual programmers will initially come up with the name of the API, which ends up being the final name for the product most of the time.

In contrast to Companies A and B which strongly encourage engineering and product manager collaboration with the API design process, Company C has chief architects and architects designing the APIs. Chief architects decide the design for their units and architects are in charge of the actual implementation of this design. In Company C, the API design process begins with the architects and project leads writing specifications, which include functional and technical requirements, for the API. The specification is then reviewed by all members of the team, as well as by product managers and owners, for whether it meets both the technical requirements and functional requirements, where the functional requirements deal with the needs of the client. The requirements are also validated by the QA department, who are responsible for checking the compatibility and functionality. Once the requirements are met and validated, the API is then implemented. Other tests run on the APIs by Company C, before it is finalized, include security tests e.g. for authentication, tests for the performance of the API, a documentation review, and generated API reviews. API designers use the Swagger tool to automatically generate their APIs, SDKs, libraries and/or documentations. Like a manually written API, automatically generated APIs go through reviews to check for their quality and compliance to company specific guidelines. Similar to Company A and B, there is no official usability testing during the development of the API, except for feedback from users after the API has been released. The process for Company D is pretty straightforward, where someone proposes the general shape of the API, and this proposal is then reviewed by a group of people. This review process is conducted by around 10 people who have previous design experience. There is an iterative process where there is continual discussion between the API designer and the review board throughout the design process. Software engineers are mainly in charge of the API design decisions, while product managers get involved sometimes.

We asked our interviewees if they currently used processes or practices at their companies that other

API designers could benefit from. Relating to design, API designers suggest that other companies publish API style guidelines, homogenize their design processes for across-company consistency, and have engineers collaborate closely with Product Managers.

Tools and Metrics for Feedback and Evaluation

When designing and maintaining APIs, a variety of tools, metrics, and data are often needed to check the quality or status of an API. Some general metrics mentioned by various participants include percent of failures, traffic using an API, request traffic for a specific method, and error rates. Some product adoption metrics, such as 30-day actives and error percentages over time, are used by designers to understand the state of their APIs. Size, performance, and number of methods in an API were suggested by an interviewee as interesting. One designer mentioned using the Cognitive Dimensions framework [3] as a metric, but said that it seemed to work best when used by someone whose passion was API design and evaluation, and who was experienced with using it. Some designers mentioned usage measures as a good indicator of what is important for users. User complaints was mentioned as a good, though slow, metric for evaluating an API and was gathered through emails to developers or StackOverflow posts.

Sites like Github and StackOverflow, or even a company's own documentation pages, provide metrics that API designers can use to evaluate their products. However, the usefulness of Github and StackOverflow was controversial. Github stars on SDKs were found by one designer to correlate well with real users' opinions, acting as a proxy since the REST APIs themselves were not hosted on Github. Another designer mentioned downloads and stars on sample projects as an indication of what services were receiving the most attention. However, another interviewee argued against trusting them:

> *"I'm not sure I've ever really trusted star ratings and things like that, that's too much of a single dimension. Having a library that solves your immediate problem might cause you to give it five stars but it doesn't mean it's a good library, it just means it got you out of a hole. That doesn't necessarily speak to its kind of structural quality."*

Another believed that people who added stars had no incentive to provide useful feedback and so stars did not have much weight. Github issues though were mentioned as helpful in understanding common problems that users had, especially if more than a few people were having the same problem. For StackOverflow, the ability to upvote and downvote questions and answers served for some as a metric for understanding how hard it was to figure out an answer and how common the question was. As one designer pointed out though, misinformation and bad comments can be found on the site.

Customer Satisfaction (CSAT) surveys placed in documentation were mentioned by one designer, and small satisfaction and feedback surveys were used as a measure of usability. However, one of the interviewees mentioned that often if something works then nothing is said, but if something does not work then people complain. This can lead to selection bias when determining satisfaction from survey responses alone. A few designers mentioned metrics regarding learnability of an API, how fast you can make your first successful set of calls - one of which mentioned that they used it as a metric for API health. Additional metrics for API health of web services include change rate, error rates, and latency.

Data collected for these metrics come from a variety of sources. Some are more quantitative, such as usage logs, particularly the number and proportion of client and server errors respectively, memory utilization, analytics on documentation regarding how people navigate, and performance. Others, such as internal feedback mechanism from events like hackathons and user studies, can provide more qualitative data for API designers.

Another way that API designers are able to gain information relevant to designing and evaluating APIs is through direct communication channels with the API users. This was said to be easy for APIs for internal use, but harder to external APIs. Not all of the designers we talked to have access to users. In some cases, there was a direct channel of communication between API users and product managers and customer engineers, but not directly with the API designers. For those that *did* have a direct channel of communication, some mentioned having a special email address listed for the owner of an external API. One designer mentioned sending surveys to current users to understand what use cases to cater towards in new APIs. Another brought up placing advertisements or utilizing internal mailing lists and communication channels to reach out to internal users for feedback.

The interviewees mentioned various tools they used to create and evaluate their APIs and documentation. These tools can be used for many purposes such as enforcing guidelines and conformance with nomenclature, running tests, checking an underlying system's stability and scalability, checking the redundancy of a system, as well as measuring latency. Specific tools mentioned include Swagger for defining an API, FXCop, Clang-Tidy, Tricorder [23], Error Prone, Shell-check, Static, ODATA, Security Annotation Language (SAL), and a "Linter" which can check some style guide rules. Another tool was mentioned, though not named, that can check and measure incoming and outgoing web traffic - which can look for spikes in specific error codes, where they are coming from, and if it is from users themselves or the system. Reference, unit, component, and end-to-end testing can be done as well to check for quality or status of an API. Finally, the design reviews mentioned above, where fellow engineers were responsible for evaluating an API, were described by many as a key way to evaluate quality.

With respect to tools used at their companies that other API designers could benefit from, three tools were recommended as being helpful. These include the "Linter" and Error Prone, as well as hand-tailoring generators such as Swagger to a company's code logic. One additional piece of advice suggested declaring an API once and then using appropriate generator tools to create multiple representations of the API without having to implement the same API multiple times.

Documentation

Documentation for the designers' APIs was another topic explored during the interviews. All designers said they were the primary authors for the documentation for their APIs, although a few had help from technical writers. The variety of kinds of documentation included reference documents, specifications, tutorials, FAQs, and overviews. Three designers mentioned creating code examples to demonstrate the use of their API, from guided tutorials with accompanying code, to small one-off code snippets. Often, the extent and depth of the documentation was impacted by the audience of the API, with more effort placed on externally facing APIs relative to internal only APIs.

DartDoc, similar to JavaDoc, was mentioned as one of the most important components to the documentation for APIs written in the Dart language. Wiki documentation, internal and external StackOverflow answers, and mailing list forums were also indicated as sources of documentation for APIs. One designer brought up future resources for their documentation that they wanted to implement, which included using automated documentation, offline versions such as PDFs, technical blogs, and white papers for products.

The way that the interviewees decide what to include in documentation differed as well. Those who had the authority to decide what was included, if mentioned in the interview, were either product managers or technical writers. One designer mentioned that the contents would depend on if the API was shared as open source or if a subscription was required, though no other designers mentioned this kind of distinction. Some interviewees noted that writing documentation was an iterative process, so feedback played a part in the evolution of what was written. Swagger was used by at least one designer to automatically generate documentation. Two designers mentioned a style guide for creating documentation that helped to decide what to include. One designer identified the documentation as part of how the cognitive burden of the API could be kept low. Finally, an interviewee recommended identifying a list of bugs or common misunderstandings that can be fixed or expanded on in the documentation.

To learn more about the challenges and potential issues with documentation, we asked the designers about what the hardest challenges they thought that API users faced when using the documentation, and if they had ever performed a usability evaluation of their documentation for discoverability, readability and ease of use. Two mentioned opposite problems: either a barebones specification or too much documentation -- the latter can cause difficulties in finding what is needed. Organization and poor searchability of documentation was noted by one of the designers. Another mentioned the challenge that API users often have where they do not know what is needed, making it hard for them to search. Other challenges include difficulty in including contextual links, simple omissions, and making the documentation conceptually easy for users to understand. Of interviewees who mentioned having evaluated their documentation, it was done either by an analysis team -- with all members of the development team and their technical writers, by iterating with partners, or investigating at hackathons and other developer-focused events. Some designers mentioned that evaluations of their documentation had revealed that discoverability was a problem.

Difficulties when designing an API

Although API designers in industry generally have a formal process for the development phase of their API (such as code reviews), the earlier design phase often tends to have less formal processes and support. Another issue is that most of the designers discussed their disappointment with the adequacy of the current API design guidelines, and one specifically mentioned that having a canonical source to refer to and look for answers when they run into difficulties when designing their APIs would be helpful. Some of the things that the designers mentioned that were not sufficiently well-covered by the guidelines included naming, abstraction, resource representation, how to design an API quickly so that it is released on time, and how to properly manage backwards compatibility. Also, it was expressed that there is little to no well-defined procedure for custom methods in design guidelines for REST APIs. An interviewee stated that in the definition of REST, one has standard verbs and the ability to create custom verbs which opens up the issue of designing APIs that are inconsistent; there are no restrictions stopping designers from deviating from the RESTfulness of an API. Interviewees said that trying to achieve the goal of a low cognitive burden on API users was one of the biggest struggles that they experience. This is especially difficult since simplicity must be balanced with power. Another struggle is how much to try to make APIs forward compatible. If designers design APIs to handle potential future use cases, then they risk wasting time since the issue might never even arise.

Designers also struggle with not having concrete answers to where the appropriate boundaries are for what constitutes a backwards compatible change and what does not. For example, it was reported that in C++ there are no changes that are 100% backwards compatible, since API users have been known to depend on aspects of the implementation as low level as the line numbers reported in error messages. However, designers who had access to the full set of code that was using their API were much less worried about backward compatible changes. Instead, they worried about how to create "refactoring" tools to automatically fix all the code that would need to be changed, because the API designers were responsible to updating the code using their APIs.

One API designer mentioned that one of the biggest struggles with API design is with the people involved (Figure 2). Since API design is usually situated in a company setting and the end goal of APIs is usually to be released for use by clients of many kinds, there are certain constraints and responsibilities that designers have to meet. One interviewee complained that sometimes a designer's management did not sufficiently value high-quality and highly usable API designs, so that was given lower priority than faster API releases or "cleverly" implemented APIs. An added difficulty experienced by API designers is the time pressure that they feel when designing an API. An interviewee said that "*APIs that are written under time pressure are often really really bad ones*."

Another source of difficulties with the design process of APIs are the tools that are available for designers to use. While there are some tools that different companies, departments and even individuals use, many issues are overlooked and even introduced by these tools. One thing that designers mentioned that cannot currently be checked or measured by the available tools is customer intent, which includes answering questions such as: Are the customers interacting in the way that the designers intend them to? Does the API fit their needs? Do customers understand the API? In addition, several designers shared the frustration of not having a tool that helps with the initial stages of API design, i.e., the sketching of the basic API model, specifications and requirements. As it is, designers use everything from Microsoft Word, email and even change logs because these were the easiest tools to work with.

A tool that has gained significant popularity within the API design community, even though some API designers stated it currently has limitations and usability issues, is the Swagger tool, which can automatically generate APIs, SDKs, libraries and/or documentation. Some of complaints about this tool include lacking of any guarantees about security, the over-verbosity of the documents created, and issues translating between different programming languages. Even though Swagger helps generate documentation for multiple programming languages and aids in reducing the amount of time spent on the API design process, the main issues are that Swagger generates outputs that do not fully adhere to company guidelines and language conventions.

Future tools

After discussing what tools the API designers currently use, we asked about their ideas for tools they wished they could have in the future. Of these tools, many would assist designers during design time. Broadly speaking, designers sought ways to better understand and visualize the design specification of their API, and better understand their users and their needs.

To help with the initial API sketching phase, a tool for listing endpoints and their inputs and outputs was suggested, which would be an improvement over the current practice of using word processing documents. A team-based glossary of terms and common attributes, IDEs with a built-in thesaurus, and a tool to detect naming inconsistencies were all mentioned in some of our interviews. Tools for better understanding an API were also mentioned, such as a tool to help visualize the objects, operations, and their relationships in an API. A tool to notify designers of newly introduced dependencies was suggested. Another designer brought up having a way to understand how a set of requirements may be mapped to different methods to see if a method is doing too much and if there are different ways to do the same thing. Some of the ideas mentioned deal with knowing more about customers and users. One designer mentioned a tool to measure user expectations, which would answer questions such as what users expect to happen, if the API is meeting their expectations, and how often users are surprised by something that has been introduced. Another designer mentioned wanting a mechanism to know who the users are that are currently using their products, who is considering using it, and who previously tried to use it but failed. Another interviewee mentioned a tool that could determine how often people try and fail at some action while using the API, as a way to understand the usability of a design during early releases of an API. One designer mentioned the problem where lots of little issues pile up and cause an API user to give up -- a "*death from 1000 cuts*," so they wanted a tool that could detect and count such little problems and identify where developers are struggling.

Other tools were suggested to assist designers during run-time. For example, one designer wanted more analytics on what parts of an API were truly being used by API users. Tools for health checks and for testing security of an API were mentioned.

Finally, two tools related to the history of an API were requested. The first would analyze the code which call the API to help find parts of the API which are misused and misunderstood. The second would look at historical changes in the design of the API, e.g., between versions, and inform designers of top APIs that people have trouble using.

DISCUSSION

The issues brought up by API designers reveal little commonality in their complaints and requests. For example, some API designers feel that formally evaluating usability is an important phase in API design, while others felt it was simply a nice-to-have or a bonus. Additionally, it was common to hear that API designers were lacking tools focused on assisting in the early sketching phase of API design. However, when asked later about any future tools that could help with API design, only one interviewee mentioned something to assist directly with this phase. This may be because they could not think of how new tools could help with the early sketching of API designs, which makes this an interesting area for future research.

Overall, we identified many areas of improvement for API designers, which could be addressed through more widely disseminating lessons that experienced designers have learned -- such as valuing simplicity, implementing correct versioning in the beginning, and obtaining feedback early -- so that others may benefit. Suggestions for helpful tools and practices were, on the whole, more abstract. As a result, it is not clear that the desired tools can be created. Some of the recommended practices may be challenging to implement because they are a change in company culture, such as homogenizing their design processes.

LIMITATIONS

This study was limited by the number of participants and the number of companies that they represent. We cannot claim that the processes and problems reported would generalize to all API designers or companies. Further, we targeted API designers we knew, and the participants self-selected whether to participate, so participants are primarily designers who care about API design quality and API usability. The results would undoubtedly differ for less motivated or experienced API designers. We understand that with the small interview population, the reported results may not serve as a comprehensive depiction of all experiences, difficulties and barriers faced by API designers. However, the interviews did reveal interesting practices, difficulties and desires that are from the lived experiences of real API designers.

CONCLUSIONS

Our interviews and analysis have shed light onto the firsthand experiences and barriers faced by API designers. Even though there exists some literature, papers and blogs that talk about API design processes, tools and guidelines, the interviews that we conducted provide insights into the real world implementation and execution of these processes, tools and guidelines. We hope that companies, researchers, and veteran and new API designers can use the information in this paper to improve their own processes, create well-designed APIs and create new tools and guidelines to help in the design process.

REFERENCES

1. Elizabeth Bjarnason, Kari Smolander, Emelie Engström, and Per Runeson. 2016. A theory of distances in software engineering. *Information and Software Technology* 70: 204–219.
2. J. Bloch. 2001. *Effective Java Programming Language Guide*. Sun Microsystems, Mountain View, CA.
3. S. Clarke. 2005. *Describing and Measuring API Usability with the Cognitive Dimensions*.
4. Krzysztof Cwalina and Brad Abrams. 2006. *Framework Design Guidelines, Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley, Upper-Saddle River, NJ.
5. H. K. Edwards and V. Sridhar. 2003. Analysis of the effectiveness of global virtual teams in software engineering projects. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. https://doi.org/10.1109/hicss.2003.1173664
6. Brian Ellis, Jeffrey Stylos, and Brad Myers. 2007. *The Factory Pattern in API Design: A Usability Evaluation*.
7. Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, 49–60.
8. Umer Farooq, Leon Welicki, and Dieter Zirkler. 2010. API usability peer reviews. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. https://doi.org/10.1145/1753326.1753677
9. Google. 2017. API Design Guide. Retrieved 2017 from https://cloud.google.com/apis/design/
10. Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. 2012. Methods towards API Usability: A Structural Analysis of Usability Problem Categories. In *Human-Centered Software Engineering*, Winckler, Marco and Et Al (eds.). Springer Berlin Heidelberg, Toulouse, France, 164–180.
11. Michi Henning. 2007. API Design Matters. *ACM queue: tomorrow's computing today* 5, 4: 24–36.
12. Daqing Hou and Lin Li. 2011. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *2011 IEEE 19th International Conference on Program Comprehension*. https://doi.org/10.1109/icpc.2011.21
13. Lauren Murphy, Tosin Alliyu, Mary Beth Kery, Andrew Macvean, Brad A. Myers. Preliminary Analysis of REST API Style Guidelines. In *8th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'2017) at SPLASH 2017*, to appear.
14. Andrew Macvean, Luke Church, John Daughtry, and Craig Citro. 2016. API Usability at Scale. In *27th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2016*, 177–187.
15. Andrew Macvean, Martin Maly, and John Daughtry. 2016. API Design Reviews at Scale. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*, 849–858.
16. Microsoft. 2016. API design. Retrieved 2017 from https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
17. Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and Youngseok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *IEEE Computer* 49, 7: 44–52.
18. Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Communications of the ACM* 59, 6: 62–69.
19. Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. 2013. An Empirical Study

of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/esem.2013.14

20. Girish Maskeri Rama and Avinash Kak. 2013. Some structural measures of API usability. *Software: practice & experience* 45, 1: 75–110.
21. Martin Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6: 703–732.
22. I. Rus and M. Lindvall. 2002. Knowledge management in software engineering. *IEEE Software* 19, 3: 26–38.
23. Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. https://doi.org/10.1109/icse.2015.76
24. Christopher Scaffidi. 2006. Why are APIs difficult to learn and use? *Crossroads* 12, 4: 4–4.
25. Thomas Scheller and Eva Kuhn. 2015. Automated measurement of API usability: The API Concepts Framework. *Information and Software Technology* 61: 145–162.
26. Jeffrey Stylos and Steven Clarke. 2007. *Usability Implications of Requiring Parameters in Objects' Constructors*.
27. Jeffrey Stylos and Brad Myers. 2007. *Mapping the Space of API Design Decisions*.
28. Jeffrey Stylos and Brad A. Myers. 2008. The Implications of Method Placement on API Learnability. In *Sixteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2008)*, 105–112.
29. Minhaz Fahim Zibran. 2008. What Makes APIs Difficult to Use? *International Journal of Computer Science and Network Security* 8, 4: 255–261.