VizAPI: Visualizing Interactions between Java Libraries and Clients

Sruthi Venkatanarayanan*, Jens Dietrich†, Craig Anslow†, and Patrick Lam*
*University of Waterloo
Waterloo, ON, Canada
{s42venkat,patrick.lam}@uwaterloo.ca
†Victoria University of Wellington
Wellington, New Zealand
{jens.dietrich,craig.anslow}@vuw.ac.nz

***Abstract*—Software projects make use of libraries extensively. Libraries make available intended API surfaces—sets of exposed library interfaces that library developers expect clients to use. However, in practice, clients only use small fractions of intended API surfaces of libraries. We have implemented the VizAPI tool, which shows a visualization that includes both static and dynamic interactions between clients, the libraries they use, and those libraries' transitive dependencies (all written in Java). We then present some usage scenarios of VizAPI, targetted at library upgrades. One application, by client developers, is to answer a query about upstream code: will their code be affected by breaking changes in library APIs? Or, library developers can use VizAPI to find out about downstream code: which APIs in their source code are commonly used by clients?**

***Index Terms*—static program analysis, dynamic program analysis, API usage, software evolution, software maintenance**

I. Introduction

Virtually all modern software projects use libraries, driven in part by the ease of use of open source component repositories such as Maven and npm. Library developers design Application Programming Interfaces, or APIs, for their libraries, and clients invoke these APIs. APIs provide clients with methods that can be invoked, fields that can be accessed, classes that can be instantiated, and annotations that can be used. "API surface" denotes the APIs that a library makes available to other code artifacts (clients and other libraries). Myers and Stylos [22], and many others, have advocated for the importance of easy-to-use and maintainable API surfaces.

The number of dependencies used by modern software has exploded, and so has their complexity [2], [13]: deeper, transitive dependencies are now common, and components are upgraded more frequently. Developers increasingly struggle to deal with issues arising from those changes. Issues include: (1) dealing with conflicting versions of the same component ("dependency hell") and dealing with supply chain vulnerabilities of deep dependencies (often notified by bots creating pull requests); (2) new issues around security and resilience of the software supply chains, e.g. problems with changes to commodity components (as in the infamous left-pad incident [6]) and novel attack patterns like typo squatting; and, (3) the use of unnecessary, bloated, and trivial dependencies [1], [24].

The benefits of using libraries (e.g. the ease of including functionality that one is not responsible for maintaining) are thus offset by the issues mentioned above. Let's consider one issue: breaking changes. *Potentially* breaking changes in library APIs are common [8], [23]. However, it is generally unclear and difficult to establish whether a change meets its potential and *actually* breaks a particular client.

In all programming environments that we are familiar with, a client developer's decision to use a particular API (e.g. library method `m()`) is a local decision, taken by the developer on-the-fly. To our knowledge, there are no existing systems that help developers visualize which APIs are used throughout an entire project. We claim that such a visualization has potential applications for both client and library developers.

We first define the terms "client", "library" and "dependency", which we use throughout this paper. A "client" is a software component which directly uses some functionality of an external component, which is the "library". Any external component that the "library" directly uses is a "dependency".

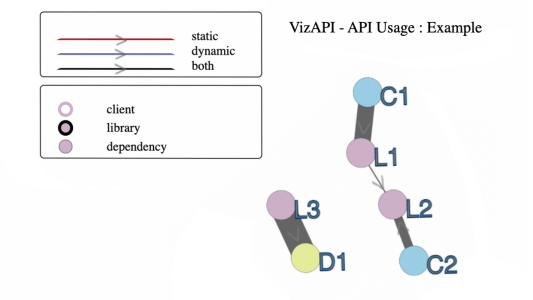


Fig. 1: VizAPI visualization of client C which calls library L, itself dependent on dependency D.

Figure 1 illustrates a VizAPI usage scenario, from the perspective of a client developer worried about breaking changes from the library. It shows plain Java client C (blue nodes) and library L (purple nodes). Library L has packages L_1, L_2, and L_3. C calls into L_1 and L_2. Internally, within L, L_1 and L_2 call into each other, but not into L_3. The VizAPI result, with no edges from C directly to L_3, allows a developer to conclude that breaking changes in L_3 will not affect C. Also, if only L_3 uses an external dependency D (yellow node), then C will not need D to be on its classpath.

More generally, from a library developer's point of view, under plain Java (i.e. no runtime containers) and considering reflection, the potential API surface of any component is huge. Essentially: every method can be called, and every field can

be read and written. Even considering only the published API surface (methods, fields, classes, and annotations with the correct visibility modifiers), libraries' API surfaces still often have hundreds to thousands of members. The breadth of the API surface is a liability with respect to continued maintenance of the library; many developers aspire to avoid breaking changes by preserving, whenever possible, library behaviour that is depended on by clients. Knowing that few clients use a particular API could liberate library developers.

On the other hand, we would expect (and have verified in our unpublished work) that each client uses only a small portion of each of its dependencies' API surfaces. Consider breaking changes again. GitHub provides the Dependabot tool [21], which monitors for upstream changes and automatically proposes pull requests to update dependency versions. That tool may well pull in breaking changes. However, we hypothesize that, most of the time, most breaking changes will not affect most clients; it is useful for clients to know whether they are using the parts of the API surface that are subject to a particular breaking change. A client with broad dependencies on a library (uses a larger fraction of its API surface) is more likely to be affected by its changes than a client with narrow dependencies (smaller fraction). A narrow library dependency would also suggest that it would be easier to swap the library for a functionally similar replacement.

Additionally, as researchers, we would like to understand how library APIs are used by clients more generally. Zhong and Mei [25] investigated API usages in a dataset of 7 experimental subjects (clients) and the libraries that they depend on. They found that clients use less than 10% of the declared APIs in libraries. Our visualization allows developers and researchers to visualize distribution information about how different parts of clients use different parts of libraries.

This paper presents the VizAPI tool, which shows visualization overviews showing API usages—from clients to libraries, but also between libraries (including transitive dependencies). The goal of VizAPI is to provide a heuristic for developers considering the impacts of changes to libraries. VizAPI incorporates information from static and dynamic analyses. We have made VizAPI publicly available[1], although it is still in development. The contributions of this paper include:

- *VizAPI*, a tool which presents visualization of API usage information; VizAPI collects static information, instruments Java code to collect dynamic instrumentation information about API uses in practice, and presents all this as a d3 visualization (Section III-A–III-B);
- *Case Study*, a discussion of VizAPI usage scenarios (Section III-C) drawn from a collection of 11 libraries and 38 clients.

II. Related Work

A representative tool from the software visualization literature is CodeSurveyor, by Hawes et al [11], which visualizes large codebases using the analogy of cartographic maps. While it incorporates dependency information into the layout of the map, VizAPI differs from CodeSurveyor in that VizAPI focusses on usage relationships between different modules (e.g. API invocations) using test cases and static analysis to identify relationships between clients and libraries, rather than investigating a particular system, as CodeSurveyor does. Earlier work includes the software cartography project by Kuhn et al [14] and software terrain maps by DeLine [7].

Hejderup and Gousios [12] explore a question which is central to our approach—how well do client tests exercise their dependencies' libraries? The dynamic part of our approach relies on client test suites exercising enough of the dependencies to get valid dynamic results from our analyses. They conclude that a combination of static and dynamic analysis of the client has some chance of detecting breaking changes in its dependencies, and we accordingly use static analysis to supplement our dynamic results.

Call graph visualization is, of course, a well-known technique, as seen e.g. by the Reacher tool [18]. VizAPI also presents static and dynamic call information. However, we designed it to support decisions about library/client interactions: the granularity of nodes is packages (typically it is methods); and the layout is influenced by frequency of interactions.

Our overall goal is to help both client and library developers understand client uses of library code. Clients benefit from sharpened warnings about unsafe upgrades, knowledge that some upgrades are safe, and having reduced attack surfaces. Library upgrades have been investigated by many researchers, including Lam et al [17] and Kula et al [16]. Kula et al found that most software had outdated dependencies, and that software developers disliked being required to constantly upgrade their libraries. Kula et al [15] also developed a tool to visualize changes in dependencies over time—but not how a particular client depends on its libraries. Our VizAPI tool's dependency visualizations will help developers prioritize required upgrades as low-effort or high-effort. Foo et al [10] proposed a static analysis which detected safe upgrades, but could only certify safety for 10% of upgrades; our combined static and dynamic approach presents the developer with more information and enables more upgrades.

Bergel et al [3] propose the GRAPH DSL for software visualizations. That language could generate static representations similar to VizAPI's; however, VizAPI chooses a specific point in the design space, and we argue that this point is useful for helping developers understand potential impacts of upgrades.

III. VizAPI

We next describe the design of VizAPI, including how we collect information and format it for the d3js visualization library. We also present two VizAPI usage scenarios.

A. Design

Our goal is to visualize software component interactions—between clients and libraries, and between libraries and other libraries. Figure 2 summarizes how VizAPI works.

[1] https://github.com/SruthiVenkat/api-visualization-tool

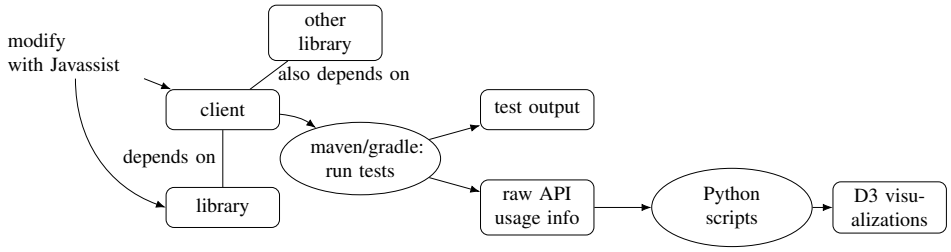


Fig. 2: VizAPI instrumentation workflow. Using Javassist, we analyze and instrument clients and run their test suites. We process the generated data with Python scripts to create D3 visualizations.

To identify interactions across the client/library boundaries, we first inspect JAR files of each software component to obtain a list of classes for every component. We then associate classes and their members to components based on these lists.

a) Static information: We perform a static analysis to record API uses. Using Javassist [5], we identify type references, which includes references at call sites, fields, annotations for classes, methods and fields, method parameters and casts. We also identify subtypes, determining possible interactions across client/library boundaries and library/library boundaries. Javassist uses a call graph produced by class hierarchy analysis to resolve method calls.

b) Dynamic information: We collect dynamic data by running client test suites under instrumentation. The instrumentation records API uses which cross client/library boundaries, as well as library/library boundaries for libraries that are transitively used. We also use Javassist to perform this instrumentation and then use the build system of each project (Maven or Gradle) to run its tests.

At every invoke instruction in every loaded method which transfers control between the client and the library, we add code to record that invoke by incrementing a counter. We handle both static and virtual (including special, virtual, interface, and dynamic) calls. Crossing the client/library boundary includes callbacks from the library to the client as well as conventional calls from the client to the library.

We also record field accesses (direct and reflective), dynamic proxies and reflective calls, Java annotations, implementations, instantiations, and casts.

B. Visualization System

Once we have generated data, we use a modified version of the d3graph[2] library in Python to generate a d3js[3] visualization. The graph in Figure 3a is an example of a graph produced by VizAPI.

VizAPI graphs are force-directed graphs based on the frequency of interactions between different software components. Each node is a set of one or more packages that belong to the same JAR. There are three categories of nodes: clients are represented by nodes with white interiors; libraries by nodes with filled interiors and black borders; and dependencies (called by libraries but not clients) by nodes with filled interiors and normal borders. We coalesce nodes if they originate from the same JAR and have the same incoming and outgoing edges.

Each edge is directed from the source package(s) to the target package(s) and represents an interaction (invocations, fields, annotations, subtyping) between packages. The thickness of each edge reflects the frequency of interactions between the source and the target. Double-clicking on a node emphasizes its direct interactions with other packages while fading out the rest of the graph.

We run a Python implementation of the Louvain clustering algorithm [4], and make the clusters visible by colouring nodes based on cluster. This means that the same colour could indicate nodes (of the same category) from the same or different JARs. Hovering on a node shows the list of packages and the JAR that they belong to, formatted as "jar : ⟨space separated list of packages⟩".

C. Case Study

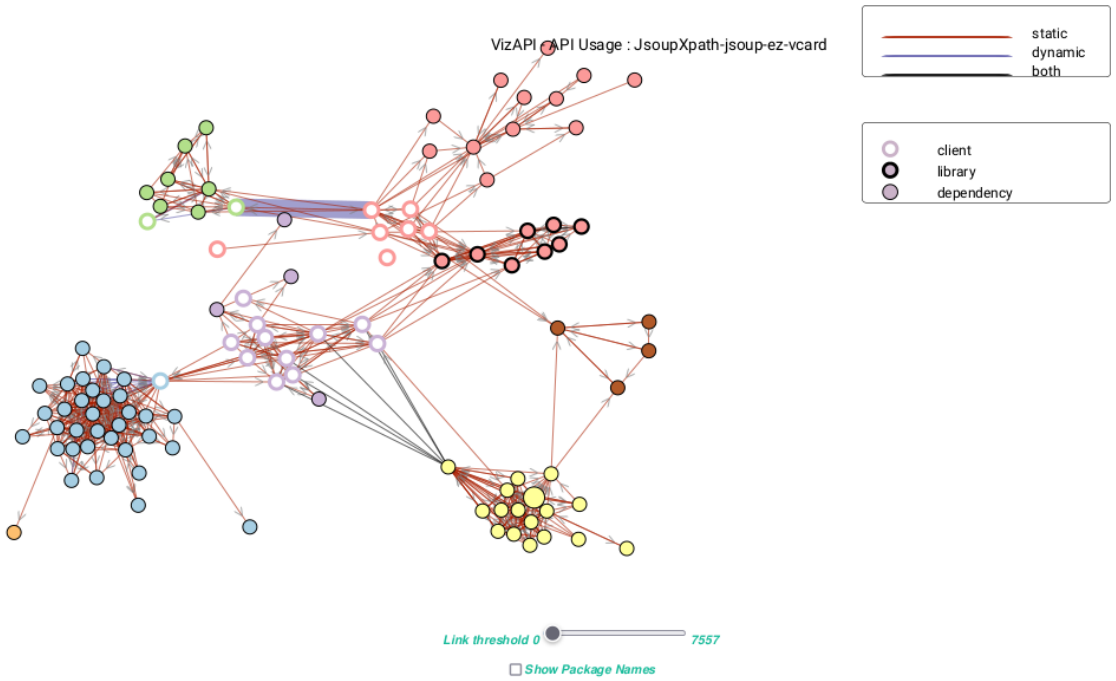
We conducted a pilot study of VizAPI. We have generated data from libraries from a subset of the Duets benchmarks [9] combined with a selection of popular Maven repositories in different categories such as logging and json parsing. Our study included 10 libraries and selected clients of these libraries (potentially overlapping), for a total of 85 projects. We have made our data publicly available[4].

We chose clients partly from popular Github repositories and partly from Duets. We have collected both static and dynamic data for these projects, and we are in a position to generate graphs for combinations of clients and libraries in these projects. We present two usage scenarios below; graphs for our usage scenarios are publicly available.[5] We intend for these usage scenarios to show how VizAPI can be useful to client developers when they want to observe library API usage and for library developers when they want to observe how their library is used by clients.

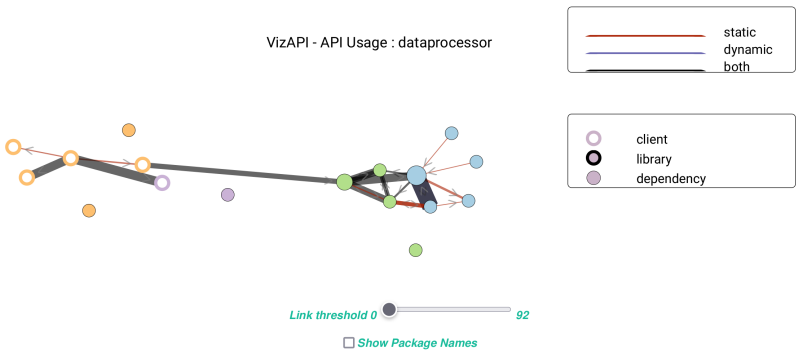
a) Usage Scenario 1: jsoup: Imagine that we are a jsoup developer and want to understand how some clients interact with it, in anticipation of making some breaking changes. We choose clients JsoupXpath[6] and ez-vcard[7]. Figure 3a shows

[2]https://pypi.org/project/d3graph/
[3]https://d3js.org/

[4]https://zenodo.org/record/6951140
[5]https://sruthivenkat.github.io/VizAPI-graph/
[6]https://github.com/zhegexiaohuozi/JsoupXpath
[7]https://github.com/mangstadt/ez-vcard



(a) Usage Scenario 1: Library *jsoup* (pink with dark borders), called by two clients, *ez-vcard* (hollow with purple border) and *JsoupXpath* (hollow with pink border). Exploration shows that internal jsoup packages aren't called directly by clients.



(b) Usage Scenario 2: Client *dataprocessor* (hollow, orange border) calls only one package in library *fastjson* (green fill).

Fig. 3: VizAPI Usage Scenarios.

static and dynamic interactions of the 2 clients with the jsoup[8] library. Recall that nodes represent packages and edges represent interactions (usually invocations) between packages.

We can start our exploration with the cluster of pink nodes. Many of these nodes belong to either JsoupXpath or jsoup. Hovering over a node tells us the package names while double-clicking shows us its direct interactions. (To search for a package, we can click on "show package names" and use the browser's find functionality.) Here, client JsoupXpath calls directly into `org.jsoup.nodes` and `org.jsoup.select`. Notably, and as we might expect, we can see that `org.jsoup.helper` and `org.jsoup.internal` aren't called directly by JsoupXpath. This would mean that breaking changes in `org.jsoup.helper` or `org.jsoup.internal` wouldn't directly affect JsoupXpath[9]

Similarly, ez-vcard, which belongs to the purple cluster in Figure 3a, directly calls into `org.jsoup`. ez-vcard also calls into jackson-core[10] and jackson-databind[11], which are very tightly coupled amongst their own packages and with each

[8]https://github.com/jhy/jsoup

[9]As a specific example, the retraction of an internal jsoup API would not break this client. Behavioural changes that are directly passed through to the external API, e.g. through delegation, can still break clients, but we can consider those to be changes in the external API.

[10]https://github.com/FasterXML/jackson-core

[11]https://github.com/FasterXML/jackson-databind

other. As a jsoup developer, we wouuld be indifferent; others, however, can observe that breaking changes in jackson-core and jackson-databind could propagate.

b) Usage Scenario 2: dataprocessor: Figure 3b presents a second usage scenario. Here, say we are the developers of client dataprocessor[12] (hollow with orange border). This client uses the fastjson[13] library (green fill). Our visualization shows calls only from dataprocessor package `com.github.dataprocessor.slice`, which is the orange-bordered client node (identity of the package available by hovering) to the package `com.alibaba.fastjson`. No other parts of dataprocessor use fastjson. This means that when we, as dataprocessor developers, need to upgrade the fastjson version, we only need to inspect the source code in our `com.github.dataprocessor.slice` package.

Note also the disconnected nodes in Figure 3b. These are all packages of fastjson that are not used by dataprocessor: any breaking changes in these packages definitely do not directly affect dataprocessor, and are less likely to affect it overall than packages that are directly used.

IV. Discussion

Our goal when developing VizAPI was to enable 1) library developers to make better decisions about pruning or modifying unused APIs and to refactor their libraries; and 2) client developers to make better decisions about library upgrades and breaking changes.

Note that client tests may not adequately represent actual client behaviours; however, our use of both static and dynamic information addresses this issue. Specifically, because we use class hierarchy analysis for our static analysis, our visualization will present all possible static calls—possibly too many. That is, the main hazard with static analysis is that our visualization may include more static edges than are actually possible. Some of those edges could be ruled out by a more precise call graph. Reflection aside, no static edges are missing (our approach is "soundy" [19] with respect to static information). On the other hand, dynamic edges have actually been observed on some execution; better tests could yield more dynamic edges. But even if a dynamic edge is missing, there will be a static edge if the behaviour is possible.

This preliminary work has presented two usage scenarios which promise to be useful for both client and library developers. We intend to carry out further user evaluations of our tool following existing techniques [20]; in particular, we aspire to perform experiments to establish the effectiveness of VizAPI, where we ask users to perform software understanding and maintenance tasks that would benefit from our tool.

Acknowledgements. This work was partially supported by Canada's Natural Science and Engineering Research Council.

[12]https://github.com/dadiyang/dataprocessor

[13]https://github.com/alibaba/fastjson

References

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *FSE*, pages 385–395, 2017.

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The Maven dependency graph: a temporal graph-based representation of Maven Central. In *MSR*, pages 344–348, 2019.

[3] Alexandre Bergel, Sergio Maass, Stéphane Ducasse, and Tudor Girba. A domain-specific language for visualizing software dependencies as a graph. In *VISSOFT*, 2014.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[5] Shigeru Chiba. Load-time structural reflection in Java. In *ECOOP*, volume 1850 of *LNCS*, pages 313–336. Springer Verlag, 2000.

[6] Keith Collins. How one programmer broke the internet by deleting a tiny piece of code. *Quartz*, March 2016. https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/. Accessed 19 October 2021.

[7] Robert DeLine. Staying oriented with software terrain maps. In *Proc. Workshop on Visual Languages and Computation*, 2005.

[8] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *WCRE*, pages 64–73. IEEE, 2014.

[9] Thomas Durieux, César Soto-Valero, and Benoit Baudry. DUETS: A dataset of reproducible pairs of java library-clients. In *MSR*, 2021.

[10] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *FSE*, 2018.

[11] Nathan Hawes, Stuart Marshall, and Craig Anslow. CodeSurveyor: Mapping large-space software to aid in code comprehension. In *VISSOFT*, Bremen, Germany, 2015.

[12] Joseph Hejderup and Georgios Gousios. Can we trust tests to automate dependency updates? A case study of Java projects. *J. Syst. Softw.*, 183:111097, 2022. doi:10.1016/j.jss.2021.111097.

[13] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *MSR*, pages 102–112. IEEE, 2017.

[14] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software cartography: thematic software visualization with consistent layout. *J. Software Maintenance and Evolution*, 22(3):191–210, April 2010.

[15] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *VISSOFT*, 2014.

[16] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Softw. Engg.*, 23(1):384–417, feb 2018.

[17] Patrick Lam, Jens Dietrich, and David J. Pearce. Putting the semantics into semantic versioning. In *Onward! Essays*, 2020.

[18] Thomas D. LaToza and Brad A. Myers. Visualizing call graphs. In *VL/HCC*, September 2011.

[19] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, jan 2015.

[20] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. A systematic literature review of software visualization evaluation. *Journal of Systems and Software*, 2018.

[21] Alex Mullans. Keep all your packages up-to-date with dependabot. github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot, June 2020.

[22] Brad A. Myers and Jeffrey Stylos. Improving API usability. *Commun. ACM*, 59(6):62–69, May 2016. doi:10.1145/2896587.

[23] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *SCAM*, pages 215–224. IEEE, 2014.

[24] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering*, 26(3):1–44, 2021.

[25] Hao Zhong and Hong Mei. An empirical study on API usages. *IEEE Transactions on Software Engineering*, 45(4):319–334, December 2019.