Studying and Leveraging API Usage Patterns

by

Sruthi Venkatanarayanan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

Statement of Contributions

This thesis consists mostly of chapters written for conference research paper submission (1 rejected, 1 accepted: VISSOFT 2022 NIER [57]), with some word changes, styling updates and other modifications.

Sruthi Venkatanarayanan was the sole author for Chapter 3, which was written under the supervision of Dr. Patrick Lam. Sruthi was responsible for developing the tool that performs the static and dynamic analyses, carrying out data collection and analysis and developing the VizAPI visualization tool. Sruthi, Dr. Patrick Lam and Dr. Jens Dietrich were co-authors for Chapters 1, 4, 2 and 5.

I understand that my thesis may be made electronically available to the public.

Abstract

Software projects make use of libraries extensively. Libraries have 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. Clients also use libraries in unexpected ways sometimes. Understanding usage patterns of library APIs by clients is beneficial to both client and library developers—targeting issues such as version upgrades, breaking changes and software bloating. We have implemented a tool to study both static and dynamic interactions between clients, the libraries they use, and those libraries' direct dependencies. We use this tool to carry out a detailed study of API usage patterns on 90 clients and 11 libraries. We present a classification framework for developers to classify API uses. We then describe two additional developer-focussed applications of the data that our tool produces: a secondary visualization tool VizAPI, as well as the concept of library fission. Conceivably, VizAPI can allow client and library developers to answer the following queries about the interaction of their code and the libraries they depend on: Will my client code be affected by breaking changes in library APIs? Which APIs in my library's source code are commonly used by clients? The concept of library fission, by which we mean the splitting of libraries into sub-modules, is based on the usage patterns that we observe. This can potentially help library developers release backward compatible versions of their libraries. It could also help client developers isolate breaking changes and reduce the likelihood of vulnerabilities and version conflicts that may be introduced through direct or transitive dependencies.

Acknowledgements

Firstly, I would like to thank Dr. Patrick Lam for patiently and constantly guiding me for the past two years. I have learnt perseverance and optimism from you, and this thesis would not have been possible without your ideas and encouragement.

I would also like to thank Dr. Jens Dietrich, our collaborator from the Victoria University of Wellington, for his invaluable inputs and for his multiple ideas to take this work in new interesting directions.

I am also grateful to Dr. Meiyappan Nagappan and Dr. Michael Godfrey for reading my thesis and providing their insights to refine it.

Dedication

This thesis is dedicated to my family.

Table of Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

The long-term aspiration for software component reuse has finally arrived. This vision—of a component ecosystem enabling ubiquitous reuse and economies of scale—was first proposed over 50 years ago [41] and has finally become reality. Today's applications are largely built from existing components (with one major exception being embedded or safety-critical systems). A key contributor to this shift was the emergence of open source component repositories. Tools such as Maven and npm lower the barriers for using third-party components through automated dependency resolution. Developers can easily include functionality from third-party components in their projects. The size and growth rate of these repositories is staggering.

Virtually all modern software projects use libraries, driven in part by the ease of dependency resolution build tools like Maven and npm. Library developers design Application Programming Interfaces, or APIs, for their libraries, and clients invoke these APIs. APIs typically provide clients with methods that can be invoked, fields that can be accessed, classes that can be instantiated or inherited from, and annotations that can be used. We use the term "API surface" to denote the APIs that a library makes available to other code artifacts (its clients). Myers and Stylos [44], and many others, have advocated for the importance of easy-to-use and maintainable API surfaces.

1.1 Motivation

Reusing functionality provided by third-party components saves time and increases modularity of software. However, there are no silver bullets in software engineering [25]. Component reuse comes with important trade-offs. The number of dependencies used by modern software has exploded, and so has their complexity [30, 4]: deeper, transitive dependencies are now common, components are upgraded more frequently, and developers increasingly struggle to deal with issues arising from those changes, such as: (1) dealing with conflicting versions of the same component (also known as 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, for example, problems with changes to commodity components (as in the infamous left-pad incident [11]) and novel attack patterns like typosquatting; and, (3) the use of unnecessary, bloated, and trivial dependencies [1, 53].

So, components are revolutionary but bring new problems. Let's consider one problem: breaking changes. By breaking change, we mean a change in an external dependency which could break its client, either syntactically or semantically. An example of a syntactic breaking change is the modification of the name of a public method in a library, which will result in a compilation error in any client that calls it. An example of a semantic breaking change is a behaviour change in a library that is undesired by clients. *Potentially* breaking changes in library APIs are common [14, 48]. However, any library change is only potentially breaking; does it *actually* break any particular client? Only if a client uses a specific component API with an incompatible change. Under plain Java (i.e. no runtime containers) and considering reflection, the API surface of any component is huge. Essentially: every method can be called, and every field can be read and written. In the history of Java (and other languages), several constructs enable component developers to better define and enforce the API surface, including access modifiers, modules, and bundles restricting access to packages, and packaging of components that only expose "services", i.e. instantiable classes implementing some abstract type that specifies the service. However, these restrictions always have to compete with the need to provide runtime introspection and code generation features. There are clear benefits in restricting the API surface: such restrictions can facilitate analyses that can calculate whether breaking changes are *likely* to actually break clients. In terms of precision, added restrictions to the API surface would facilitate breakage analyses with fewer false positives (i.e., increased precision).

As a second problem, consider the detection of vulnerabilities in dependencies. Detection is relatively straight-forward: compute the transitive closure of all dependencies, and cross-reference the transitive dependencies with vulnerability databases like CVEs. Tools like *snyk* and *dependabot* are based on this general idea. Some languages and build systems like npm have built-in language-specific support (*npm audit*). The problem is again precision—listing something as a dependency does not mean that all of its functionality is used. So, if dependencies are sufficiently large and deep, a conservative approach inevitably results in false positives. Indeed, Elizalde Zapata et al [19] found that 73% of their studied clients with theoretically vulnerable dependencies were not actually at risk from CVEs in those dependencies, and Chinthanet et al [10] implemented a code-based vulnerability detection tool for Node.js applications. Like the boy who cried wolf, false positives can lead to a potentially devastating impact on application security when true positives start being ignored, as demonstrated in the infamous Equifax incident [39]. Sadowski et al [50], among others, also cite the necessity for low false positive rates in developer tools.

We study software component usage and explore its usefulness in this work. Specifically, our research aims to explore the following questions: is the API surface huge in practice? How much of it is actually used and in what ways is it used? Ought we better control component use? What are some ways to do so?

We look at API surface usage and API bypasses in our study. We also present a classification framework for classifying API uses and present different usage patterns, including expected and unexpected ones.

We also apply our results and identify two applications based on them—VizAPI and library fission. We introduce VizAPI to help library developers, client developers and researchers to visually understand API usage. To mitigate the issue of huge API surfaces and multiple, complex dependencies, we introduce the concept of library fission. By observing how clients use libraries in practice, we can propose splitting libraries into loosely confederated modules. Then, a client can depend on some subset of the library's modules. This has implications both on safe upgrades and on security vulnerabilities. Upgrades are safe if potentially-breaking changes are in unused components. And, security vulnerabilities in unused components are less likely to cause problems in their clients.

1.2 Contributions

The contributions of this work include:

- a tool to record both static and dynamic interactions between clients, the libraries they use, and those libraries' direct dependencies
- a detailed empirical study of API usage patterns on 11 libraries and 90 clients
- VizAPI, a tool which presents visualization of API usage information
- a case study on the concept of library fission

Our tool collects static information and also instruments Java code to collect dynamic instrumentation data about API uses in practice. This includes different patterns of interactions across components, such as vanilla invocations, field usage, annotation usage, subtyping, dynamic proxies, reflective calls and service loaders. The tool records these interactions across the boundaries of clients, libraries and transitive dependencies.

Using the tool's output, we empirically study how developers access libraries in practice, so that we (and others) can develop tools to help developers achieve more stability. Additionally, we present a classification framework to help developers classify different API uses. We conduct our study on a corpus of 11 open-source libraries and 90 clients. We have generated API usage data for this corpus of 101 projects, which we have made publicly available.

Our visualization tool, VizAPI, presents this information about API uses in practice as a `D3.js` visualization, which can be useful to component developers when they want to introspect about their software.

We also apply the API usage information to explore the concept of library fission. We define library fission as splitting up a component into smaller sub-modules. The possible advantages of library fission includes lesser chances of breaking changes in clients of the library, and lesser chances of vulnerabilities and version conflicts introduced through dependencies.

Chapter 2

API Usage Study

We aim to take a snapshot of where current software practices stand: we investigate how APIs are used and misused. Our goal is to investigate API usage patterns, both conceptually and in practice. Such patterns can provide interesting hints for API designers in the future.

We begin by defining two terms that we use in this work, followed by an example. A library's *intended API surface* is the set of classes, methods and fields that it expects clients to use. Other classes, methods and fields are *internal* to the library. Clients using internal parts of the library perform some sort of *bypass*. On the other hand, a client uses an *actual API surface* of the library. In the absence of bypasses, the actual surface is a subset of the declared surface.

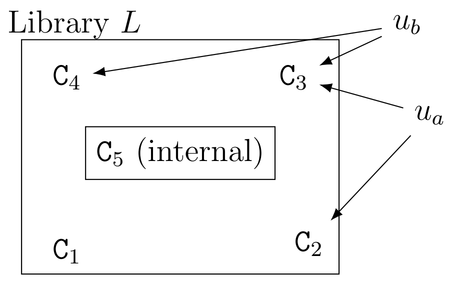


Figure 2.1: API surface of library L, which exports classes $\texttt{C}_1$ through $\texttt{C}_4$. Class $\texttt{C}_5$ is internal and direct calls to it are not allowed. Client u_a uses classes $\texttt{C}_3$ and $\texttt{C}_2$, while client u_b uses classes $\texttt{C}_4$ and $\texttt{C}_3$.

Figure 2.1 illustrates the situation where library L is used by clients (users) u_a and u_b. L's intended API surface includes classes $\texttt{C}_1$ through $\texttt{C}_4$, while class $\texttt{C}_5$ is internal to L, and uses of it involve bypasses. From Figure 2.1, we can see that the actual API surface includes classes $\texttt{C}_2$ through $\texttt{C}_4$, and we don't know about whether $\texttt{C}_1$ is used by any extant client.

2.1 Classification of API Uses

Before we look at API usage patterns, we first present our classification framework for classifying API uses and also to understand whether an API use is a misuse. We intend for this framework to serve as a guideline for component developers who want to understand their software's interactions with other components.

The framework aims to enumerate different ways that a library exposes APIs to clients and to classify their usage. It enables library developers to think about whether the actual APIs being exported and used by clients are intended uses. As an example, a library developer knows that certain public methods are exported as an interface. However, they may not intend to export a specific type as part of the interface, but observe it being used by a client. They can reflect on whether such client uses of their APIs are expected or unexpected based on their original design; this could prove useful when they design new APIs. We hope that this framework can be useful to client developers for comparing and contrasting the design of different libraries with similar functionality.

This conceptual framework outlines four dimensions along which a potential API use can be classified. We phrase the four dimensions as questions; each API usage has an answer for each of the questions. They are as follows: *what*, *how which way* and *why not*?

1. *What* thing is being accessed?

    - types: user declares a subtype of a provider type;
    - classes: user instantiates an object of a provider's class;
    - annotations: user annotates a class or member with a provider-defined annotation;
    - methods: user invokes a method from the provider;
    - fields: user accesses a field defined in the provider; and,
    - casts: user casts to a provider type.

2. *How* is it accessed?
    - direct: user directly names thing being accessed;
    - indirect: (methods only) thing being accessed differs from thing on declared type of the receiver object;
    - reflection/unsafe: user creates a handle to thing being accessed and uses that handle to perform the access;
    - advanced dynamic: none of the above—user accesses thing in some other way, for example, via proxies.
3. *Which way* is the access going?
    - client user to library provider;
    - library user to client provider, via callbacks.
4. *Why not*, i.e. is the user bypassing access control?
    - access modifiers prevent the access, but are overridden;
    - modularity conventions or mechanisms prevent the access: "internal" package, Java 9 modules, or OSGi;
    - service loader restrictions prevent the access.

The "why not" dimension differentiates uses from misuses; an API use that is allowed and expected by the API developer does not have an answer for "why not". Considering intended versus actual API surfaces gives another perspective on the "why not" question. The other dimensions serve to classify the API use or misuse and understand which kinds of uses and misuses are most common. We suggest that it is useful to think of misuse as on a continuum rather than as a strict binary allowed-versus-not-allowed. Some uses are more appropriate than others: when a client passes an object to a serialization library, it is asking the library to access even the private fields of that object, and so this does not constitute a misuse. Calling a deprecated API is less clear-cut. Finally, explicitly calling into internal classes is likely to be a misuse. All bypasses some incur technical debt and add to the project's risk, but some bypasses are more risky than others.

We produce lists for the "what" and "how" dimensions by enumerating different ways a class and its members can be used. Similarly, for "why not", we enumerate access control mechanisms and ways to bypass them. These lists are not exhaustive and can be constantly added to.

2.2 API Usage Patterns

We now discuss API usage patterns with examples. Our examples use the *connector-j* library[1], which enables Java programs to communicate with MySQL databases. Namespaces are omitted for brevity. The standard Java library JDBC types belong to package `java.sql` and MySQL classes to packages `com.mysql.cj.jdbc.*`. Most of the code snippets are not considered best practice, and therefore illustrate API misuses.

2.2.1 Vanilla API Usage

MySQL JDBC driver class `com.mysql.cj.jdbc.Driver` implements the `java.sql.Driver` interface from the standard Java library. This is a case where the user, *connector-j*, is a client of the standard library API. It is not bypassing any access control, directly accessing a *type* from its library, and the access is going from the client to the library.

Shifting perspectives, *connector-j* is intended for use as a library by its own clients. Its `Driver` implementation has a public constructor that can be *directly instantiated* by its clients, as shown in line 1 of Listing 2.1. It turns out that clients are not supposed to instantiate `Driver` classes themselves—JDBC documentation states that, for recent versions of JDBC, clients are supposed to call `DriverManager.getConnection()`. However, there are no compile-time or run-time checks prohibiting the instantiation of this object. We call this a *modularity convention violation*. The part of the library that is being accessed is a class and the access is going from client (shown) to library (`connector-j`). Although direct instantiation is not the intended use of the API, the library must provide a public constructor to enable service discovery.

Continuing with our example, line 2 of the same Listing 2.1 shows a *direct invocation*. Although this call is a virtual method invoke, the declared type and actual types will always coincide in this example. There is no access control bypass here, and this is a direct invocation of a method call from the client to the library. We also observe *indirect invocations*, i.e. virtual invokes where declared and actual differ.

```
1 com.mysql.cj.jdbc.Driver driver = new com.mysql.cj.jdbc.Driver();
2 driver.connect("jdbc:mysql:foo", null);
```

Listing 2.1: Direct Instantiation and Invocation

[1]https://github.com/mysql/mysql-connector-j/, release 8.0.26

Our classification also allows for direct field accesses—from client to library or vice-versa (in the library to a client-provided parameter object). Of course, these may come with or without access control bypasses.

2.2.2 Reflection and Reflective API Bypasses

Java reflection provides an alternate way for users to access APIs, i.e. it is an alternate "how" something is accessed. Reflective accesses exist in practice and are more challenging to analyze statically than vanilla usages, as investigated by Landman et al [35]; however, Bodden et al [7] have presented pragmatic workarounds based on recording dynamic information and using it statically. Clients request a reflective handle to classes, methods (and constructors), and fields, and call certain Java API methods to access the thing behind the handle. Listing 2.2 shows a reflective instantiation of a `Driver`; prior to JDBC 4, this used to be the recommended way of creating a JDBC driver, but has been superceded by using a `DriverManager` to get a connection. We understand that this is not explicitly forbidden, but it is deprecated.

```
Class clazz = Class.forName("com.mysql.cj.jdbc.Driver");
java.sql.Driver driver=(java.sql.Driver)clazz.newInstance();
```

Listing 2.2: Instantiating a `java.sql.Driver` via reflection

Even if constructors, methods and fields and the classes declaring them are not visible, reflection can still be used to instantiate, invoke or access the respective things, subject to permission being granted by security managers. Listing 2.3 shows the use of reflection to bypass access modifiers. While the class `ConnectionImpl` is public, the no-argument constructor has access modifier `protected`. However, reflection allows users to override access restrictions and call that constructor anyway. These are examples of a *reflective API bypass pattern*. This call is thus a reflective-API-bypassing access to a constructor via reflection, from the client to the library. Line 3 makes the constructor accessible, using a pattern known as "deep reflection". Under Java 9, a user must provide a specific parameter to the JVM to allow such calls.

Java has mechanisms for preventing unwanted reflective accesses in general. In addition to preventing deep reflection and providing security managers, Java 9 also displays warnings about illegal reflective accesses to JDK internals. (Reflection can, of course, also be used to make calls that are part of a library's published API.)

```
1 Class clz=Class.forName("com.mysql.cj.jdbc.ConnectionImpl");
2 Constructor constructor = clz.getDeclaredConstructor();
```

```
constructor.setAccessible(true);
Connection conn = (Connection)constructor.newInstance();
```

Listing 2.3: "Deep reflective" instantiation bypassing visibility constraints

At this stage, we'd like to reiterate our goal in investigating API accesses. We've discussed vanilla API usages and reflective accesses to APIs. These two types of accesses have different affordances in terms of program evolution, and we aim to empirically evaluate how developers access libraries in practice, so that we (and others) can develop tools to help developers achieve more stability.

Upon a syntactic breaking upgrade, a vanilla API usage will fail to compile. Reflection is more dynamic—developers have the power to query the system and to adapt to the actual versions of the components that they are interacting with. If they use this power, then their code can be more resilient. However, failures that do arise can be more subtle and severe, these include semantic failures. The same phenomenon is present in Rinard's acceptability-oriented computing [49].

Dynamic techniques like reflection in data binding and persistence mapping pose additional perils. Even minor changes to classes can render data produced with previous versions un-readable. Adding, removing or modifying a field in a class mapped to a relational database table requires altering the table and migrating data. Adding a custom constructor (and thereby removing the default constructor) or removing a setter method may derail a deserialization mechanism based on the Java bean model (for example, `java.bean.XMLDecoder`). Binary serialization has long suffered from this problem.

2.2.3 Restricting API surfaces: Services, Package Names and Package Exports

We mentioned that reflective instantiations of `Driver`s were recommended prior to JDBC 4. Post-JDBC 4, the recommendation is now to use the service loading mechanism to further reduce dependencies of clients on particular drivers. Specifically, the client is to use the `DriverManager` abstraction, which allows multiple drivers to bid on establishing a connection for a given database URL. (The client does not use the service loader directly.)

```
ServiceLoader<java.sql.Driver> services = ServiceLoader.load(java.sql.
    Driver.class);
java.sql.Driver driver = services.iterator().next();
java.sql.Connection conn = driver.connect("jdbc:mysql:foo",null);
```

Listing 2.4: `Driver` instantiation through a service loader

Service loading (described by Fowler in [23]) is now widely used to register and access service implementations, for example, parsers, character sets, JDBC drivers, and JUnit5 extensions. Listing 2.4 shows an example of a client requesting an available `Driver` from a service loader (and not the `DriverManager`) and using that driver to request a jdbc MySQL connection.

Access modifiers are not the only way to protect APIs from being called directly by clients. Java's namespace is segregated by hierarchical package names, but there is no way to allow or deny others access to specific packages (at a package level granularity) in plain Java. The Java standard library does state the modularity convention that some of its implementation (specifically the classes living in the `sun.*` namespace) is not to be used by clients, but this convention is not enforced. Many libraries also use the convention that packages named `internal` are not for use by clients.

Java 9 modules [45] and OSGi [46] offer other ways to delimit an intended API, in terms of a set of exported packages. However, without enforcement, clients can use all of the methods and fields of their libraries.

Enforcing the stated constraints requires additional support—either a Java 9+ runtime or an OSGi container implementation. The Java 9+ compiler and runtime ensure that modules do not access modules to which they do not have access, for example, non-exported modules. Under Java 9+, modules can also rely on having fine-grained control over reflection permissions. Similarly, under an OSGi container implementation, non-exported packages are invisible to other code in the same Java Virtual Machine, and this is enforced using the class loader mechanism.

We run all of our clients and libraries unprotected by Java 9 modules and OSGi containers—this is an opportunity to observe whether developers bypass the stated constraints or not. As we'll see in Section 2.4, they almost universally do not bypass constraints.

By advertising services or exported packages, components define an intended API. Clients should not have any additional compile-time dependencies on the component beyond the intended API: any such additional dependencies become API bypasses. In particular, any of the following is an API bypass if it occurs in client code:

1. instantiation of a type declared in the component but not defined in the intended API;

2. reference to a type declared in the component (`T.class`; casts `(T)`; or uses of annotation types) not defined in the intended API;

3. invocations of method implementations which are not part of the intended APIs, following a downcast from an intended API to a private API; or,

4. accesses to fields not defined in the intended API.

2.2.4 Reversing Directions: Callbacks and Dependency Injection

We think of clients calling libraries. However, callbacks from libraries to clients are common in practice: the client provides an object to the library, and then library calls back a pre-defined method on the client. Callbacks are especially ubiquitous in languages like JavaScript, but do occur in Java code. Fowler [24] uses the term "framework" to describe callback-heavy libraries that practice this Inversion of Control.

JDBC uses callbacks to tell its clients about events that happen to connections, as shown in Listing 2.5. Call sites for callbacks live in the library *connector-j*, and the respective invocations are virtual calls in JDBC resolved at runtime to invocations of methods that live in the client.

```
ConnectionEventListener listener = new ConnectionEventListener() {
    public void connectionClosed(ConnectionEvent event) {}
    public void connectionErrorOccurred(ConnectionEvent event) {}
};
ConnectionPoolDataSource datasource = new MysqlConnectionPoolDataSource()
    ;
PooledConnection conn = datasource.getPooledConnection();
conn.addConnectionEventListener(listener);
```

Listing 2.5: Callbacks in JDBC

Callbacks can be reflective. This is widely used in JSON and XML data binding and object-relational mapping. For instance, the *jackson* object mapper and the XML serializer in the Java standard library can serialize instances of Java classes by exploiting reflection and Java Bean programming patterns. That is, the library accesses non-public fields in the client by discovering getters and invoking them dynamically, rather than reflectively reading the fields. Note the direction of the access—from library to client. This is an example of an API bypass that is not a misuse.

A popular alternative to *jackson* is *guava*. That library instead directly accesses even non-public fields, bypassing access restrictions[2] Note that the general idea of reversing the direction of API access is not restricted to method calls but also extends to fields.

[2]See `UnsafeReflectionAccessor::makeAccessible` in package `com.google.gson.internal.reflect`.

2.2.5 Beyond Reflection

There are several other techniques that allow dynamically using code provided by libraries. Dynamic proxies (`java.lang.reflect.Proxy`) can be used to dynamically subtype existing types, with invocation handlers being used to provide any required method implementations. This mimics protocols like Smalltalk's `doesNotUnderstand`, and was originally used to provide stubs for remote object protocols like RMI and CORBA, and later in some mock object frameworks. However, usage is rather rare [15], and modern remoting and mock testing frameworks (in particular, protocol buffers and mockito) use some form of code generation instead.

An older dynamic mechanism is the `sun.misc.Unsafe` API. Mastrangelo et al [40] pointed out that Java's undocumented `Unsafe` API was extensively used in practice to circumvent the safety properties guaranteed by the Java Virtual Machine. Since then, this API has been migrated to a `jdk.unsupported` module, and Evans [20] noted that many of the features have been migrated into supported Java APIs.

The `invokedynamic` instruction can also be used as an intermediate mechanism to access APIs via custom dispatch. However, the usage patterns defined by the current Java compiler restrict the use to the compilation of lambdas and string concatenation only. There might be more interesting API access patterns in clients compiled with non-Java compilers (such as Kotlin), but this is outside the scope of this study.

In principle, `Unsafe` or its newer equivalents could be used to bypass API access restrictions. However, we believe that the patterns observed in practice are unlikely to correspond to actual violations.

2.3 Methodology

Now that we have looked at some API usage and misuse patterns, we next describe the implementation of our tool which we run on our benchmarks to find these patterns. We also discuss our benchmark selection process in this section.

2.3.1 Static Analysis and Instrumentation

We use Javassist [9] to perform class hierarchy analysis on clients and create a static call graph. We collect 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 that we describe in Section 2.2. We also use Javassist for instrumentation, modifying the build system of each project (Maven) to run instrumented tests and obtain dynamic call graphs.

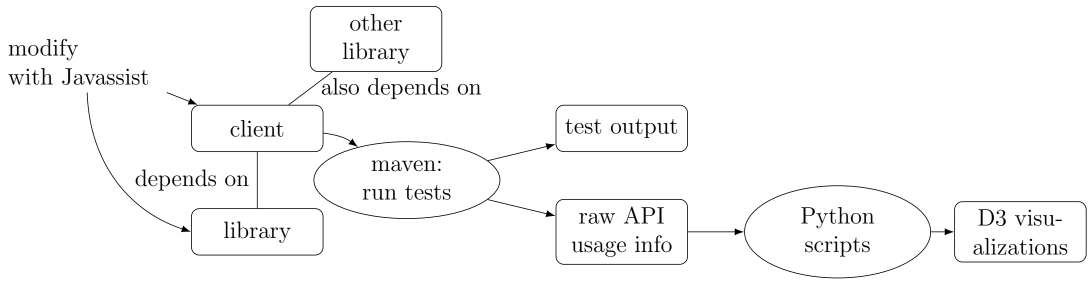


Figure 2.2: Our instrumentation workflow. Using Javassist, we analyze and instrument clients and run their test suites. (We process the generated data with Python scripts to create D3 visualizations for VizAPI.)

Figure 2.2 summarizes our instrumentation and data capture workflow. We next describe our instrumentation implementation in detail.

We identify interactions across the client/library boundaries by inspecting JAR files of each software component to obtain a list of classes for every component. We associate classes and their members to components based on these lists. Since the JAR files contain source code, we ensure that none of the library uses meant solely for unit testing are captured.

Vanilla invocations The standard case is simple. 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 conventional calls from the client to the library as well as callbacks from the library to the client.

Field accesses We capture direct (field sets and gets) and reflective (via invocations of `java.lang.reflect.Field.get()` and `.set()`) field accesses.

Dynamic proxies and reflective calls We specially handle invocations of the distinguished method `java.lang.reflect.Method.invoke()` method used to invoke dynamic proxies and reflective calls, recording details of the calls that we intercept. We identify dynamic proxies by checking whether the invocation of `Method.invoke()` originates from a class that implements `java.lang.reflect.InvocationHandler`. If so, we inspect the call stack to find the caller and callee of `Method.invoke()` and record the call if it crosses the client/library boundary. All other calls to `Method.invoke()` are standard reflective calls, and we record the respective callers and callees. (We also specifically ignore calls to `Method.invoke()` made by the Maven surefire plugin as it runs tests.)

Instrumenting methods also allows us to capture several other library uses, as we describe below.

Class usages We capture reflective uses of the `Class` object by intercepting calls to `java.lang.Class.forName()` and `java.lang.ClassLoader.loadClass()`.

Service Loaders We are particularly interested in bypasses of services that use the `ServiceLoader` API. Before the instrumentation, we record a list of services and their implementations by inspecting files in `src/main/resources/META-INF/services`. With this information, we look for service bypasses which are direct uses of service implementation classes in clients, either through instantiations, casts or reflection. We also intercept calls to method `load()` in classes with name `Service*Loader` and record any calls to methods beyond the published interface.

setAccessible() Java provides the `setAccessible()` method to allow reflective access to class members despite access modifiers. After a call to this method, the program may then (subject to security manager restrictions) reflectively access the class member. We thus record calls to `setAccessible()` along with the previous visibility of the class member.

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. We also record an association between the class and its memers' annotations.

Inheritance and interface implementation At load time, we also record information about all superclasses and implemented interfaces that cross the library/client barrier.

Instantiations and casts We also instrument the `NewExpr` and `Cast` bytecodes to record library/client instantiations and casts.

2.3.2 Benchmark Selection

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. Table 2.1 presents our set of libraries. We measured lines of code (kLOC) using SLOCcount[3] and number of classes by building libraries and counting resulting `.class`es. 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[4], and this manifest contains `Export-Package` declarations.

Table 2.1: 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[5] 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 less than 10 stars or less than 10 forks on Github. Apart from this, we also pick a subset of projects from the Duets benchmarks [18]. We exclude components that do not contain unit tests, components that use our chosen library only for testing and components that

[3]https://dwheeler.com/sloccount/

[4]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.

[5]https://libraries.io/

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 is, other libraries that are also called by the clients ("also depends on" in Figure 2.2) 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 [29] found that 17.9% of builds break, and our context is even more challenging.

While our current benchmark set consists of 101 projects, it is possible to run both our static and dynamic analysis tool and the VizAPI visualization tool on new libraries and clients. However, when either tool is run by a library developer, they are required to provide a specific set of clients that they wish to observe as input to the tools. Libraries.io can be used to find popular clients of libraries—it provides a dependency tree based on projects' packaging information.

We have made our existing data publicly available[6].

2.4 Results

We look for our usage patterns in our benchmark collection of 11 libraries (Table 2.1).

2.4.1 API bypasses

We first investigate the use of API bypasses in our clients. This is useful to library developers if they want to identify which parts of their libraries that are supposed to be internal are actually used by clients. This can help them with API design in future versions.

In Section 2.1, we enumerate three broad categories of bypasses: access modifiers, modularity conventions, and service loaders. We discuss each of them in turn.

[6] https://zenodo.org/record/6951140

Library	Client	Member	Visibility	Count
libthrift	benchmark-thrift	Field	private	1
spring-context	fastjson	Constructor	private	1
javax.servlet-api	fastjson	Field	private	9
spring-context	fastjson	Field	private	22
javax.servlet-api	fastjson	Method	public	17
spring-beans	fastjson	Method	public	11
spring-context	fastjson	Method	public	38
spring-web	fastjson	Method	public	3
jsoup	JsoupXpath	Method	protected	1
rocketmq-common	rocketmq-acl	Field	private	7

Table 2.2: setAccessible Calls—Client to Library

Table 2.3: setAccessible Calls—Library to Client

Library	Client	Fields	Constructors	Methods	private	default	protected	public	total
fastjson	rocketmq-*	159	37	304	158	1	1	340	500
jackson-databind	capitalone.dashboard:core	35	12	0	34	1	0	12	47
jackson-databind	ez-vcard	0	2	0	0	0	0	2	2
jackson-databind	swagger-codegen	117	1	34	0	0	0	152	152
gson	capitalone.dashboard:core	220	24	0	196	9	17	22	244
groovy	logback-classic	0	5	0	0	1	0	4	5
hk2-locator	fastjson	0	3	0	0	0	0	3	3
mockito-all	capitalone.dashboard:core	2	6	0	2	0	0	6	8
jmustache	swagger-codegen	70	0	1	0	0	0	71	71
jaxb-impl	capitalone.dashboard:core	28	0	0	0	0	28	0	28
mirror	vraptor	1	0	25	1	0	0	25	26
commons-lang3	rocketmq-acl	1	0	0	1	0	0	0	1
commons-lang3	rocketmq-client	7	0	0	7	0	0	0	7
cxf-rt-frontend-jaxrs	fastjson	1	0	0	0	0	1	0	1
rocketmq-common	rocketmq-client	28	0	0	27	0	1	0	28
spring-core	capitalone.dashboard:core	628	0	0	625	3	0	0	628

Access modifiers and reflection Table 2.2 shows selected uses of the reflection API `setAccessible()` to enable reflective access from clients to libraries, and Table 2.3 shows selected uses of `setAccessible()` to enable reflective callbacks from libraries back to clients. We can see that accesses to fields, constructors, and methods are all reasonably common, though some codebases only reflectively access fields, while others only access methods.

In our data, 1.9% of the `setAccessible()` calls are used to ensure accessibility of a class member in the same `rocketmq` project version 4.9.1 but in a different module (Maven module). For instance, `rocketmq-acl` makes private field `SendMessageRequestHeaderV2::a` from `rocketmq-common` accessible before accessing it. Such inter-module accesses are more controlled than client/library accesses since they are within the same project, but are still a form of technical debt.

We investigated all `setAccessible()` calls on fields that belong to in *rocketmq*. (*rocketmq* is a client that calls *fastjson*). These calls are made from the library to the client, i.e., accesses are from library *fastjson* to client *rocketmq* using callbacks. We see hundreds of `setAccessible()` calls being executed when the client tests are run. The *rocketmq* source code shows 6 classes with calls to `setAccessible()`: a serialization/deserialization pair for the `CommandCustomHeader` class, 2 methods which appear to print out the state of an object for debugging or logging purposes, and 2 methods which store a reference to a specified field or which call a specified method, provided in those methods' parameters. We would not characterize the first 4 API bypasses as misuses, and it is not obvious that the last 2 are misuses either. Another interesting use of `setAccessible()` on a field is by *benchmark-thrift* to access the field `maxLength_` which is a class member of the class `TFramedTransport` belonging to Apache *thrift*. This is a private field and the default maximum length is overridden by the client.

We found that 23% of the calls to `setAccessible()` were on methods. Of those, only 2 distinct methods that were reflectively invoked were previously not accessible. We manually inspected all of the reflective callers of these 2 methods. Protected method `Node.setParentNode()` in *jsoup* is called from code in `org.seimicrawler.xpath` in *JsoupXpath*. This appears to be test code committed to the main repository. The next method is the default-visibility method `getInstance()`. This method is used to obtain a singleton instance of `ReflectionNavigator` from within the same project. The motivation for developers calling `setAccessible` within the same project is unclear to us, rather than modifying the code themselves. This work can help developers identify such instances and possibly refactor their code.

We do find that some methods are already accessible before being invoked, and some methods are made accessible but never actually invoked.

An API bypass requires a `setAccessible()` call followed by an actual reflective access. As expected, we see a good overlap between the `setAccessible()` calls to methods and reflective invocations. We find that our benchmarks contain reflective invocations of 20 constructors following a call to `setAccessible`, which were not already public. Of the 20 callsites, some are generated by the groovy dynamic language; some are for serialization and some instantiate objects where the constructor had default visibility. All of these reflective constructor invocations are callbacks from the library to the client: the library is providing a factory method to create instances of a client type that it has been supplied with. This appears to be an acceptable API use. Tables 2.4 and 2.5 show how many reflective usages are actually made.

Table 2.4 shows some of our data on reflective field accesses. We observe that many of these reflective field accesses are for serialization. We also observe that in the case of fields, there are only a few calls that access fields that were previously not accessible.

Table 2.5 shows the reflective callback API usage pattern. We see that most accesses in this pattern are made on public methods. We examined the list of methods that are accessed using this usage pattern and this pattern is popular for logging, serialization and deserialization. We also observe a lot of getters and setters accessed this way.

Containers, modules, and modularity conventions For OSGi, none of our libraries are used by our clients in the context of OSGi containers, so the clients are free to violate OSGi access control. Our results show that, even though they can, they almost universally do not. The sole exception is a pair of calls from client `xsoup` to internal class `StringUtil` of library `jsoup`; the calling class in `xsoup` was copied from `jsoup` and still uses its internal helper functions. These calls violate both modularity conventions and OSGi export declarations. When we found these calls, we submitted a pull request[7] duplicating the `jsoup` methods into `xsoup`, and it was quickly merged, showing that the `xsoup` developer was not in favour of violating modularity conventions.

Similarly, although we have Java 8 clients which can violate the unenforced module export rules of Java 9 libraries (they run in environments that don't enforce the rules), none of the clients do so. We believe that the most likely explanation is that such modularity violations are uncommon; however, it is also possible that Java 9 module definitions are too permissive and do not prevent calls that should be prevented.

[7]https://github.com/code4craft/xsoup/pull/53

Library	Client	default	private	protected	public	total
vraptor	mirror	0	1	0	0	1
dble	gson	0	0	12	0	12
capitalone.dashboard:core	jackson-databind	0	26	0	0	26
capitalone.dashboard:core	gson	3	142	0	0	145
capitalone.dashboard:core	jaxb-impl	0	0	4	0	4
capitalone.dashboard:core	mockito-all	0	4	0	0	4
capitalone.dashboard:core	spring-beans	1	21	0	0	22
capitalone.dashboard:core	spring-core	2	157	0	0	159
benchmark-thrift	jcommander	0	5	0	0	5
pagehelper-...-autoconfigure	spring-beans	0	2	0	0	2
MultiChainJavaAPI	gson	21	0	0	0	21
motan-core	hessian	0	0	2	0	2
crushpaper	hibernate-core	0	33	0	0	33
dubbo-cluster	dubbo-common	0	31	0	0	31
dubbo-common	gson	0	20	0	0	20
dubbo-config-api	dubbo-common	0	1	0	0	1
dubbo-metadata-api	gson	0	22	0	0	22
dubbo-registry-api	gson	2	0	0	0	2
rocketmq-broker	commons-lang3	0	8	0	0	8
rocketmq-client	commons-lang3	0	7	0	0	7
rocketmq-client	rocketmq-common	0	27	1	0	28
rocketmq-common	rocketmq-acl	0	7	0	0	7
rocketmq-common	rocketmq-remoting	0	60	0	0	60
rocketmq-remoting	rocketmq-common	0	23	0	0	23
rocketmq-store	rocketmq-common	0	69	0	0	69
libthrift	benchmark-thrift	0	1	0	0	1
mybatis	pagehelper	0	1	0	0	1
thymeleaf	ognl	0	0	0	1	1
thymeleaf	spring-expression	0	0	0	1	1

Table 2.4: Reflection on fields

Table 2.6 shows uses of packages labelled “internal” from outside a given module. In some cases (for example, netty-buffer and netty-common), the uses are across different modules in the same project. We looked at one case, *rocketmq-remoting* and *netty*. In this case, *rocketmq-remoting* uses internal logging infrastructure from *netty* in its NettyLogger module; such a module might be expected to be more closely coupled to its callee than other parts of the code. On the other hand, the usage of *groovy* internals in *rest-assured* would appear to be due to the choice of Groovy as an implementation language, and thus compiler-generated references to internals in the *rest-assured* code.

Service bypasses We find that all clients of libraries that use service loaders bypass the defined services. This is done most commonly through instantiation, but also through casts, subtyping and reflection.

For instance, component *com.h2database:h2* advertises a service (`org.h2.Driver` implementing `java.sql.Driver`), making it a JDBC4-compliant driver. Its clients can obtain connections through the JDBC driver manager, which selects and instantiates instances based on driver URLs. However, client `glowroot.agent` (specifically class `org.glowroot.agent.embedded.util.DataSource`) directly instantiates `org.h2.jdbc.JdbcConnection` and therefore becomes dependent on using the particular database *h2*. This leads to further direct calls to `org.h2.jdbc` APIs being observed. This is a clear bypass of a defined API.

Now consider *fastjson*. This component declares that it provides several services, including three services defined by interfaces in the `javax.ws.rs.ext` package, part of the JEE support for RESTful services. The respective services are implemented in classes in the package `com.alibaba.fastjson.support.jaxrs`. However, looking at clients, we also see calls to APIs provided in *fastjson* APIs outside packages implementing the interfaces declared as services. For instance, `com.alibaba.fastjson.JSON::toJSONString` is called from `org.springframework:spring-web`. This is a legitimate use of a JSON parser via a non-standard API, and in this sense, it is a false positive as we detect it as a API misuse. So *fastjson* could be easily split into two components, one providing an "open API" for the JSON parser functionality, and one implementing and providing the RESTful services. The services would then depend on the open API. This would significantly reduce the overall API surface of fastjson, splitting it into two components, each with well defined functionalities and internal coherence.

We have mainly focussed on "why not", i.e. what mechanisms clients use to bypass API restrictions: reflection, violating conventions, and service loader bypasses, and reported results with respect to "what thing". When relevant, we discussed "how" the bypass occurred, especially for reflection, and we reported the directionality of the bypasses.

Finding 1: Programs are generally well-behaved: reflection is mostly used for serialization not API bypasses; OSGi and Java 9 module definitions are always respected; internal packages are only called sparingly; but service loaders are almost always bypassed.

2.4.2 Extent of API usage

Continuing with our investigation of API surfaces, but moving from misuses to uses, we present results on API use.

Table 2.7 presents the usage distribution of libraries' API elements (methods, fields, and classes subtyped) as covered by client tests. The "total in lib" refers to the number of public and protected members, including static ones. To calculate these totals, we use the latest stable version of each library and consider it to be representative of the versions used by clients. The "distinct used" column counts an element once regardless of how many clients use that element, while "total used" counts an element once per client using it. We identify method calls by the declared type of the receiver object, for example, calls to `o1.f()` and `o2.f()` are the same if `o1` and `o2` have the same declared type. Our uses include both vanilla and reflective uses of API elements.

Library	Client	default	private	protected	public	total
mirror	vraptor	0	0	0	1	1
jboss-el-api_3.0_spec	vraptor	0	0	0	4	4
dble	dble	0	0	0	11	11
fastjson	dble	0	0	0	12	12
junit	dble	0	2	0	0	2
alipay-sdk-java	alipay-sdk-java	0	0	0	63	63
jackson-databind	bandwidth-java-sdk	0	0	0	2	2
fongo	capitalone.dashboard:core	0	0	0	3	3
spring-beans	capitalone.dashboard:core	0	0	0	7	7
spring-core	capitalone.dashboard:core	0	0	0	2	2
spring-data-commons	capitalone.dashboard:core	0	0	0	1	1
mirror	vraptor	3	2	2	13	20
jboss-el-api_3.0_spec	vraptor	0	0	0	4	4
logback-core	logback-classic	0	0	0	13	13
janino	logback-classic	0	0	0	3	3
slf4j-api	logback-classic	0	0	0	1	1
spring-beans	druid	0	0	0	4	4
cxf-core	fastjson	0	0	0	1	1
spring-web	fastjson	0	0	0	1	1
spring-beans	core	0	0	0	7	7
spring-data-commons	core	0	0	0	1	1
freemarker	ez-vcard	0	0	0	64	64
querydsl-core	querydsl-sql	0	0	1	2	3
groovy	json-path	0	10	0	5	15
groovy	rest-assured	0	34	1	141	176
springside-utils	springside-core	0	0	0	1	1
junit	springside-metrics	0	0	0	7	7
jackson-databind	swagger-codegen	0	0	0	24	24
jmustache	swagger-codegen	0	0	0	1	1
fastjson	rocketmq-*	0	0	0	232	232
netty-all	rocketmq-remoting	0	0	0	4	4

Table 2.5: Reflective Callbacks

Table 2.6: Uses of .internal APIs externally

To	From	count
fastjson	jersey-common	1
netty-all	rocketmq-remoting	2
netty-common	lettuce-core	2
netty-common	netty-buffer	4
netty-common	netty-transport	1
json-path	groovy	13
rest-assured	groovy	147
rest-assured-common	json-path	2
rest-assured-common	rest-assured	2
rest-assured-common	groovy	15
jasperreports	ecj	11
rocketmq-remoting	netty-all	9
ecj	jasperreports	6
jersey-common	fastjson	1
jersey-common	hk2-utils	2
kotlin-stdlib	okio	3
jsoup	xsoup	2
mockito-core	ognl	1
mockito-core	spring-expression	1
mongo-java-driver	fongo	3

Table 2.7: Usage Distribution of API Elements by Clients: Method Invocations, Field Accesses, Classes Subtyped and Annotations. Clients almost universally do not use annotations.

	Methods				Fields			
Library	# clients	total in lib	distinct used	total used	clients using	total in lib	distinct used	total used
fastjson	26	1631	59	130	3	512	5	6
commons-collections4	6	3561	759	1731	2	142	9	9
commons-io	9	1257	19	24	2	116	3	3
joda-time	9	3406	58	86	3	202	2	3
gson	17	679	85	176	4	79	10	11
json	4	258	25	34	2	18	1	2
jsoup	4	1032	79	117	0	142	0	0
slf4j-api	89	378	80	1337	0	22	0	0
jackson-databind	23	675	88	115	12	376	42	46
jackson-core	13	1996	296	873	9	611	39	65
h2	3	6712	16	16	1	1699	1	1

	Subtyping			
Library	# clients	total in lib	distinct used	total used
fastjson	2	190	1	2
commons-collections4	3	358	1	1
commons-io	2	163	1	1
joda-time	0	145	0	0
gson	6	60	4	6
json	1	21	1	1
jsoup	3	54	1	1
slf4j-api	9	29	9	25
jackson-databind	10	148	20	35
jackson-core	13	117	12	15
h2	1	662	1	1

We can observe that at most 22% (and on average 9%) of the methods in the API surface are used by clients, with *slf4j-api* having the highest use proportion. Table 2.7 suggests that there is a small overlap between methods used by clients.

We also looked at the *json* library and found that its own tests reach 151 of the 258 API methods, compared to the 25 methods used by our selected clients. One possible reason for the sparsity of API use is that, at least in *json*, over half of the API methods are overloaded, i.e. share a name with some other API method.

Despite standard OO practices calling for fields to be encapsulated, more than half of the libraries have their fields accessed by clients. Only 1 of the *joda-time* and 25 of the *jackson-core* fields are static (so it's not just constants); the rest of the accesses are to instance fields. However, the number of fields used is often in the single digits, the *jackson* libraries being an exception with about 11% of declared fields used. We inspected the use of *jackson* fields and found that these usages mostly exist within the *jackson* libraries themselves. We count calls that happen across the *jackson* libraries as crossing library/client boundaries. An example of these usages is the `WritableTypeId` class belonging to *jackson-core*, which is used often, and usually by *jackson-databind*. `WritableTypeId` is a Java class that acts like a C-style struct; it exposes fields and not methods. It is intended to be used for passing information, and the use of its fields is expected.

About half of the libraries have clients subclassing a single-digit number of library types, again with the exception of *jackson* with dozens of subtypes in clients, which are also often other *jackson* libraries.

We observe only one use of library annotations across all our benchmarks. *crushpaper* uses `com.fasterxml.jackson.databind.annotation.JacksonStdImpl`, which is used for indicating implementation classes and is typically used by serializers and deserializers in *jackson*.

Figure 2.3 presents a boxplot of pairwise Jaccard similarities between the portion of the APIs used by different clients for each library. Two clients u_1 and u_2 of library L which use exactly the same parts of L's API would result in a similarity of 1; more generally, it is

$$\frac{|\text{used.API}_L(u_1) \cap \text{used.API}_L(u_2)|}{|\text{used.API}_L(u_1) \cup \text{used.API}_L(u_2)|}.$$

Although Table 2.7 showed that some of the members overlap, we can see from this Figure that the overall level of overlap is generally less than 10%. We observe a half-dozen client pairs (data points) where there is about 50% overlap between client API uses, including for *fastjson*, *commons-io*, and *json*. Although the mean overlap for *slf4j-api* is near 0, there are also 4 pairs of clients which have more than 50% overlap.

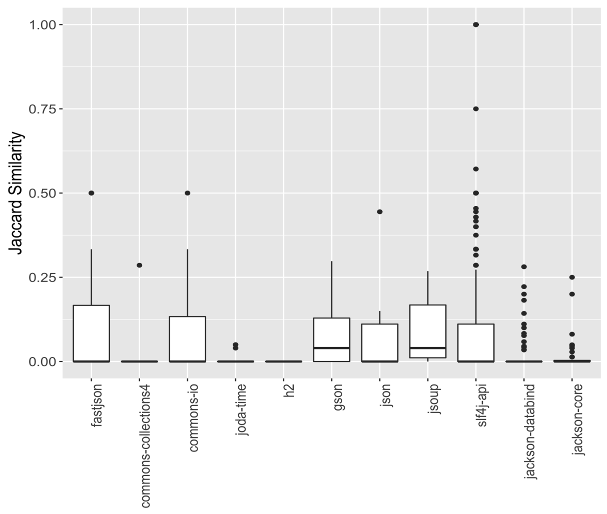


Figure 2.3: Jaccard similarities between clients' API usages

We were not surprised by the generally small amounts of overlap, because the libraries' exposed API surfaces tended to have thousands of exposed elements and hundreds of used elements. This result makes a convincing argument for libraries being fissioned into modules, which we discuss further in Section 3.2.

Table 2.8 presents another view of API overlap between clients. We constructed the largest set of methods shared by more than 1 client of a library ("max-set") and report the size of that set as well as the percentage of clients which use all of that set of methods. So, for *json*, we can see that there is one method called by 3 out of 4 clients, and for *jsoup*, ten methods are all called by 3 out of 4 clients. We characterize the amount of overlap as generally low but not nonexistent: a few methods are repeatedly used.

Finding 2: APIs are sparsely used by clients—mostly methods (9% utilization), but a few fields (4%) and supertypes (6%). There is limited but nonzero overlap between the methods used by different clients.

Library	Total No. of clients	% Clients calling same methods	No. of methods called by (%) clients
fastjson	26	50	1
commons-collections4	6	50	1
commons-io	9	44	1
joda-time	9	56	1
h2	3	0	0
gson	17	41	1
json	4	75	1
jsoup	4	75	10
slf4j-api	89	51	2
jackson-databind	23	26	1
jackson-core	13	46	4

Table 2.8: % of clients calling same methods

Chapter 3

Applications of API Usage Information

In this chapter, we discuss two applications of our results from Chapter 2. The first is the visualization tool VizAPI which uses the results of our static and dynamic analyses tool to create visualizations for developers to understand API usage in their components. The second is library fission, which we also perform using the results of our static and dynamic analyses tool.

3.1 Visualization

We now present the VizAPI tool, which shows visualization overviews depicting API usages—namely, usages of clients by libraries for the most part, but also inter-library usages (which includes usages of transitive dependencies by libraries). The goal of VizAPI is to provide information for developers considering the impacts of changes to libraries.

We have verified that each client uses only a small portion of each of its dependencies' API surfaces. Consider breaking changes again. GitHub provides the Dependabot tool [43], 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.

Our visualization allows developers and researchers to visualize distribution information about how different parts of clients use different parts of libraries. A limitation of this tool is that with larger, complex systems, the visualization graphs become denser and more difficult to interpret. Another limitation is that the tool is highly dependent on the size of the project in terms of time and can take hours to produce graphs. However, we do not intend for this tool to be used regularly as part of a software development cycle. It is only intended to be used when developers want to examine their use of libraries or usage by clients. VizAPI is still in a preliminary phase and in development.

VizAPI incorporates information from static and dynamic analyses. We have made VizAPI publicly available[1]. We have also archived the artifact at https://doi.org/10.5281/zenodo.7023911

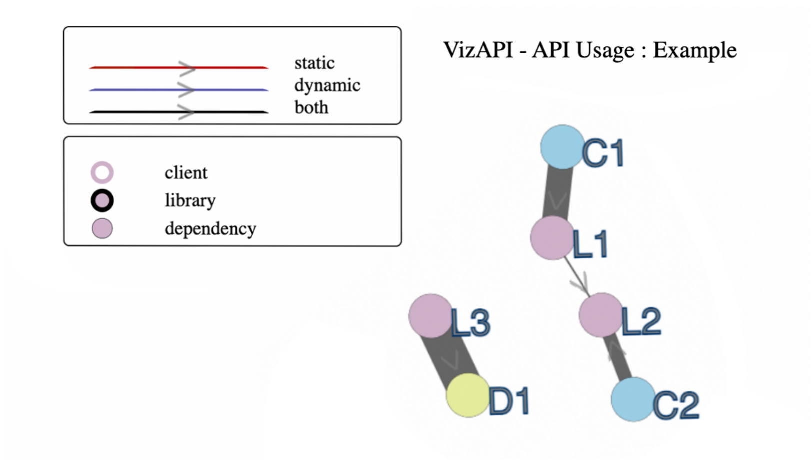


Figure 3.1: An Example VizAPI Visualization

We first define the terms “client”, “library” and “dependency”. 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”.

Figure 3.1 illustrates a possible VizAPI usage scenario, from the perspective of a client developer. Consider a client C (blue nodes) and a library L (purple nodes), in the context of plain Java. 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

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

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 we know that C will not need D to be on its classpath.

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.

3.1.1 Visualization System

Once we have generated data from our tool that runs the static and dynamic analyses, we use a modified version of the d3graph[2] library in Python to generate a d3js[3] visualization. The modifications that we made to the d3graph library in Python in Python include multiple 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 3.1, 3.2a and 3.2b are examples of graphs 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 [6], 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⟩".

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

3.1.2 Case Study

We conducted a pilot study of VizAPI. We have generated data from our benchmarks. 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.[4] 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.

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[5] and ez-vcard[6]. Figure 3.2a shows static and dynamic interactions of the 2 clients with the jsoup[7] 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[8]

Similarly, ez-vcard, which belongs to the purple cluster in Figure 3.2a, directly calls into `org.jsoup`. ez-vcard also calls into jackson-core[9] and jackson-databind[10], which are very tightly coupled amongst their own packages and with each other. As a jsoup developer, we would be indifferent since it does not affect us; others, however, can observe that breaking changes in jackson-core and jackson-databind could propagate.

[4]https://sruthivenkat.github.io/VizAPI-graph/

[5]https://github.com/zhegexiaohuozi/JsoupXpath

[6]https://github.com/mangstadt/ez-vcard

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

[8]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, for example, through delegation, can still break clients, but we can consider those to be changes in the external API.

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

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

Usage Scenario 2: dataprocessor Figure 3.2b presents a second usage scenario. Here, say we are the developers of client dataprocessor[11] (hollow with orange border). This client uses the fastjson[12] 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 and cross-reference against release notes for fastjson.

Note also the disconnected nodes in Figure 3.2b. 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.

3.2 Library Fission

Modern software extensively reuses existing functionality using third-party libraries. Over time, libraries tend to extend their functionality, introducing more features and modifying existing ones. This leads to huge library sizes, a lot of which goes unused by a client that imports it. We have observed that modern API surfaces are vast and include thousands of potential access points, of which only hundreds are used by any typical client.This is called software bloating and there has been work around debloating. Debloating focuses on a single client at a time; it depends on dynamic analysis based on the client's execution. We instead propose library fission, which focuses on splitting libraries based on client usage. This is a more permanent solution to bloating and does not require running analyses on clients for every execution. (We aim to split libraries based on client behaviour in a way that sub-modules of the fissioned library have common functionality.)

We use the data generated by our static and dynamic analyses to experiment with fissioning a subset of our corpus of libraries. On a high level, by library fission, we mean that we separate packages of a library into different Maven submodules. The following are the steps we follow to perform library fission for a given library, L:

1. Filter out all interactions with L, i.e., client uses of L, L's uses of dependencies and intra-library uses of L.

[11]https://github.com/dadiyang/dataprocessor
[12]https://github.com/alibaba/fastjson

2. Run the Louvain clustering algorithm on the filtered data where each node is set to be a package belonging to a jar.
3. Split up L into Maven submodules based on the clusters.
4. Test on a random subset of L's clients.

We now discuss specific examples from our libraries.

3.2.1 *fastjson*

We have 26 clients for *fastjson*. Based on the usages of these clients, we obtained the following initial set of 5 clusters:

- Cluster 1:
  - `com.alibaba.fastjson.annotation`
  - `com.alibaba.fastjson.asm`
  - `com.alibaba.fastjson.parser`
  - `com.alibaba.fastjson.serializer`
  - `com.alibaba.fastjson.support.hsf`
  - `com.alibaba.fastjson.support.config`
  - `com.alibaba.fastjson.support.retrofit`
  - `com.alibaba.fastjson.support.springfox`
- Cluster 2:
  - `com.alibaba.fastjson`
- Cluster 3:
  - `com.alibaba.fastjson.util`
  - `com.alibaba.fastjson.parser.deserializer`
- Cluster 4:
  - `com.alibaba.fastjson.support.spring`

  - `com.alibaba.fastjson.support.spring.annotation`

- Cluster 5:
  - `com.alibaba.fastjson.support.jaxrs`

We observe that the clusters are loosely based on functionality, that is, the packages within a cluster have similar functionality. This is not strictly true for all packages, but it is a good approximation. We see that the `jaxrs` and `spring` functionalities are clusters of their own and it makes sense to separate these into submodules since they provide their own independent functionality. However, we observed that there exist cyclic dependencies between Cluster 1, 2 and 3. If a library developer performs library fission and they run into cyclic dependencies, we recommend that they resolve these dependencies by moving classes into appropriate submodules. Due to our lack of domain knowledge (we don't know about what individual *fastjson* classes do), we combined these 3 clusters into one submodule. Our final submodules are as follows:

- Submodule 1 — utils:
  - `com.alibaba.fastjson.annotation`
  - `com.alibaba.fastjson.asm`
  - `com.alibaba.fastjson.parser`
  - `com.alibaba.fastjson.serializer`
  - `com.alibaba.fastjson.support.hsf`
  - `com.alibaba.fastjson.support.config`
  - `com.alibaba.fastjson.support.retrofit`
  - `com.alibaba.fastjson.support.springfox`
  - `com.alibaba.fastjson`
  - `com.alibaba.fastjson.util`
  - `com.alibaba.fastjson.parser.deserializer`

- Submodule 2 — spring:
  - `com.alibaba.fastjson.support.spring`
  - `com.alibaba.fastjson.support.spring.annotation`

- Submodule 3 — jaxrs:
  - `com.alibaba.fastjson.support.jaxrs`

When we split *fastjson* into submodules, we observed that each submodule needed fewer dependencies in the pom compared to the current version of *fastjson* that is not split. The current version of *fastjson* has 61 dependencies, while *utils* has 47, *spring* has 27 and *jaxrs* has 29. Importing one of these newer submodules means a smaller probability of propagated vulnerabilities, version conflicts and breaking changes.

We sent an email to the *fastjson* contributors proposing a split and did not receive a response, after which we created a pull request[13]. Our PR text was as follows: "This PR splits fastjson into 3 submodules - utils, spring, jaxrs. We're looking into Java API usage as part of research and propose this split. These submodules are based on usage patterns that we've observed clients making of fastjson. It is beneficial to clients – they can import only the submodule they need to use (utils seems to be most popular). After this change, clients will not be affected by breaking changes in unused submodules and can avoid version conflicts and vulnerabilities propagated through dependencies. We believe that this split makes it easier to release backward compatible versions of fastjson and to isolate breaking changes."

We looked for *fastjson*'s vulnerabilites using the "Synk Vulnerability DB" and found the "Unsafe deserialization in com.alibaba:fastjson" vulnerability which affects versions ¡ 1.2.83. Following this, we inspected the source code for the fix and found that the vulnerability originates in the *utils* cluster. Using Github's advanced search, we went through 10 projects that use *fastjson* and found that while 6 of them use *utils* and will continue to be vulnerable, the vulnerability will not be reported for 4 of the projects after fission.

3.2.2 *jsoup*

Based on usage by the 7 clients of jsoup we have in our benchmark set, we observed the following clusters:

- Cluster 1:
  - `org.jsoup.parser`

[13]https://github.com/alibaba/fastjson/pull/4276

  - `org.jsoup.safety`
  - `org.jsoup.helper`
  - `org.jsoup.internal`
- Cluster 2:
  - `org.jsoup`
  - `org.jsoup.examples`
- Cluster 3:
  - `org.jsoup.select`
  - `org.jsoup.nodes`

We observed multiple cyclic dependencies. In the case of cyclic dependencies, it is best for the library developers to split classes in packages and move them to clusters based on functionality.

We now discuss a few interesting things we noted in the *jsoup* clusters. Cluster 1 contains `org.jsoup.internal`. It is clear from the naming that this package is meant for internal use only and not for clients. We suggest that such packages be moved to a separate submodule so that clients are not tempted to import them. Similarly, Cluster 2 contains `org.jsoup.examples` which also almost certainly will not be needed by clients and can also be moved to a separate submodule.

Due to the multiple cyclic dependencies that we observed, we did not raise a pull request for *jsoup*.

3.2.3 *guava*

The *guava* library makes an interesting case for library fission since it provides multiple functionalities such as I/O, caching, collections and so on, all within the same library. Due to this, we see multiple clusters for the library as expected. In this library too, we observe that packages have loosely similar functionality within clusters. Our final submodules for *guava* are as follows:

- Submodule 1 — annotations:

  - `com.google.common.annotations`

- Submodule 2 — eventbus:
  - `com.google.common.eventbus`
- Submodule 3 — reflect:
  - `com.google.common.reflect`
- Submodule 4 — html-xml:
  - `com.google.common.html`
  - `com.google.common.xml`
- Submodule 5
  - `com.google.common.escape`
  - `com.google.common.io`
  - `com.google.common.net`
  - `com.google.common.graph`
- Submodule 6
  - `com.google.common.base`
  - `com.google.common.cache`
  - `com.google.common.collect`
  - `com.google.common.hash`
  - `com.google.common.math`
  - `com.google.common.primitives`
  - `com.google.common.util`

We looked at the "Synk Vulnerability DB" for *guava*. We found two vulnerabilties, "Information Disclosure" which affects the "com.google.common.io" package and "Deserialization of Untrusted Data" which affects the "com.google.common.collect" package. When we combed through projects that depend on *guava* using Github's advanced search, we found multiple projects that depend on the invulnerable packages of *guava* and would potentially benefit from fissioning *guava*, in terms of security vulnerabilities. We observe that interactions with transitive dependencies play a bigger role in deciding whether a vulnerability affects a client, compared to when we actually create clusters for library fission.

We believe that library fission is potentially beneficial to developers. However, we expect developer resistance when it comes to making these changes in their libraries, since they are breaking changes. Indeed, Tran et al [56] looked into improving file layouts of 2 open source systems (Linux and VIM); while this information is not in the cited work, we were told that, while the developers of these systems were appreciative of the changes they had made, they were still reluctant to accept them as they were breaking changes [26]. We believe that the appropriate window to introduce fission changes is when other major breaking changes are being planned.

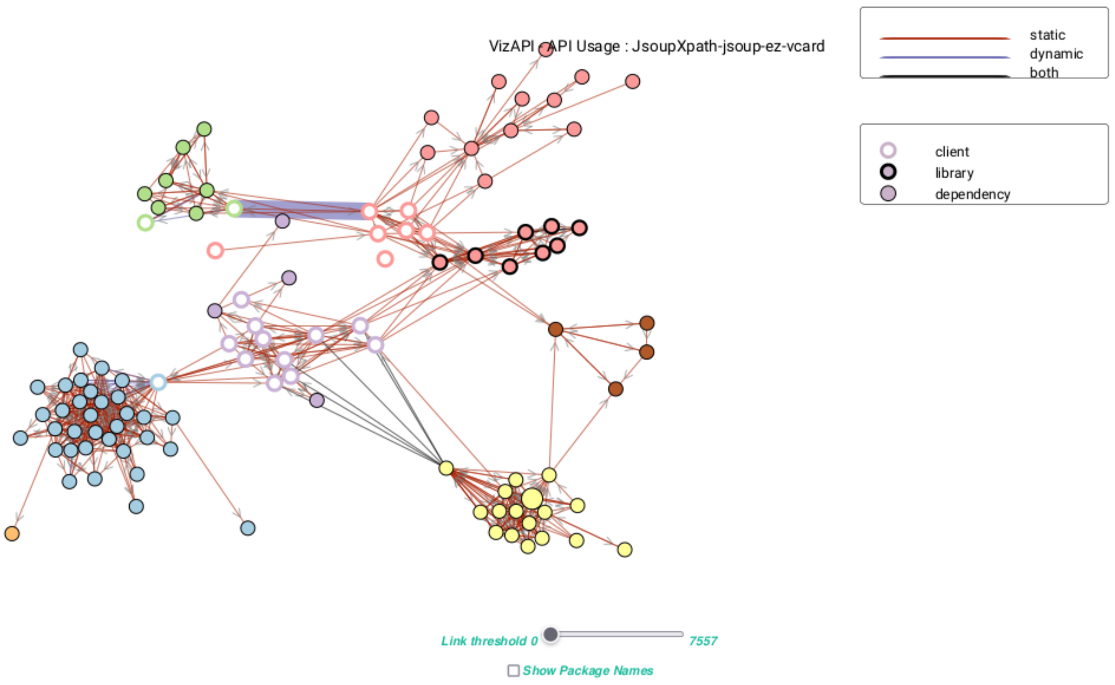
The main breaking change that library fission introduces in a client is that the client needs to identify the submodules that it requires and import only those submodules. We expect no impacts on clients in terms of performance, since we are only adding visibility restrictions. The worst case that we foresee is that a client must include all the submodules of the library, in which case there is no difference from the previous (unfissioned) version of the library. We performed manual exploratory testing of library fission on a subset of libraries and a random subset of their clients from our benchmarks. The unit test results were the same and we observed no change in build times.

3.3 Upgrades, Breaking Changes and Backward Compatibility

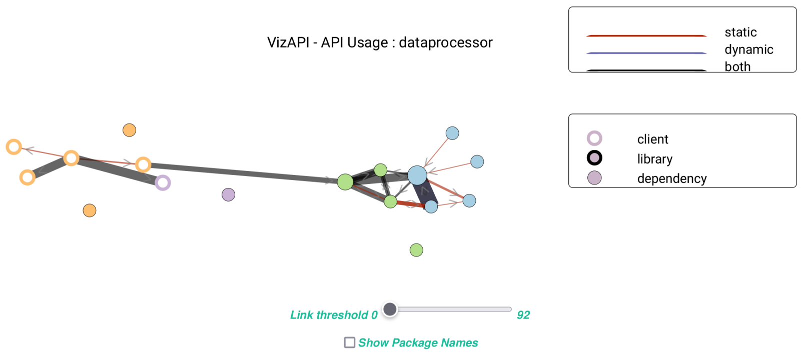
Our data for 101 Java software components contains different kind of interactions across component boundaries. This data can be analyzed and prove useful in the following ways:

- Library developers can observe different usage patterns of their APIs. The clusters observed in library packages can help with new version releases. Breaking changes might be restricted to clusters and library developers can choose whether to make clusters from new versions backward compatible with other clusters from older versions or not.
- Clients can also check if breaking changes in new library versions will affect their code during upgrades and cross check this with library changelogs.

- Unused dependencies can be identified and removed to prevent them from possibly introducing version conflicts and vulnerabilties.



(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 3.2: VizAPI Usage Scenarios.

Chapter 4

Related Work

We now discuss related work, which is divided into three sections. Section 4.1 presents work related to API usage while Sections 4.2.1 and 4.2.2 present work related to our two applications; library fission and the VizAPI visualization tool respectively.

4.1 API Usage

This work revolves around the usage of library APIs by clients. There is a large body of work investigating API usages. Mining API specifications was first introduced by Ammons et al [3]. In their work, frequent interaction patterns are obtained from program execution. More closely related to our present work is that by Zhong and Mei [60], who have investigated API usages in a dataset of 7 experimental subjects and the libraries that they depend on. Some of those findings are relevant to us: they find that clients use less than 10% of the declared APIs in libraries. Our results corroborate these findings. Our work differs from [60] in that we are specifically not investigating sequencing relationships between API calls. Saied et al [51] studied which API calls tended to co-occur in client code and inferred co-usage relationships between these calls. Our work goes beyond previous work by investigating not only which *intended* APIs get used but also APIs which are not declared to be part of the interface but are used in practice. We record different types of usage patterns, which we explain further in Chapter 2.

Thummalapenta and Xie [54] presented the related SpotWeb tool, which identifies framework hotspots (APIs that are often used) and coldspots (APIs that are never used); they do not consider framework APIs that are used but not intended to be. Hotspots and coldspots are, however, related to our investigations about client use of APIs; part of our study could be seen as investigating the prevalence of hotspots on our benchmark suite. Their notion of API usage is similar to ours, but they perform a static search to identify uses, while we statically record uses but also dynamically record test executions. They also identify the top N percent of used APIs as hotspots, and unused APIs as coldspots. Viljamaa [58] also aimed to find hotspots but used concept analysis to do so.

Good API design is important, and Piccioni et al [47] carried out a detailed study to determine what contributed to API usability. Our work also takes an empirical approach and studies what clients use in practice, as well as parts of the API which are hidden from users and yet are still used.

On the subject of API evolution, Yasmin et al [59] investigate deprecation and removal of APIs in the RESTful context, while Zhou and Walker [61] investigate it in the context of Java APIs with examples on the Web, and Li et al [37] investigate deprecation in the Android context. Finding deprecated APIs is useful for preventing breaking changes. Calls to removed APIs will definitely fail, especially in the RESTful context, where the server retracts the API (there is no previous library version to fall back on). Internal APIs share some similarities with deprecated APIs, in that calls to deprecated APIs may fail in the future—there are no guarantees. Our tool records all client-to-library interactions, as well as intra-client and intra-library interactions, and so we record usage of internal APIs.

Our study of API usage patterns includes investigating API bypasses. Dynamic language features are one way to bypass protections, and Dietrich et al [15] explored the extent of their use in the significant XCorpus. A related bypass technique is unsafe Java (especially used for performance), and Mastrangelo et al [40] characterized the use of unsafe Java in their benchmark suite. Amann et al [2] perform an evaluation of API mis-use detectors and provide a classification of API mis-uses in their work. However, they focus on misuses that result in exceptions while we focus on mis-uses that are unexpected but do not lead to any errors or exceptions.

4.2 Our Applications

We investigate client uses of library code, targeting two applications: the VizAPI visualization tool and library fission. 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 [34] and Kura et al [33]. Kura et al found that most software had outdated dependencies, and that software developers had negative feelings about being required to constantly upgrade their libraries. Foo et al [22] 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. The applications of this information—the VizAPI visualization tool and library fission can potentially enable easier upgrades. From the library side, library fission can help with maintainability—smaller libraries can be released with fewer worries about downstream effects.

4.2.1 Library Fission

The related notion of "tree-shaking" has been well-known since at least the 1990s, and recently rebranded as "debloating". The Jax project by Tip et al [55] was early work on debloating in the Java space. More recently, JShrink by Bruce et al [8] has applied more modern techniques to programs using more modern Java features (for example, lambdas). Both our approach and JShrink's integrate static and dynamic data to present recommendations to developers about what they should update and what they should exclude. However, debloating depends on executing and analyzing the client. Library fission, on the other hand, does not require running analyses on clients for every execution, since it splits libraries based on client usage.

A related concept is library shading, as done for instance by the Maven shade plugin[1]. We are looking for code that is included (due to indiscriminate inclusion of dependencies) but not used. A well-known problem that arises with dependencies is that different versions of a library can sometimes be included in the same client, and this is more likely to happen when there are more dependencies. Shading mitigates this problem, to some extent, by renaming included libraries such that each included version appears to be different.

Hejderup and Gousios [28] explore a question which is central to our approach—how well do client tests exercise their dependencies' libraries? To some extent, we rely on client test suites exercising enough of the dependencies to get valid results from our analyses. Their conclusion is 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.

Shah et al [52] present refactorings which enable dependency breaking. In our work, we instead use dynamic and static data to investigate library fission and enable clients to depend on the same libraries as before, but less of them, using techniques that are similar

[1]https://maven.apache.org/plugins/maven-shade-plugin/

to library shading. The difference is that we create versions of libraries that are smaller, not simply renamed.

4.2.2 VizAPI

We now discuss existing work related to VizAPI, our visualization tool. A representative tool from the software visualization literature is CodeSurveyor, by Hawes et al [27], 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 (for example, API invocations) using dynamic analysis by executing 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 [31] and software terrain maps by DeLine [13].

Call graph visualization is, of course, a well-known technique, as seen for example, in the Reacher tool [36]. 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. Another key difference is that VizAPI also displays interactions other than method calls, which include field usage, annotation usage, subtyping, reflective calls and dynamic proxies.

Kula et al [32] 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.

Bergel et al [5] 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.

Chapter 5

Conclusion

The goal of this work was to enable 1) library developers to make better decisions when designing new APIs, pruning or modifying unused APIs and to refactor their libraries; and 2) client developers to make better decisions about library upgrades and breaking changes.

5.1 Threats to Validity

Our threats to validity include the usual threat to external validity of insufficient sample size or variety—many of our seed libraries are JSON parser/generators, although our transitive closures result in a wider range of domains.

There is also the construct validity issue: tests may not adequately represent actual client behaviours. Dietrich et al discuss this in their work [16]; they find that even when coverage by unit tests is low, the low coverage is made up for by the fact that programmers are likely to write tests for the parts of the code that they believe to be most important. The other issue is considering client test suites as a reasonable representation of how libraries are used by the client. Our use of both static and dynamic information addresses both this issue and that of low coverage.

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" [38] 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.

We may have missed other categories of bypass patterns—though we believe that we have chosen at least a representative sample of mechanisms to ensure modularity.

Finally, our results apply best to Java-like languages, and may vary dramatically for other languages.

5.2 Actionable Outcomes

Based on our exploration of API uses and mis-uses, we make some recommendations for both API and language/analysis designers.

1. API scope: Both we and Thummalapenta and Xie [54] find that APIs are sparsely used. While some APIs are included in libraries for the sake of completeness (e.g. implementing all methods on an interface even if they are never expected to be called), API designers can seek to prune unused APIs that have no conceptual reason to exist;

2. Deprecation: One could investigate the scope for aggressive deprecation of unused APIs in released libraries, giving more liberty to API designers to modify their code;

3. Refactoring: Our results show that library fission could be useful, i.e., some existing APIs can be split into loosely-connected parts, reducing effective API surface and potentially developer cognitive burden;

4. Modularity: API and language designers can be confident that stated encapsulation boundaries are respected.

5.3 Directions for Future Work

In our study of API usage in our set of benchmarks, we find that APIs are sparsely used. This corroborates the same finding in Thummalapenta and Xie [54]'s work. We also see that our clients call into parts of libraries with a limited but nonzero overlap. An interesting experiment would be to plot cumulative usage as the number of clients increases and observe if the percentage of APIs used reaches a saturation point.

We can also extend our API usage study further by investigating why certain APIs are used the way that they are. It was interesting to observe that reflection and `setAccessible()` are commonly used for serialization. This is done to gain access to fields of objects to be serialized. Other observations from our study include reflection and `setAccessible()` being used on methods, reflection and `setAccessible()` being used on things that are already accessible (for example, public methods), accesses to non-constant fields of a library by a client and so on. Understanding the reason for these usage patterns can aid in better API design in the future.

VizAPI is in development and future work includes features such as zooming and filtering. User evaluations of our VizAPI tool can establish and improve the effectiveness of VizAPI. This can be performed following existing techniques [42]; in particular, experiments where users perform software understanding and maintenance tasks.

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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 385–395, 2017.

[2] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. A systematic evaluation of static API-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2018.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 4–16, 2002.

[4] 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. IEEE, 2019.

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

[6] 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.

[7] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.

[8] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. JShrink: In-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 135–146, New York, NY, USA, 2020. Association for Computing Machinery.

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

[10] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in Node.js applications: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1199–1203, New York, NY, USA, 2020. Association for Computing Machinery.

[11] 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.

[12] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12. IEEE, 2017.

[13] Robert DeLine. Staying oriented with software terrain maps. In *Proceedings of the Workshop on Visual Languages and Computation*, 2005.

[14] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73. IEEE, 2014.

[15] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. Xcorpus–an executable corpus of Java programs. *jot.fm*, 2017.

[16] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. On the construction of soundness oracles. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 37–42, 2017.

[17] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 266–276. IEEE Press, 2012.

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

[19] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563, 2018.

[20] Ben Evans. The Unsafe class: Unsafe at any speed. *Java Magazine*, May 2020. https://blogs.oracle.com/javamagazine/post/the-unsafe-class-unsafe-at-any-speed.

[21] Bob Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, March 2000.

[22] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *FSE*, page 791–796, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, January 2004. https://martinfowler.com/articles/injection.html. Accessed 22 October 2021.

[24] Martin Fowler. InversionOfControl, June 2005. https://martinfowler.com/bliki/InversionOfControl.html. Accessed 22 October 2021.

[25] Jr. Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[26] Michael W. Godfrey. personal communication.

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

[28] 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.

[29] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? An empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, 2014.

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

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

[32] 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.

[33] 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.

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

[35] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection—Literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518, 2017.

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

[37] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 254–264, New York, NY, USA, 2018. Association for Computing Machinery.

[38] 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.

[39] Jeff Luszcz. Apache Struts 2: How technical and development gaps caused the Equifax breach. *Network Security*, 2018(1):5–8, 2018.

[40] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The Java Unsafe API in the wild. *SIGPLAN Not.*, 50(10):695–710, October 2015.

[41] M Douglas McIlroy, J Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch-Partenkirchen, Germany*, pages 88–98, 1968.

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

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

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

[45] Oracle Corporation. Java Platform Module System (JSR 376). http://openjdk.java.net/projects/jigsaw/spec/, Sep 2017.

[46] OSGi Alliance. OSGi core release 8 specification. http://docs.osgi.org/specification/, July 2020.

[47] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. An empirical study of API usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14, 2013.

[48] 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.

[49] Martin C. Rinard. Acceptability-oriented computing. In Ron Crocker and Guy L. Steele Jr., editors, *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion), Onwards! Session*, pages 221–239. ACM, 2003.

[50] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61(4):58–66, 2018.

[51] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. Mining multi-level API usage patterns. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, Canada, March 2015. IEEE.

[52] Syed Muhammad Ali Shah, Jens Dietrich, and Catherine McCartin. On the automation of dependency-breaking refactorings in Java. In *2014 IEEE International Conference on Software Maintenance*, pages 160–169, Eindhoven, NL, Sep 2013.

[53] 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.

[54] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *ASE*, pages 327–336. IEEE Computer Society, 2008.

[55] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, nov 2002.

[56] John B Tran, Michael W Godfrey, Eric HS Lee, and Richard C Holt. Architectural repair of open source software. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 48–59. IEEE, 2000.

[57] Sruthi Venkatanarayanan, Jens Dietrich, Craig Anslow, and Patrick Lam. VizAPI: Visualizing interactions between Java libraries and clients. In *10th IEEE Working Conference on Software Visualization (VISSOFT 2022)*, 2022.

[58] Jukka Viljamaa. Reverse engineering framework reuse interfaces. In *FSE*, page 217–226, New York, NY, USA, 2003. Association for Computing Machinery.

[59] Jerin Yasmin, Yuan Tian, and Jinqiu Yang. A first look at the deprecation of RESTful APIs: An empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 151–161, 2020.

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

[61] Jing Zhou and Robert J. Walker. API deprecation: A retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, page 266–277, New York, NY, USA, 2016. Association for Computing Machinery.

APPENDICES

Appendix A

Artifacts

Our Github repository for the static and dynamic analyses tool is at

https://github.com/SruthiVenkat/calls-across-libs

The artifact documentation for VizAPI is presented below.

This document explains our artifact for the VizAPI tool, including how to obtain, install, and use the artifact. VizAPI generates a d3js visualization of library API usage by clients, using a modified version of Python's d3graph library. Our working Github repository for VizAPI is at https://github.com/SruthiVenkat/api-visualization-tool. We have also archived the artifact at https://doi.org/10.5281/zenodo.7023911 and our dataset at https://doi.org/10.5281/zenodo.7023872.

A.1 Getting Started

To acquire the repository, clone the repository (or download the archived version) and then copy the `apis-data` directory from the dataset into the root directory of the artifact. The API data directory contains our data for 101 benchmarks. If the input to VizAPI exists in `apis-data`, then VizAPI directly generates visualizations from the data files. If not, our package first runs the instrumentation to create the data files, and then generates visualizations from them.

We next describe the input to VizAPI. To run on a benchmark, the tool expects a JSON file called `input.json` as its input. The JSON file must contain an array of objects where each object describes a project; it can be in one of the following formats:

1. The first format, shown immediately below, is to be used when data for the project already exists in `apis-data`. In this case, the artifact ID of the project (which can be arbitrarily chosen by the user) and the type of the project (whether it is a client or library) need to be specified.

```
[{
  "artifact": artifact ID of project,
  "type": "client" or "library"
}]
```

Listing A.1: Input Format

2. The second is to be used when data for the project does not exist in `apis-data`. Again, the artifact ID of the project and the type of the project (whether it is a client or library) still need to be specified. Since the data do not exist, VizAPI also needs to capture instrumentation data, and hence VizAPI expects the URL and commit ID of the project. VizAPI expects the project to be in Maven format and can automatically execute the project's tests.

```
[{
  "url": Github link to repo,
  "commit": Commit ID,
  "artifact": artifact ID of project,
  "type": "client" or "library"
}]
```

Listing A.2: Input Format

A.2 Running VizAPI with Docker

We recommend using Docker to run VizAPI; we have tested the configuration and believe that it should be portable.

1. Run `docker build -t img_name .` from the directory where you have cloned the Github repository.

2. Run `docker run`
   `-v /path/to/this/repo/api-`

`visualization-tool:/api-visualization-tool img_name`

The path before the : in the command is your local path to the repo. The path after the : in the command is the path in the container, which is `/api-visualization-tool`.

A.3 Running VizAPI without Docker

It is also possible to run VizAPI outside of Docker.

1. Install the following Python packages: pandas, jupyterlab_server, networkx, colourmap, python-louvain, sklearn, ismember, d3graph, PyGithub.
2. Change paths starting with `/api-visualization-tool` to point to the location of your repository in the following files: `api-viz.py`, `config/config.properties`.
3. Run `api-viz.py`.

In both cases, i.e., with and without Docker, the final graph is generated with the name `api-usage.html`, in the VizAPI main directory.

The following is an example input.json needed to reproduce the Graph 1 at https://sruthivenkat.github.io/VizAPI-graph/:

```
[{
  "artifact": "dataprocessor",
  "type": "client"
}]
```

Listing A.3: Input Example

Some points to note are:

1. The size of the Docker image is around 4.1 GB.
2. The more data points, the longer the graphs take to generate.
3. When running VizAPI to generate graphs, you may see many Python Future warnings. They can be ignored.