July 10, 2023

Dear Drs. Papoutsoglou and Treude,

We are submitting the attached paper, "Design and implementation of the VizAPI tool for visualizing Java static and dynamic analysis results," for publication in the JSS Special Issue on Open Science in Software Engineering Research. The main contributions of this paper are:

- it describes the design and implementation of the artifacts behind the VizAPI tool (Section 3);
- it discusses design decisions behind VizAPI—particularly the relevant tools in the visualization and Java static and dynamic analysis spaces, and how to choose an appropriate tool (Section 4). We particularly hope that this section will be useful to other researchers;
- it describes the benchmark suite that we collected to evaluate VizAPI.

We have made our artifacts available both on zenodo (for archival purposes) and Github (for ease of extensibility). The paper contains all relevant links.

We have attached the VISSOFT NIER paper "VizAPI: Visualizing Interactions between Java Libraries and Clients" to this submission for reference. Conceptually, the VISSOFT paper described the tool from the perspective of a user or researcher, while the present paper describes the tool's implementation. The most significant repeated content is in Section 3.1 and Figure 3, giving examples of VizAPI usage. These account for two pages (out of 12 pages in this submission). There are also a few more paragraphs across Section 3 with repeated content. Some of the material here comes from the first author's MMath thesis, which is not a peer-reviewed publication. We are confident that this paper contains well over 70% (8.4 pages) original content.

Thank you for your consideration.

Sruthi Venkatanarayan, Craig Anslow,
and Patrick Lam

Design and implementation of the VizAPI tool for visualizing Java static and dynamic analysis results

Sruthi Venkatanarayanan[a], Craig Anslow[b], Patrick Lam[a,*]

[a] *University of Waterloo*
200 University Ave W
Waterloo, ON N2L 3G1
Canada
[b] *Victoria University of Wellington*
PO Box 600
Wellington 6140
New Zealand

Abstract

The VizAPI tool shows visualizations for better understanding Java software systems. The data in the visualizations include both static and dynamic interactions between clients, the libraries they use, and those libraries' transitive dependencies, for the portions of these components that are written in Java. Client developers can use VizAPI to answer queries about upstream code: will their code be affected by breaking changes in library APIs? Library developers can use VizAPI to find out about downstream code: which APIs in their source code are commonly used by clients?

This paper focusses on describing the design and implementation of VizAPI, including our approach to static and dynamic analysis using the Javassist library. The goal of this paper is to help future developers of program analysis and visualization tools. We discuss artifact-level issues; discuss viable alternative libraries that we could have used to implement this tool; and explain when different libraries would be a better choice. We then describe how we collected a dataset for VizAPI consisting of 11 libraries and 90 clients, transitively including 4297 components in all. Finally, we suggest potential future uses of VizAPI.

Keywords: tool design, static program analysis, dynamic program analysis, software visualization, benchmark selection

1. Introduction

In previous work, we have presented our VizAPI tool [1, 2], which creates visualization overviews showing API usages—mostly from clients to libraries, but also between libraries (including transitive dependencies). VizAPI provides a heuristic for developers considering the impacts of changes to libraries. VizAPI incorporates information from static and dynamic analyses and uses the D3 visualization toolkit [3] to present this information to developers.

This work aims to explain the VizAPI artifact and to share some of our knowledge in developing software analysis and transformation tools with researchers who would also like to develop new tools for analysis and visualization. We describe the design and implementation of VizAPI (Section 3). We then continue by discussing design alternatives that we could have taken, and especially other libraries that we could have used (Section 4). Dataset construction is challenging, and we explain the dataset that we used to evaluate VizAPI (Section 5). We also outline some potential uses of VizAPI's techniques in future research (Section 6).

2. Artifacts

We have archived our artifacts for the two parts of VizAPI:

*Corresponding author
Email address: patrick.lam@uwaterloo.ca (Patrick Lam)

- Static and dynamic analysis tool to generate data: https://zenodo.org/record/8104759
- Data visualization tool: https://zenodo.org/record/7023911
- Dataset: https://zenodo.org/record/7023872

For those who seek to build on our artifact, we have made both our static and dynamic analysis tool[1] and the visualization tool[2] available as public GitHub repositories, though the version of record is at Zenodo.

3. VizAPI Implementation

We now describe VizAPI, which includes two phases: (1) data extraction and (2) visualization.

The data extraction phase uses static and dynamic analysis to collect data about client usages of libraries. We generate both static and dynamic call graphs, and we also capture other client-library interactions. The static analysis captures the following interactions: vanilla invocations, field accesses, class usages, setAccessible(), annotations and inheritance information. The dynamic analysis captures all dynamic uses of those patterns and additionally reflection, dynamic proxies and service loader bypasses.

The visualization phase transforms the data extracted in the data extraction phase into d3 format by first converting the data into a graph and then using our customized version of the d3graph[3] library in Python to generate d3js[4] visualizations.

In this section, we start by describing VizAPI. Next, we discuss the implementation of the static and dynamic analyses, as well as the visualization tool. Then, in Section 4, we explore alternatives that we could have chosen, including their advantages and drawbacks. The goal of this paper is to produce an explanation of our artifact that will be helpful to other researchers, including in particular an exploration of the design space for program analysis and visualization tools like ours.

3.1. Overview

We briefly give an overview of VizAPI to enable discussions of its design and implementation. The VISSOFT paper [1] and the first author's MMath thesis [2] explain VizAPI in more detail.

We first define the terms "client", "library", "dependency" and "interaction", 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" of that library; components may also have "transitive dependencies". The pattern observed when a client/library boundary is crossed is an "interaction".

VizAPI creates graphs where the nodes are packages that belong to components (clients, libraries and dependencies) and the edges represent the types of interactions between the packages. We create these graphs with the goal of being useful to developers who are thinking about API design questions for their clients and libraries.

Figure 1 illustrates a VizAPI usage scenario, from the perspective of a client developer worried about breaking changes from a new version of a library. It shows 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.

To produce its graphs, VizAPI needs to collect information about interactions between clients, libraries, and dependencies. We provide a few examples of interactions—enough to give the idea of what VizAPI is looking for, but not an exhaustive list.

[1] https://github.com/SruthiVenkat/calls-across-libs
[2] https://github.com/SruthiVenkat/api-visualization-tool
[3] https://pypi.org/project/d3graph/
[4] https://d3js.org/

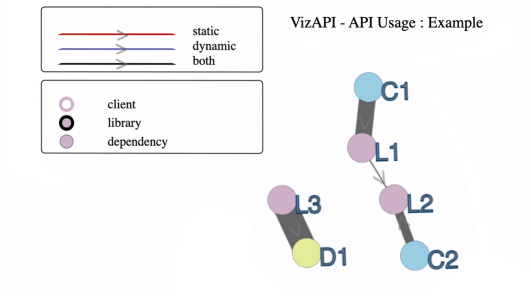


Figure 1: VizAPI visualization of client C which calls library L, itself dependent on dependency D.

Vanilla invocations. The goal here is to capture information about straightforward ("vanilla") method calls between components, in opposition to the more complicated interactions discussed immediately below. This information can be approximated statically using Class Hierarchy Analysis, as we describe below. Furthermore, it can also be captured precisely at runtime, at least for the invocations that happens on a particular set of executions.

Dynamic proxies and reflective calls. We use a dynamic approach for recording calls made using Java dynamic proxies and Java reflection to avoid needing to make hopelessly conservative approximations. The soundiness manifesto [4] points out that many papers in the literature claim soundness and are, in fact, sound for most language features, but exclude highly dynamic features like reflection. VizAPI uses dynamic information and instrumentation to capture certain patterns of reflection and dynamic proxies.

Service Loaders. This Java API allows Java programs to dynamically load additional code, typically plugins. The plugins would declare a published interface, which we can record statically. VizAPI captures uses of the published interface as well as dynamic calls that exceed the public interface.

3.2. Overall Design of VizAPI

We now explain the design of our VizAPI tool, which integrates data from both static and dynamic analysis. We use Javassist [5] for our static analysis, which we discuss in Section 3.3. We also use Javassist to instrument the code so that we can collect dynamic analysis data; we discuss that part in Section 3.4.

Our static and dynamic analysis tool takes in a set of clients and libraries as input. We programatically obtain a list of dependencies for each client and library by invoking Maven, and locate the JARs that Maven downloads in its build output. For each client component and dependency, we ask Maven to create JAR files with non-test code only (e.g. code in "src" directories). This ensures that we do not capture library uses meant solely for unit testing. We then record the extents of components (clients, libraries, and dependencies) by inspecting JAR files of each software component to obtain a list of classes for that component. We associate classes and their members to components based on these lists. This allows us to identify interactions across the client/library boundaries (i.e. between classes that appear on different lists).

Figure 2 summarizes our (static) data capture and (dynamic) instrumentation workflow.

3.3. Static Analysis

It turns out that a straightforward static analysis suffices for our purposes, and we describe parts of it in this section. Section 4 discusses when the use of more sophisticated analyses is appropriate, and how to incorporate such analyses in an analysis pipeline.

For VizAPI, using Javassist, we perform Class Hierarchy Analysis on components and create a static call graph. We implement Class Hierarchy Analysis ourselves. We first read the class files from JARs of clients, libraries, and dependencies (provided by Maven) and we record inheritance relationships between classes. Then, when we walk through the different kinds of interactions, we add an edge between the caller and its potential callees—because we

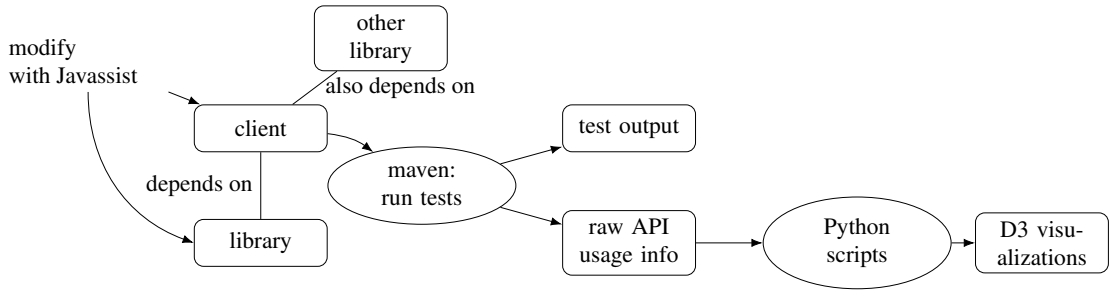


Figure 2: Our instrumentation workflow. Using Javassist, we analyze and instrument clients and run their test suites. (We process the generated data with Python scripts to create D3 visualizations for VizAPI.)

use Class Hierarchy Analysis, the set of potential callees follows from the recorded inheritance relationships. Some of the interactions we record include vanilla invocations, subtyping and annotations.

Vanilla invocations. The standard case is simple. At every invoke instruction in every loaded method which transfers control between a client and a library, we record a static dependency using the class hierarchy analysis call graph that we compute.

Subtyping. We record information about all superclasses and implemented interfaces that cross the library/client barrier.

Annotations. We have a quasi-static approach for finding class, field and method annotations: we observe all annotations for a class or class member when it is loaded, and record cases where a class or member declares an annotation from the library of interest.

3.4. Dynamic Analysis

We collect dynamic data about client usages of libraries by running client test suites under instrumentation. The instrumentation records API uses which cross client/library boundaries, closely mirroring the API usage patterns in [2]. We first describe how we capture specific dynamic interactions between components, and then discuss our instrumentation implementation.

Vanilla invocations. At every invoke instruction in every loaded method which transfers control between a client and a library, we add code to dynamically record the invoke by incrementing a counter just before the invoke executes. Note that we handle all Java invocation types, including virtual and even dynamic invokes. Crossing the client/library boundary includes conventional calls from the client to the library as well as callbacks from the library to the client. Because we instrument libraries, we can record both invocation directions.

Dynamic proxies and reflective calls. Our instrumentation intercepts calls to `java.lang.reflect.Method.invoke()`, a distinguished method used to invoke dynamic proxies and reflective calls, recording details of the calls that we intercept at runtime. It is possible for static analyses to resolve a subset of such calls using clever techniques [6]. Our static analysis machinery does not provide enough information to statically resolve this dependency, but we have full visibility on calls that actually happen dynamically.

Service Loaders. Before the instrumentation, we record a list of services and their implementations by statically inspecting files in `src/main/resources/META-INF/services`. This gives a static list of dependencies; anything beyond this list that we observe dynamically would be a service bypass. In the dynamic phase, we use the statically recorded information to dynamically detect service bypasses which are direct uses of service implementation classes in clients, either through instantiations, casts or reflection. To do so, we intercept calls to method `load()` in classes with name `Service*Loader` and record any calls to methods beyond the published interface that we have recorded statically.

How we implemented instrumentation. In the dynamic analysis phase, we modify the build system of each client (we chose clients that all use Maven) to run its provided unit tests on instrumented code. Asking the modified build system to run the client's tests gives us dynamic call graphs as a side-effect output.

Specifically, using the Java instrumentation APIs and getting them to call Javassist APIs that modify bytecode, we instrument classes and their members to increment counters for every interaction source with a destination lying outside the JAR that it belongs to. Our system creates code to carry out instrumentation and places it into a Java agent JAR. The agent is then passed as an argument to Maven when running client unit tests, which attaches it to the JVM. Once classes are loaded, the agent modifies the bytecode and it is then executed. The code that we insert during the instrumentation phase keeps track of interactions and counters, and writes them out upon program termination.

While our current benchmark set consists of 101 projects, it is possible to run VizAPI (both the data extraction and visualization phases) on new libraries and clients. However, when a library developer wishes to use VizAPI, they are required to provide a specific set of clients that they wish to observe as input to the tools. Developers can use libraries.io to find popular clients of libraries—it provides a dependency tree based on projects' packaging information.

3.5. Visualizations

Once we have generated static and dynamic analysis data using our tool, we use a modified version of the d3graph[5] library in Python to generate a d3js[6] visualization. The modifications that we made to the d3graph library in Python include styling changes (for example, changing node styles based on whether it is a client, library or dependency), legends, and a toggle to show all package names. The graphs in Figures 1, 3a, and 3b are examples of graphs produced by VizAPI.

VizAPI graphs are force-directed graphs based on the frequency of interactions between different software components. Here, frequency of interactions indicates the sum of number of times different kinds of interactions occur. 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 (e.g. invocations, fields, annotations, subtyping) between packages. The colour of an edge indicates whether it was observed statically, dynamically, or both. 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. To search for a package, the user can click on "show package names" and use the browser's find functionality. The "Link threshold" slider allows the user to form new clusters based on frequencies of interactions.

We run a Python implementation of the Louvain clustering algorithm [7], and make the clusters visible by colouring nodes based on cluster. Nodes of the same cluster may come 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⟩". Hovering on an edge shows the type of interaction (invocations, fields, annotations, subtyping or a combination of these) between nodes.

4. Design Decisions

Having described VizAPI's implementation, we now broaden our attention and discuss possible design alternatives, particularly in terms of toolkits that we could have used.

VizAPI works on Java bytecode. Java was a good choice for us because the ecosystem of Java code available on the Internet is vast, and Java bytecode is amenable to analysis. npm also has a lively ecosystem, but JavaScript is much harder to analyze than Java, in part because it is much more dynamic. It is plausible to us that WebAssembly will take over much of Java's niche in the medium-term future, and it is also amenable to analysis.

[5] `https://pypi.org/project/d3graph/`

[6] `https://d3js.org/`

As mentioned in Section 3, VizAPI presents both static and dynamic information. Static analysis will typically present over-approximations (e.g. both branches of a conditional might be taken), except for under-approximations in special cases like reflection and "eval". VizAPI makes almost maximally coarse over-approximations, for instance by using Class Hierarchy Analysis to approximate method calls. The soundiness manifesto [4] discusses the underapproximations used by typical static analysis in more detail.

On the other hand, dynamic analysis routinely under-approximates possible program behaviours: it reports only the behaviour that is induced by a particular program input. It completely reports that behaviour, though, and sees through constructs like "eval" that are difficult for static analyses to handle soundly. TamiFlex [8] dynamically captures information about (among other things) which classes are loaded at runtime, and feeds that back to the static analysis.

VizAPI integrates static and dynamic information. The approach is still not sound with respect to all potential executions, and it does not integrate dynamic class loading information into the static results, but it does make both static and dynamic information (as captured by the tool) visible to the user.

4.1. Static analysis and program transformation

We next describe potential static analysis and transformation approaches, from low-level bytecode manipulation through to declaratively declaring desired analysis facts. VizAPI extracts static facts from the programs under analysis. VizAPI also carries out some program transformation so that it can collect dynamic facts.

Java bytecode is commonly used by program analysis tools as an input format, since it is widely available and yet preserves the semantics of the original source code. Unless obfuscated, it contains some source-level entities such as classes and methods, with their original names.

Although some source-level information information is present, it is not complete. Original code structure (e.g. structured Abstract Syntax Tree-level source-level loops rather than control-flow graphs) and local variable names may not be present. However, the code structure can be recovered, and local variable names can be inferred using Big Code techniques similar to those used for JavaScript by Raychev et al [9].

Bytecode tools provide varying levels of abstraction. We discuss three bytecode-level tools here, first from the perspective of fact extraction and then from that of code manipulation and creation.

Low-level analyses. The Apache Commons Byte Code Engineering Library[7] (BCEL) is a low-level library that provides a very thin abstraction layer to its user. For instance, it requires users to manually manage constant pool entries and provides verbatim access to stack-based bytecode instructions. In that sense, users need to re-invent the wheel to extract higher-level information.

The ASM library[8] provides many of the same capabilities as BCEL, but also a more robust abstraction layer; e.g. ASM manages the constant pool itself. In addition to an object-oriented ("tree-based") API like that of BCEL, it also provides an event-based API which trades increased performance for decreased flexibility for the user (they must write their analysis code in a certain way).

We chose to use Javassist[9] [5] for VizAPI. It provides more of an abstraction layer than BCEL (which has no constant pool) but fewer abstractions than ASM (Javassist has only one API, not two). For our purposes, any of the libraries would have worked, though BCEL would have required more effort to use. Our application is not performance-sensitive; if it were, the ASM event-based API would be most suitable.

At a higher (non-bytecode) level, the Soot framework [10] provides more support for sophisticated intraprocedural analyses: instead of having to work with stack-based bytecode, it transforms the input bytecode into a typed three-address code, with explicit arguments for instructions. Soot also provides a framework for writing intraprocedural dataflow analyses. At the intraprocedural level, because of the three-address code and the dataflow analysis framework, writing a sophisticated analysis is far easier in Soot than in bytecode-level libraries. On the other hand, transforming the code to three-address code involves more computational overhead.

To provide a concrete example, it is relatively easy to detect method invocations in all of these frameworks. It is more difficult to, for instance, detect method invocations with constant parameters, because the analysis has to reason

[7] `https://commons.apache.org/proper/commons-bcel/`
[8] `https://asm.ow2.io/`
[9] `https://www.javassist.org/`

about the stack contents at the invocation site. Soot's three-address code hides the stack from the user and instead provides local variables for the user to work with. In VizAPI's case, we only need to detect the method invocation, which is easy with any framework.

Low-level transformation. While VizAPI's use case includes separate transformation and execution phases, some tools transform code on-the-fly. All of BCEL, ASM, and Javassist allow their users to transform code just before executing it, in the same Virtual Machine. Soot was not designed to support on-the-fly transformations.

In terms of generating code, the frameworks vary in their API support. ASM and BCEL provide APIs that require the user to specify the exact bytecode instructions to be used. Soot allows the user to provide three-address code instructions by (somewhat awkwardly) constructing them through its API. Javassist provides a "simple Java compiler" which can compile some code fragments into bytecode; the compiler does not support all Java constructs. The ByteBuddy toolkit[10] builds on ASM but supports a more declarative way of specifying code, e.g. "create a method that returns X".

Interprocedural considerations. More sophisticated static analysis can benefit from interprocedural information. For instance, it is useful to know about possible definition points for a local variable, even when that variable was assigned from a method parameter and ultimately from another method.

Unfortunately, in Java, precise interprocedural analysis requires a call graph and pointer analysis information, and, as pointed out by Lhoták when explaining SPARK [11], call graphs and pointer analysis information must be computed in tandem. Getting more precise results than Class Hierarchy Analysis or, perhaps, Rapid Type Analysis [12], requires significant computation, and most analysis users would benefit from using an off-the-shelf implementation. Soot's SPARK toolkit is still used today, and Doop [13] provides more sophisticated options, which we discuss shortly.

Assuming that we have a call graph available, the next challenge in designing a tool is to correctly propagate data interprocedurally. Even with Soot's call graph information, it is still not that easy to do whole-program analyses: it is up to the user to manually propagate information at method invocation sites.

Heros [14] implements the IFDS/IDE framework for interprocedural analysis of Java code. Being interprocedural, it is more complicated to use than the dataflow analysis framework built into Soot. However, it provides a lot of scaffolding that a user would otherwise have to create; the user can focus on the essentials of the analysis.

To our knowledge, all call graph implementations for Java require minutes of startup time, even if the call graph computation itself could be relatively fast. Such analyses would not be appropriate for use at runtime. (But, any runtime analysis could instead use easy-to-collect dynamic information and could roll back unsound assumptions that it might have made).

Also, note that these interprocedural analyses need to know about program entry points. For an application whose execution starts with the `main()` function, this is straightforward. For Web middleware or for a library, this is more complicated, and the user will need to specify entry points. VizAPI's test-based dynamic approach essentially uses all test cases as potential entry points, and it is straightforward to generate a driver that invokes all test cases.

Declarative approaches. So far, we have discussed analysis and transformation approaches that have been primarily imperative, with ByteBuddy a minor exception. By contrast, declarative approaches like Doop are concise and surprisingly powerful. In Doop's declarative approach, it suffices to declare the rules that define the analysis. For instance, this utility rule relates methods that have the same name and descriptor but have different declaring types:

```
.decl SameMethodExceptDeclaringType(m:Method, m0:Method)
SameMethodExceptDeclaringType(m, m0) :-
  Method_SimpleName(m, n),
  Method_SimpleName(m0, n),
  Method_Descriptor(m, d),
  Method_Descriptor(m0, d).
```

In practice, one challenge of using Doop is that it is not obvious to find the relations that Doop provides to users; the documentation is somewhat sparse.

[10] `https://bytebuddy.net/`

In any case, Doop reads the input program and the analysis definition, computes necessary pointer and callgraph information, and feeds the facts to the Soufflé Datalog solver [15]. It is fairly simple to read out the Doop results. Doop does not support program transformation and it would be challenging but not impossible to apply Doop results to drive a subsequent transformation phase. Doop also has substantial runtime cost.

4.2. Dynamic analysis

As discussed in Section 3.4, VizAPI uses Java instrumentation APIs (`java.lang.instrument` interface) and Javassist to transform the code before running the instrumented code as a Java agent to collect dynamic data.

Java provides two different mechanisms for gathering dynamic information: Java agents and the Java Virtual Machine Tool Interface (JVMTI).

Java agents. Java agents, along with Java instrumentation APIs, allow tool builders to modify classes' bytecode at runtime. Multiple libraries, such as ASM and Javassist, provide APIs for bytecode manipulation; having considered the options described above, we chose Javassist for VizAPI. The Java agent is attached to the JVM, which means it shares the heap and the garbage collector with the application being run. Since the agent also runs in the JVM, Java agents are portable between different Virtual Machine implementations. We use a Java agent for VizAPI's dynamic analysis because it allows for straightforward bytecode manipulation and also because our dynamic analysis focuses on API interactions. Memory management, for instance, would be harder to observe from within the Virtual Machine.

JVMTI. JVMTI is a set of native APIs which provide the ability to inspect the JVM in multiple ways, including memory management, thread synchronization, and code manipulation. JVMTI agents are native and do not reside within the JVM. They are thus a good choice for monitoring JVM internals such as memory and garbage collection without introducing overhead from the agent itself. However, JVMTI agents are not portable between JVMs running on different hosts, since JVMTI is a native interface and calls outside the JVM.

4.3. Visualization

It is common for software developers to use visualizations for development aspects but there are no common standard tools. The options are either building one's own, or using existing toolkits [16]. There is a plethora of information visualization tools and toolkits [17, 18] that enable developers to create visualizations from different kinds of data types. Although GraphViz[11] is the dominant free graph layout software, we chose to use d3graph[12], which is a Python library for D3 [3], to present our graphs. We made this choice due to the number of visualization techniques the toolkit supports, the fact that visualizations are deployable to the web, the ease of modifications of the Python code, and its good uptake within the community. There are also other variants of D3 for different purposes such as Vega [19], VegaLite [20], and Plotly[13], among others, with bindings for different languages.

Our visualizations are still mostly-static representation of a single graph. However, d3graph renders to HTML and allows the viewer to interact with the graph by rotating the view, moving, and selecting nodes. It supports multiple different types of graph techniques. We found it useful to be able to change the viewing angle of more complicated graphs. d3graph provided us with expansion possibilities for the future: it is easy to add more interaction capabilities and to capture and display evolution aspects of the data.

4.4. Open design questions

We point out some of the design decisions that we have embedded into VizAPI's design and which could be examined in future work.

- Granularity: We chose Java packages as the level of granularity, and coalesce nodes from the same JAR file and with the same edges. One could have a different coalescing policy. An overly-fine granularity makes graphs that are hard to understand, while an overly-coarse granularity provides no information. Also, some sort of cross-cutting modularity mechanism might make more sense, for instance a filter selecting all classes that use the `Node` class defined in some library.

[11] `https://graphviz.org/`
[12] https://pypi.org/project/d3graph/
[13] https://plotly.com/

- Kinds of dependencies: We have chosen what we believe is a fairly exhaustive set of ways that components can depend on each other. Some of these ways may turn out to be unimportant to developers, and we may have missed others.
- Combining static and dynamic information: Although we display both types of information on the same graph, we currently treat static and dynamic information separately. One could imagine an analysis design where dynamic information feeds into static information à la TamiFlex. (By default, we show both static and dynamic edges; users may hide either static or dynamic information.)

5. Dataset

We believe that information about our dataset collection process and the dataset itself will be useful to other researchers, and thus describe it in this section. Our benchmark set consists of 11 libraries and 90 clients. For libraries, we pick the most popular Maven repositories in different categories such as logging, JSON parsing and databases. While this is still a convenience sample, we aimed for some representativeness in categories and also some overlap (i.e. more than one entry per category).

Table 1 presents our set of libraries. We measured lines of code (kLOC) using SLOCcount[14] and number of classes by building libraries and counting resulting `.classes`. Since VizAPI aims to understand relationships between components, we classified component types as follows. A project uses ServiceLoaders if it has a `META-INF/services` directory and Java 9 modules if it has a `module-info.java` file. A library is an OSGi component if it contains a `MANIFEST.MF` file in its build output[15], and this manifest contains `Export-Package` declarations.

Table 1: Libraries that we investigated for API usage and mis-usage patterns

Library	version	description	non-test kLOC	# classes	Service Loader	Java 9 modules	OSGi
commons-collections4	4.4	data structure implementations	28.9	524			✓
commons-io	2.8.0	IO functionality library	12.6	182			✓
joda-time	2.10.10	date and time handling library	28.9	247			✓
slf4j-api	1.7.9	logging library	1.5	28			✓
jsoup	1.13.1	HTML parser	12.5	249			✓
fastjson	1.2.76	JSON parser/generator	43.6	260	✓		
gson	2.8.8	JSON parser/generator	14.4	182		✓	✓
json	20210307	JSON parser/generator	11.8	27			
jackson-core	2.12.3	JSON parser/generator	27.1	124	✓	✓	
jackson-databind	2.12.3	bindings for JSON parser/generator	68.2	700	✓	✓	
h2	1.4.200	database	147.2	1010	✓		✓

We use the libraries.io dataset[16] to construct a dependency graph and look for the most used upstream components (highest number of other components depends on [any version of] those), and the top downstream components (clients). We exclude any clients that have fewer than 10 stars or fewer than 10 forks on Github. Apart from this, we also pick a subset of projects from the Duets benchmarks [21]. We exclude components that do not contain unit tests, components that use our chosen library only for testing, and components that declare the library as a dependency in their POM file, but do not actually use it. Our benchmark set contains both Maven single module and multi-module components.

We executed each of the clients' test suites to collect data about how the clients use all of their dependencies; our data therefore includes not just interactions between our clients and the 11 libraries sampled, but also "bycatch"—that

[14] `https://dwheeler.com/sloccount/`

[15] Some libraries (for example, *connector-j*) create OSGi metadata during the build, so we look for the metadata in the build output, not in the source.

[16] `https://libraries.io/`

is, other libraries that are also called by the clients ("also depends on" in Figure 2) and the libraries. The total static transitive closure of dependencies of our clients includes 4297 components.

Collecting execution data from programs is more challenging than it seems: downloading software and collecting static numbers is fairly straightforward, but running this software to instrument it involves fixing numerous uninteresting environment glitches which nevertheless block progress—even in the stable environment of a continuous integration system at a large software company, Kerzazi et al [22] found that 17.9% of builds break, and our context is even more challenging, since we use libraries of random provenance. Static information is generally easier to collect than dynamic information: instead of running the code (and thus getting all necessary dependencies), it suffices to make conservative assumptions about the dependencies.

We have made our existing code pipeline (analysis tool and VizAPI) and data publicly available (links in Section 2), and invite readers to execute VizAPI both on the demonstration configuration that we provide and on further inputs of interest, as well as to extend VizAPI. Discussion of the results is beyond the scope of this artifact-focussed paper.

6. Potential Applications

While the main contribution of this paper is to share our experiences about writing program analysis and transformation tools with the hope that they are useful to other researchers, we also briefly outline some potential applications of VizAPI itself. In the future, along with exploring potential applications, we would also like to conduct user studies with developers to evaluate the effectiveness of VizAPI.

In our original paper, we discussed how 1) library developers could use VizAPI to guide their API maintenance decisions—based on actual usage, which APIs were best prunable or modifiable, and where good refactoring candidates might exist, thus helping library users' experiences; and 2) client developers could use VizAPI to guide decisions about when to upgrade their dependencies while avoiding upgrade pains.

An extension of VizAPI could help with the problem of library selection. In Section 4 we discussed considerations for library selection for VizAPI-like tools. An extension to VizAPI could also help general developers pick which libraries would be helpful for their needs—e.g., "projects like yours used libraries X and Y". In particular, VizAPI can surface the specific packages that similar projects used, helping developers understand the functionalities of candidate libraries.

Our static and dynamic analyses record versions of different libraries used. Generalizing our work to surface whether clients use multiple versions of the same library and to analyze repository histories to observe how often clients upgrade library versions could be useful information for library developers in considering when to make releases.

Another area of potential applications would be security; we already identify ServiceLoader bypasses, and could also identify other suspicious patterns. VizAPI would be particularly good for identifying cases where vulnerabilities from dependencies do and do not propagate; lack of an edge from a vulnerability-containing source to a sink in a client would be a hint that the client is unlikely to be vulnerable to the vulnerability. Unexpected edges could also indicate abnormal behaviours that should be further investigated. VizAPI could detect violations of security policies, though it is admittedly not the best tool for that.

More fundamentally, we believe that the way VizAPI combines static and dynamic information could be useful in understanding the limitations of static and dynamic approaches in practice. In Section 4, we talked about conceptual limitations of both approaches. Sui et al [23] have investigated the recall of static call graph construction in practice, but further work in this area would help in terms of having confidence in the analysis results that we compute and rely on.

7. Conclusion

In this work, we have discussed the design and implementation of our VizAPI artifact. VizAPI collects static and dynamic information about interactions between components (clients, libraries, and dependencies) and makes it visible to developers. We hope that our discussions will be of value to future researchers who are looking to design static and dynamic code analysis and visualization tools.

Funding. This work was supported in part by Canada's Natural Science and Engineering Research Council.

References

[1] S. Venkatanarayanan, J. Dietrich, C. Anslow, P. Lam, VizAPI: Visualizing interactions between Java libraries and clients, in: IEEE Working Conference on Software Visualization, Limassol, Cyprus, 2022.

[2] S. Venkatanarayanan, Studying and leveraging API usage patterns, MMath, University of Waterloo, Waterloo, ON, Canada (September 2022).

[3] M. Bostock, V. Ogievetsky, J. Heer, D3 data-driven documents, IEEE Transactions on Visualization and Computer Graphics 17 (12) (2011) 2301–2309. doi:10.1109/TVCG.2011.185.
URL https://doi.org/10.1109/TVCG.2011.185

[4] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, D. Vardoulakis, In defense of soundiness: A manifesto, Commun. ACM 58 (2) (2015) 44–46. doi:10.1145/2644805.
URL https://doi.org/10.1145/2644805

[5] S. Chiba, Load-time structural reflection in Java, in: ECOOP 2000 — Object Oriented Programming, Vol. 1850 of LNCS, Springer Verlag, 2000, pp. 313–336.

[6] A. S. Christensen, A. Møller, M. I. Schwartzbach, Precise analysis of string expressions, in: R. Cousot (Ed.), SAS, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 1–18.

[7] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of statistical mechanics: theory and experiment 2008 (10) (2008) P10008.

[8] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, M. Mezini, Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 241–250. doi:10.1145/1985793.1985827.
URL https://doi.org/10.1145/1985793.1985827

[9] V. Raychev, P. Bielik, M. Vechev, A. Krause, Learning programs from noisy data, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 761–774. doi:10.1145/2837614.2837671.
URL https://doi.org/10.1145/2837614.2837671

[10] P. Lam, E. Bodden, O. Lhoták, L. Hendren, The Soot framework for Java program analysis: a retrospective, in: Cetus Users and Compiler Infrastructure Workshop, Galveston Island, TX, 2011.

[11] O. Lhoták, Spark: A flexible points-to analysis framework for Java, Master's thesis, McGill University (December 2002).

[12] D. F. Bacon, P. F. Sweeney, Fast static analysis of C++ virtual function calls, in: OOPSLA, 1996, pp. 324–341.

[13] M. Bravenboer, Y. Smaragdakis, Strictly declarative specification of sophisticated points-to analyses, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 243–262. doi:10.1145/1640089.1640108.
URL https://doi.org/10.1145/1640089.1640108

[14] E. Bodden, Inter-procedural data-flow analysis with IFDS/IDE and Soot, in: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 3–8. doi:10.1145/2259051.2259052.
URL https://doi.org/10.1145/2259051.2259052

[15] H. Jordan, B. Scholz, P. Subotic, Soufflé: On synthesis of program analyzers, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, Vol. 9780 of Lecture Notes in Computer Science, Springer, 2016, pp. 422–430. doi:10.1007/978-3-319-41540-6_23.
URL https://doi.org/10.1007/978-3-319-41540-6_23

[16] J. Paredes, C. Anslow, F. Maurer, Information visualization for agile software development, in: 2014 Second IEEE Working Conference on Software Visualization, 2014, pp. 157–166. doi:10.1109/VISSOFT.2014.32.

[17] J. Heer, M. Bostock, V. Ogievetsky, A tour through the visualization zoo, Commun. ACM 53 (6) (2010) 59–67. doi:10.1145/1743546.1743567.
URL https://doi.org/10.1145/1743546.1743567

[18] S. Liu, W. Cui, Y. Wu, M. Liu, A survey on information visualization: Recent advances and challenges, Vis. Comput. 30 (12) (2014) 1373–1393. doi:10.1007/s00371-013-0892-3.
URL https://doi.org/10.1007/s00371-013-0892-3

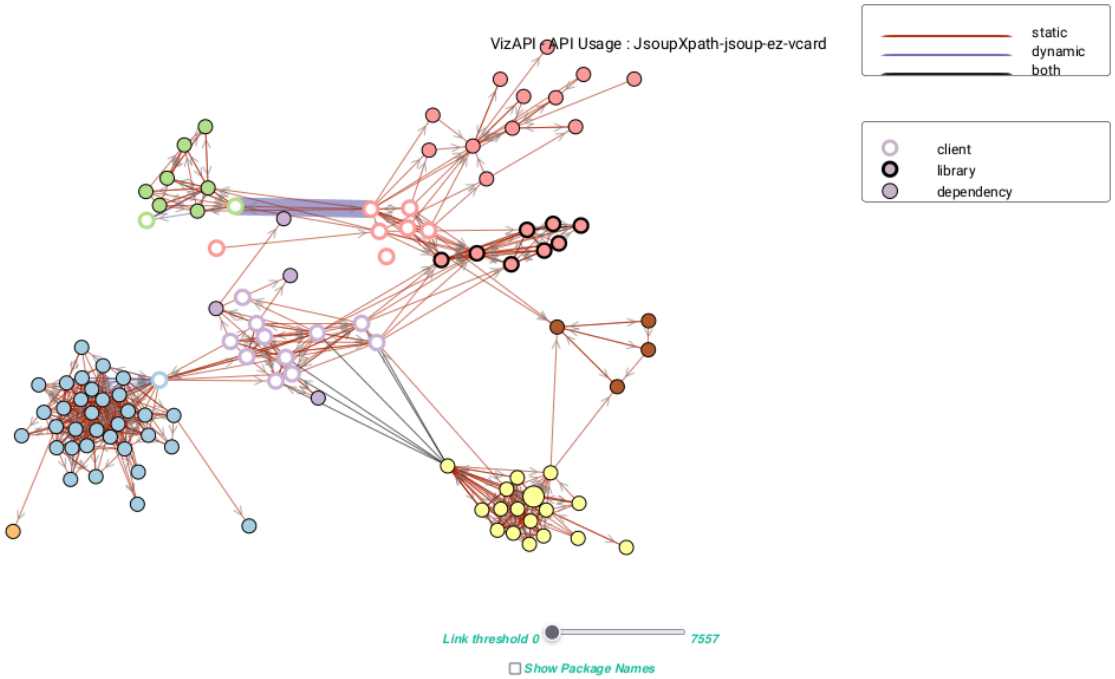
[19] A. Satyanarayan, R. Russell, J. Hoffswell, J. Heer, Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization, IEEE Transactions on Visualization & Computer Graphics (Proc. IEEE InfoVis) (2016). doi:10.1109/TVCG.2015.2467091.
URL http://vis.csail.mit.edu/pubs/reactive-vega

[20] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, J. Heer, Vega-lite: A grammar of interactive graphics, IEEE Transactions on Visualization and Computer Graphics 23 (1) (2017) 341–350. doi:10.1109/TVCG.2016.2599030.
URL https://doi.org/10.1109/TVCG.2016.2599030

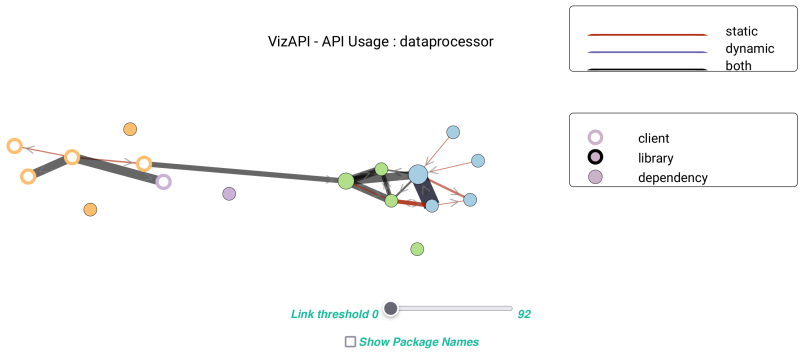
[21] T. Durieux, C. Soto-Valero, B. Baudry, DUETS: A dataset of reproducible pairs of Java library-clients, in: MSR, 2021.

[22] N. Kerzazi, F. Khomh, B. Adams, Why do automated builds break? An empirical study, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 41–50. doi:10.1109/ICSME.2014.26.

[23] L. Sui, J. Dietrich, A. Tahir, G. Fourtounis, On the recall of static call graph construction in practice, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1049–1060. doi:10.1145/3377811.3380441.
URL https://doi.org/10.1145/3377811.3380441



(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).

Figure 3: VizAPI Usage Scenarios.

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".

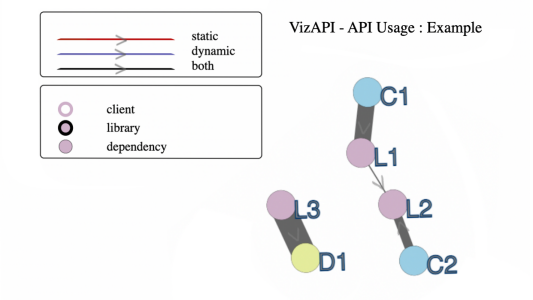


Fig. 1: VizAPI visualization of client C which calls library L, itself dependent on dependency D.

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

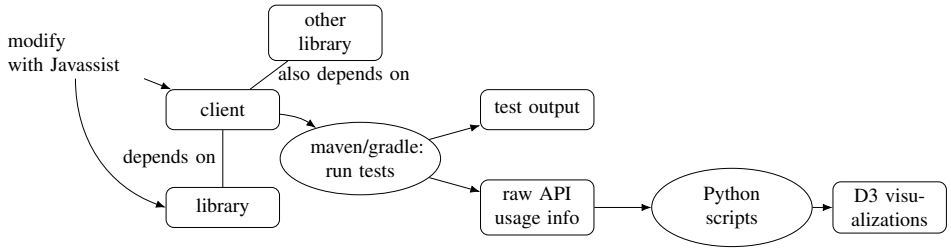


Fig. 2: VizAPI instrumentation workflow. Using Javassist, we analyze and instrument clients and run their test suites. We process the generated data with Python scripts to create D3 visualizations.

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

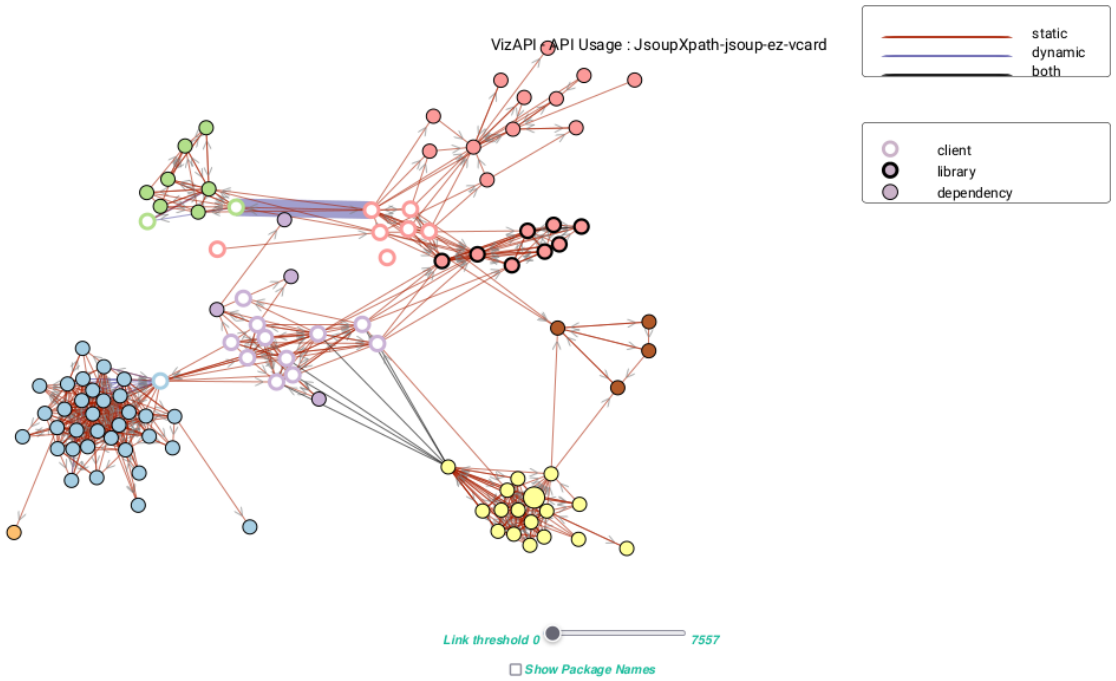
We conducted a pilot study of VizAPI. We have generated data from libraries from a subset of the Duets benchmarks [9] combined with a selection of popular Maven repositories in different categories such as logging and json parsing. Our study included 10 libraries and selected clients of these libraries (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.

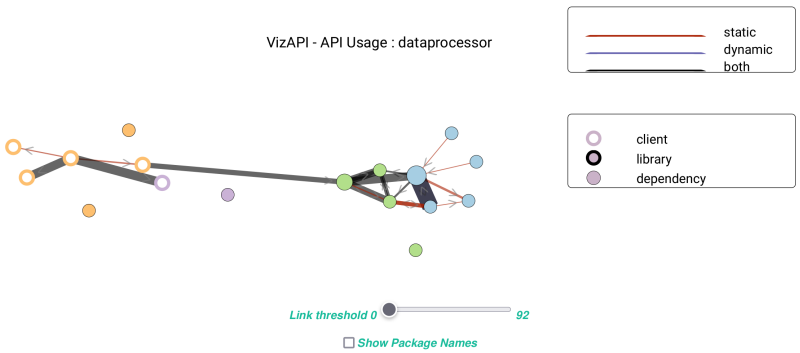
a) Usage Scenario 1: jsoup: Imagine that we are a jsoup developer and want to understand how some clients interact with it, in anticipation of making some breaking changes. We choose clients JsoupXpath[6] and ez-vcard[7]. Figure 3a shows

[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.