Discovering API Usability Problems at Scale

Emerson Murphy-Hill[∞], Caitlin Sadowski[♡], Andrew Head[⋄],
John Daughtry[♡], Andrew Macvean[♡], Ciera Jaspan[♡], and Collin Winter[♡*]
[∞]North Carolina State University, [♡]Google, [⋄]University of California, Berkeley
emerson@csc.ncsu.edu,supertri@google.com,andrewhead@berkeley.edu
{daughtry,amacvean,ciera,collinwinter}@google.com

ABSTRACT

Software developers' productivity can be negatively impacted by using APIs incorrectly. In this paper, we describe an analysis technique we designed to find API usability problems by comparing successive file-level changes made by individual software developers. We applied our tool, StopMotion, to the file histories of real developers doing real tasks at Google. The results reveal several API usability challenges including simple typos, conceptual API misalignments, and conflation of similar APIs.

ACM Reference Format:
Emerson Murphy-Hill[∞], Caitlin Sadowski[♡], Andrew Head[⋄], John Daughtry[♡], Andrew Macvean[♡], Ciera Jaspan[♡], and Collin Winter[♡]. 2018. Discovering API Usability Problems at Scale. In *Proceedings of International Workshop on API Usage and Evolution (WAPI'18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Usable APIs are important. Egele and colleagues found that API misuse is widespread, with 88% of applications in the Google Play store having at least one API usage mistake [3]. The authors suggest that the problem could be mitigated had the API designers made more usable API choices, such as by providing better default values and documentation. Security is one aspect of programming where API usability makes a clear difference, but other domains like reliability and performance may also suffer from poor API usability.

Researchers have created several labor-intensive methods of uncovering API usability issues, including user-centered design [12], heuristic evaluation [2], peer review [8], lab experiments [13], and interviews [11]. While these approaches can uncover important problems, they cannot be expected to scale up to uncover usability issues across a broad sample of programmers using dozens of APIs.

Researchers have created four main techniques for uncovering API usability issues at scale. The first is conducting surveys [9]; while this approach can scale up to a large number of developers, it requires social capital on the part of researchers to elicit responses and effort on the part of busy developers. Moreover, it can only work effectively if respondents have sufficient memory recall of the API usability challenges they've faced in the past, a problem regularly explored in the social sciences (e.g. [6]). The second technique is to investigate API usability challenges by summarizing them on public discussion forums, like StackOverflow [9]. The disadvantage is that this technique requires developers to be sufficiently reflective and sufficiently blocked to bother posting a question about the API that they're using. The third technique is mining software repositories such as the Google Play store or GitHub to look for instances of API misuse [3]. While this approach scales well, such techniques elide a large amount of valuable software evolution data that happened *between* commits or releases [10]. The fourth scalable technique is mining data provided by "try it out" web API platforms, where researchers can look at 404 errors that indicate which APIs developers have problems with [7]. However, this approach does not reveal what those problems are.

Our insight is that before a developer correctly uses an API, they may struggle to find the right way to use that API. That struggle reveals the nature of the API usability problem. We capitalize on this insight by creating a technique that we call StopMotion. StopMotion analyzes snapshots of developers' work history – typically on every editor save – to reconstruct API usability challenges. By comparing adjacent snapshots, StopMotion finds instances where developers replace one API call with another. When multiple developers replace one API call with another, we infer that there is something confusing about the underlying API. Our approach is a promising approach to discover API usability problems at scale, inexpensively, and at a high level of granularity.

This paper contributes a sketch of this new approach and initial results. We next discuss our approach, describe our experience in applying it at Google, and discuss some areas for future work.

2 APPROACH

The goal of StopMotion is to identify API usability problems automatically. It works in several steps, which we describe below.

The first step is to identify code patches of interest. At Google, patches are peer reviewed before being merged into the company's main codebase. Patches contain metadata, such as the person proposing the change, when the change was proposed, and the files that the patch changes. Our tool selects recent patches that change at least one existing Java file.

The second step is to identify all edits that have gone into a patch. For most software development at Google, file versions are recorded automatically by a FUSE-based file system, such that every save from the developer's editor will be recorded as a separate version. Furthermore, some editors at Google save automatically at regular intervals, such as every 30 seconds. To access this data, we process the workspace associated with the patch. The workspace contains a history of every file changed in that workspace by a developer.

*This work took place while Emerson Murphy-Hill was a Visiting Scientist and Andrew Head was an Intern at Google.

WAPI'18, June 2018, Gothenburg, Swedon

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

Sometimes developers create a new workspace for each patch, but workspaces may also be reused for multiple patches. To deal with workspace reuse, we analyze only the files changed in association with the patch by analyzing the changes between when the patch was submitted for peer review and when the developers' prior patch was merged into the main codebase.

For each edit, we analyze each file that was changed to its prior version. To find changes to API client code, we use the abstract syntax tree (AST) differencing tool, Gumtree [5], which for Java uses Eclipse's Java Development Toolkit (JDT) as a back end. We found that Gumtree often produces inaccurate results when syntax errors are present. We also found that many snapshots contained errors, and we discard such snapshots. Through manual inspection, we observed that if we excluded programs with errors, we would miss many interesting API usability problems, such as when a developer has typed the name of the method they want to call, but has not yet added the parentheses and semicolon that would make the program syntactically valid. However, we judged the false positive rate was too high to continue analysis when errors are present.

Our approach looks for particular fixed patterns of client API changes. Our tool currently detects two similar patterns:

- The developer changes a method call, in the form `obj.a(...)` to `obj.b(...)`. The tool records the fully qualified type of object and the method name before and after.
- The developer changes a static method call, in the form `Class.a(...)` to `Class.b(...)`. The tool records the fully qualified class name, and the two method names.

The tool also records timestamps of when these changes occured, who made the changes, and the associated patch.

3 STUDY

3.1 Methodology

We ran our tool in an informal study at Google on a single multicore workstation. We analyzed the most recent changes from thousands of developers working on real development projects over several weeks in July of 2017. We analyzed the output of the tool manually. There was a long tail of results – many changes were made only once in our analysis period, so we discarded these results in our analysis. In the reporting in this paper, we include some count data, in the form ($n = \ldots$) to give a rough estimate of frequency with which changes occurred.

3.2 Collections

Collections are one of the most commonly used APIs at Google. Google developers use both an open source Google version of collections (`com.google.common.collect`) and Java collections (`java.util`).

One of the most common changes we observed ($n = 174$) is changing an immutable collection, such as `ImmutableList`, from calling the method `of` to `copyOf`. Both methods are static and return an instance of the collection. The method of takes any number of arguments of type `T`, and `copyOf` takes different kinds of collections of `T` (array, `Iterator, Iterable, Collection`). Our intuition about the `of`/`copyOf` change is that the names of the methods don't distill the essence of the difference between the two.

A similar change ($n = 81$) was changing a Java collection's instance method `add` to `addAll`. These methods add an item of type `T` to the collection and add a collection of items of type `T`, respectively. Again, these methods are conceptually similar, but arguably do a better job than `of`/`copyOf` of conveying their purpose. One might posit a simpler solution of overloading the methods so as they are all called `add`; however, this would remove the ability to create collections of collections.

Another common change ($n = 150$) was changing a call to `size` to `isEmpty`. While these two methods have quite different return values, why a developer might make this change is clear; in a conditional with a collection `c`, `if(c.size()==0)` is made more intention-revealing by writing `if(c.isEmpty())`. This refactoring is suggested by a popular static analysis tool at our company. The relative infrequency of changes from `isEmpty` to size ($n = 22$) provides some confirmation of this interpretation. Similarly, we observed developers changing `add` to `put` on maps ($n = 56$). And likewise, `add` does not exist in `Map`, but does in other collections.

3.3 Protocol Buffers

Remote procedure calls are critical to Google given the number of calls being made. When you make 10 billion API calls every second, optimizing calls is important [1]. Protocol buffers are a language neutral structure optimized for serialization. Given that protocol buffers are focused on serialization, they have their own representation of a byte string (`com.google.protobuf.ByteString`). We found evidence of confusion about how to copy data into a `ByteString`. There are 8 different methods that start with the words `copyFrom`. We find evidence of programmers using `copyFrom` and later changing their method call to `copyFromUtf8` ($n = 27$). Interestingly, the specificity of `copyFromUtf8` is a design choice that wasn't required, as there is no corresponding `copyFrom(String)` method on the class.

3.4 `Optional`

One of the most common APIs that our tool returned results for was `java.util.Optional` and `com.google.common.base.Optional` (from Google's Guava libraries), which we'll refer to as Java `Optional` and Guava `Optional`, respectively. Both of these `Optional` classes serve the same purpose; wrapping an object so that it is possible to check for whether the object has been set via an explicit API call without resorting to using `null` values, which can lead to runtime exceptions. Although Java `Optional` is recommended for new code, Google developers use both `Optional` APIs since Guava `Optional` predates Java `Optional` and there is a lot of legacy code. Here is an example use of how either `Optional` class might be used:

```
Optional<String> optStr = getString();
if (optStr.isPresent()) {
  return optStr.get();
}
```

Figure 1 shows a graph that illustrates API usability challenges using the two `Optional` APIs. This graph was created using manually filtered data from StopMotion, fed into GraphViz [4], then tweaked manually for visual clarity. The graph should be read as:

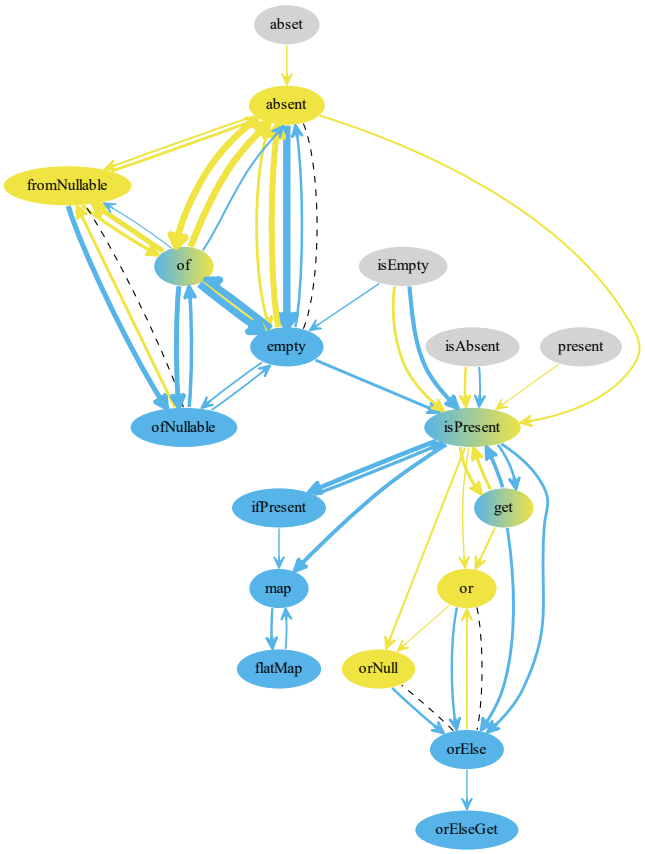

Figure 1: Changes in clients using `Optional` APIs.

- Nodes represent methods on one of the two APIs. Yellow methods exist in Guava `Optional`; blue methods on Java `Optional`; blue-yellow gradient methods exist in both APIs; and grey methods do not exist in either. For example, the method `of` exists in both Java and Guava `Optional`, but `isEmpty` exists in neither.
- Edges from method `a` to method `b` represent developers calling `a` in one snapshot, then calling `b` in the next. The edge color indicates the type of object the developer is using, either Java or Guava's `Optional`. For example, the topmost edge in the graph represents developers who had a Guava `Optional` object, called a non-existent `abset` method, but then changed it to a call to `absent`.
- Edge thickness represents the number of instances of the indicated change. The edge weight is calculated as $log2(n) - 1$, where n is the number of instances of the change. The thickest edge (`absent` to `empty` with a Java `Optional` object) represents 189 changes. Multiple changes may have been made by the same developer.
- Dashed edges represent sister methods of equivalent or approximately equivalent functionality. For example, `absent` and `empty` are equivalent methods on each API.
- For simplicity, only changes that occurred more than 3 times times are shown.
- The plot does not explicitly distinguish between static and instance methods, though they are visible implicitly; the subgraph in the upper left represent static methods, and the subgraph in the lower right represent instance methods.

The graph illustrates several confusions Google developers have about using these APIs: Developers confuse one `Optional` API for the other, as evidenced by looking at the sister methods. For example, developers often interchange `fromNullable` and `ofNullable`, given their similar names. Likewise, they interchange `absent` and `empty`. The case of Java's `orElse` and Guava's `or` and `orNull` is similar, but somewhat more complicated. Overall, the similarity – but not exact similarity – between these two APIs appears to be regularly confusing to developers.

The `abset` node indicates a simple – but repeated – typo, where developers intend to be calling Guava `Optional`'s `absent`. For both APIs, many developers expected an `isEmpty` or `isAbsent` method, when instead they end up using the `isPresent` method, which returns a boolean indicating whether the contained object is present. The mistaken expectation reveals an interesting inconsistency in both APIs; after using the static constructor methods `absent` and `empty`, one might expect an `Optional` object to support an `isAbsent` or `isEmpty` method. However, the APIs do not; instead, the semantics are reversed, supporting only an `isPresent` method. The addition of an isEmpty method has been discussed for Guava's `Optional`, but was eventually dismissed as not having a clear enough advantage.[1] Our tool provides some empirical data to help inform the opinions expressed in that discussion.

3.5 Logging

Developers add logging statements to get some visibility into what is happening inside running programs. When debugging an issue, developers may log fine-grained information about an executing program, but debug logging may be too expensive or verbose to use when running in production. At Google, this problem is addressed by adding log statements as specific levels including error, warning, and info. The log level to use is specified at runtime and only those log statements at or above that level are printed (e.g., programs run at "warning" level also will display error logs but not info logs).

We discovered that developers often do not know which log level to use when creating log statements. Figure 2 shows a graph for the Android logs API, showing transitions between logging methods at different levels: `v` for a verbose message, `d` for a debug message, `i` for info, `w` for a warning, `e` for an error, and `wtf`[2] for a failure that should never happen. In this graph, the number of change instances are labeled on the edges, and no edges are elided.

We note that no non-existent methods are present here, meaning typos were rare. We also observe that the graph is largely evenly bidirectional, meaning that for most edges from a to b, there's an edge from b to a of approximately equal weight. This suggests that developers weren't necessarily confused about what each method does, but instead were unsure which one was appropriate in a given context. The only exception is the d(ebug) to e(rror) transition; there were half as many opposite transitions.

[1] https://github.com/google/guava/issues/734
[2] What a Terrible Failure. https://developer.android.com/reference/android/util/Log.html

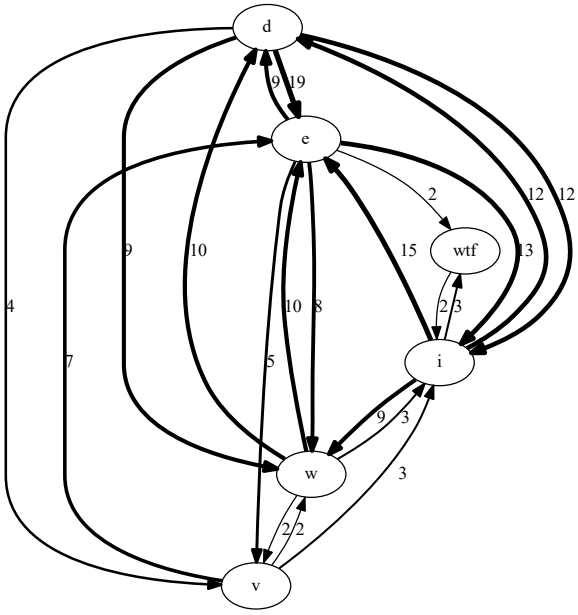

Figure 2: Changes in clients using Android's logging API.

4 LIMITATIONS AND FUTURE WORK

Although our initial work is promising, a variety of improvements are needed to ready this tool for practical use. Long-term, we envision this tool running constantly over developers' snapshots, providing designers feedback on how developers are using, and struggling to use, their APIs. Although our tool capitalizes on the development environment at Google, we predict that in the future, all software development will be performed in the cloud, enabling this kind of analysis in a variety of organizations. The recent emergence of powerful cloud-based IDEs hints at this future. Nonetheless, the value of improving the usability of APIs must be balanced with developers' needs for privacy and ability to experiment freely.

Our existing tools could be enhanced in a variety of ways. One way is to support more patterns of API change, beyond simple method changes. Another way is to support more languages than Java. Using more frequent snapshots (say, automatically snapshotting a developer's code every second) holds the promise to theoretically enhance our ability to identify API usability problems. However, it also makes analysis more challenging, because more snapshots would contain syntax errors. Reliably performing program analysis in the presence of syntax errors, perhaps leveraging past snapshots of the same code, could substantially improve StopMotion's ability to find syntax errors.

A significant area for improvement of our approach is in the analysis, summarization, and visualization of results. The analysis of the results done here was largely manual, which we found was laborious and requiring expertise in the APIs under inspection.

The analysis presented in this paper also focused exclusively on failures as opposed to entropy. For example, methods that are used often without failure might help illuminate patterns in the cases where we do see failure. For example, we saw multiple cases of issues with differentiation by specificity (`add` vs. `addAll` and `copyFrom` vs. `copyFromUtf8`). Does differentiation by specificity work well in other places, and what makes those designs different?

It also remains difficult to automatically distinguish between true API usability problems and other kinds of code changes. For example, we often found that developers would copy and paste a statement, then replace the statement's method call with a different call. This does not necessarily indicate any problem with the API being used, but instead reflects the frequency of which the API is used among our population of developers.

Likewise, it remains difficult to suss out the "worst" API usability problems. We originally analyzed our results by starting with the more frequent changes first (e.g `absent` to `empty` in `Optional` was the most frequent change), these turned out to not necessarily be the biggest API usability problems; instead, they reflected simply the most common APIs used in the company. We believe that a variety of heuristics could be used to find the most severe API usability problems. For instance, the longer a developer takes to converge on the final, desired API usage, the more time they are wasting, and arguably the more severe the problem. Another heuristic we think is promising is looking at developer API experience; changes from developers who have not used the API before are more likely to reflect actual API usability problems, compared to developers who are already familiar with the API.

REFERENCES

[1] Tim Burks. 2017. I got a golden ticket: What I learned about APIs in my first year at Google. (September 2017). medium.com.
[2] Steven Clarke. 2005. Describing and measuring API usability with the cognitive dimensions. In *Cog. Dimensions of Notations: 10th Ann. Workshop*. 131–132.
[3] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the Conference on Computer & Communications Security*. 73–84.
[4] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2001. Graphviz – open source graph drawing tools. In *International Symposium on Graph Drawing*. Springer, 483–484.
[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 313–324.
[6] Cynthia A Graham, Joseph A Catania, Richard Brand, Tu Duong, and Jesse A Canchola. 2003. Recalling sexual behavior: a methodological analysis of memory recall bias via interview using the diary as the gold standard. *Journal of sex research* 40, 4 (2003), 325–332.
[7] 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.
[8] 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*. ACM, 849–858.
[9] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 935–946.
[10] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E Johnson, and Danny Dig. 2012. Is it dangerous to use version control histories to study source code evolution?. In *European Conference on Object-Oriented Programming*. 79–103.
[11] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on*. IEEE, 5–14.
[12] Jeffrey Stylos, Benjamin Graf, Daniela K Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. 2008. A case study of API redesign for improved usability. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 189–192.
[13] Jeffrey Stylos and Brad A Myers. 2008. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 105–112.