Lightweight Syntactic API Usage Analysis with UCov

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, Jean-Rémy Falleri

Lightweight Syntactic API Usage Analysis with UCov

Gustave Monce
gustave.monce@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Thomas Couturou
thomas.couturou@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Yasmine Hamdaoui
yasmine.hamdaoui@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Thomas Degueule
thomas.degueule@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Jean-Rémy Falleri
falleri@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Institut Universitaire de France
France

ABSTRACT

Designing an effective API is essential for library developers as it is the lens through which clients will judge its usability and benefits, as well as the main friction point when the library evolves. Despite its importance, defining the boundaries of an API is a challenging task, mainly due to the diverse mechanisms provided by programming languages that have non-trivial interplays. In this paper, we present a novel conceptual framework designed to assist library maintainers in understanding the interactions allowed by their APIs via the use of *syntactic usage models*. These customizable models enable library maintainers to improve their design ahead of release, reducing friction during evolution. The complementary *syntactic usage footprints* and coverage scores, inferred from client code using the API (*e.g.*, documentation samples, tests, third-party clients), enable developers to understand in-the-wild uses of their APIs and to reflect on the adequacy of their tests and documentation. We implement these models for Java libraries in a new tool UCov and demonstrate its capabilities on three libraries exhibiting diverse styles of interaction: JSOUP, COMMONS-CLI, and SPARK. Our exploratory case study shows that UCov provides valuable information regarding API design and fine-grained analysis of client code to identify under-tested and under-documented library code.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**; *Software design engineering*.

KEYWORDS

software library, API design, API usage, API documentation

ACM Reference Format:
Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. 2024. Lightweight Syntactic API Usage Analysis with UCov. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643916.3644415

ICPC '24, April 15–16, 2024, Lisbon, Portugal
2024. ACM ISBN 979-8-4007-0586-1/24/04
https://doi.org/10.1145/3643916.3644415

1 INTRODUCTION

Application Programming Interfaces (APIs) govern the interactions between a library and client projects using it. In particular, they specify which elements of a library (*e.g.*, types, methods, fields) can be accessed from the outside and the valid interactions with them. Designing an effective API is essential for library developers as it is the lens through which clients judge its usability and benefits and the main friction point when the library evolves [23, 28]. Despite its importance, defining the boundaries of an API is a challenging task, mainly due to the diverse mechanisms provided by programming languages and paradigms that either aim specifically at designing APIs or indirectly affect them as a side effect (*e.g.*, visibilities, subclass restriction and sealing, name binding rules, overloading and overriding, polymorphism).

Given the complex interplay among those mechanisms, it is difficult for library maintainers to maintain an exhaustive and accurate mental model of their API and the interactions it enables. This complexity is compounded by the various *styles* libraries can adopt (*e.g.*, frameworks and inversion of control, fluent interfaces) and the variety of ways that client code can interact with individual API elements. For instance, methods may either be invoked directly or overridden, and non-abstract classes may be instantiated, extended, referenced, *etc.* This impairs the maintainers' ability to fully understand how their APIs are—or could be!—utilized in the wild. As a result, libraries sometimes inadvertently allow some kinds of interactions that their developers did not intend. This led to best API design practices such as "minimize accessibility" or "forbid subclassing by default" that are widespread in different communities (*e.g.*, in Java [1]). To minimize the likelihood of bad outcomes on the client side, library developers should restrict the possible interactions with their APIs to those that are *intended* and *validated*. Indeed, unintended interactions are a recipe for buggy client code and frustrated developers who may opt for an alternative library providing similar services with better support. However, there is no lightweight, effective way of extracting and maintaining a transparent, comprehensive API model, nor is there a way to validate the extent to which the interactions with an API are validated, documented, and representative of actual uses. Current approaches in the literature can successfully analyze the use of API symbols but do not account for the variety of possible interactions with an individual symbol [12, 26].

In this paper, we introduce a novel conceptual framework aimed at helping library maintainers better understand the boundaries of their APIs via the use of so-called *syntactic usage models* (SUMs). SUMs enable maintainers to reason about the interactions allowed by their APIs throughout their evolution. Based on the SUMs, we define *syntactic usage footprints* (SUFs), which measure the extent to which a given piece of code using the library (*e.g.*, third-party clients, documentation samples, library tests) exercises its API and covers its possible uses. We show how different SUFs can be easily compared, enabling library maintainers to answer questions such as "*do our tests validate the interactions found in third-party clients?*" or "*do the examples in our documentation align with in-the-wild uses of our API?*" (Section 3).

We present UCov (*Usage COVerage*), an implementation of these models for the Java programming language that leverages static analysis to automatically infer them from Java source code (Section 4). We illustrate the benefits of UCov and the underlying models with an exploratory case study of three libraries implementing various styles of interaction: JSOUP, a library for HTML manipulation;[1] Apache COMMONS-CLI, a library for implementing command-line interfaces;[2] and SPARK, a simple web framework[3] (Section 5). From our experiments with UCov, we lay out a research agenda for syntactic models and envision application scenarios in other areas (Section 6). [20]

Overall, our syntactic models and UCov provide a novel means for library maintainers to understand the extent of possible interactions allowed by their APIs and to oversee their use in client code. By offering a robust conceptual framework for usage coverage and a first implementation for Java, we aim to empower library maintainers with the insights needed to improve their API's design, documentation, and tests, thereby reducing friction with client code and fostering long-term maintenance.

2 BACKGROUND AND MOTIVATION

Understanding how client code interacts with APIs is instrumental in prioritizing development and documentation efforts. Therefore, numerous studies have examined methods to extract usage data from client code to identify notable *hotspots* and *coldspots*, *i.e.*, parts of the APIs that are over- or under-utilized [30–32]. In a recent study, Qiu et al. delved into the usage of Java's standard library and third-party libraries across a corpus of over 5,000 projects, encompassing 150M+ lines of code [26]. Various tools and studies adopt distinct methodologies for gathering usage data, whether from bytecode [12], source code and resolved ASTs [6, 18, 26], or other sources [31]. A unifying theme across these works and their underlying models of API usage is their focus on determining *whether a specific API symbol is accessed* in client code, rather than exploring *how it is used*. This emphasis stems from the primary objectives of these studies: assess the frequency, popularity, and coverage of API symbols in client code, rather than exploring the diverse *uses* of these symbols. Indeed, the way a symbol is used has no influence on its popularity. Extending, instantiating, or simply referencing a class all contribute equally to its popularity.

We argue that this approach to modeling library usage is too coarse-grained to allow library maintainers to understand fully the implications of the design of their API. For instance, let us consider a library designer providing a simple `public class C` and a client instantiating it with `C c = new C()`. Existing usage models would indicate that there is one exported symbol (`C`) and that this symbol is used in client code. Based on this information, library maintainers could conclude that the design of their API is satisfactory. However, this declaration does not prevent clients from subclassing the provided class with `class ClientC extends C`. The aforementioned models do not provide any means to distinguish this particular use from other uses in client code. This is a missed opportunity, as library maintainers could have reconsidered the API early on to prevent subclassing, disallowing clients from extending the class, and thus freeing themselves from supporting this scenario. The approaches mentioned above are oblivious to this distinction, while our work emphasizes it.

There is a wide variety of literature addressing the problem of mining and recommending API usage patterns and protocols [35, 42]. These studies typically analyze library code and client code to infer automata and probabilistic graphs that represent legal sequences of API invocations and check whether client code complies with them. While these approaches go beyond our objectives and consider some kind of semantic relation between API elements, they are unable to differentiate between the different interactions allowed for a specific element. In contrast to previous works, our approach emphasizes the diverse ways a particular symbol may be utilized in client code, leveraging the *syntactic usage models* introduced in the next section.

3 SYNTACTIC API USAGE

Our analysis of API usage is built upon two distinct models: syntactic usage models, which are extracted from library code and represent the *legal* (possible) uses of an API (Section 3.1), and syntactic usage footprints, which are extracted from client code and represent the *actual* uses of an API (Section 3.2). From these two models, we derive various metrics, including a dedicated API coverage metric (Section 3.3). These models are formulated independently of any specific programming language and can be adapted for diverse analysis scenarios. However, we reference Java syntax and semantics throughout this section for illustrative purposes.

3.1 Syntactic Usage Model (SUM)

We consider that a library defines a set of *symbols* $\mathcal{S}$ (*i.e.*, named entities), each of a particular *kind* and with a *declaration* that specifies its properties. For instance, the Java standard library defines the symbol `public class ArrayList<E>` of kind *Class*, the symbol `public final static PrintStream out` of kind *Field*, and the symbol `public void println()` of kind *Method*. These symbols can be exported, allowing client code to access them. Together, the exported symbols of a library form its API. Given a function $exported : \mathcal{S} \rightarrow \{\top, \bot\}$ indicating whether a given symbol is exported ($\top$) or not ($\bot$), derived from the language's semantics, the API of a library is the set $\mathcal{A} \subseteq \mathcal{S}$ such that $\mathcal{A} = \{s \in \mathcal{S} : exported(s) = \top\}$. In Java, for instance, visibilities and modules are the primary mechanisms for controlling symbol exports. As an

[1] https://jsoup.org/
[2] https://commons.apache.org/proper/commons-cli/
[3] https://sparkjava.com/

Table 1: Syntactic usage footprints for the API of Listing 1

Client	Statement	Symbol	Use
Listing 2	`ArrayList<Integer> lst`	`public class ArrayList<E>`	*Referenced*
	`new ArrayList<Integer>()`	`public class ArrayList<E>`	*Instantiated*
	`new ArrayList<Integer>()`	`public ArrayList<E>()`	*Invoked*
	`lst.add(42)`	`public boolean add(E e)`	*Invoked*
	`lst.add(1337)`	`public boolean add(E e)`	*Invoked*
Listing 3	`class MyArrayList<E> extends ArrayList<E>`	`public class ArrayList<E>`	*Extended*
	`@Override public boolean add(E e)`	`public boolean add(E e)`	*Overridden*

illustrative example, consider the simplified excerpt of the JDK's `java.util.ArrayList` depicted in Listing 1. In this example, the set $\mathcal{S}$ holds three symbols: `public class ArrayList<E>`, `public boolean add(E)`, and `private int size` while $\mathcal{A}$ holds the two first symbols only.

Listing 1: A simplified excerpt of `java.util.ArrayList`

```
public class ArrayList<E> implements List<E> {
    public boolean add(E e) { ... }
    private int size;
    ...
}
```

Depending on its properties, the same kind of symbol may be used in various ways in client code. Therefore, syntactic usage models employ a function $uses : \mathcal{A} \rightarrow \mathcal{P}(\mathcal{U})$, with $\mathcal{U}$ the set of all possible uses, that associates a given symbol with the set of its legal uses. For instance, following Java's semantics, a `public class A` can be *Instantiated*, *Referenced*, or *Extended* in client code. In contrast, a `public abstract class B` can only be *Referenced* or *Extended*, and a `public class E extends Exception` can even be *Thrown*. There is no universally correct definition of the *uses* function: it is the SUM designer's responsibility to specify the uses of interest that they would like to associate with each individual symbol. For instance, the use *Referenced* above may be refined for each kind of reference to a type (*e.g.*, as a parameter, variable, or return type) if subsequent analyses require it. Finally, the syntactic usage model $\mathcal{M}$ of a library is the union of the possible uses for every exported symbol in its API $\mathcal{M} = \bigcup_{s \in \mathcal{A}} uses(s)$. In the example of Listing 1, the SUM holds two exported symbols and five possible uses of the API, as shown in Table 2.

Table 2: Syntactic usage model of Listing 1

Symbol	Kind	Exported	Uses
`public class ArrayList<E>`	Class	✓	{*Instantiated, Referenced, Extended*}
`public boolean add(E e)`	Method	✓	{*Invoked*, *Overridden*}
`private int size`	Field	✗	∅

3.2 Syntactic Usage Footprint (SUF)

While the SUM holds the legal, possible uses of an API, the SUF materializes actual uses of the API in a given piece of client code. Formally, the footprint of a client on a SUM is the set of triples $\mathcal{F} = \{\langle s, u, l\rangle : s \in \mathcal{A}, u \in uses(s), l \in \mathcal{L}\}$ where s denotes the API symbol being used, u the kind of use (*e.g.*, *Referenced*, *Invoked*), and l the location of the use in client code (*e.g.*, a physical line-column location). As the definition implies, there can be multiple identical uses of the same symbol in different locations.

Syntactic usage footprints materialize the diversity of ways a given API can be used in client code. For instance, Listing 2 depicts a classical interaction with the `ArrayList` API through instantiation and invocations. On the other hand, Listing 3 depicts a typical framework-like interaction with the same API through extension and overriding. This latter style is prevalent in frameworks such as web servers or batch processing frameworks (*e.g.*, Spring, Hadoop, Spark) that heavily employ inversion of control and the Hollywood principle ("*Don't call us, we'll call you*"). As Table 1 shows, these two snippets yield two disjoint and complementary SUFs. Importantly, when looking at multiple clients, their SUFs can trivially be joined through set union and compared through set difference.

Listing 2: Classical usage of the API of Listing 1

```
ArrayList<Integer> lst = new ArrayList<Integer>();
lst.add(42);
lst.add(1337);
```

Listing 3: Framework-like usage of the API of Listing 1

```
class MyArrayList<E> extends ArrayList<E> {
    @Override
    public boolean add(E e) { ... }
}
```

3.3 Usage Coverage and Metrics

The SUM gives the universe of legal uses to cover for a given API and the SUF pinpoints those that are actually covered in client code. The set of API symbols that are covered in a SUF is denoted $C_{\mathcal{A}} = \{s : \langle s, u, l\rangle \in \mathcal{F}\}$ and the set of covered API uses is denoted $C_{\mathcal{M}} = \{u : \langle s, u, l\rangle \in \mathcal{F}\}$. Then, the API symbol coverage score of the SUF with respect to the API is $\frac{|C_{\mathcal{A}}|}{|\mathcal{A}|}$ and its API use coverage score is $\frac{|C_{\mathcal{M}}|}{|\mathcal{M}|}$. Following these definitions, Listing 2 covers 50% of Listing 1's API uses and 100% of its symbols, so does Listing 3. Together, they cover 100% of the API uses. One can identify the exported symbols that are not covered in client code by exploring $\mathcal{A} \setminus C_{\mathcal{A}}$ and the legal uses that are not realized by exploring $\mathcal{M} \setminus C_{\mathcal{M}}$. Interestingly, one may reproduce the popularity metrics used to identify hotspots and coldspots in the literature [26, 30] by counting the occurrences of $\langle s, u, l\rangle \in \mathcal{F}$ for a given symbol of interest s.

We distinguish between three levels of coverage for each individual API symbol. An API symbol $s \in \mathcal{A}$ is fully covered if all of its uses are present in the SUF, *i.e.*, if $\forall u \in uses(s), \exists \langle s, u, l \rangle \in \mathcal{F}$. Conversely, an API symbol is not covered if none of its uses are present in the SUF, and it is partially covered if only some of its uses are present in the SUF.

4 UCOV: JAVA SYNTACTIC USAGE ANALYSIS

UCov is our proof-of-concept implementation of syntactic API usage analysis for Java libraries.[4] It parses and analyzes the source code of Java libraries to produce syntactic usage models and the source code of Java clients to produce syntactic usage footprints and their coverage. When implementing such a tool, one must decide on the symbols of interest to include in the API and the types of uses to represent, based on language semantics and analysis goals, as highlighted in Section 3. In this section, we discuss some of the choices we made while implementing UCov to support our case studies and the overall architecture of the tool, depicted in Figure 1.

4.1 Exported Symbols

SUM models employ a simple $exported : \mathcal{S} \rightarrow \{\top, \bot\}$ function that indicates whether a given symbol s is exported. From the library's code, UCov first extracts the list of all API symbols (namely, for Java, types, methods, and fields) uniquely identified by their fully qualified name (and their signature in the case of methods). Then, it distinguishes those that can be accessed from the outside, in client code, and those that cannot. Following the rules described in the Java 17 Language Specification [11], the following symbols can be accessed from client code:

- **Public symbols:** (transitively) `public` types, as well as `public` methods and fields within `public` types, can be accessed from anywhere without restriction;
- **Protected symbols:** `protected` fields and methods within effectively extensible `public` types can be accessed through subtyping or by code located in the same package. An effectively extensible type is one that is not `final` nor `sealed` and that, in the case of classes, possesses a `public` or `protected` constructor the subclass can access;
- **Package-private symbols:** package-private symbols (Java's default visibility in the absence of a visibility modifier) can be accessed by code located in the same package.

The only legal visibilities for a top-level type declaration are `public` and package-private, so API designers can only use the latter when they want to hide a type declaration from the outside. UCov's implementation of the *exported* function thus considers the `public` and `protected` symbols and discards the package-private ones, as client code intentionally using a package of the same name as the library's to access its package-private symbols is a pathological case that most likely breaches the API's intent.

4.2 Supported API Symbol Uses

Once the exported symbols of a library are identified, UCov maps each of these to a set of legal uses, based on its declaration specifics, thereby implementing the *uses* function. Table 3 gives an excerpt of the uses we consider for each kind of symbol.

In a nutshell, UCov looks for every possible use of type references (as parameter types, return types, exceptions, *etc.*) and categorizes them all as *Type reference* uses. Classes can be instantiated if they provide a `public` (possibly default) constructor and are not `abstract`. They can be inherited unless declared `final`. Interfaces can be implemented by client classes or extended by other interfaces. Methods can be invoked (statically or dynamically) and overridden unless declared `final`. When polymorphism comes into play in client code and the static analysis cannot determine the dynamically invoked method beyond the static type of its receiver object, UCov registers an invocation to the corresponding method in the static type of the receiver object, as well as invocations to all super-methods with the same signature in its hierarchy. As a result, abstract methods in `abstract` classes and interfaces are considered covered whenever a more concrete implementation is invoked in client code. For instance, an invocation of `ArrayList.size()` in client code would cover `List.size()`. Finally, UCov distinguishes between read and write accesses to an exported field.

Note that these choices aim to reflect customary uses of Java APIs and are, to some extent, arbitrary. Another implementation may, for instance, distinguish between various references to a type or unify read and write accesses to a field under the same kind of use. These choices will vary depending on the task at hand and analysis objectives.

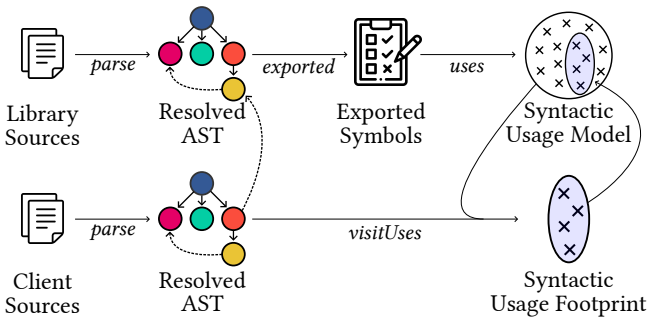


Figure 1: UCov parses and analyses Java code to build SUM and SUF models. Each cross symbol (×) represents one possible use of an API symbol.

4.3 Implementation

UCov is implemented in Java and uses the Spoon framework [25] to parse Java code and create visitors that implement the necessary static analyses.

When supplied with library code, UCov uses Spoon to build a resolved AST and applies a dedicated visitor on types, methods, and fields to list the exported API symbols, thereby implementing the *exported* function (upper part of Figure 1). Then, it maps each of these symbols to the corresponding uses (×) according to the rules in Section 4.2, forming the final syntactic usage model.

When supplied with client code (library tests, library samples, code of third-party clients, or any arbitrary snippet using the library), UCov builds its resolved AST and visits it to identify every

[4] UCov is available at https://github.com/alien-tools/ucov. A snapshot of UCov alongside all artifacts discussed in this paper is available on Zenodo [20].

Table 3: The kinds of uses considered in UCov, inspired and refined from the work of Qiu et al. [26]

Symbol Type	Use	Example	Resolved API Symbol
Type	Reference	`String s`	`java.lang.String`
		`void f(Integer i)`	`java.lang.Integer`
		`Integer f()`	`java.lang.Integer`
		`void f()throws IOException`	`java.io.IOException`
		`catch (IOException e)`	`java.io.IOException`
		`List<String> l`	`java.lang.String`
		etc.	*etc.*
Class	Instantiation	`new Integer(42)`	`java.lang.Integer`
	Inheritance	`class T extends Thread`	`java.lang.Thread`
Interface	Implementation	`class R implements Runnable`	`java.lang.Runnable`
		`Runnable r = ()→{...}`	`java.lang.Runnable`
	Extension	`interface R extends Runnable`	`java.lang.Runnable`
Constructor	Invocation	`new Integer(42)`	`java.lang.Integer(int)`
Method	Invocation	`"a".length()`	`java.lang.String.length()`
	Static invocation	`String.valueOf(42)`	`java.lang.String.valueOf(int)`
	Overriding	`@Override void run()`	`java.lang.Thread.run()`
		`Runnable r = ()→{...}`	`java.lang.Runnable.run()`
Field	Field read	`Integer.MAX_VALUE`	`java.lang.Integer.MAX_VALUE`
	Field write	`Point.x = 2`	`java.awt.Point.x`

node that references, in one way or another, symbols of the API. For example, it analyzes method invocations, field accesses, and type references to determine if they correspond to any of the uses identified in the syntactic usage model. Table 3 lists an excerpt of the elements in client code that the static analyzer searches for and the information it collects for each. The resulting set of actual uses in client code constitutes its syntactic usage footprint.

5 EXPLORATORY CASE STUDY

In this section, we evaluate the capability of syntactic usage models and footprints, as well as their implementation for Java libraries in UCov, to produce meaningful information that assists maintainers in understanding the boundaries of their libraries and their usage in the wild. To this end, we follow an exploratory case study approach [27] to analyze three popular Java libraries that exhibit diverse interaction styles: JSOUP, a library for HTML manipulation; Apache COMMONS-CLI, a library for implementing command-line interfaces; and SPARK, a simple web framework. We introduce our subject libraries in Section 5.1 and our methodology in Section 5.2, and then detail our results in Section 5.3. We direct the reader to our reproduction package to access the data, scripts, and analyses conducted in this section [20].

5.1 Subject Libraries

The implementation of syntactic usage models and footprints in UCov aims to explore the boundaries of APIs and their actual uses in client code. Every API is unique, and it is neither possible nor desirable to study all of them. Instead, we establish a set of criteria to select subject libraries that are diverse and representative of mature Java libraries.

Interaction styles. Based on our modeling of syntactic usage, we hypothesize that different interaction styles favor different kinds of uses in client code and yield different usage profiles. To explore the diversity of uses, we aim for libraries that expose diverse styles. We categorize interaction styles into three types that are widespread in practice: classical, framework, and fluent. Below, we provide an example of each and explain their significance in the context of API usage. Naturally, these styles are not mutually exclusive and a library may offer different styles to interact with the same features.

In the *classical* style, APIs expose a set of public classes that are then instantiated in client code to invoke their methods and access their services. Client code retains control of the execution flow. The first snippet of Listing 4 depicts a typical classical use of COMMONS-CLI where a parser and options are instantiated and configured using their methods. The classical style is the most common and is especially popular in libraries that offer a collection of utilities (*e.g.*, Google's Guava and Apache's Commons).

In the *framework* style, APIs implement inversion of control and the Hollywood principle to allow client code to extend and specialize types of the API (usually interfaces and abstract classes) by providing implementation code. The library retains control over the execution flow and only hands it to client code when needed. This style is prevalent in frameworks (*e.g.*, Spring, Hadoop), hence its name. The second snippet of Listing 4 shows a typical framework-like interaction with SPARK, where users configure routes for their application by passing lambda expressions that implement the functional interface `Route` and its unique method `handle(Request, Response)` which gives the implementation of endpoints.

In the *fluent* style, APIs heavily employ programming tricks such as method chaining and cascading, static methods, the *Builder* design pattern, and static imports to mimic the look and feel of a domain-specific language [9]. The third snippet of Listing 4 shows

Listing 4: Diverse API interaction styles exemplified using actual documentation samples from our subject libraries

```
// Classical usage of commons-cli
// https://commons.apache.org/proper/commons-cli/usage.html
CommandLineParser parser = new DefaultParser();
Options options = new Options();
options.addOption("a", "all", false, "do not hide entries");
options.addOption("C", false, "list entries by columns");
try {
  CommandLine line = parser.parse(options, args);
  if (line.hasOption("block-size")) { ... }
}
catch (ParseException exp) { ... }

// Framework-like usage of Spark
// https://sparkjava.com/documentation#routes
get("/",     (request, response) -> { ... });
post("/",    (request, response) -> { ... });
put("/",     (request, response) -> { ... });
delete("/",  (request, response) -> { ... });
options("/", (request, response) -> { ... });

// Fluent-like usage of jsoup
// https://jsoup.org/cookbook/input/load-document-from-url
Document doc = Jsoup.connect("http://example.com")
  .data("query", "Java").userAgent("Mozilla")
  .cookie("auth", "token").timeout(3000).post();
```

a typical fluent interaction with JSOUP that shows how the *Builder* pattern is used to configure a JSOUP connection and load a document from a remote URL.

Client code. To make the analysis possible, we look for libraries that have sufficient amounts of client code available: third-party clients, tests, and documentation samples. This rules out immature libraries that lack sufficient documentation or clients, or that are not well-tested.

While we expect to find third-party clients and tests that can be parsed and analyzed with UCov, documentation and samples suffer from additional issues. Some libraries include their samples as proper compilable files in their source directory, while others include their samples in readme files (*e.g.*, on GitHub) or dedicated documentation websites. These samples are typically simple snippets cleaned from the surrounding boilerplate code (imports, structure, type declarations, *etc.*) that often cannot be parsed on their own. When these cases arise, we create a Java file for each case, encapsulating the snippet within a main method and manually inserting the missing imports. For our subject libraries, with the help of the surrounding documentation, the missing imports are always unambiguous.

Selected libraries. To obtain high-quality libraries that meet these requirements, we start from the Duets dataset [7]. Duets contains 395 libraries and 2,874 clients extracted from GitHub that compile, can be executed, have passing test suites and a minimum of five stars. To identify libraries with documentation samples, we narrow our search to libraries with a readme file and an official documentation website. If these contain documentation samples, we collect them. Otherwise, we follow their hyperlinks and repeat the process. We also explore the source tree of each library to find samples stored alongside the library code. Finally, we handpick a set of three libraries that exhibit diverse interaction styles and have sufficient test cases, documentation samples, and third-party clients: JSOUP (classical and fluent styles), COMMONS-CLI (classical and fluent styles), and SPARK (framework style). Table 4 provides some descriptive statistics for these libraries.

Table 4: Descriptive statistics of the subject libraries, extracted from GitHub on October 30, 2023

	COMMONS-CLI	JSOUP	SPARK
Last release date	2021/10/29	2023/04/29	2020/10/08
Version	1.5.0	1.16.1	2.9.3
Since	2002	2010	2011
Stars	309	10.4k	9.5k
Commits	1,374	1,889	1,067
Size (LoC)	6,307	27,817	11,298
Contributors	46	97	100
Clients	49k	131k	30k

For these three libraries, we retrieve the following documentation samples:

- For COMMONS-CLI, its readme links to the official documentation, which includes a page showcasing some sample uses.[5] This page contains partial code snippets without imports or structure. We manually reconstruct parseable Java files for these snippets.
- For JSOUP, we find a set of samples directly in its source code (`src/main/java/org/jsoup/examples`). Its readme file also leads to the official website and a cookbook that presents different sample snippets.[6] We reconstruct complete Java files with proper imports and structure for these samples.
- For SPARK, its readme includes fully parseable sample code and links to the official documentation.[7] The documentation points to a list of user-authored tutorials,[8] which consist of parseable Java files, and a list of template projects,[9] which we also include.

5.2 Methodology

Our methodology aims to explore the interactions offered by various APIs and their actual uses in third-party clients, tests, and documentation samples. Since the number of clients per library stored in Duets is relatively low (31 for JSOUP, 202 for COMMONS-CLI, and 8 for SPARK), and as we do not require clients to have passing test suites, we collect new clients for each of the libraries using GitHub's dependency graph.[10] We compile a list of every repository in the dependency graph that declares a dependency towards one

[5] https://commons.apache.org/proper/commons-cli/usage.html
[6] https://jsoup.org/cookbook/
[7] http://sparkjava.com/documentation
[8] https://sparkjava.com/tutorials
[9] http://sparkjava.com/tutorials/application-structure, https://github.com/tipsy/spark-basic-structure, https://github.com/perwendel/spark-template-engines
[10] https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph

Table 5: Syntactic usage models extracted from COMMONS-CLI, JSOUP, and SPARK, and syntactic usage footprints extracted from their third-party clients, tests, and samples. Symbols used are presented as % of API symbols and unique uses as % of legal uses.

			CLI	JSOUP	SPARK
SUM		API symbols	291	1,138	771
		Legal uses	755	2,941	1,960
SUF	Symbols used	All	205 (70%)	608 (53%)	301 (39%)
		Clients	153 (53%)	282 (25%)	179 (23%)
		Tests	179 (62%)	586 (51%)	248 (32%)
		Samples	24 (8%)	47 (4%)	134 (17%)
	Unique uses	All	367 (49%)	1,028 (35%)	476 (24%)
		Clients	266 (35%)	456 (16%)	278 (14%)
		Tests	315 (42%)	967 (33%)	400 (20%)
		Samples	38 (5%)	69 (2%)	211 (11%)
	Total uses	All	79,284	29,979	23,882
		Clients	74,453	15,246	20,827
		Tests	4,690	14,511	1,605
		Samples	141	222	1450

of the libraries of interest. After discarding repositories that cannot be retrieved and forks, we obtain 11,159 clients for JSOUP, 11,612 for COMMONS-CLI, and 12,530 for SPARK. Naturally, it is not necessary to analyze all of these, so we apply the standard Cochran formula with a confidence level of $c = 95\%$, an error margin of $e = 5\%$, and a conservative proportion of $p = 0.5$. We obtain sample sizes of 372, 372, and 373, which we draw at random from the corresponding client sets. These sets contain the third-party clients, which we analyze in the remainder of this section.

We use UCov to construct the SUM of each library. Following the specification in Section 4, UCov extracts the list of all exported API symbols and maps them to their corresponding legal uses. The resulting models materialize the universe of interactions enabled by the libraries (Table 5). For each library, we classify client code as either tests, documentation samples, or third-party clients, and we run UCov on these three corpora to produce their SUF. Since SUFs can be easily joined through set union, we also construct a merged SUF for each library representing every use from all client code, denoted *All* in Table 5.

5.3 Analysis and Results

This section discusses the results obtained with UCov for the three subject libraries and their client code.

Syntactic usage models and footprints. Table 5 presents simple statistics for the SUMs and SUFs extracted from our three subject libraries. JSOUP exposes the most extensive API, with 1,138 symbols and 2,941 legal uses in its SUM, followed by SPARK and COMMONS-CLI. This is consistent with their overall size in lines of code (Table 4). The three libraries exhibit a similar ratio of exported API symbols to legal uses, averaging around 2.5 legal uses per symbol.

The coverage scores for uses are consistently lower than for API symbols. This is expected, as covering a use implies covering the corresponding symbol. However, the sizeable difference in coverage indicates that many of the interactions legally allowed in the APIs are not realized in either tests, documentation samples, or third-party clients, even when the corresponding symbols are known and used. Although the relatively low coverage of API symbols in client code has been extensively studied in the literature (*e.g.* [12, 26]), this suggests that symbol coverage does not fully reflect the extent of interactions permitted by APIs.

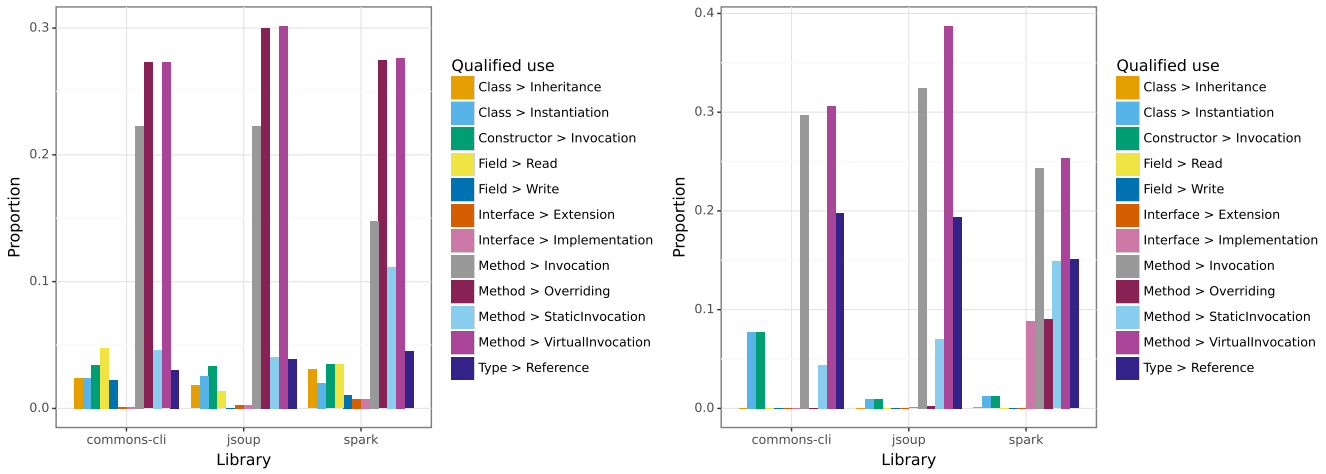
Across all libraries, tests achieve the highest coverage scores for symbols and uses, followed by third-party clients and samples. This suggests that the tests cover a sizeable subset of the APIs. According to Hyrum's law, "*with a sufficient number of users of an API [...] all observable behaviors [...] will be depended on by somebody*" [36], suggesting that third-party clients might eventually achieve the highest coverage score, assuming a sufficient number of clients. This has already been empirically verified for the libraries hosted in Maven Central [12]. Documentation samples are sparse, so naturally they cover much less than tests and third-party clients. SPARK stands out with a much better coverage score of its API by samples, thanks to the rich documentation on its official website.

Overall, COMMONS-CLI is the most focused library, with fewer exported symbols and possible uses, which results in better coverage of its symbols and uses in client code. Its API also exhibits repetitive patterns: client code often instantiates numerous command-line options and configures them similarly, resulting in frequent and uniform usage of identical symbols (averaging 216 instances per use, compared to 29 for JSOUP and 50 for SPARK).

Library profiles. Figure 2a depicts the usage profiles of our three libraries. The profile of a library is the distribution of the legal uses in their SUMs, such that the proportion of each kind of use adds up to 1. It highlights which kinds of uses are allowed and which are the most frequent. The profiles are mostly similar, with some specificities for each library. In all cases, (virtual) method invocation and method overriding dominate the frequencies. This is expected as methods are by far the most frequent kind of symbol in the three libraries, with fewer types and fields exposed. COMMONS-CLI and JSOUP share a very similar profile. Indeed, they both implement the classical and fluent styles in their interactions. Interestingly, SPARK exposes a larger number of methods that can be statically invoked and interfaces that can be extended and overridden in client code. These correspond to the primary interfaces and methods used in client code to configure routes (`get()`, `post()`, `path()`, *etc.*), as shown in Listing 4. This is consistent with its framework-like interaction style.

Usage analysis. Figure 2b shows the usage profiles of the three libraries, defined as the distribution of actual uses in their SUFs (including third-party clients, tests, and samples). Comparing Figure 2a and Figure 2b reveals the discrepancies between what the APIs allow and what client code actually uses.

Among the three libraries, method overriding is the least covered type of use by a large margin. This suggests that the APIs offer possibilities for extension and specialization that are not yet utilized in client code or, more likely, that many of the API methods could be closed for extension using the `final` keyword at the level of the method or its containing type. Until version 17 (2021) and the introduction of sealed classes, Java lacked the ability to restrict extension and overriding to a predetermined set of types. Without



(a) As a distribution of the legal uses in their APIs (SUM)

(b) As a distribution of the actual uses in client code (SUF)

Figure 2: Usage profiles of the three libraries COMMONS-CLI, JSOUP, and SPARK

this possibility, library maintainers had to open types for extension and specialization to everyone, even when their intent was to allow extension and specialization in library code only.

Here again, COMMONS-CLI and JSOUP expose a similar profile. However, JSOUP's clients rely more on its fluent style, with fewer class instantiations and more frequent use of static invocations and builders to create objects. We also observe that although COMMONS-CLI and JSOUP expose some fields that can be read and written, as well as some types that can be extended or implemented, clients rarely use these interactions. The only uses of these interfaces in client code are through type references.

SPARK, on the other hand, exhibits a different profile with many static invocations, method overrides, and interface implementations. Indeed, as advocated in its documentation, the preferred method of declaring routes in SPARK is to pass a lambda expression that implements a single method in the `Route` interface, resulting in each route declaration involving one static invocation, one interface implementation, and one method override.

Digging deeper into the most popular uses for each library, we find that the SUFs accurately reflect the expected uses of each library, as documented in their samples. For COMMONS-CLI, the top three most popular interactions involve referencing and instantiating the symbols `Option` and `Options`, and invoking the method `addOption`, which developers use to build their command-line interface. For JSOUP, the top three most popular interactions involve referencing the symbols `Document`, `Element`, and invoking the method `Element.select(String)`, which developers use to navigate HTML documents. For SPARK, the top three most popular interactions involve implementing the interface `Route` and overriding its method `Route.handle()`, which developers use to declare routes, as well as referencing the types `Request` and `Response`, which are passed to and from the routes. We refer the reader to the reproduction package for a comprehensive list of the most and least popular API symbols and interactions.

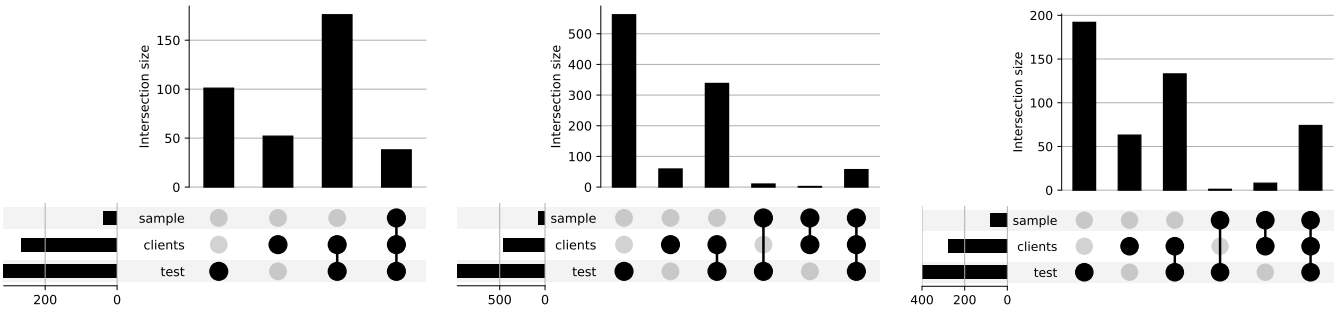
Interestingly, we observe that JSOUP exposes parts of its *internal* API publicly (`org.jsoup.internal`), most likely for technical reasons: isolating types in a package forces them to be `public` to allow other packages of the library to access them. While the maintainers take great care to discourage clients from using these APIs using source code comments ("*Jsoup internal use only, please don't depend on this API.*"), we find several uses in the code of third-party clients. Perhaps more surprisingly, JSOUP's official documentation also uses these internal APIs in one of its samples.[11] UCov allows library maintainers to identify these problematic cases at a glance.

Comparative analysis of third-party clients, tests, and samples. The SUF profiles of third-party clients, tests, and samples (not shown here for conciseness but available in the reproduction package) are very similar for each library. However, when looking at individual symbols and uses, we observe that each has its specificities. Figure 3 depicts the individual contributions to unique uses of third-party clients, tests, and samples, represented as UpSet plots to visualize the size of their intersections. This visualization enables library maintainers to immediately identify gaps in the coverage of client code by their tests and documentation.

The official samples of COMMONS-CLI[12] are particularly sparse compared to JSOUP and SPARK: all the uses they document are also found in tests and third-party code. Conversely, we identify 176 uses that are common to tests and third-party clients but not documented (23% of legal uses). These include the deprecated `OptionBuilder` type, which is unsurprisingly not documented, but also symbols that are popular in client code such as `org.apache.commons.cli.Option.isRequired()`. As the API of COMMONS-CLI is well-focused, there is still a significant amount of uses that are common to all three. However, we observe that 52 uses found in third-party clients

[11] https://github.com/jhy/jsoup/blob/84fd43766992401057b73f740acec4b82f1e3dd6/src/main/java/org/jsoup/examples/HtmlToPlainText.java

[12] https://commons.apache.org/proper/commons-cli/usage.html



(a) SUF intersections for COMMONS-CLI (b) SUF intersections for JSOUP (c) SUF intersections for SPARK

Figure 3: UpSet plots [17] depicting the common and unique uses between third-party clients, tests, and samples for JSOUP, COMMONS-CLI, and SPARK

are not exercised: for instance, the method `org.apache.commons.cli.Option.setValueSeparator(char)` is neither tested nor documented.

The samples of JSOUP are more varied, but there is still a significant amount of uses that are undocumented. Among these 338 uses, we identify some popular interactions such as invoking the `org.jsoup.nodes.Node.hasAttr(java.lang.String)` method. Its tests are comprehensive, and only a few uses are exclusive to third-party clients. Among these, we find clients extending and overriding the `org.jsoup.Connection` type to implement their own logic and innocuous cases such as invocations to the many `toString()` methods that are (predictably) untested and undocumented. Interestingly, some of the uses we find in third-party clients are also documented in its samples, but absent from the tests. This is, for instance, the case for the internal APIs of JSOUP discussed previously.

Despite the extensive samples found in its readme and tutorials, SPARK still presents 39 uses that are common to clients and tests but undocumented. Even more surprisingly, we find some uses that are documented but neither tested nor used in third-party clients, such as invocations of the `spark.Request.bodyAsBytes()` method. In contrast with COMMONS-CLI and JSOUP, we find some uses that are documented, tested, but for which we could not find any instance in client code: we did not find any static invocation of the method `spark.Spark.modelAndView(Object, String)` which is the documented way of instantiating a `ModelAndView` object. Similarly to JSOUP, we find 27 API interactions that are documented in samples and used by third-party clients but remain untested. These include setting the listening port of the web server (`spark.Spark.port(int)`) and examining the underlying raw Java requests beneath SPARK's `Request` objects (`spark.Request.raw()`).

A significant part of the value of syntactic models resides in the intersection of the uses of different kinds of client code. UCov allows library maintainers to identify undocumented or untested interactions in their APIs and take appropriate action. Naturally, it is not practical to expect that all API interactions are covered by the library's tests or samples. Most developers use IDEs to discover features, and the associated JavaDoc is often sufficient. Besides, trivial APIs such as the `toString()` methods mentioned above may not necessarily require documentation. Nevertheless, we believe that UCov can help maintainers prioritize their documentation and testing efforts by focusing on the interactions that are commonly used in the code of third-party clients, and identify blind spots in their design.

6 DISCUSSION

Syntactic usage models and syntactic usage footprints offer a lightweight and intuitive way to analyze the interactions allowed by an API and their coverage in client code. While our case study primarily explores their ability to accurately represent different library profiles and identify unused, undocumented, and untested interactions, they can inform and benefit other scenarios.

Support for API design in programming languages. SUM models represent the interactions permitted by an API. When using them, it becomes clear that the capabilities provided by Java for designing APIs are limited. Library developers are often required to expose numerous legal interactions to client code for purely technical reasons that have little to do with API design. For example, although library developers can prevent extension and overriding for everyone by using the `final` modifier, they have only recently gained the ability to open extension to a predetermined set of implementers using `sealed` classes. On the other hand, it is not possible to fine-tune the interactions allowed in client code, such as allowing method overriding but not method invocation, which can violate the principles of inversion of control in framework-like libraries. Similarly, there are no mechanisms to restrict access to certain types, fields, and methods beyond the basic scoping mechanisms provided by visibilities. As a result, developers must sometimes declare types as package-private or even public for technical reasons and rely on code comments and naming conventions such as internal packages and `@Internal` annotations to warn their users. SUM models make all these issues explicit and could facilitate reflection on language design when adapted to other programming languages.

Compatibility and breaking changes. When libraries evolve, they sometimes introduce breaking changes that affect clients and force them to adapt their code. How breaking changes impact client code depends on the interactions with the broken API. For example, a class that evolves into an abstract class does not break client code that references it but does break client code that attempts to instantiate it. Simply knowing which API symbol the client code

uses is insufficient to assess the impact of breaking changes. Some approaches attempt to measure the impact of potential breaking changes by analyzing client code. While some only rely on import declarations to determine if client code may be impacted [37], more recent approaches gather usage information to improve the accuracy of impact detection [23, 24]. SUF models provide accurate information regarding the uses made in client code and we believe they could improve the accuracy of these tools.

Another interesting direction is to utilize SUM models to automatically generate synthetic client code that thoroughly exercises every possible interaction with the API. This code would exhibit a complete footprint of the API and could be recompiled whenever the library is updated, allowing the compiler to automatically check if any of the interactions break, signaling a breaking change. Using the compiler as ground truth for breaking change detection would address the accuracy issues of existing detection tools [13] and automatically cope with the evolution of programming language specifications and their implementation in compilers and virtual machines.

API evolution. Predicting the consequences of API changes can be challenging. When an API evolves, it can not only introduce breaking changes but also alter the way clients interact with it, even when the changes are backward-compatible. Because SUM models can be efficiently computed (e.g., during code review or continuous integration), maintainers can reason about the impact of their changes and refactorings by exploring the differences between pre- and post-change SUMs. This could allow maintainers to easily assess the impact of external contributions on the interactions permitted by their API and determine whether to incorporate the changes.

7 RELATED WORK

In this section, we discuss related work on the analysis and applications of API usage. Specifically, we review the tools and studies that analyze usage at the symbol level, the extensive literature on usage patterns and protocols, and their applications to API evolution and breaking changes analysis.

API symbols analysis. Traditionally, the analysis of API usage has primarily been conducted at the level of individual symbols. Many studies analyze how API symbols are used to identify notable *hotspots* and *coldspots*, *i.e.*, parts of the APIs that are over- or under-utilized [30–32]. A recent study by Qiu et al. delves into the usage of Java's standard library and third-party libraries across a corpus of over 5,000 projects, encompassing 150M+ lines of code, and finds that usage follows a Zipf distribution [26]. Another study by Harrand et al. examines 2.2M dependencies on Maven Central and confirms the observations of Hyrum's law: a small subset of the API attracts most usages in client code [12]. As illustrated by our exploratory case study, the coverage of uses is much smaller than that of symbols. Various tools and studies employ different methodologies for collecting usage data, whether from bytecode [12], source code and resolved ASTs [6, 18, 26], or other sources [31]. A unifying theme across these works and their underlying models of API usage is their focus on determining whether a specific API symbol is accessed in client code, rather than exploring how it is used. Syntactic models and UCov go beyond this level of granularity by distinguishing the various interactions allowed by individual symbols. As our case study shows, the coverage of uses in client code is much smaller than that of symbols.

Usage patterns and protocols. A variety of work in the literature addresses the problem of mining and recommending unordered or sequential API usage patterns and protocols to users [3, 22, 29, 35, 42]. These studies typically analyze library code and client code to infer automata and probabilistic graphs that represent legal sequences of API invocations and check whether client code complies with them. Some even gather data from Q&A websites to infer and recommend usage patterns [33, 40], while others empirically study these usages in the wild [41]. These approaches consider sequences of API calls, often with temporal properties, and go beyond the goals of UCov. On the other hand, UCov still provides a more detailed view of interactions with API symbols, which could benefit these approaches. In our work, we leverage syntactic usage models and footprints to help developers understand which API uses are allowed and to help them improve their API design, eliminating unintended uses or documenting undocumented areas of their APIs. A promising area for future research is studying how syntactic usage models and footprints can contribute to extracting more fine-grained usage patterns and protocols.

API evolution breaking changes. There is a wide range of literature on the nature and driving forces of API evolution [14, 15, 19], as well as on breaking changes and their impact [2, 23, 37]. Some studies focus on detecting breaking changes through API usage patterns [8, 10, 12, 16, 21, 34, 38, 39]. A key enabler for these studies is the use of accurate models that can represent both the interactions allowed by an API and their uses in client code. The impact of a breaking change, for example, depends on both the symbol being used and the way it is used. We believe that the rich information provided by syntactic usage models and footprints can better inform breaking change detection tools [13] and improve their accuracy in assessing the impact of breaking changes on client code. This information can also better inform studies that aim to understand API usage and evolution by refining the level of granularity at which APIs are considered [4, 5].

8 CONCLUSION

In this paper, we introduced syntactic usage models and footprints to support library maintainers in understanding the boundaries of their APIs and the interactions they allow. We presented an implementation of these models for the Java programming language in UCov, a static analysis tool that analyzes the source code of libraries to collect their exported symbols and allowed uses, and client code to collect fine-grained uses of the library.

Our exploratory case study of three popular Java libraries that exhibit diverse interaction styles (commons-cli, jsoup, and spark) showed that UCov provides valuable information regarding the usage profiles of libraries and their uses in documentation samples, tests, and third-party clients. Specifically, we showed how UCov can pinpoint undocumented and untested interactions, as well as misalignments between legal uses of the API and actual uses in client code.

For future work, we will explore the suitability of syntactic usage models and footprints to support graceful API evolution and the detection of breaking changes and their impact. We will also employ UCov for large-scale empirical analysis of the evolution of Java libraries and investigate the benefits of syntactic models in other programming languages and ecosystems.

ACKNOWLEDGMENTS

This work was partially funded by the French National Research Agency through grant ANR ALIEN (ANR–21–CE25–0007). We thank the anonymous reviewers for their precious comments.

REFERENCES

[1] Joshua Bloch. *Effective java*. Addison-Wesley Professional, 2008.

[2] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why and how java developers break apis. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265. IEEE, 2018.

[3] Raymond PL Buse and Westley Weimer. Synthesizing api usage examples. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 782–792. IEEE, 2012.

[4] John Businge, Alexander Serebrenik, and Mark GJ Van Den Brand. Eclipse api usage: the good and the bad. *Software Quality Journal*, 23:107–141, 2015.

[5] John Businge, Simon Kawuma, Moses Openja, Engineer Bainomugisha, and Alexander Serebrenik. How stable are eclipse application framework internal interfaces? In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, pages 117–127. IEEE, 2019.

[6] Coen De Roover, Ralf Lämmel, and Pek Pek. Multi-dimensional Exploration of API Usage. In *21st International Conference on Program Comprehension*, pages 152–161. IEEE, 2013. doi:10.1109/ICPC.2013.6613843. ISSN: 1092-8138.

[7] Thomas Durieux, César Soto-Valero, and Benoit Baudry. Duets: A dataset of reproducible pairs of java library-clients. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 545–549. IEEE, 2021.

[8] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 791–796, Lake Buena Vista FL USA, October 2018. ACM. ISBN 978-1-4503-5573-5. doi:10.1145/3236024.3275535. URL https://dl.acm.org/doi/10.1145/3236024.3275535.

[9] Martin Fowler. Fluent interface, 2005. URL https://www.martinfowler.com/bliki/FluentInterface.html.

[10] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. APIfix: output-oriented program synthesis for combating breaking changes in libraries. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, October 2021. ISSN 2475-1421. doi:10.1145/3485538. URL https://dl.acm.org/doi/10.1145/3485538.

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java language specification: Java se 17 edition, 2021.

[12] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. Api beauty is in the eye of the clients: 2.2 million maven dependencies reveal the spectrum of client–api usages. *Journal of Systems and Software*, 184:111134, 2022.

[13] Kamil Jezek and Jens Dietrich. Api evolution and compatibility: A data corpus and tool evaluation. *J. Object Technol.*, 16(4):2–1, 2017.

[14] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding type changes in java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 629–641, 2020.

[15] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. An empirical study on the impact of refactoring activities on evolving client-used apis. *Information and Software Technology*, 93:186–199, 2018.

[16] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A Systematic Review of API Evolution Literature. *ACM Comput. Surv.*, 54(8):1–36, November 2022. ISSN 0360-0300, 1557-7341. doi:10.1145/3470133. URL https://dl.acm.org/doi/10.1145/3470133.

[17] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. Upset: Visualization of intersecting sets. *IEEE Transactions on Visualization and Computer Graphics (InfoVis)*, 20(12):1983–1992, 2014. doi:10.1109/TVCG.2014.2346248.

[18] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1317–1324. ACM, March 2011. ISBN 978-1-4503-0113-8. doi:10.1145/1982185.1982471.

[19] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE, 2013.

[20] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. Artifacts for "Lightweight Syntactic API Usage Analysis with UCov", February 2024. URL https://doi.org/10.5281/zenodo.10571867.

[21] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 819–830, Montreal, QC, Canada, May 2019. IEEE. ISBN 978-1-72810-869-8. doi:10.1109/ICSE.2019.00089. URL https://ieeexplore.ieee.org/document/8812071/.

[22] Phuong Thanh Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. FOCUS: a recommender system for mining API function calls and usage patterns. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1050–1060. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00109. URL https://doi.org/10.1109/ICSE.2019.00109.

[23] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study. *Empirical Software Engineering*, 27 (3):61, 2022.

[24] Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. Breakbot: Analyzing the impact of breaking changes to assist library evolution. In Liliana Pasquale and Christoph Treude, editors, *44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results ICSE (NIER) 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 26–30. IEEE/ACM, 2022. doi:10.1109/ICSE-NIER55298.2022.9793524.

[25] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. doi:10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

[26] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the API usage in Java. *Information and Software Technology*, 73:81–100, May 2016. ISSN 09505849. doi:10.1016/j.infsof.2016.01.011.

[27] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. Empirical standards for software engineering research. *arXiv preprint arXiv:2010.03525*, 2020.

[28] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[29] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. *IIEEE Trans. Software Eng.*, 39(5):613–637, May 2013. ISSN 0098-5589, 1939-3520. doi:10.1109/TSE.2012.63. URL http://ieeexplore.ieee.org/document/6311409/.

[30] Anand Ashok Sawant and Alberto Bacchelli. fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage. *Empirical Software Engineering*, 22(3):1348–1371, 2017.

[31] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126. IEEE, September 2009. ISBN 978-1-4244-4876-0. doi:10.1109/VLHCC.2009.5295283. URL http://ieeexplore.ieee.org/document/5295283/.

[32] Suresh Thummalapenta and Tao Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336. IEEE, 2008.

[33] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining api usage scenarios from stack overflow. *Information and Software Technology*, 122:106277, 2020.

[34] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A. Gerosa, and Igor Scaliante Wiese. I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. *ACM Trans. Softw. Eng. Methodol.*, 32(4):94:1–94:26, May 2023. ISSN 1049-331X. doi:10.1145/3576037. URL https://doi.org/10.1145/3576037.

[35] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328. IEEE, 2013.

[36] Hyrum Wright. Hyrum's law, 2017. URL https://www.hyrumslaw.com/.

[37] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, 2017.

[38] Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. How libraries evolve: A survey of two industrial companies and an open-source community. In *29th Asia-Pacific Software Engineering Conference, APSEC*

2022, Virtual Event, Japan, December 6-9, 2022, pages 309–317. IEEE, 2022. doi:10.1109/APSEC57359.2022.00043.

[39] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing, September 2022. URL http://arxiv.org/abs/2209.00393.

[40] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th international conference on software engineering*, pages 886–896, 2018.

[41] Hao Zhong and Hong Mei. An empirical study on API usages. *IEEE Trans. Software Eng.*, 45(4):319–334, 2019. doi:10.1109/TSE.2017.2782280.

[42] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*, pages 318–343. Springer, 2009.