LaBRI

université de BORDEAUX

THÈSE

PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR

DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE
MATHÉMATIQUES ET INFORMATIQUE

SPÉCIALITÉ INFORMATIQUE

Par **Gustave Monce**

Diminuer les frictions entre les utilisateurs et les mainteneurs de bibliothèques logicielles

---

Soutenue le Mercredi 1er Avril, 2026

Membres du Jury :

Mme Anne Etien	Professeure des universités, Université Lille 1	Rapporteure
M. Jannik Laval	Maître de Conférences, Université Lumière Lyon 2 ..	Rapporteur
M. Djamel Eddine-Khelladi	Chargé de Recherche, CNRS	Examinateur
M. Hicham Lakhlef	Professeur des universités, Bordeaux INP	Président
M. Jean-Rémy Falleri ...	Professeur des universités, Bordeaux INP	Directeur
M. Thomas Degueule	Chargé de Recherche, CNRS	Co-directeur

To my grand fathers...

Acknowledgement

[TODO Write this section]

Remerciement

[TODO Write this section]

Contents

1 Introduction

This chapter provides an overview of the research context of this thesis, including its background, motivation, and key findings and contributions. Lastly, this chapter concludes by describing the outline of this thesis in Section 1.4. An English summary of our research is present in Chapter 1, as well as a French summary in Chapter 2.

Chapter Contents

1.1 Context

Modern software development heavily relies on third-party libraries. These libraries have become a cornerstone of contemporary software engineering, enabling developers to rapidly build complex systems by reusing existing functionality. For example, the most widely adopted library in the Java ecosystem counts more than 136,645 registered clients on Maven Central as of Wednesday 27th August, 2025. Such numbers illustrate the central role that libraries play in today's software ecosystems.

A key advantage of libraries is their re-usability across multiple projects. Rather than re-implementing common functionality, clients depend on libraries to acquire features that would otherwise require significant development effort. By leveraging third-party libraries, client developers can simplify their development workflows and focus on what matters most: their application-specific logic. The design, implementation, and maintenance of reusable functionality are instead delegated to dedicated library development teams [Cox+15; Cox19; Ngu+10].

To make use of a third-party library, clients declare a dependency towards a specific version or version range in their build system or package manager. These dependencies are then automatically retrieved from remote package repositories and included during compilation, packaging, or execution. Well-known repositories include npm for JavaScript, NuGet for C#, PyPi for Python, or Maven Central for Java [Foo+18; MNT20]. Managing dependencies is therefore

straightforward: adding, removing, or updating a library involves modifying the corresponding declaration in the build configuration.

Following seminal work on information hiding and modularity [Par72], client developers interact with libraries exclusively through well-defined interfaces. These interactions are governed by an API (Application Programming Interface), which constitutes the explicit boundary between a client and a library. The API exposes a set of visible symbols (such as classes, methods, or fields in the Java programming language) that client developers can interact with without knowledge of the underlying implementation. Depending on whether the library is distributed in source or compiled form, this boundary may be defined at the source-code level or at the binary level, in which case it is referred to as an ABI (Application Binary Interface).

The API specifies not only which symbols are accessible, but also how they are intended to be used. Through language-level mechanisms such as visibilities and module definitions, library developers explicitly configure which symbols in their code are exported and made available to clients, and how these symbols are meant to be manipulated in client code.

As clients and libraries are linked together through a dependency relationship, clients and libraries co-evolve together over time. When libraries release new versions (to incorporate new features, fix bugs, conduct refactorings, fix vulnerabilities, etc.), this co-evolution relationship (Figure 1.1) requires clients to adapt to new versions of their dependencies. However, new versions may introduce BCs (Breaking Changes) in the library's API, which can invalidate existing client code and force developers to update their code to deal with compilation errors or unintended behavioral changes [Och+22; Jay+24].

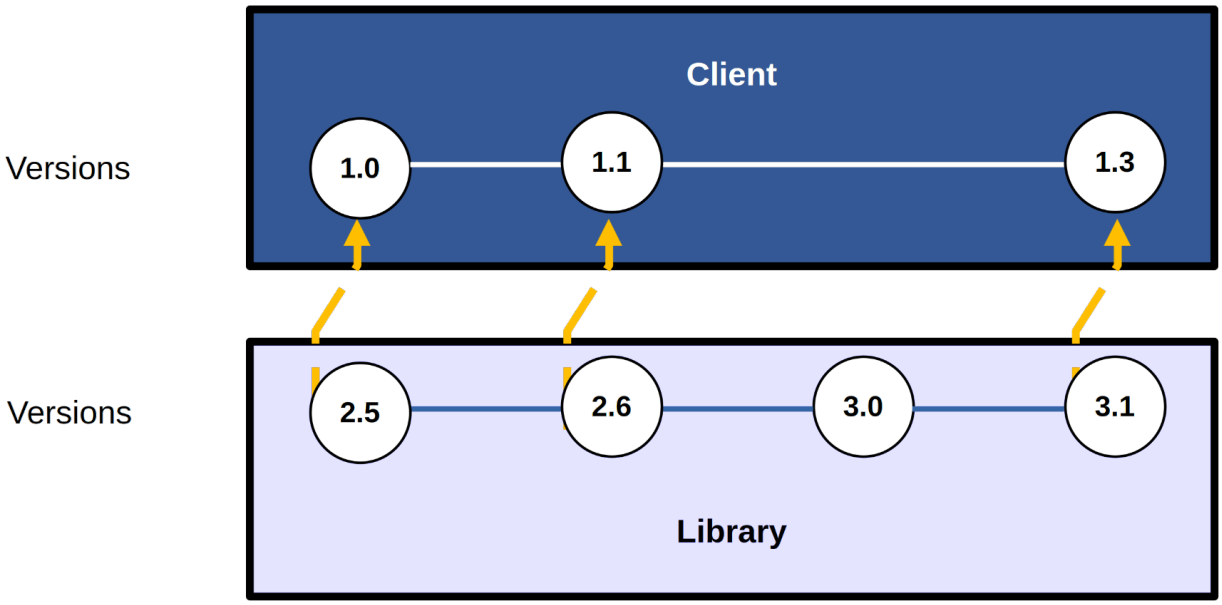


Fig. 1.1 A client and a library co-evolving together with each update of the library or the client.

Each yellow arrow symbol (←) represents the client version (destination) adopting a new version of a library (origin). After version `3.0` of the library, the client did not directly update because BCs (Breaking Changes) added more cost to client developers.

The challenges induced by library–client co-evolution affect both sides of the relationship. On the library side, it is often difficult for maintainers to understand precisely which API elements are

exported, intended for use, tested, and documented. As a result, libraries may unintentionally expose elements that are relied upon by clients. On the client side, it is difficult for developers to assess the impact of library updates on their codebase. Updating dependencies can therefore expose clients to unexpected breaking changes, threatening software stability and maintainability.

1.2 Problem Statement

The context of software engineering today presents numerous challenges related to software dependencies. From dependency management and maintenance to security issues, new functionality, or bug fixes, maintaining dependencies today is complicated and often a source of additional work for developers. In this section, we detail some of these problems.

1.2.1 Problem 1

> **Problem ❶**
>
> Library developers can hardly reason about the interactions allowed by their API and how their libraries are used in the wild

Defining precise boundaries for a library API is a fundamental yet challenging task. The API governs how clients access library functionality by specifying which symbols are exposed and which interactions with them are permitted. As such, it directly influences both the usability and long-term maintainability of the library.

Designing an API that accurately reflects the developer's intent is difficult. Modern programming languages provide mechanisms such as visibility rules, subclassing, sealing, overloading, and polymorphism that implicitly enable interactions beyond those explicitly intended. The interaction of these features can lead to the accidental exposure of API symbols or to unintended forms of interaction with them. As a result, library developers often struggle to maintain a complete and accurate understanding of all usages their API allows.

This challenge is further exacerbated by variations in programming styles and architectural patterns, including IoC (Inversion of Control), fluent APIs, and framework-oriented designs. These styles encourage indirect or declarative interactions, making it harder to reason about the full set of permitted API usages.

Unintended API usages have significant consequences. They may cause clients to depend on behaviors that were not part of the original design intent, increasing coupling and restricting the library's ability to evolve. Over time, such dependencies can lead to subtle breakages and backward-compatibility issues.

This lack of understanding directly hinders library evolution. Because multiple forms of BCs may arise from library changes, developers are often reluctant to modify their code for fear of

breaking existing clients. Without a clear model of which usages are allowed and which are actually relied upon, it becomes difficult to evaluate the impact of changes or to evolve one API symbol without risking others. Understanding both the space of permitted usages and their prevalence in practice is therefore essential for performing syntactic and behavioral evolution safely.

To reduce these risks, developers sometimes adopt defensive practices, such as minimizing accessibility or avoiding certain language features altogether (for example, in Java [BBlo08]). While these practices may limit accidental exposure, they do not provide a systematic way to verify that an API admits only the intended symbols and interactions.

A related challenge concerns documentation and testing. Because developers often lack visibility into how their API is actually used, documentation and examples may not reflect all permitted interactions. Similarly, without understanding which usages are exercised by clients, it is difficult to prioritize documentation, guide test development, or identify the most critical parts of the library to maintain.

In summary, library developers currently lack effective means to:

- characterize the full range of usages their API allows,
- verify that these usages align with their design intent
- leverage real-world usage information to guide safe evolution, documentation, and testing.

Several approaches in the literature aim to extract API usage data from client code to identify over- or under-utilized portions of an API [Sty+09; SB17; TX08]. These approaches rely on different sources, including bytecode [Har+22], source code, resolved ASTs (Abstract Syntax Trees) [QLL16; LPS11; DLP13], and other artifacts [Sty+09]. However, they primarily focus on whether an API symbol is accessed by client code. They do not capture how a symbol is used (for example, in Java, whether a method is invoked or overridden, or whether a class is instantiated or extended), nor do they characterize the full set of interactions permitted by the API.

This is why we propose the following new challenge:

> **Challenge ❶**
>
> How to precisely model and infer library usage for developers, to assist developers in making evolutions to their library?

We answer this specific challenge via our first contribution, detailed in the next section. Answering the above challenge will better guide library developers towards safely making evolutions for their own software.

1.2.2 Problem 2

> **Problem ❷**
>
> Clients can hardly reason about the impact of new releases on their code

Client developers must continuously decide whether to update their library dependencies. Although updates provide bug fixes, security patches, and new functionality, they may also introduce BCs, including semantic and behavioral changes. Such changes affect runtime behavior and can alter the behavior of a client or cause failures.

This creates a fundamental trade-off. Updating dependencies reduces exposure to known bugs and vulnerabilities, but risks introducing regressions that are difficult to detect or debug. As a result, developers often delay updates, pin specific versions, or avoid automatic updates altogether [HVK25]. Recent security incidents in widely used libraries, such as `SolarWinds`, `xz`, and the `Log4Shell` vulnerability, further intensify this dilemma. While these incidents discourage updates due to fear of regressions, the availability of fixes often forces rapid updates without sufficient impact assessment.

Conversely, avoiding updates prevents clients from benefiting from improvements and security patches, leading to technical debt and prolonged vulnerability exposure. Thus, both updating and not updating dependencies impose costs on developers.

To reduce the manual effort of dependency management [Jai+24], developers increasingly rely on automatic dependency update tools. However, this practice amplifies exposure to BeBCs. Developers commonly validate updates using release notes and test suites, yet release notes may be incomplete or inaccurate [Wu+22], and test suites often fail to capture changes in runtime behavior. Unlike SyBCs, BeBCs cannot be reliably detected using static analysis, making them particularly difficult to identify at update time. As a result, such regressions may only become apparent in production, thereby increasing maintenance and debugging costs.

Assessing behavioral changes is further complicated by the fact that their impact depends on how a client uses a library API. A behavioral change may be harmful, beneficial, or neutral depending on the client's expectations and usage patterns. For example, a bug fix may resolve issues for some clients while breaking others. Currently, there is no practical way to assess the impact of behavioral changes for a specific client, as client expectations are rarely explicit or machine-readable.

Although client test suites can be executed after updates, they typically provide limited coverage of library API calls. Clients often test their own logic rather than directly asserting API call results. Writing dedicated tests for every library interaction would improve detection, but it is costly and difficult to maintain. Empirical studies show that test suites are frequently insufficient for detecting behavioral changes [Jay+24; Gyo+18; HG22]. Similarly, library test suites are written by library developers and do not reflect the diverse usage patterns of dependent clients; passing them does not guarantee behavioral compatibility.

As a result, client developers lack effective mechanisms to detect and assess BeBCs introduced by dependency updates. This uncertainty forces them to choose between risky updates and

costly stagnation, despite the presence of known security vulnerabilities and regressions in library dependencies. Detecting behavioral regressions is therefore essential for enabling informed update decisions and protecting clients from unintended breakage.

Several approaches have attempted to address this problem. Sembid uses static analysis to diff API methods across versions and flag potential BeBCs [Zha+22a], but it often produces excessive false positives. DeBBI aggregates and executes test suites from multiple clients using the same library [Che+20], but this approach is resource-intensive and may miss regressions due to weak or tolerant assertions. Uppdatera analyzes control- and data-flow differences in methods reached by a client [HG22], yet its strict diffing rules can lead to false positives. Other approaches leverage crowd-sourced tests from different clients [Zhu+23], but they may suffer from false negatives due to insufficiently sensitive assertions.

Overall, existing techniques struggle to balance precision, scalability, and usefulness for client developers. We argue that a new approach is needed–one that better captures client-relevant behavior, limits false results, and remains practical to adopt–thereby enabling safer and more confident dependency updates.

This is why we propose the following new challenge:

> **Challenge ❷**
>
> How can clients defend themselves against potential regressions?

We answer this specific challenge via our second contribution, detailed in the next section. Answering the above challenge will help client developers defend against behavioral changes in the wild within their dependencies.

1.3 Contributions

To address the specific challenges outlined in the previous section and resolve the problems highlighted previously, we make two main contributions as part of this thesis. The first contribution is oriented towards library developers, while the second is oriented towards library consumers (the clients).

> In this thesis, we focus exclusively on solving the previously mentioned challenges for the Java programming language. Java is a popular, object-oriented, cross-platform programming language, and is used extensively in the literature with numerous datasets available. While the concepts can be applied to other programming languages, they would each require dedicated attention. We provide proof of concept implementations of our contributions for the Java programming language and evaluate them on Java client and library datasets.

1.3.1 Lightweight Syntactic API Usage Analysis

Chapter 4 focuses on the static analysis of usages in the wild in clients of libraries. This chapter notably presents UCov, a proof of concept implementation we implemented for modeling the uses of Java libraries from clients, documentation, or tests. This constitutes our first significant contribution, oriented towards library developers.

A significant highlight of the results of this contribution is the discovery of unexpected uses from 3 popular Java libraries, each featuring a different API usage style. We observe discrepancies between what the clients is looking for in some cases, as well as documentation blind spots or deprecated API usage from clients. We demonstrate how modeling the list of possible uses, as well as those currently in use, is relevant and helpful to library API designers.

This chapter is based on published works mentioned below and answers **Challenge ❶**.

Published Conference Paper

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. “Lightweight Syntactic API Usage Analysis with UCov”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC ’24: 32nd IEEE/ACM International Conference on Program Comprehension. Lisbon Portugal: ACM, Apr. 15, 2024, pp. 426–437. DOI: 10.1145/3643916.3644415. URL: https://dl.acm.org/doi/10.1145/3643916.3644415

As part of this contribution, we also make available our own proof of concept implementation on GitHub [SMon+], as well as the dataset used for the study available in the published conference paper [SMon+24a].

Published Artifacts

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. *Artifacts for "Lightweight Syntactic API Usage Analysis with UCov"*. Version 1.0.0. Jan. 26, 2024. DOI: 10.5281/ZENODO.10571867. URL: https://zenodo.org/doi/10.5281/zenodo.10571867

Proof of Concept Implementation

[SW] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri, *UCov, Alien-Tools – GitHub*. URL: https://github.com/alien-tools/ucov

1.3.2 Client–Library Compatibility Testing with API Interaction Snapshots

Chapter 5 focuses on API regression testing for BeBCs in libraries.

This chapter notably presents Gilesi, a proof of concept implementation for our novel approach to BeBC detection and testing. This constitutes our second significant contribution, oriented towards clients, consumers of libraries.

A significant highlight of this contribution's results is the ability to detect behavioral changes with fewer false positives compared to current solutions. Our approach involves using the client own test suite on a select set of Java client and library pairs. Still, it focuses on analyzing the changes at the boundary, *i.e.*, the API between the client and the library, and not through static analysis, data flow/code flow analysis, or via the results of the client test suite. This should enable developers to get more accurate results without needing to dive into the code of the library as well as quickly determine if the observed changes are relevant to their own usage.

This chapter is based on published works mentioned below, and contains extended contents and results compared to the published NIER (New Ideas and Emerging Results) version [SMon+25b], which included only emerging results. It also answers **Challenge ❷**.

Published Conference Paper

Gustave Monce, Thomas Degueule, Jean-Rémy Falleri, and Romain Robbes. "Client–Library Compatibility Testing with API Interaction Snapshots". In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME). Auckland, New Zealand: IEEE, 2025

As part of this contribution, we also make available our own proof of concept implementation on GitHub [SMDF], as well as the dataset used for the study available in the published conference paper [SMon+25a].

Published Artifacts

Gustave Monce, Thomas Degueule, Jean-Rémy Falleri, and Romain Robbes. *Artifacts for "Client–Library Compatibility Testing with API Interaction Snapshots"*. Version 1.0.0. July 24, 2025. DOI: 10.5281/ZENODO.16411966. URL: https://zenodo.org/doi/10.5281/zenodo.16411966

Proof of Concept Implementation

[SW] Gustave Monce, Thomas Degueule, and Jean-Rémy Falleri, *Gilesi, Alien-Tools – GitHub*. URL: https://github.com/alien-tools/gilesi

1.4 Outline

The remaining chapters in this thesis are structured as follows (the state of the art is spread between our two distinct contribution areas):

Part I **Introduction and Background**

Chapter 3 ***Background***
presents the background of this research as it exists in the literature, defines key concepts relevant to this work, and outlines the current state of software ecosystems.

Part II **Contributions**

Chapter 4 ***Lightweight Syntactic API Usage Analysis***
presents the methodology and results for analyzing interaction use cases in libraries from client, and it discusses how to address the current lack of visibility and actions a library has on its usages. It also features the **state of the art** related to this chapter.

Chapter 5 ***Client-Library Compatibility Testing with API Interaction Snapshots***
presents the methodology and results for analyzing the behavioral changes that can occur between a library and one of its client. It also addresses such behavioral changes by showcasing a novel method for detecting such perturbations in the wild. This contribution also highlights the results of this methodology. It also features the **state of the art** related to this chapter.

Part III **Conclusion and Perspectives**

Chapter 6 ***Conclusion and Perspectives***
summarizes all findings, methodology, and new approaches; presents potential future research directions. It also concludes this thesis.

2 Introduction (En Français)

Ce chapitre fournit une vue d'ensemble du contexte de la recherche, incluant son historique, sa motivation, ainsi que les principaux résultats et contributions. Enfin, ce chapitre se conclut en présentant la structure de cette thèse dans Section 2.4. Un résumé en anglais de notre recherche est disponible dans Chapitre 1, ainsi qu'un résumé en français dans Chapitre 2.

Contenu du Chapitre

2.1 Contexte

Le développement logiciel moderne fait usage de bibliothèques tierces. Ces bibliothèques sont devenues une pierre angulaire de l'ingénierie logicielle contemporaine, permettant aux développeurs de construire rapidement des systèmes complexes en réutilisant des fonctionnalités existantes. Par exemple, la bibliothèque la plus largement adoptée dans l'écosystème Java compte plus de 136,645 clients enregistrés sur Maven Central à la date du mercredi 27 août 2025.De tels chiffres illustrent le rôle central que jouent les bibliothèques dans les écosystèmes logiciels actuels.

Un avantage clé des bibliothèques est leur réutilisabilité à travers de multiples projets. Plutôt que de réimplémenter des fonctionnalités courantes, les clients dépendent des bibliothèques pour acquérir des fonctionnalités qui nécessiteraient autrement un effort de développement important. En tirant parti des bibliothèques tierces, les développeurs de client peuvent simplifier leurs processus de développement et se concentrer sur l'essentiel : leur logique applicative spécifique. La conception, l'implémentation et la maintenance de fonctionnalités réutilisables sont ainsi déléguées à des équipes dédiées au développement de bibliothèques [Cox+15 ; Cox19 ; Ngu+10].

Pour utiliser une bibliothèque tierce, les clients déclarent une dépendance vers une version spécifique ou un intervalle de versions dans leur système de build ou leur gestionnaire de paquets. Ces dépendances sont ensuite automatiquement récupérées depuis des dépôts de

paquets distants et incluses lors de la compilation, du packaging ou de l'exécution. Parmi les dépôts bien connus figurent npm pour JavaScript, NuGet pour C#, PyPi pour Python, ou Maven Central pour Java [Foo+18 ; MNT20]. La gestion des dépendances est donc simple : ajouter, supprimer ou mettre à jour une bibliothèque consiste à modifier la déclaration correspondante dans la configuration de build.

Suite aux travaux fondateurs sur la dissimulation d'information et la modularité [Par72], les développeurs de client interagissent avec les bibliothèques exclusivement à travers des interfaces bien définies. Ces interactions sont régies par une API (Interface de Programmation Applicative), qui constitue la frontière explicite entre un client et une bibliothèque. L'API expose un ensemble de symboles visibles (tels que des classes, des méthodes ou des champs dans le langage de programmation Java) avec lesquels les développeurs de client peuvent interagir sans connaissance de l'implémentation sous-jacente. Selon que la bibliothèque est distribuée sous forme de code source ou compilée, cette frontière peut être définie au niveau du code source ou au niveau binaire, auquel cas elle est appelée une ABI (Interface Binaire Applicative).

L'API spécifie non seulement quels symboles sont accessibles, mais aussi comment ils sont destinés à être utilisés. Au moyen de mécanismes propres au langage tels que les visibilités et les définitions de modules, les développeurs de bibliothèque configurent explicitement quels symboles de leur code sont exportés et rendus disponibles aux clients, et comment ces symboles doivent être manipulés dans le code des client.

Étant donné que les clients et les bibliothèques sont liés par une relation de dépendance, ils co-évoluent au fil du temps. Lorsque les bibliothèques publient de nouvelles versions (afin d'intégrer de nouvelles fonctionnalités, corriger des bugs, effectuer des refactorings, corriger des vulnérabilités, etc.), cette relation de co-évolution (Figure 2.1) oblige les clients à s'adapter aux nouvelles versions de leurs dépendances. Cependant, les nouvelles versions peuvent introduire des changements cassants (BC (Breaking Change)) dans l'API de la bibliothèque, ce qui peut invalider le code existant des clients et contraindre les développeurs à mettre à jour leur code pour gérer des erreurs de compilation ou des changements de comportement non intentionnels [Och+22 ; Jay+24].

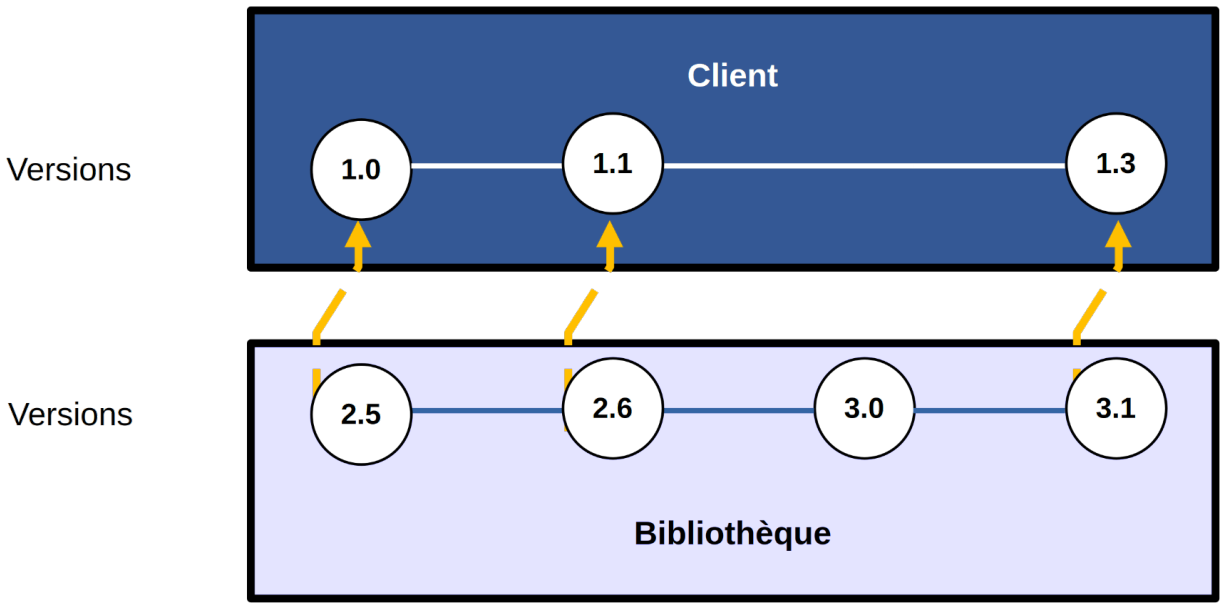


Fig. 2.1 Un client et une bibliothèque co-évoluant ensemble à chaque mise à jour de la bibliothèque ou du client.

Chaque symbole de flèche jaune (←) représente la version du client (destination) adoptant une nouvelle version d'une bibliothèque (origine). Après la version `3.0` de la bibliothèque, le client n'a pas été mis à jour directement car les changements cassants (BCs (Breaking Changes)) ont ajouté un coût supplémentaire pour les développeurs de client.

Les défis induits par la co-évolution bibliothèque–client affectent les deux côtés de la relation. Du côté des bibliothèques, il est souvent difficile pour les mainteneurs de comprendre précisément quels éléments de l'API sont exportés, destinés à l'usage, testés et documentés. En conséquence, les bibliothèques peuvent exposer involontairement des éléments sur lesquels les clients s'appuient. Du côté des clients, il est difficile pour les développeurs d'évaluer l'impact des mises à jour de bibliothèque sur leur base de code. La mise à jour des dépendances peut ainsi exposer les clients à des changements incompatibles inattendus, menaçant la stabilité et la maintenabilité des logiciels.

2.2 Problématique

Le contexte de l'ingénierie logicielle actuelle présente de nombreux défis liés aux dépendances logicielles. De la gestion et de la maintenance des dépendances aux problèmes de sécurité, en passant par les nouvelles fonctionnalités ou les correctifs de bogues, maintenir des dépendances est aujourd'hui une tâche complexe et souvent une source de travail supplémentaire pour les développeurs. Dans cette section, nous détaillons certains de ces problèmes.

2.2.1 Problème 1

> **Problème ❶**
>
> Les développeurs de bibliothèque peuvent difficilement raisonner sur les interactions autorisées par leur API et sur la manière dont leurs bibliothèques sont utilisées en pratique

Définir des frontières précises pour l'API d'une bibliothèque est une tâche fondamentale mais difficile. L'API régit la manière dont les clients accèdent aux fonctionnalités de la bibliothèque en spécifiant quels symboles sont exposés et quelles interactions avec ceux-ci sont autorisées. À ce titre, elle influence directement à la fois l'utilisabilité et la maintenabilité à long terme de la bibliothèque.

Concevoir une API qui reflète fidèlement l'intention du développeur est difficile. Les langages de programmation modernes fournissent des mécanismes tels que les règles de visibilité, l'héritage, le scellement, la surcharge et le polymorphisme, qui permettent implicitement des interactions allant au-delà de celles explicitement prévues. L'interaction de ces fonctionnalités peut conduire à l'exposition accidentelle de symboles de l'API ou à des formes d'interaction non intentionnelles avec ceux-ci. En conséquence, les développeurs de bibliothèque peinent souvent à maintenir une compréhension complète et précise de l'ensemble des usages que leur API autorise.

Ce défi est encore accentué par la diversité des styles de programmation et des patterns architecturaux, notamment l'inversion de contrôle (IoC (Inversion of Control)), les APIs fluent et les conceptions orientées framework. Ces styles encouragent des interactions indirectes ou déclaratives, rendant plus difficile le raisonnement sur l'ensemble des usages permis par l'API.

Les usages non intentionnels de l'API ont des conséquences importantes. Ils peuvent amener les clients à dépendre de comportements qui ne faisaient pas partie de l'intention de conception initiale, augmentant le couplage et restreignant la capacité de la bibliothèque à évoluer. Au fil du temps, de telles dépendances peuvent conduire à des ruptures subtiles et à des problèmes de compatibilité ascendante.

Ce manque de compréhension entrave directement l'évolution des bibliothèques. Comme de multiples formes de changements cassants (BCs) peuvent résulter de changements dans une bibliothèque, les développeurs hésitent souvent à modifier leur code par crainte de casser des clients existants. Sans un modèle clair des usages autorisés et de ceux effectivement exploités, il devient difficile d'évaluer l'impact des changements ou de faire évoluer un symbole de l'API sans risquer d'en affecter d'autres. Comprendre à la fois l'espace des usages permis et leur prévalence en pratique est donc essentiel pour réaliser des évolutions syntaxiques et comportementales de manière sûre.

Pour réduire ces risques, les développeurs adoptent parfois des pratiques défensives, telles que la minimisation de l'accessibilité ou l'évitement pur et simple de certaines fonctionnalités du langage (par exemple, en Java [BBlo08]). Bien que ces pratiques puissent limiter les expositions accidentelles, elles ne fournissent pas de moyen systématique de vérifier qu'une API n'admet que les symboles et les interactions intentionnellement prévus.

Un défi connexe concerne la documentation et les tests. Les développeurs manquant souvent de visibilité sur la manière dont leur API est réellement utilisée, la documentation et les exemples peuvent ne pas refléter l'ensemble des interactions permises. De même, sans comprendre quels usages sont exercés par les clients, il est difficile de prioriser la documentation, d'orienter le développement des tests ou d'identifier les parties les plus critiques de la bibliothèque à maintenir.

En résumé, les développeurs de bibliothèque manquent actuellement de moyens efficaces pour :

- caractériser l'ensemble des usages que leur API autorise,
- vérifier que ces usages sont alignés avec leur intention de conception,
- exploiter des informations issues des usages réels pour guider l'évolution sûre, la documentation et les tests.

Plusieurs approches dans la littérature visent à extraire des données d'usage des APIs à partir du code des clients afin d'identifier les parties sur- ou sous-utilisées d'une API [Sty+09 ; SB17 ; TX08]. Ces approches s'appuient sur différentes sources, notamment le bytecode [Har+22], le code source, les ASTs (Abstract Syntax Trees) résolus [QLL16 ; LPS11 ; DLP13], ainsi que d'autres artefacts [Sty+09]. Cependant, elles se concentrent principalement sur le fait qu'un symbole de l'API est accédé par le code des clients. Elles ne capturent pas la manière dont un symbole est utilisé (par exemple, en Java, si une méthode est invoquée ou redéfinie, ou si une classe est instanciée ou étendue), ni ne caractérisent l'ensemble complet des interactions autorisées par l'API.

C'est pourquoi nous proposons le nouveau défi suivant :

> **Défi ❶**
>
> Comment modéliser et inférer précisément les usages d'une bibliothèque pour les développeurs, afin de les assister dans la réalisation d'évolutions de leur bibliothèque ?

Nous répondons à ce défi spécifique par notre première contribution, détaillée dans la section suivante. Répondre au défi ci-dessus permettra de mieux guider les développeurs de bibliothèque vers des évolutions sûres de leur propre logiciel.

2.2.2 Problème 2

> **Problème ❷**
>
> Les clients peuvent difficilement raisonner sur l'impact des nouvelles versions sur leur code

Les développeurs de client doivent décider en permanence s'ils doivent mettre à jour leurs dépendances vers des bibliothèques. Bien que les mises à jour apportent des corrections de bogues, des correctifs de sécurité et de nouvelles fonctionnalités, elles peuvent également introduire des BCs, y compris des changements sémantiques et comportementaux. De tels

changements affectent le comportement à l'exécution et peuvent modifier le comportement d'un client ou provoquer des défaillances.

Cela crée un compromis fondamental. Mettre à jour les dépendances réduit l'exposition à des bogues et vulnérabilités connus, mais comporte le risque d'introduire des régressions difficiles à détecter ou à déboguer. En conséquence, les développeurs retardent souvent les mises à jour, figent des versions spécifiques ou évitent complètement les mises à jour automatiques [HVK25]. Des incidents de sécurité récents dans des bibliothèques largement utilisées, tels que `SolarWinds`, `xz` et la vulnérabilité `Log4Shell`, intensifient encore ce dilemme. Bien que ces incidents découragent les mises à jour par crainte de régressions, la disponibilité de correctifs impose souvent des mises à jour rapides sans évaluation suffisante de l'impact.

À l'inverse, éviter les mises à jour empêche les clients de bénéficier des améliorations et des correctifs de sécurité, entraînant une dette technique et une exposition prolongée aux vulnérabilités. Ainsi, tant la mise à jour que l'absence de mise à jour des dépendances imposent des coûts aux développeurs.

Afin de réduire l'effort manuel lié à la gestion des dépendances [Jai+24], les développeurs s'appuient de plus en plus sur des outils de mise à jour automatique des dépendances. Cependant, cette pratique accroît l'exposition aux changements de comportement cassants (BeBCs). Les développeurs valident couramment les mises à jour à l'aide des notes de version et des suites de tests, mais les notes de version peuvent être incomplètes ou inexactes [Wu+22], et les suites de tests échouent souvent à capturer les changements de comportement à l'exécution. Contrairement aux changements syntaxiques cassants (SyBCs), les BeBCs ne peuvent pas être détectées de manière fiable à l'aide de l'analyse statique, ce qui les rend particulièrement difficiles à identifier au moment de la mise à jour. En conséquence, de telles régressions peuvent ne devenir apparentes qu'en production, augmentant ainsi les coûts de maintenance et de débogage.

L'évaluation des changements comportementaux est encore compliquée par le fait que leur impact dépend de la manière dont un client utilise l'API d'une bibliothèque. Un changement comportemental peut être nuisible, bénéfique ou neutre selon les attentes et les schémas d'utilisation du client. Par exemple, une correction de bogue peut résoudre des problèmes pour certains clients tout en en cassant d'autres. À l'heure actuelle, il n'existe pas de moyen pratique d'évaluer l'impact des changements comportementaux pour un client spécifique, car les attentes des clients sont rarement explicites ou lisibles par machine.

Bien que les suites de tests des clients puissent être exécutées après des mises à jour, elles offrent généralement une couverture limitée des appels à l'API des bibliothèques. Les clients testent souvent leur propre logique plutôt que d'asserter directement les résultats des appels à l'API. Écrire des tests dédiés pour chaque interaction avec une bibliothèque améliorerait la détection, mais cela est coûteux et difficile à maintenir. Des études empiriques montrent que les suites de tests sont fréquemment insuffisantes pour détecter des changements comportementaux [Jay+24 ; Gyo+18 ; HG22]. De même, les suites de tests des bibliothèques sont écrites par les développeurs de bibliothèque et ne reflètent pas la diversité des schémas d'utilisation des clients dépendants ; les réussir ne garantit pas la compatibilité comportementale.

En conséquence, les développeurs de client manquent de mécanismes efficaces pour détecter et évaluer les BeBCs introduites par les mises à jour de dépendances. Cette incertitude les contraint à choisir entre des mises à jour risquées et une stagnation coûteuse, malgré la présence de vulnérabilités de sécurité et de régressions connues dans les dépendances de bibliothèque. La détection des régressions comportementales est donc essentielle pour permettre des décisions de mise à jour éclairées et protéger les clients contre des ruptures non intentionnelles.

Plusieurs approches ont tenté de répondre à ce problème. Sembid utilise l'analyse statique pour différencier les méthodes de l'API entre versions et signaler des BeBCs potentielles [Zha+22a], mais produit souvent un nombre excessif de faux positifs. DeBBI agrège et exécute les suites de tests de plusieurs clients utilisant la même bibliothèque [Che+20], mais cette approche est coûteuse en ressources et peut manquer des régressions en raison d'assertions faibles ou tolérantes. Uppdatera analyse les différences de flot de contrôle et de données dans les méthodes atteintes par un client [HG22], mais ses règles de différenciation strictes peuvent conduire à des faux positifs. D'autres approches exploitent des tests issus du crowdsourcing provenant de différents clients [Zhu+23], mais elles peuvent souffrir de faux négatifs dus à des assertions insuffisamment sensibles.

Dans l'ensemble, les techniques existantes peinent à concilier précision, passage à l'échelle et utilité pour les développeurs de client. Nous soutenons qu'une nouvelle approche est nécessaire – une approche qui capture mieux les comportements pertinents pour les clients, limite les résultats erronés et demeure praticable à adopter – afin de permettre des mises à jour de dépendances plus sûres et plus confiantes.

C'est pourquoi nous proposons le nouveau défi suivant :

> **Défi ❷**
>
> Comment les clients peuvent-ils se défendre contre des régressions potentielles ?

Nous répondons à ce défi spécifique par notre deuxième contribution, détaillée dans la section suivante. Répondre au défi ci-dessus aidera les développeurs de client à se défendre contre les changements comportementaux observés en pratique au sein de leurs dépendances.

2.3 Contributions

Pour relever les défis spécifiques décrits dans la section précédente et résoudre les problèmes mis en évidence auparavant, nous apportons deux contributions principales dans le cadre de cette thèse. La première contribution est orientée vers les développeurs de bibliothèque, tandis que la seconde est orientée vers les consommateurs de bibliothèque (les clients).

Dans cette thèse, nous nous concentrons exclusivement sur la résolution des défis mentionnés précédemment pour le langage de programmation Java. Java est un langage de programmation populaire, orienté objet, multiplateforme, et il est largement utilisé dans la littérature avec de nombreux jeux de données disponibles. Bien que les concepts puissent être appliqués à d'autres langages de programmation, chacun nécessiterait une attention particulière. Nous fournissons des implémentations de preuve de concept de nos contributions pour le langage de programmation Java et les évaluons sur des jeux de données de client et de bibliothèque en Java.

2.3.1 Analyse syntaxique légère des usages d'API

Le Chapitre 4 se concentre sur l'analyse statique des usages observés en pratique dans les clients de bibliothèques. Ce chapitre présente notamment UCov, une implémentation de preuve de concept que nous avons développée pour modéliser les usages de bibliothèques Java à partir des clients, de la documentation ou des tests. Cela constitue notre première contribution majeure, orientée vers les développeurs de bibliothèque.

Un résultat marquant de cette contribution est la mise en évidence d'usages inattendus provenant de 3 bibliothèques Java populaires, chacune présentant un style d'usage de l'API différent. Nous observons, dans certains cas, des divergences entre ce que recherchent les clients, ainsi que des angles morts dans la documentation ou des usages obsolètes de l'API par les clients. Nous montrons en quoi la modélisation de la liste des usages possibles, ainsi que de ceux effectivement observés, est pertinente et utile pour les concepteurs d'API de bibliothèque.

Ce chapitre s'appuie sur les travaux publiés mentionnés ci-dessous et répond au **Défi ❶**.

Article de conférence publié

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule et Jean-Rémy Falleri. "Lightweight Syntactic API Usage Analysis with UCov". In : *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC '24 : 32nd IEEE/ACM International Conference on Program Comprehension. Lisbon Portugal : ACM, 15 avr. 2024, p. 426-437. DOI : 10.1145/3643916.3644415. URL : https://dl.acm.org/doi/10.1145/3643916.3644415

Dans le cadre de cette contribution, nous mettons également à disposition notre propre implémentation de preuve de concept sur GitHub [SMon+], ainsi que le jeu de données utilisé pour l'étude, disponible dans l'article de conférence publié [SMon+24a].

Artefacts publiés

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule et Jean-Rémy Falleri. *Artifacts for "Lightweight Syntactic API Usage Analysis with UCov"*. Version 1.0.0. 26 jan. 2024. DOI : 10.5281/ZENODO.10571867. URL : https://zenodo.org/doi/10.5281/zenodo.10571867

Implémentation de preuve de concept

[Log.] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule et Jean-Rémy Falleri, *UCov, Alien-Tools – GitHub*. URL : https://github.com/alien-tools/ucov

2.3.2 Tests de compatibilité client–bibliothèque à l'aide d'instantanés d'interactions d'API

Le Chapitre 5 se concentre sur les tests de régression d'API pour les BeBCs dans les bibliothèques.

Ce chapitre présente notamment Gilesi, une implémentation de preuve de concept de notre nouvelle approche de détection et de test des BeBCs. Cela constitue notre deuxième contribution majeure, orientée vers les clients, consommateurs de bibliothèques.

Un résultat marquant de cette contribution est la capacité à détecter des changements comportementaux avec moins de faux positifs que les solutions existantes. Notre approche consiste à utiliser la propre suite de tests du client sur un ensemble sélectionné de paires Java client et bibliothèque. Elle se concentre toutefois sur l'analyse des changements à la frontière, *i.e.*, l'API entre le client et la bibliothèque, et non au moyen d'analyses statiques, d'analyses de flot de données ou de flot de contrôle, ni via les résultats de la suite de tests du client. Cela devrait permettre aux développeurs d'obtenir des résultats plus précis sans avoir à plonger dans le code de la bibliothèque, ainsi que de déterminer rapidement si les changements observés sont pertinents pour leurs propres usages.

Ce chapitre s'appuie sur les travaux publiés mentionnés ci-dessous et contient des contenus et des résultats étendus par rapport à la version NIER (New Ideas and Emerging Results) publiée [SMon+25b], qui ne présentait que des résultats émergents. Il répond également au **Défi ❷**.

Article de conférence publié

Gustave Monce, Thomas Degueule, Jean-Rémy Falleri et Romain Robbes. "Client–Library Compatibility Testing with API Interaction Snapshots". In : *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME). Auckland, New Zealand : IEEE, 2025

Dans le cadre de cette contribution, nous mettons également à disposition notre propre implémentation de preuve de concept sur GitHub [SMDF], ainsi que le jeu de données utilisé pour l'étude, disponible dans l'article de conférence publié [SMon+25a].

Artefacts publiés

Gustave Monce, Thomas Degueule, Jean-Rémy Falleri et Romain Robbes. *Artifacts for "Client–Library Compatibility Testing with API Interaction Snapshots"*. Version 1.0.0. 24 juill. 2025. doi : 10.5281/ZENODO.16411966. url : https://zenodo.org/doi/10.5281/zenodo.16411966

Implémentation de preuve de concept

[Log.] Gustave Monce, Thomas Degueule et Jean-Rémy Falleri, *Gilesi, Alien-Tools – GitHub*. url : https://github.com/alien-tools/gilesi

2.4 Plan

Les chapitres restants de cette thèse sont structurés comme suit (l'état de l'art est réparti entre nos deux domaines de contribution distincts) :

Partie I **Introduction et contexte**

Chapitre 3 ***Contexte***
présente le contexte de cette recherche tel qu'il existe dans la littérature, définit les concepts clés pertinents pour ce travail et décrit l'état actuel des écosystèmes logiciels.

Partie II **Contributions**

Chapitre 4 ***Analyse syntaxique légère des usages d'API***
présente la méthodologie et les résultats pour l'analyse des cas d'usage d'interaction dans les bibliothèques à partir des clients, et discute des moyens de répondre au manque actuel de visibilité et d'actions qu'une bibliothèque possède sur ses usages. Elle présente également l'**état de l'art** lié à ce chapitre.

Chapitre 5 ***Tests de compatibilité Client-Bibliothèque à l'aide d'instantanés d'interactions d'API***
présente la méthodologie et les résultats pour l'analyse des changements comportementaux pouvant survenir entre une bibliothèque et l'un de ses clients. Elle traite également ces changements comportementaux en présentant une nouvelle méthode pour détecter de telles perturbations en pratique. Cette contribution met aussi en avant les résultats de cette méthodologie. Elle présente également l'**état de l'art** lié à ce chapitre.

Partie III **Conclusion et perspectives**

Chapitre 6 ***Conclusion et perspectives***
synthétise l'ensemble des résultats, de la méthodologie et des nouvelles approches ; présente des pistes de recherche futures potentielles. Elle conclut également cette thèse.

3 Background

This chapter presents the background notions of this research. We begin by explaining the fundamentals of modules in software engineering, detailing key concepts such as APIs (Application Programming Interfaces) and BCs (Breaking Changes), and showcasing them with examples. We also detail in Java the techniques used to enable software modules today, their publishing, discovery, consumption, and maintenance.

Chapter Contents

3.1 Software libraries

Parnas, as part of his seminal work on software design, introduced the principle of information hiding as a foundation for achieving software modularity [Par72]. Parnas argued that software systems should be decomposed into independent modules. These independent modules would each be responsible for a well-defined set of functionalities and would hide their internal design decisions from the rest of the system. Modules interact exclusively through explicitly defined interfaces, which specify how other parts of the system may use their services without relying on implementation details. This separation allows developers to reason about, modify, and evolve individual modules independently, making design decisions more critical than low-level code structure. By clearly defining module boundaries and interfaces, information hiding reduces ambiguity, limits the impact of change, and improves system maintainability.

Third-party software libraries represent a modern realization of these foundational ideas. A software library is an independently developed and maintained software component that provides reusable functionality to other programs. Unlike the modular decomposition envisioned by Parnas, where a single development team designs all modules of a system, libraries are typically developed by external parties and reused across multiple applications. Libraries expose an API (Application Programming Interface) intended for consumption by developers who are not involved in their implementation. These APIs may, however, not always be well-defined. While the organizational context differs, the underlying principle remains aligned with information

hiding: consumers interact with libraries solely through their interfaces, without knowledge of or dependence on their internal design or implementation choices.

When an application incorporates a library, the library becomes a dependency of that application. Dependencies enable developers to reuse existing functionality rather than re-implement it from scratch, allowing applications to build upon a growing ecosystem of specialized components. This reuse brings substantial benefits: development effort and costs are reduced, development workflows are simplified, and software quality can improve as libraries are maintained and evolved by dedicated teams [Ngu+19b]. The widespread adoption of third-party libraries reflects these advantages, and modern software systems routinely depend on numerous external components [Cox+15; Cox19; Ngu+10]. For example, applications written in Java rely on an average of 14 dependencies [Wan+18]. As a result, third-party libraries have become an essential element of contemporary software engineering.

The provided functionality can be diverse, as shown in a list of popular software libraries used in Java in Table 3.1.

We notice a few categories of libraries in this popular software library list:

Test frameworks such as `JUnit` for implementing test suite libraries like `Jupiter`.

Utilities or helpers that extend functionality of the base language, implement commonly wished functionalities missing from it, like `Commons Lang` or `Commons IO`.

JSON format parsers like `Jackson` or `Gson`.

Libraries offering similar functionalities are possible. For example, `Log4J` implements a logger for Java, but `SLF4J` also does. Having multiple libraries providing similar functionalities favors choice with differences in terms of API design, or options.

> **Contextualization**
>
> As part of this thesis, we focus exclusively on libraries written in the Java programming language. We also only consider consumers of those libraries written in the same programming language: Java. As a consequence, we also only consider local interactions, and we do not consider cross-language interactions, such as those found in network protocols and servers.

#	Name	Author	URL	Functionality offered	Clients
#1	JUnit	JUnit team	`junit.org`	Unit test framework	138,645
#2	SLF4J API Module	QOS.ch Sarl (Switzerland)	`slf4j.org`	Simple abstraction for various logging frameworks	77,814
#3	Guava	Google	`github.com/google/guava`	Utility classes, Google's collections, I/O classes, *etc.*	42,143
#4	Mockito Core	Szczepan Faber and friends	`site.mockito.org`	Core API and implementation for object mocking	35,907
#5	Jackson Databind	FasterXML, LLC	`github.com/FasterXML/jackson`	Jackson data-binding, turns serialized files into objects and vice versa	35,285
#6	`Commons Lang`	Apache	`commons.apache.org/proper/commons-lang`	`java.lang`'s utility classes	33,270
#7	Logback Classic Module	QOS.ch Sarl (Switzerland)	`logback.qos.ch`	Logback logging framework SLF4J API Implementation	31,254
#8	`Commons IO`	Apache	`commons.apache.org/proper/commons-io`	IO Utility classes, streams, file filters, comparators, endian transformers, *etc.*	31,002
#9	Project Lombok	The Project Lombok authors	`projectlombok.org`	Annotations that enables the generation of boilerplate code	30,167
#10	Gson	Google	`github.com/google/gson`	JSON serializer and deserializer	27,060
#11	AssertJ Core	AssertJ authors	`assertj.github.io/doc/#assertj-core`	Rich and fluent assertions for testing in Java	21,338
#12	Log4J	Apache	`logging.apache.org/log4j/1.2`	Legacy version of Log4J logging framework	19,806
#13	JUnit Jupiter API	JUnit team	`junit.org`	JUnit Jupiter is the API for writing tests using JUnit 5	19,574

Tab. 3.1 Top 13 most popular libraries for Java in the Maven Central repository.

Generated by visiting `mvnrepository.com/popular` on Wednesday 27th August, 2025. We excluded libraries targeting a non-Java programming language. The *Clients* column represents the number of consumers of said library aggregated across all of its versions on `mvnrepository.com`.

3.1.1 Application programming interfaces (APIs)

The API serves as the critical boundary between a library and its consumers, defining the contract–both syntactic and semantic–by specifying allowed interactions and providing controlled access to library functionalities [Rob+13]. Library developers author the API within their own code. Their role is crucial, as they both govern the allowed interactions and remain responsible for maintaining them. Correct API design is essential because it shapes how consumers judge the library's usability and usefulness. Furthermore, the API is the main friction point when the library evolves and consumers must adapt [Rob+13; Och+22; Hor+15; CW12]. Despite how vital the API is, defining it is challenging for library developers.

3.1.1.1 API symbols

APIs are composed of API symbols. These API symbols are *methods*, *fields*, or *types*. Language features let developers control which symbols are accessible and which interactions are allowed. For consumers, these symbols and their allowed uses constitute the library's visible interface, allowing internal changes without affecting consumers.

3.1.1.2 Controlling symbol interactions

To minimize bad outcomes, developers should restrict API interactions to those they have explicitly intended and validated. Unintended interactions can cause buggy behavior, frustrate consumers, and lead them to switch to alternative libraries. Many interactions exist and are implicitly allowed by default. Due to the diversity and interplay of mechanisms, combined with limited tools for modeling permitted uses, unintended interactions can exist within the developer API.

In Java, keywords such as `public`, `private`, or `protected`, or the use of the JPMS (Java Platform Module System) (introduced on Thursday 21st September, 2017) control the visibility and determine whether a symbol is part of the API. However, library developers may want to prevent specific interactions and not all of them, something that visibility modifiers are unable to help with. Java features other mechanisms to restrict the kinds of interactions that consumers can perform for a given API symbol. For example, methods by default can be *overridden*, *overloaded*, or *invoked* in Java. The use of the `final` keyword enables developers to restrict these interactions on a given symbol. Specifically, overriding methods, subclassing classes, and reassigning fields become impossible by marking a symbol as `final` for both the client and the library developers. Other mechanisms exist, such as *subclass restrictions and sealing*, *overloading and overriding*, and *polymorphism*. Constructs such as *interfaces*, *abstract classes*, *default methods*, and *generics* further impact the way symbols are used. Such keywords/mechanisms prevent meaningless or potentially harmful interactions that could compromise the library's intended exported API. For example, overriding a method that assumes internal invariants could break the library's state, in a way that the developer of the library has not accounted for.

In some cases, these mechanisms may be lacking or difficult for developers to use. Some communities have introduced best practices that recommend minimizing accessibility or forbidding subclassing by default [BBlo08] as a consequence. While these restrictions help prevent unintended interactions, they complicate development, may block legitimate uses, and provide only a partial solution, further complicating library evolution. The difficulty of capturing the semantic contract of API symbols, which is critical for safe evolution, further increases the lack of support for listing allowed interactions and thus limits the developer's ability to keep interactions to those explicitly intended and validated. Design by contract methods for API design exist that reduce the difficulty of capturing the semantic contract, like Google Cofoja [@Lê]. Some developers instead opt to document their intended use cases as part of their documentation or provide samples to account for the lack of mechanisms while still allowing consumers to use interactions they do not account for. Other techniques to hint consumers besides documentation also exist in the form of annotations (*e.g.*, `@Deprecated`, `@Internal`). Still, consumers can, if they want to, bypass the restrictions or documentation often and perform unintended uses. Runtime mechanisms, such as reflection and dynamic proxies, are examples of bypass methods that can help lift restrictions.

3.1.2 Protocol for interactions

The notion of protocol for the interactions a consumer has with a library is a precise order and set of operations to achieve a specific outcome. For example, a library developer may design their API to write to a file on disk. This action should be performed according to a strict set of rules, starting with an open call and concluding with a flush and close call, all of which utilize a common context parameter. These rules define the protocol for interactions that the library developer intends consumers to use to edit a file on disk. Failure to follow these rules could result in unintended behavior, preventing the file from being edited successfully [KBM14]. The protocol for interaction, therefore, represents the pattern a program exhibits when making use of symbols part of the library API. The type, order, and passed-in arguments (in and out, including returned values) of those symbols used are all considered and form the protocol exercised by the consumer of the API. The considered interactions can be *method invocations*, *class instantiations*, *field reads*, *field writes*, *etc.*, and are contextualized by their parameters. The notion of the "correct" protocol to use can be hinted at by the developers of the library in question as part of their documentation. Still, different protocols can be performed by consumers due to a lack of mechanisms to enforce a correct flow of instructions in Java, contradicting the library developers' intentions. We showcase two interaction flows of two simple Java programs in Figure 3.1.

```
Program 1
├── int open(String path)
├── write(int fd, String data)
└── close(int fd)
```

(a) Example of a program A interaction flow

```
Program 2
├── int fd = open("/tmp/file") ..................................(precedes)
├── ...
├── write(int fd, String data)
├── close(int fd)
├── int open(String path)
├── ...
└── close(fd) ..................................................(succeeds)
```

(b) Example of a program B interaction flow

Fig. 3.1 Example of two different programs' interaction flows.

Let's consider the first program of the above example in Figure 3.1a: These method calls would open a file on the system, write to it, and close it. The above example, therefore, represents a consumer who wants to write to a specific file.

However, let's now consider the second program of the above example in Figure 3.1b: Here, the program has already opened the file before the three invocations being considered. The program writes to the file, closes it, and then reopens it. The use of the `write()` method would not make sense without having the file already open. The same applies to the use of the `close()` method. These methods should take as a parameter a context variable hinting at which file should be edited.

These two scenarios involve different programs performing distinct actions on the user's system. One opens a file to write to it and then closes it. The other takes an existing open file, writes to it, closes it, and then reopens it. While both interactions can be valid, the library developer's intentions may not take into consideration what the second program is doing. The notion of protocol is therefore essential when focusing on library interactions as it encapsulates the exact usage and expectations a consumer has against the library. Said use and expectation can be different from what the library developers have intended and can be a source of friction in the future as the library evolves. This notion is further complemented by the presence of side effects and IoC within these protocols.

3.1.2.1 Side effects

When considering interaction protocols, specific calls on symbols may have consequences on the results of subsequent calls in an API. The variables being used (or reused) also influence the overall state of the program, resulting in different outcomes when the calls are performed with different input. Indeed, when a consumer calls an API symbol, nothing prevents the library's own code from using symbols or fields not exported as part of their API for their consumers. Triggering interactions on internal elements can cause runtime side effects.

A side effect occurs when invoking an exported symbol modifies the program's internal state, thereby influencing the behavior of subsequent invocations. Such behavioral changes may not be immediately apparent to the consumer–either directly, during the execution of the symbol that introduces the side effect, or indirectly, through the behavior of later symbol invocations. This internal state modification is generally harmless unless it alters the program's control flow or produces output variations substantial enough to affect the consumer's code behavior. Even if the consumer's current sequence of interactions remains unaffected, future modifications–such as invoking additional exported symbols, introducing new interactions, or changing the order of operations–may reveal the underlying side effect. An example of such a side effect is shown in Figure 3.2.

```java
// Client making use of the methods offered by the library
public class ClientMain {
    public static void main(String[] args) {
        System.out.println(Lib.getCounter()); // Outputs 0

        // This makes all future getCounter calls not return 0 anymore
        Lib.incrementCounter();

        System.out.println(Lib.getCounter()); // Outputs 1
    }
}
```

Snp. (3.1) Consumer calling into API methods of the library, the first call creates a side effect observed in the last call.

```java
// Library offered methods
public class Lib {
    private static Integer counter = 0;

    public static Integer getCounter() { return counter; }

    // Causes a side effect on getCounter by altering the state of the counter variable
    public static void incrementCounter() { counter++; }
}
```

Snp. (3.2) Library method definitions, exposing a side effect as a result of calling `incrementCounter`, visible afterwards by calling `getCounter`.

Fig. 3.2 Example of a side effect created as a result of a consumer interacting with a library's API.

3.1.2.2 Inversion of Control (IoC)

IoC (Inversion of Control) is a design pattern where the flow of interactions is controlled by the library targeting the consumer and not the other way around, as it is classically done (hence the term "inversion"). In Java, IoC can be achieved via multiple language mechanisms such as lambda/callback usage, for example. To detect all interactions between a consumer and a library (and their order), simply listing all methods invoked from the consumer (by means of static analysis) is not enough. To obtain the list of all interactions, developers need to understand the flow of interactions between the consumer and the library, in the precise order they come, with the exact variables (and same instances) in use as parameters, and capture it at runtime. Reproducing these interactions requires accurately reproducing passed-in lambdas or interfaces for the inverted calls to occur. Calls can go from the consumer to the library (which is natural), and can come to the consumer via code not exported via an API symbol directly due to callbacks in the library. This multi-directional aspect makes IoC challenging to analyze, but it is essential to consider when focusing on interaction protocols.

3.1.3 Library packaging and distribution

Library developers can package the source code to facilitate the distribution and use of their library. Traditionally, making use of a library would involve telling the JVM (Java Virtual Machine) where to find the classes for loading them at runtime. Using a package makes the process simpler, as it involves passing the path to a single file that encapsulates all the library code.

The library packages already feature uniquely assigned package names, directly following the folder structure of the source code. For example, a class named `Foo` in a package named `org.world`, will be physically present on disk in a path named `/org/world/Foo.java`, this forces proper organization of the code and enables consumers to consume dependencies. Compiled Java source code (as `.class`) follows the same principle; compiling `Foo.java` will result in an output compiled binary in `/org/world/Foo.class`.

The compiled code can be further packaged into an archive with the extension JAR. These files are Zip archives containing pre-compiled classes as well as JVM manifest files. These files can be consumed by applications using a command-line argument passed to the JVM to resolve the classes requested by other pieces of code. These JAR files contain no version information, no dependency information, nor any information that would help developers determine if a new version is available, how to update it, or where to obtain it.

3.1.3.1 Versioning

Libraries can be versioned by their developers to tag a specific project milestone for release. Versioning dependencies helps consumers know the freshness of the dependency they are using and helps them make decisions regarding which revision of the library they want to use. Versions usually are composed of numbers (sometimes, multiple sets of numbers separated by

a dot), and in rare occurrences (such as pre-release, in-development versions), a string. Here are a few version numbers to illustrate: `1`, `2`, `2.0`, `2.0.1`, `2.0.1-rc1` Let's consider the simplest form of versions for a specific library symbolizing 3 different milestones:

Version	Release Date
1	Wednesday 1st January, 2025
2	Saturday 1st March, 2025
3	Tuesday 1st July, 2025

Tab. 3.2 List of consecutive versions of a fictional software library with their respective release dates.

Versions are intended to be comparable because a specific order is set for the version numbers. The order also typically follows the release dates chronologically, making it easy to understand for consumers if one version of the library is newer than another.

3.1.3.2 Package management

Developers want to make their library available to other developers for consumption in their projects. Developers also want to version their dependencies and make such information available to consumers when fetching or updating them. To facilitate discovery and updates, developers utilize package repositories and package managers. These package repositories make use of the versioning concept and the JAR packaging to offer libraries to consumers.

The task of discovering packages has been delegated to build tools such as Gradle or Apache Maven. These build tools also utilize package repositories (such as the Maven Central one) for locating and downloading dependencies. They fetch the dependencies a consumer needs (and declares) and cache them on disk for execution at runtime after successful compilation. The task of linking the dependencies is therefore delegated to the build tools and not the developer consuming the dependencies. Build tools help developers curate a list of dependencies their project depends on in their respective project configuration files (`pom.xml` for Apache Maven, `build.gradle` for Gradle). They also help them abstract away all the complexity behind linking dependencies for execution, downloading, and maintaining dependencies available in a specific location.

Apache Maven (or simply Maven) [@Apae] is one of many build tools available for Java projects. It enables a more straightforward build process (by abstracting Java compiler usage and details about the compilation by default, without barring developers from going in depth if they want to), enables a unified build system (via POM and plugins), and encourages better development practices for Java projects (by providing a better separated project structure, having separate environments for tests. Maven features repositories, such as Maven Central, which host third-party dependencies that can be discovered and consumed by clients by declaring the corresponding dependency package coordinates in their project files. Repositories like Maven Central notably enable the distribution of JAR files, with different variants offered (compiled code, source code, tests, *etc.*).

Package coordinates (Group, artifacts, version (GAV)) Maven makes use of the package coordinates (GAV (Group, Artifact, Version)). The package coordinates enable developers to provide an identity to their dependency. Maven uses those notably to enable fetching and discovering dependencies a project needs. The package coordinates are illustrated in Figure 3.3.

Maven in some places, however, may use a slightly different package coordinate scheme, adapted from the generally adopted one. We present such a variant in Figure 3.4.

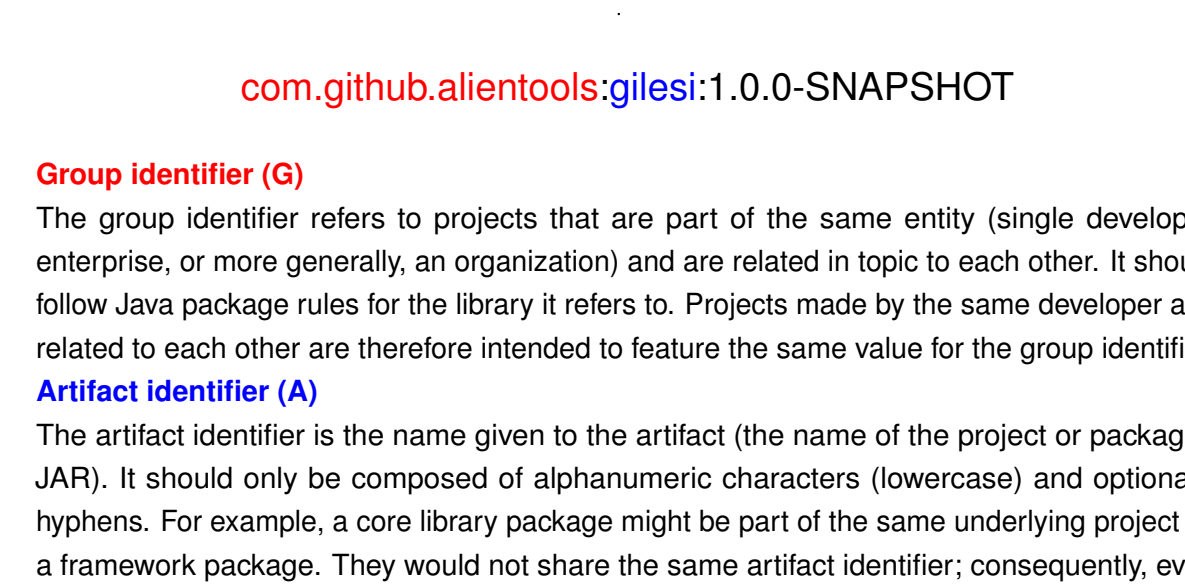


Fig. 3.3 Package coordinate (GAV (Group, Artifact, Version)) scheme, as used generally across the Java library ecosystem.

`maven.apache.org/guides/mini/guide-naming-conventions.html`

Fig. 3.4 Package coordinate scheme, as used specifically in Maven Central.

`maven.apache.org/guides/mini/guide-naming-conventions.html`

Package object model (POM) The POM (Package Object Model) is the Maven representation of a Java project. It includes fields such as the project identity, the version of Java the project

is compatible with, its dependencies, all build scopes, plugin definitions, and more. Projects typically configure this model using a XML file named `pom.xml`, which is the XML serialization of the POM. Developers can thus easily configure their projects with just a text editor.

An example `pom.xml` file is shown in Snp. 3.3. In this example, the project identity is `com.github.gilesi.samples:SNAPSHOT` (`<groupId>com.github.gilesi.samples</groupId>`, `<artifactId>samplelibrary</artifactId>` and `<version>1.0.0-SNAPSHOT</version>`), the project is meant to be packaged as a JAR file (`<packaging>jar</packaging>`). We also see a description (`<description>Sample project</description>`) and a name (`<name>Sample</name>`) set, which would be shown in select package repositories for Maven, such as Maven Central, to help developers discover and understand what the library is about.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>samplelibrary</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <groupId>com.github.gilesi.samples</groupId>

    <name>Sample</name>
    <description>Sample project</description>

    <packaging>jar</packaging>
</project>
```

Snp. 3.3 Example of a simple POM (Package Object Model) file.

3.2 Library clients

The term we use to refer to the consumers of a library is client. While libraries can depend on other libraries, we will also consider them to be clients in that context. Clients making use of a library make the library a dependency of their own software. Likewise, libraries depending on other libraries make those dependencies transitive dependencies for the client (as opposed to direct dependencies, the ones the client declares on its own). The client has no possibility of directly editing or controlling library code, but can make use of the library to the extent the library developer has allowed it to do so in its own code, as explained before. However, clients are usually the last actor in the chain leading up to a finished software application. These clients have huge responsibilities towards their users for the stability, integrity, and functionality of the software they offer. Dependencies, while convenient for developers, expose the clients to greater risks, originating from code written by third-party developers, over which they have no control.

Transitive dependencies also matter just like direct dependencies, due to the tendency of issues present in them to affect the client and the direct dependencies like a ripple effect.

Several vulnerabilities have surfaced over the years, affecting numerous clients. `SolarWind`, `xz`, and the `Log4Shell` vulnerability are all examples of vulnerabilities that exist due to a misbehaving dependency in client software. Furthermore, BCs (Breaking Changes) (*cf.*, Section 3.3.1) occurring in libraries can impact client's correct functionality in a non-trivial manner to distinguish. All of these issues push client developers to maintain their dependencies up to date. Still, fear and mistrust of the author of those libraries exist, making the task harder for client developers about what to do with their dependency updates. Stuck between updating or pinning dependencies, client developers have a big responsibility that they are struggling to solve today due to a lack of adequate tools to detect such misbehavior in their dependencies, or non-trivial updates due to BCs.

On the other side, library developers lack insight into the use of their library, what documentation is lacking, and what hot spots or cold spots [TX08] may exist in their API. This lack of insight is mainly due to the developers of libraries being distinct from the ones developing the client. Knowing the uses done by clients is further complicated by the lack of adequate tooling for modeling the uses present in the library' API. Indeed, as we discussed earlier, accidental API element inclusion, as well as accidental interactions, is a thing that can easily occur in Java. While some remediation exists, restricting the clients too much may not be desired and may make development more complicated for both parties. Furthermore, if client developers want new functionality or want to fix some issue. The presence of potential issues in the library can also make the developers of the library create issues in the client code. And because library developers are unaware, they can be reluctant to update a specific portion of their code. Significant client usage of specific portions of the API puts considerable pressure on the developers' hands to avoid accidentally breaking clients, at the risk of losing their users to other libraries.

3.2.1 Consumption of packaged libraries

3.2.1.1 Dependency declaration

Maven allows developers to curate the list of dependencies they depend on right into the `pom.xml` file.

Developers declare the `<dependencies />` XML element, containing a list of `<dependency />` elements. `<dependency />` elements contain the GAV (Group, Artifact, Version) coordinate of the dependency, as well as the scope the dependency applies to. Scopes in Maven enable developers to define which build step the dependencies apply to. For example, some dependencies may only be needed during test execution (such as the case of test frameworks like JUnit). In this case, developers will declare a dependency with a scope value of *test* to ensure the dependency is only included when packaging the unit tests of the project, not when packaging the main application.

We show an example of dependencies declaration in Snp. 3.4. In this example, the project declares a dependency on the `junit:junit:4.13.2` package, the `org.junit.jupiter:junit-jupiter-engine:5.11.0-M2` package,
and the `com.github.gilesi.samples:samplelibrary:1.0.0-SNAPSHOT` package. The JUnit packages are only used during test compilation and are not included as a dependency for the main project package.

Library developers can also leverage dependency declaration to enable an ecosystem of addons to extend the functionality of a given library by providing a core library, that other libraries make use of. For example, with the `SLF4J` framework, clients can implement new support for existing logging dependencies, like done with the `Logback Classic Module` library.

```
...

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.github.gilesi.samples</groupId>
      <artifactId>samplelibrary</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.11.0-M2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

...
```

Snp 3.4 Example of dependencies declaration in the POM (Package Object Model) file.

3.2.1.2 Transitive dependencies

Libraries can define, like clients do, a list of dependencies on other libraries. Libraries therefore turn into "clients " for the dependencies they define. In this case, these dependencies of a library are called transitive dependencies for any client of the main library. Likewise, the dependencies declared by a client are called direct dependencies. Transitive dependencies create a large set of dependencies that a client can potentially be dependent on, without knowing beyond the ones it explicitly declares, often resulting in big dependency trees.

We show the dependency tree of a popular Java application in Figures 3.5 to 3.7. Notice how the list of dependencies, as directly defined by the client, consists of 8 dependencies, but extends rapidly to more dependencies due to transitive dependencies declared by `Spark Core` and `SLF4J nop`. This growth can lead to an increase in the vulnerable surface that the client is subject to.

```
project :client.diff
├── org.atteo.classindex:classindex
├── project :core
├── project :client
├── org.slf4j:slf4j-nop
├── it.unimi.dsi:fastutil
├── com.fifesoft:rsyntaxtextarea
├── com.j2html:j2html
└── com.sparkjava:spark-core
```

Fig. 3.5 GumTree diff client's direct dependencies

```
org.slf4j:slf4j-nop
└── org.slf4j:slf4j-api
```

Fig. 3.6 GumTree diff client's indirect dependencies due to a direct dependency on `org.slf4j:slf4j-nop`

```
com.sparkjava:spark-core
├── org.slf4j:slf4j-api
├── org.eclipse.jetty:jetty-server
│   ├── javax.servlet:javax.servlet-api
│   ├── org.eclipse.jetty:jetty-http
│   │   ├── org.eclipse.jetty:jetty-util
│   │   └── org.eclipse.jetty:jetty-io
│   │       └── org.eclipse.jetty:jetty-util
│   └── org.eclipse.jetty:jetty-io
├── org.eclipse.jetty:jetty-webapp
│   ├── org.eclipse.jetty:jetty-xml
│   │   └── org.eclipse.jetty:jetty-util
│   └── org.eclipse.jetty:jetty-servlet
│       ├── org.eclipse.jetty:jetty-security
│       │   └── org.eclipse.jetty:jetty-server
│       └── org.eclipse.jetty:jetty-util-ajax
│           └── org.eclipse.jetty:jetty-util
├── org.eclipse.jetty.websocket:websocket-server
│   ├── org.eclipse.jetty.websocket:websocket-common
│   │   ├── org.eclipse.jetty.websocket:websocket-api
│   │   ├── org.eclipse.jetty:jetty-util
│   │   └── org.eclipse.jetty:jetty-io
│   ├── org.eclipse.jetty.websocket:websocket-client
│   │   ├── org.eclipse.jetty:jetty-client
│   │   │   ├── org.eclipse.jetty:jetty-http
│   │   │   └── org.eclipse.jetty:jetty-io
│   │   ├── org.eclipse.jetty:jetty-util
│   │   ├── org.eclipse.jetty:jetty-io
│   │   └── org.eclipse.jetty.websocket:websocket-common
│   ├── org.eclipse.jetty.websocket:websocket-servlet
│   │   ├── org.eclipse.jetty.websocket:websocket-api
│   │   └── javax.servlet:javax.servlet-api
│   ├── org.eclipse.jetty:jetty-servlet
│   └── org.eclipse.jetty:jetty-http
└── org.eclipse.jetty.websocket:websocket-servlet
```

Fig. 3.7 GumTree diff client's indirect dependencies due to a direct dependency on `com.sparkjava:spark-core`

3.2.1.3 Scope of dependencies

Maven enables multiple scopes for Java software. Scopes define in which class path a dependency is available for the JRE (Java Runtime Environment) to load at runtime. Developers can declare the availability of their dependencies for any scope they want.

```
src
├── main/java
│   └── com/github/gilesi/samples/sampleclient
│       └── Client.java
├── test/java
│       └── Tests.java
└── pom.xml
```

Fig. 3.8 Project source tree of a sample for the `com.github.gilesi.samples:samplelibrary` library.

In Figure 3.8 is represented the file structure of a sample for the `com.github.gilesi.samples:samplelibrary` fictional library. There are a few scopes, among which are two principal ones, both represented in the above example:

The main scope is the main software code. This portion of code is executed at runtime, packaged, or part of the primary software distribution. The availability of the code part of this scope is restricted to the main application environment class path. In Figure 3.8, the files related to this scope are located within `src/main/java`, which can access the `com.github.gilesi.samples:samplelibrary` library within its own code as defined in Snp. 3.4.

The test scope is the code designed to test the software. Code in the test scope is intended to test the software distribution. The availability of the code part of this scope is restricted to the test environment class path. In Figure 3.8, the files related to this scope are located within `src/test/java`, which can access the main code of the sample, as well as the `JUnit` library.

It is important to note that due to the dependency declaration shown in Snp. 3.4, the main scope is unable to make use of the `JUnit` library, but because the test scope is dependent on the main scope, the `com.github.gilesi.samples:samplelibrary` library is also accessible directly within the `Tests.java` file. If the sample developer tries to use `JUnit` symbols in code within the main scope, an error would occur during compilation of the project because the dependency declaration specifies `JUnit` only to be a dependency for the test scope.

Furthermore, when the `<scope/>` element is omitted, Maven defaults to the "compile" scope, which automatically adds the dependency to the *runtime*, *test*, and *compilation* class paths.

In Snp. 3.6, we can see the code part of the main scope of the sample. This code makes use of the `com.github.gilesi.samples:samplelibrary` dependency directly in its code. Notably, the `Lib` class, `staticIntMethod`, and `staticDefaultIntMethod` methods of the `Lib` class are being used to implement specific functionality. This is possible thanks to the declaration of the dependency on `com.github.gilesi.samples:sampl` within the POM file as seen in Snp. 3.4.

```
1  import com.github.gilesi.samples.sampleclient.Client;
2  import org.junit.jupiter.api.Test;
3  ...
4
5  public class Tests {
6  |   @Test
7      public void testUseStaticIntMethod() {
8  |       Client.useStaticIntMethod();
9      }
10
11 |   @Test
12     public void testUseStaticDefaultIntMethod() {
13 |       Client.useStaticDefaultIntMethod();
14     }
15     ...
16 }
```

Snp. 3.5 Test code of the `com.github.gilesi.samples:samplelibrary` library sample. The test code makes use of the `com.github.gilesi.samples.sampleclient.Client`, `useStaticIntMethod`, and `useStaticDefaultIntMethod` exported symbols present within the `com.github.gilesi.samples:sampleclient`'s API, part of the main scope of the sample; and of the `org.junit.jupiter.api.Test` exported symbol present within the `JUnit`'s API.

```
1  package com.github.gilesi.samples.sampleclient;
2  import com.github.gilesi.samples.samplelibrary.Lib;
3
4  public class Client {
5      public static void useStaticIntMethod() {
6  |       Lib.staticIntMethod();
7      }
8
9      public static void useStaticDefaultIntMethod() {
10 |       Lib.staticDefaultIntMethod();
11     }
12     ...
13 }
```

Snp. 3.6 Sample for the `com.github.gilesi.samples:samplelibrary` library. The sample code makes use of the `com.github.gilesi.samples.samplelibrary.Lib`, `staticIntMethod`, and `staticDefaultIntMethod` exported symbols present within the `com.github.gilesi.samples:samplelibrary`'s API.

In Snp. 3.5 we instead see the code part of the test scope of the sample. This code uses the main scope code as a dependency, along with the `JUnit` dependency to implement unit tests. Notably, for the main scope dependency, the `Client`, `useStaticIntMethod`, and `useStaticDefaultIntMethod` exported symbols are being used. For the `JUnit` dependency, the `org.junit.jupiter.api.Test` exported symbol is being used as an annotation. The use of `JUnit` is possible here thanks to the declaration of the dependency for the test scope within the POM file as seen in Snp. 3.4. Due to the nature of the test scopes in Maven, the main scope is automatically made a dependency without needing a specific declaration.

3.2.1.4 Variety in dependency forms for consumption

Multiple forms of the software code can be consumed. The source code or the compiled Java code (into ByteCode) can be used independently. For example, IDEs (Integrated Development Environments) can fetch and consume the distributed source code of a library to provide client developers with comments, or documentation (as part of JavaDoc) directly into the developer text editor. The pre-compiled binary distribution is mainly used as a dependency when executing the client at runtime, but can also be used for syntax highlighting (via ByteCode analyzers) when the source code is not available, or it can be used for merging the library code directly into a single package (using JAR shading).

3.2.2 Variety in client uses

Library developers craft their API with specific usage protocols in mind. However, clients do not necessarily have to follow said intended protocol (willingly or due to a lack of documentation). Wright claims that, unfortunately, at a specific point clients will depend on all available behavior of the library, including its bugs. For library developers, however, the API surface should be entirely covered by all clients, thus maximizing the uses that can be performed on a library and ensuring a large variety in interactions, extending far beyond what the library's unit tests may cover [@Wri17]. At the same time, Wright further claims that a small portion of the API of the library is actually used by the clients. Harrand et al. demonstrate that both seemingly contradictory claims are valid on a dataset of the 94 most popular Java library of the time in Maven Central dependencies [Har+22].

This variety in client usage poses a problem for developers because it is impossible to accurately model all interactions performed by every client and test against those scenarios. The variety in interaction, in turn, can create divergence and friction between what the library developers actively test for and what the client developers do, leading to issues. Current studies also demonstrate that API coverage is well achieved by clients today [Har+22]. We showcase an example of a real-world divergence found in actual projects in Snps. 3.7 and 3.8.

```java
package org.jsoup.internal;

...

|  /**
|   A minimal String utility class.
|   Designed for <b>internal</b> jsoup use only
|   - the API and outcome may change without notice.
|   */
public final class StringUtil {
    ...
}
```

Snp. 3.7 `JSoup` developers' intents for the exported `org.jsoup.internal.StringUtils` API portion, used within `JSoup`'s code samples.

```java
package org.jsoup.examples;
...
import org.jsoup.internal.StringUtil;
...

public class HtmlToPlainText {
    ...

    // the formatting rules, implemented in a breadth-first DOM traverse
    private static class FormattingVisitor implements NodeVisitor {
        ...
        // hit when all of the node's children (if any) have been visited
        @Override
        public void tail(Node node, int depth) {
            String name = node.nodeName();
            if (StringUtil.in(name, "br", "dd", "dt", "p", "h1", "h2", "h3", "h4", "h5"))
                append("\n");
            else if (name.equals("a"))
                append(String.format(" <%s>", node.absUrl("href")));
        } ...
    } ...
}
```

Snp. 3.8 Example of an unexpected use of the `JSoup` library, misaligned with the developers' intents, present within `JSoup`'s code samples.

While clients cover most of a library API coverage, Harrand et al. show that for the migration of `gson` to the `jackson-databind` library, for 90% of all clients, instead of performing migration rules for 162 types, only 20 are needed [Har+22]. This shows that the coverage of clients as a whole spans across the entire API of the library, but the majority of them only depend on a core subset of all available symbols. Harrand et al. compute the usage percentage for a large dataset for a % of all clients and shows similar findings on other libraries. They note that for 50% of all libraries present in their study, more than 71.8% of the exported symbols are used by no client. Despite this, they also demonstrate that as soon as a type is exported in the API, a client is bound to make use of it [Har+22].

3.3 Client–Library Co-evolution

In ecosystems of clients and libraries, libraries evolve independently from clients. However, the independent evolution of the libraries leads to changes in clients because clients follow the new versions of libraries. The need to follow the evolution of their library dependencies is motivated by the benefits brought by the latest features or fixes offered by the newer versions of libraries. However, the changes made in libraries complicate the task of updating the clients due to frictions that occur in the client code. These changes lead to delays in dependency updates for client developers. They can also force client developers to pin their dependencies on a specific older version until they find a solution for using the latest library version. In general, this dynamic of clients updating together following changes in libraries is called the client–library co-evolution. We showcase an example of a client being subject to co-evolution in Figure 3.9. The client follows the first two versions of the library directly without delays, but skips library version 3.0, due to some incompatibility that needed more time to address, making them migrate with version 3.1 of the dependency later, with enough time to workaround their issues. In general, the client–library co-evolution dynamic is motivated by the library evolving, which induces evolution in the client afterwards. It is a dynamic that is not bi-directional, even if clients can, in practice, ask library developers for changes or influence changes in the library code due to their own usage. The cost for the evolution is thus more in the client developers' hands than the library developers' hands, but is induced by library developers' own evolution decisions. The work clients have to perform to keep their libraries up to date can vary from a simple version bump when the new version is retro-compatible, to more cumbersome changes to adapt to new code changes the library made. For example, a library developer may decide to remove a symbol from their API. A client making use of this symbol would have to find alternatives on their own following the update, or else their software would not compile and run successfully with the newer version. Sometimes, the clients need to reevaluate their use of a dependency completely or move to a competitor if the changes become too drastic. The work the libraries perform is not without consequences, and it propagates a series of issues down to their clients. Sometimes, issues in library code (like bugs or newly thrown exceptions) may even reflect down to the client and impact its proper functionality.

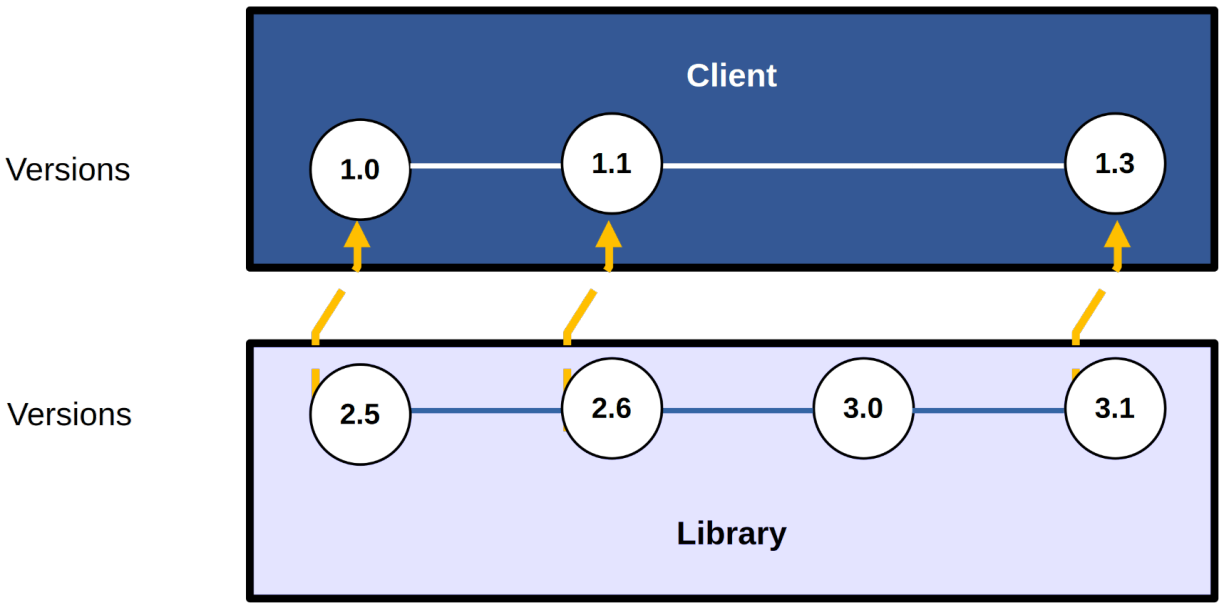


Fig. 3.9 A client and a library co-evolving together with each update of the library or the client.

Each yellow arrow symbol (←) represents the client version (destination) adopting a new version of a library (origin). After version `3.0` of the library, the client did not directly update due to BCs (Breaking Changes) adding more cost on client developers.

It is therefore essential for library developers to account for the issues their clients may face. Some developers have attempted to mitigate compatibility issues with automatic dependency updates by leveraging their unit tests to verify that the dependency update is compatible; however, this approach may not always be effective. In general, clients remain blind to the consequences of dependency updates on their own software, and library developers remain blind to the consequences their updates bring to their clients. Transitive dependencies are also affected; however, this time, clients have even less input on the remediation and must trust their direct dependencies to perform such work correctly.

The dynamic brought by co-evolution and its pace sometimes leads to BCs (Breaking Changes). While some BCs are identified and handled by client developers, not all BCs are detected before merging the dependency update declaration changes. Discovering, debugging, or understanding the BCs can be costly, bringing in friction for both client and library developers. In the next section, we explore BCs, including the types that exist and their impact on software developers.

3.3.1 Breaking changes (BCs)

Updating dependencies can lead to new, unexpected behavior, which may manifest at compile time, link time, or runtime. This type of incompatible change is known as a BC (Breaking Change). A dependency update may contain more than one, but the client will only be affected by the ones touching functionality they use. BCs are classified into two categories: SyBCs (Syntactic Breaking Changes), and BeBCs (Behavioral Breaking Changes) (also called SeBCs (Semantic Breaking Changes)).

3.3.1.1 Syntactic breaking changes (SyBCs)

SyBCs (Syntactic Breaking Changes) can be detected using static analysis techniques as shown in [Lat+25; Bri+18a; Foo+18]. Significant work was also made on prevalence [Och+22; Jay+25; Bri+20] and on remediation [Gao+21; Nav+23; ZM24] of SyBCs. Two distinct categories further divide SyBCs: Binary Syntactic Breaking Changes, and Source Syntactic Breaking Changes. SyBCs can be both source and binary incompatible at the same time. For clarity in the definition of both, we present two illustrated cases of SyBCs, one that is only source incompatible, and the other only binary incompatible.

Source syntactic breaking changes Source SyBCs are incompatibilities that manifest themselves upon compilation of the client. They notably affect the API of the library, the source one. These would not happen by loading compiled code, nor during the execution of the code, because the program would fail to compile successfully as a result of this BC. For example, in Figure 3.10, an update changes the method to contain a new `throws` keyword. Because bytecode does not encode the exception information, this cannot be a binary-incompatible change. Indeed, the class would load fine, as the JVM (Java Virtual Machine) will be able to resolve the specified symbol, given that the bytecode is identical. However, it is source incompatible because the Java compiler would forbid making use of a method without either throwing such an exception or surrounding the call with a `try { } catch { } finally { }` block. This type of BC is made possible only when attempting to recompile the client against the newer version of the library.

```
import java.util.Collection;

// Library offered task method
public class LibraryTask {
-    public static Collection<Item> performTask() {
+    public static Collection<Item> performTask() throws Exception {
         ...
    }
}
```

Snp. (3.9) Library diff between the previous version (compatible) and the updated version (breaking).

```
import java.util.Collection;

// Client making use of the task method offered by the library
public class ClientMain {
    public static void main(String[] args) {
        // Compiler error when upgrading due to the exception not being caught
        java.util.Collection<Item> resultItems = LibraryTask.performTask();
        ...
    }
}
```

Snp. (3.10) Client usage of the library breaks upon updating to the newer version.

Fig. 3.10 Example of a source SyBC (Syntactic Breaking Change) that is not binary incompatible.

Binary syntactic breaking changes Binary SyBCs are incompatibilities that manifest when attempting to load the code at runtime. They notably affect the ABI (Application Binary Interface) of the library. Typically, the link between a symbol definition present in the library binary and the symbol used by the client does not match, creating the BC. For example, in Figure 3.11, an update changes the method signature to contain a different return type. This change is source compatible because the new return type is more restrictive than the older one, `List` being a subtype of `Collection`. For clients only specifying the `List` type, or `var`, this change would not create a source-incompatible change. However, it is binary incompatible because the method descriptor, which is composed of the method's return type, differs. The client upon loading the compiled class would therefore throw an exception because it would not be able to find the previous symbol definition, which lacked the added argument. This type of BC is also made possible due to the use of pre-compiled JAR files. Simply replacing the old library version's JAR with the new one and re-running the client, to the point of triggering loading the class, is enough to trigger the BC.

```
- import java.util.Collection;
+ import java.util.List;

// Library offered task method
public class LibraryTask {
-    public static Collection<Item> performTask() {
+    public static List<Item> performTask() {
             return null;
    }
}
```

Snp. (3.11) Library diff between the previous version (compatible) and the updated version (breaking).

```
import java.util.Collection;

// Client making use of the task method offered by the library
public class ClientMain {
    public static void main(String[] args) {
        java.util.Collection<Item> resultItems = LibraryTask.performTask();
        ...
    }
}
```

Snp. (3.12) Client usage of the library breaks upon updating to the newer version.

Fig. 3.11 Example of a binary SyBC (Syntactic Breaking Change) that is not source incompatible.

3.3.1.2 Behavioral breaking changes (BeBCs)

BeBCs (Behavioral Breaking Changes) are changes that occur during the execution of the code. Class loading and compilation are not problematic after updating a client dependency, and the BC manifests itself as a result of a different program behavior. Some approaches attempt to detect BeBCs using static analysis only [Zha+22a], but we cannot be certain that the detection of a BC is accurate due to the undecidability of program equivalence. Indeed, it is possible that the change is equivalent to the older version of the program and does not impact the client in a breaking way. BeBCs can occur, for example, as a result of a method being rewritten in a way that results in different return data with the same entry parameters. They can also occur when undocumented side effects are added and impact the behavior of a method call as a result. BeBCs can also consist of newly thrown exceptions. The variety of root causes can be significant and is influenced by the usage of specific methods by the client (within specific entry and context parameters). Figure 3.12 depicts a BeBC introduced in `common-text` and reported in TEXT-219 [@Apac]. The API method `StringTokenizer#getTokenList` originally returns a list copy of an internal array of tokens. In commit 2d1ab7 [@Apab], released with version `1.10`, a maintainer simplified various parts of the code and used the built-in utility `Arrays#asList` to convert the array into a list. This seemingly innocuous change introduced two adverse effects: the returned list implementation is now fixed-size, and it is backed by the original array, meaning that changing either will alter the other. Thus, in contrast with the previous version, attempting to insert or remove an element in the returned list will raise a runtime exception, and reordering the list may have unintended side effects. This change is both binary and source compatible and therefore not a SyBC because the method descriptor and method signature remain unchanged in both bytecode and source code forms.

```
public class StringTokenizer ... { public List<String> getTokenList() {
-   List<String> list = new ArrayList<>(tokens.length);
-   Collections.addAll(list, tokens);
-   return list;                                          // Returns ArrayList
+   return Arrays.asList(tokens);                         // Returns Arrays$ArrayList
  }}
```

Snp. (3.13) Diff between API `1.9` (compatible) and API `1.10` (breaking) for Figure 3.12.

```
static List<String> fetchProducts(
                    String products) {

  StringTokenizer st =
         new StringTokenizer(products);
  st.setDelimiterChar(',');
  return st.getTokenList(); }
```

Snp. (3.14) Client code for Figure 3.12.

```
@Test void test_fetchProducts() {
  String str = "apple,banana,cherry";
  List<String> fruits =
                  Products.fetchProducts(str);
  assertEquals(3, fruits.size());
  assertThat(fruits.
     contains("apple", "banana", "cherry")); }
```

Snp. (3.15) Client test for Figure 3.12.

Fig. 3.12 An example BeBC (Behavioral Breaking Change) introduced in Apache `Commons Text` and released in version `1.10`.

The regression was introduced in commit `2d1ab7`[1], identified in TEXT-219 and later fixed in commit `f9846b`[2].
Add and *remove* operations will throw `UnsupportedOperationException` upon update. The client test fails to detect this change.

[1] https://github.com/apache/commons-text/commit/2d1ab7ea72298949900df47f65b4f71d56411f0b
[2] https://github.com/apache/commons-text/commit/f9846b10b2365a36f95f63ff9f90e0f8847f901b

3.3.2 Tool support

To remediate issues brought about by BCs, developers are implementing mitigations. We make the distinction between preventive mitigations, applied by library developers to prevent their clients from potentially being subject to BCs, from defensive mitigations, applied by client developers to prevent them from being subject to incoming BCs from their dependencies. This brings two different points of view, one where the trust is in the library developer's hands and another where the clients are taking a zero-trust approach against their dependencies.

3.3.2.1 Mitigation on the client side

Client developers may want to identify BCs with new incoming dependency updates. Even though library developers should communicate clearly to their clients the existence of BCs, this may not always be the case or known by library developers. As a safety measure, clients therefore, look into automatically detecting compatibility issues using multiple techniques.

Automated Dependency Updates Developers use dependency update bots to keep their dependencies up to date. These offer the ability to automatically update dependencies as new versions get released and automatically run the software application test suite before merging the change. Some of these bots include Dependabot, Renovate, or Uppdatera, and often offer deep integration into existing developers' workflows [He+23]. These bots aim to reduce the cost associated with dependency updates but also expose clients to frequent updates of their dependencies and thus, library evolutions propagating into the client software automatically. These bots feature massive adoption over the years, due among other things to the desire from client developers to benefit from bug fixes, security patches, new or improved features, *etc.* [Bog+21].

The use of dependency automatic update bots also increases the vector for behavioral impact on clients. The increase is especially notable when developers do not spend time thoroughly reviewing the library changes, if at all. Library developers can put information into their release notes, but this is generally lacking and focuses more on the new features and fixes specified in their release notes. Release notes are also not necessarily accurate [Wu+22]. Clients trust their test suite results fully. However, this test suite may not be able to catch all potential BeBCs that can occur in used dependencies.

Because client developers want to solve the cost caused by needing to update their dependencies all the time manually [Jai+24], they tend to trust their CI pipelines and thus, their own test suite to ensure the dependency update is compatible. The amplified exposure to BeBCs therefore increases further costs on the clients associated with the potential issues BeBCs brings later after the update.

Clients' CI/CD Each time client developers modify their code, dependency information, or project configuration, there is an increase in automated unit test execution, thanks to CI (Continuous Integration) / CD (Continuous Development). These unit tests are designed to test the client main written code, and usually do not attempt to test dependency code. Libraries may also feature unit tests of their own to test their own functionality. Tests are composed of a series of interactions against the program code, as well as assertions on what expected values versus what's returned/computed.

- A **passing** (or ***Green***) test is a test where the expected values in the assertion(s) match the computed values at runtime.
- A **failing** (or ***Red***) test is a test where the expected values do not match or an exception occurs.

Developers use test suites as a measure of integrity and quality, but they are highly dependent on the code and assertions that developers write in them. A way to evaluate unit test quality is to compute the code coverage metric of the entire test suite. Code coverage refers to the percentage of lines of code that the execution of the test suite can run, compared to the total number of executable lines of code in the program.

CI (Continuous Integration) ensures that each time the client developer updates a dependency, the client executes its unit tests automatically. Code coverage metrics may mislead clients on that front, because they would not account for the total lines of code behind each API symbol invocation.

Tools exist that leverage client test suites to test for dependency compatibility [Dan+20]. These tools leverage both the client test suite and the library API symbol coverage from the client test suite to provide the client with a compatibility report. These tools show that detecting BCs is possible for clients. However, these tools remain limited by the coverage and assertions the client test suite offers.

3.3.2.2 Mitigation on the library side

Library developers may want to prevent BCs with their new releases. On top of making the client experience better, it also enables greater trust from their client in regards to the reliability of the dependency. Developers have used multiple techniques to both alert client in case of intentional BCs, as well as identify and detect unexpected BCs during the development of their new library versions.

Semantic Versioning (SemVer) To alert clients of intentional BCs, libraries leverage their version numbers to convey information to client developers directly. Libraries already feature a version to comply with various package repository requirements. Version numbers are specified as part of the package repository manifest. Their use thus extends beyond just verifying if a package is different or newer than another. Such conveyed information typically reflects if changes were made that were BCs, or minor additions, or bug fixes.

Fig. 3.13 SemVer (Semantic Versioning) `2.0.0` Scheme.

Traditionally, developers have adopted the SemVer (Semantic Versioning) scheme [@Pre] to version their dependencies to communicate the presence of BCs, or bug fixes, or enhancements. We illustrate SemVer in Figure 3.13. For example, for a library using SemVer, an old package with version `2.6.10`, and a newer package with version `3.0.0` indicates that a BC is to be expected when upgrading from the previous versions to the latest version because the major version is higher (2 *vs.*3).

Reverse-Dependency Compatibility Testing (RDCT) To prevent client incompatibilities, library developers can rely on their regression tests to ensure the behavior of their library is unchanged from one version to the next. Thanks to RDCT (Reverse-Dependency Compatibility Testing), library developers can detect BCs impacting their clients by leveraging their client unit tests. Developers gather a few of their clients, upgrade their declaration on their library, and proceed to run their unit tests to ensure compatibility. However, using RDCT in practice requires obtaining a list of clients of a library, which can be hard. On top of which, library developers also need to get the clients projects in a buildable and runnable state, upgrade the dependency declaration, adapt the client code if it is incompatible, and have insight into the client architecture to understand issues if they arise, all of this running on the library developer's own environment. This complexity makes RDCT unsuitable for library developers to target, and is also subject to the same test suite weaknesses that the client developers alone are subject to. Furthermore, this requires library developers also to perform the necessary code changes if they intentionally introduced BCs, something they would mention to their clients via their release notes or versions, and that will have to be replicated by the clients themselves. Additionally, the quality of the assertions the clients make may not be suitable for accurately determining compatibility with the new version due to weaknesses in change coverage. Tools exist that leverage RDCT to provide compatibility reports for library developers today [ODF22]. These tools, however, are limited by the coverage

of the client test suites and the setup complexity that may be required for running the client projects on the library developer environment.

4 Lightweight Syntactic API Usage Analysis

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 chapter, we present a novel conceptual framework designed to assist library maintainers in understanding the interactions allowed by their APIs via the use of *SUMs (Syntactic Usage Models)*. These customizable models enable library maintainers to improve their design ahead of release, reducing friction during evolution. The complementary *SUFs (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: `Commons Cli`, `JSoup`, and `Spark`. Our exploratory case study reveals that UCov provides valuable insights into API design and the fine-grained analysis of client code, enabling the identification of under-tested and under-documented library code.

Chapter Contents

This chapter in the literature

This chapter is initially published in the literature as

Published Conference Paper

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. “Lightweight Syntactic API Usage Analysis with UCov”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC ’24: 32nd IEEE/ACM International Conference on Program Comprehension. Lisbon Portugal: ACM, Apr. 15, 2024, pp. 426–437. DOI: 10.1145/3643916.3644415. URL: https://dl.acm.org/doi/10.1145/3643916.3644415

and features artifacts published as

Published Artifacts

Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. *Artifacts for "Lightweight Syntactic API Usage Analysis with UCov"*. Version 1.0.0. Jan. 26, 2024. DOI: 10.5281/ZENODO.10571867. URL: https://zenodo.org/doi/10.5281/zenodo.10571867

as well as a proof of concept implementation of the concepts presented, available as

Proof of Concept Implementation

[SW] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri, *UCov, Alien-Tools – GitHub*. URL: https://github.com/alien-tools/ucov

4.1 Introduction

APIs govern the interactions between a library and client projects using it. In particular, they specify which symbols 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 [Rob09; Och+22]. 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 IoC, fluent interfaces) and the variety of ways that client code can interact with individual API symbols. 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 understand how their APIs are—or could be fully!—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 [BBlo08]). 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 [QLL16; Har+22].

In this chapter, we introduce a novel conceptual framework aimed at helping library maintainers better understand the boundaries of their APIs via the use of so-called *SUMs (Syntactic Usage Models)*. SUMs enable maintainers to reason about the interactions allowed by their APIs throughout their evolution. Based on the SUMs, we define *SUFs (Syntactic Usage Footprints)*, 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 4.3).

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 4.4). 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 [@JSoa]; Apache `Commons Cli`, a library for implementing command-line interfaces [@Apaa]; and `Spark`, a simple web framework [@Spaa] (Section 4.5). From our experiments with UCov, we discuss applications in other areas (Section 4.6) [SMon+24a].

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.

4.2 State of practice 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 [Sty+09; SB17; TX08]. In a 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 [QLL16]. Various tools and studies adopt distinct methodologies for gathering usage data, whether from bytecode [Har+22], source code and resolved ASTs [QLL16; LPS11; DLP13], or other sources [Sty+09]. 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 does not influence 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 models mentioned above 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 [Zho+09; Wan+13]. 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 extend beyond our objectives and consider some form of semantic relation between API symbols, they are unable to differentiate between the various interactions allowed for a specific symbol. In contrast to previous works, our approach emphasizes the diverse ways a particular symbol may be utilized in client code, leveraging the *SUMs (Syntactic Usage Models)* introduced in the next section.

4.3 Syntactic API Usage

Our analysis of API usage is built upon two distinct models: SUMs (Syntactic Usage Models), which are extracted from library code and represent the *legal* (possible) uses of an API (Section 4.3.1), and SUFs (Syntactic Usage Footprints), which are extracted from client code and represent the *actual* uses of an API (Section 4.3.2). From these two models, we derive various metrics, including a dedicated API coverage metric (Section 4.3.3). 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.

4.3.1 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 $exported : \mathcal{S} \to \{\top, \bot\}$ 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 illustrative example, consider the simplified excerpt of the JDK's `java.util.ArrayList` depicted in Snp. 4.1. 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 first two symbols only.

```
1 public class ArrayList<E> implements List<E> {
2     ...
3     public boolean add(E e) { ... }
4
5     private int size;
6     ...
7 }
```

Snp. 4.1 A simplified excerpt of `java.util.ArrayList`.

Depending on its properties, the same kind of symbol may be used in various ways in client code. Therefore, SUMs (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 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 SUM (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 Snp. 4.1, the SUM holds two exported symbols and five possible uses of the API, as shown in Table 4.1.

4.3.2 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.

SUFs (Syntactic Usage Footprints) materialize the diversity of ways a given API can be used in client code. For instance, Snp. 4.2 depicts a classical interaction with the `ArrayList` API through instantiation and invocations. On the other hand, Snp. 4.3 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 IoC and the Hollywood principle ("*Don't call us, we'll call you*"). As Table 4.2 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.

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

Tab. 4.1 SUM (Syntactic Usage Model) of Snp. 4.1.

```
1 ArrayList<Integer> lst =
2         new ArrayList<Integer>();
3 lst.add(42);
4 lst.add(1337);
```

Snp. (4.2) Classical usage of the API of Snp. 4.1.

```
1 class MyArrayList<E> extends ArrayList<E> {
2     ... @Override
3     public boolean add(E e) { ... }
4 }
```

Snp. (4.3) Framework-like usage of the API of Snp. 4.1.

Client	Statement	Symbol	Use
Snp. 4.2	`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*
Snp. 4.3	`class MyArrayList<E> extends ArrayList<E>`	`public class ArrayList<E>`	*Extended*
	`@Override public boolean add(E e)`	`public boolean add(E e)`	*Overridden*

Tab. 4.2 SUFs (Syntactic Usage Footprints) for the API of Snp. 4.1.

4.3.3 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, Snp. 4.2 covers 50% of Snp. 4.1's API uses and 100% of its symbols, so does Snp. 4.3. 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 [QLL16; SB17] 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 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.

4.4 UCov: Java Syntactic Usage Analysis

UCov is our proof-of-concept implementation of syntactic API usage analysis for Java libraries [SMon+]. A binary compile of UCov alongside all artifacts discussed in this chapter is available on Zenodo [SMon+24a]. It parses and analyzes the source code of Java libraries to produce SUMs (Syntactic Usage Models) and the source code of Java clients to produce SUFs (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 4.3. 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 4.2.

4.4.1 Exported Symbols

SUM models employ a simple $exported : \mathcal{S} \to \{\top, \bot\}$ 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 [Gos+21], 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.

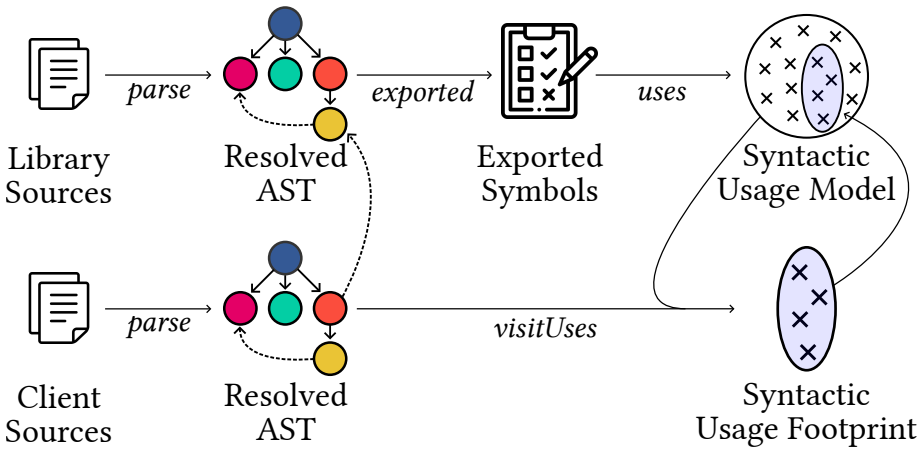


Fig. 4.2 UCov parses and analyses Java code to build SUM (Syntactic Usage Model) and SUF (Syntactic Usage Footprint) models.

Each cross symbol (×) represents one possible use of an API symbol.

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 not `final` nor `sealed`, and that, in the case of classes, possesses a `public` or `protected` constructor that 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.

4.4.2 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 4.3 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.

Symbol Type	Use	Example	Resolved API symbol

Type	*Reference*	`String s`
		`void f(Integer i)`
		`Integer f()`
		`void f()throws IOException`
		`catch (IOException e)`
		`List<String> l`
		etc.
			`java.lang.String`
			`java.lang.Integer`
			`java.lang.Integer`
			`java.io.IOException`
			`java.io.IOException`
			`java.lang.String`
			etc.

Class	*Instantiation* *Inheritance*	`new Integer(42)` `class T extends Thread`	`java.lang.Integer` `java.lang.Thread`

Interface	*Implementation*
	
	Extension
		`class R implements Runnable`
		`Runnable r = ()→{...}`
		
		`interface R extends Runnable`
			`java.lang.Runnable`
			`java.lang.Runnable`
			
			`java.lang.Runnable`

Constructor	*Invocation*	`new Integer(42)`	`java.lang.Integer(int)`

Method	*Invocation*
	Static invocation
	
	Overriding
		`"a".length()`
		`String.valueOf(42)`
		
		`@Override void run()`
		`Runnable r = ()→{...}`
			`java.lang.String.length()`
			`java.lang.String.valueOf(int)`
			
			`java.lang.Thread.run()`
			`java.lang.Runnable.run()`

Field	*Field read* *Field write*	`Integer.MAX_VALUE` `Point.x = 2`	`java.lang.Integer.MAX_VALUE` `java.awt.Point.x`

Table 4.3 The kinds of uses considered in UCov.

Inspired and refined from the work of Qiu et al. [QLL16].

4.4.3 Implementation

UCov is implemented in Java and uses the Spoon framework [Paw+16] 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 4.2). Then, it maps each of these symbols to the corresponding uses ($\times$) according to the rules in Section 4.4.2, forming the final SUM (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 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 SUM. Table 4.3 lists an excerpt of the symbols 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 SUF (Syntactic Usage Footprint).

4.5 Exploratory Case Study

In this section, we evaluate the capability of SUMs (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 [Ral21] to analyze three popular Java libraries that exhibit diverse interaction styles: Apache `Commons Cli`, a library for implementing command-line interfaces; `JSoup`, a library for HTML manipulation; and `Spark`, a simple web framework. We introduce our subject libraries in Section 4.5.1 and our methodology in Section 4.5.2, and then detail our results in Section 4.5.3. We direct the reader to our reproduction package to access the data, scripts, and analyses conducted in this section [SMon+24a].

4.5.1 Subject libraries

The implementation of SUMs (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.

4.5.1.1 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, Snp. 4.4, illustrates a typical classical use of `Commons Cli`, where a parser and options are instantiated and configured using their respective 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 IoC 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, Snp. 4.5, 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 [@Fow05]. The third snippet, Snp. 4.6, demonstrates a typical fluent interaction with `JSoup`, illustrating how the *Builder* pattern is used to configure a `JSoup` connection and load a document from a remote URL.

4.5.1.2 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.

4.5.1.3 Selected libraries

To obtain high-quality libraries that meet these requirements, we start from the Duets dataset [DSB21]. 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 4.4 provides some descriptive statistics for these libraries.

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 [@Apad]. 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 links to the official website and a cookbook that presents various sample snippets [@JSob]. 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 [@Spac]. The documentation points to a list of user-authored tutorials [@Spae], which consist of parseable Java files, and a list of template projects [@Spab; @tip; @per], which we also include.

	Commons Cli	JSoup	Spark
Last release date	Friday 29th October, 2021	Saturday 29th April, 2023	Thursday 8th October, 2020
Version	1.5.0	1.16.1	2.9.3
Since	Wednesday 6th November, 2002	Sunday 31st January, 2010	Thursday 7th February, 2013
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

Tab. 4.4 Descriptive statistics of the subject libraries.

Extracted from GitHub on Monday 30th October, 2023.

```java
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) { ... }
```

Snp. (4.4) Classical usage of `Commons Cli`, retrieved from [@Apad].

```
get("/",    (request, response) → { ... });
post("/",   (request, response) → { ... });
put("/",    (request, response) → { ... });
delete("/", (request, response) → { ... });
options("/", (request, response) → { ... });
...
```

Snp. (4.5) Framework-like usage of `Spark`, retrieved from [@Spad].

```java
Document doc = Jsoup
    .connect("http://example.com")
    .data("query", "Java")
    .userAgent("Mozilla")
    .cookie("auth", "token")
    .timeout(3000)
    .post();
```

Snp. (4.6) Fluent-like usage of `JSoup`, retrieved from [@JSoc].

Fig. 4.3 Diverse API interaction styles exemplified using actual documentation samples from our subject libraries.

4.5.2 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 (202 for `Commons Cli`, 31 for `JSoup`, 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 [@Git]. We compile a list of every repository in the dependency graph that declares a dependency towards one of the libraries of interest. After discarding repositories that cannot be retrieved and forks, we obtain $11,612$ clients for `Commons Cli`, $11,159$ for `JSoup`, 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 4.4, 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 4.5). 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 4.5.

4.5.3 Analysis and Results

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

4.5.3.1 Syntactic usage models and footprints

Table 4.5 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 4.4). 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.*, [Har+22; QLL16]), 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*" [@Wri17], 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 [Har+22]. 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`).

			Commons 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

Tab. 4.5 SUMs (Syntactic Usage Models) extracted from `Commons Cli`, `JSoup`, and `Spark`, and SUFs (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.

4.5.3.2 Library profiles

Figure 4.5a 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 Snp. 4.5. This is consistent with its framework-like interaction style.

4.5.3.3 Usage analysis

Figure 4.5b 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 4.5a and Figure 4.5b 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 (Tuesday 21st September, 2021) and the introduction of sealed classes, Java could not restrict extension and overriding to a predetermined set of types. Without this possibility, library maintainers had to open types for extension and specialization to everyone, even when they intended 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 reflects the expected uses of each library, as documented in their samples.

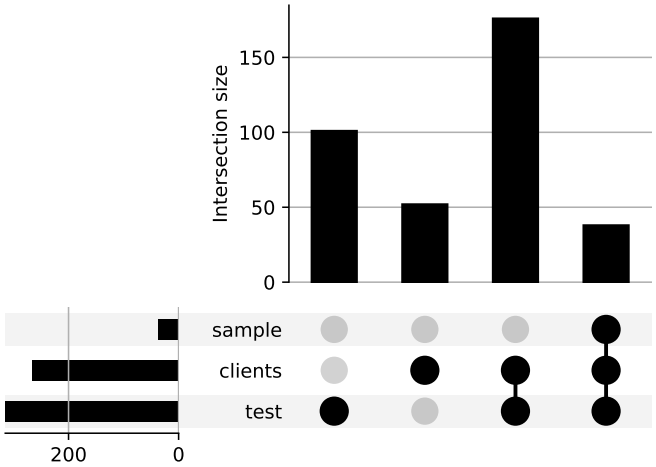
For `Commons Cli` the top three most popular interactions involve referencing and instantiating the symbols `Option` and `Options`, and invoking the method `addOption`, which developers use to build their command-line interface.

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.

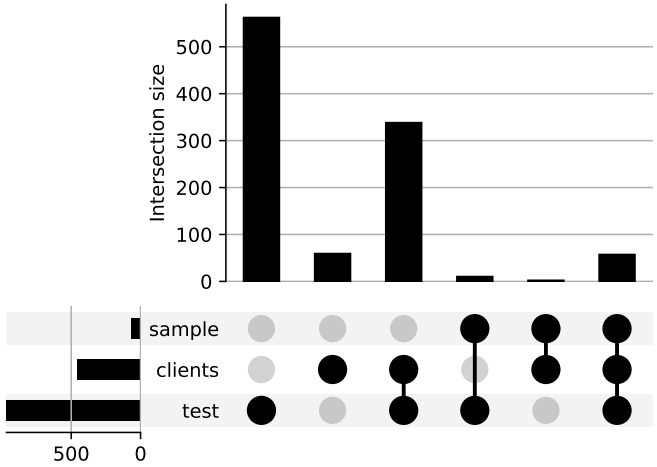
For `Spark` the top three most popular interactions involve implementing the interface `Route` and overriding its method `Route.handle()`, which developers use to declare routes, as well as referencing the types `Request` and `Response`, which are passed to and from the routes.

We refer the reader to the reproduction package for a comprehensive list of the most and least popular API symbols and interactions.

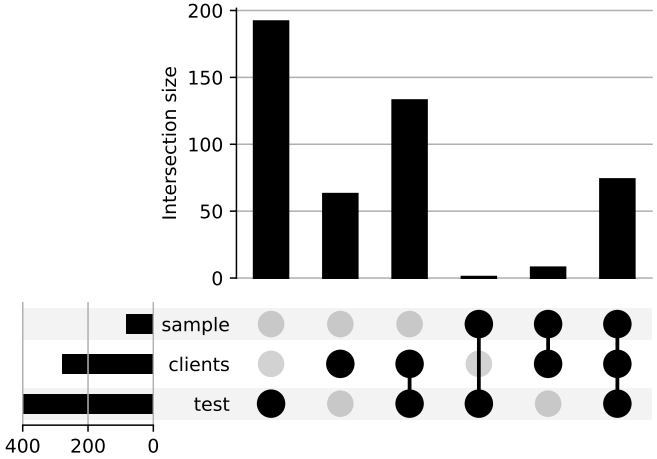
Interestingly, we observe that `JSoup` exposes parts of its *internal* API publicly (`org.jsoup.internal`), most likely for technical reasons: isolating types in a package forces them to be `public` to allow other packages of the library to access them. While the maintainers take great care to discourage 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 [@JSod]. UCov allows library maintainers to identify these problematic cases at a glance.



(a) SUF (Syntactic Usage Footprint) intersections for `Commons Cli`.

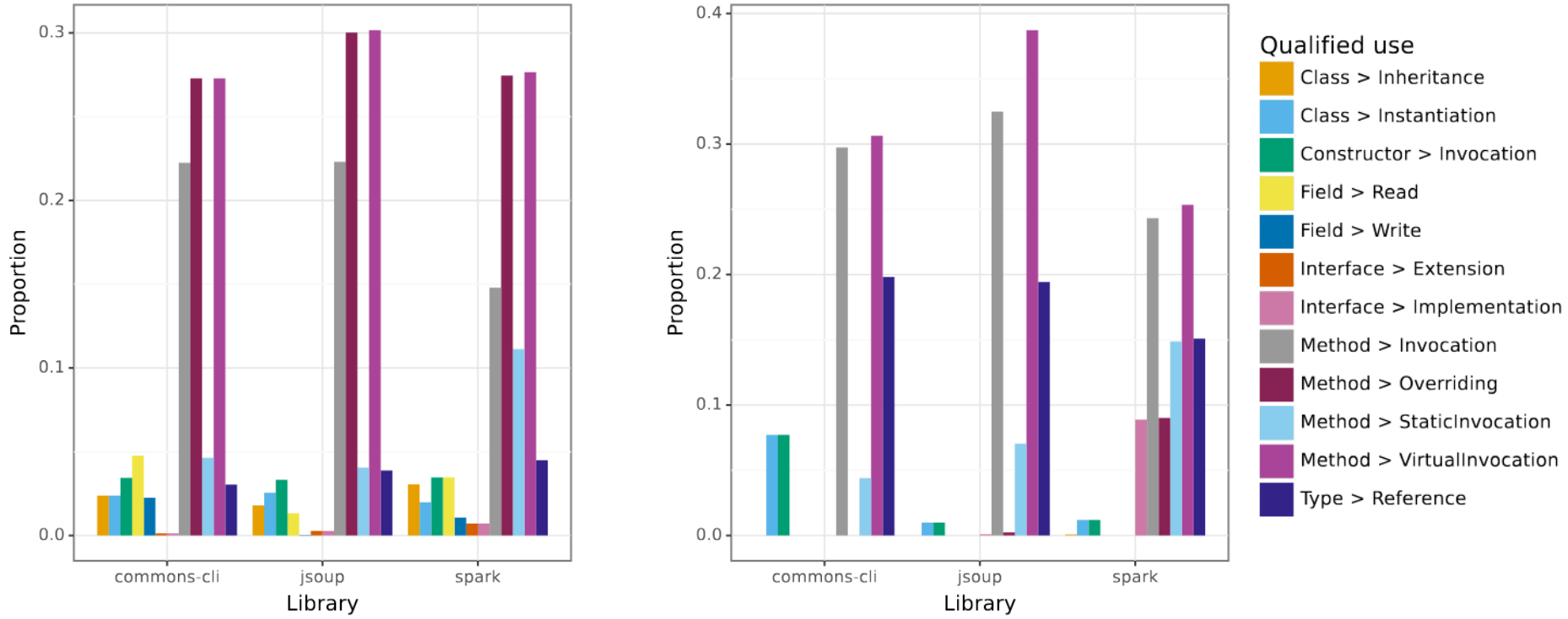


(b) SUF (Syntactic Usage Footprint) intersections for `JSoup`.



(c) SUF (Syntactic Usage Footprint) intersections for `Spark`.

Fig. 4.4 UpSet plots [Lex+14] depicting the common and unique uses between third-party clients, tests, and samples for `Commons Cli`, `JSoup`, and `Spark`.



(a) As a distribution of the legal uses in their APIs (SUM (Syntactic Usage Model)).

(b) As a distribution of the actual uses in client code (SUF (Syntactic Usage Footprint)).

Fig. 4.5 Usage profiles of the three libraries `Commons Cli`, `JSoup`, and `Spark`.

4.5.3.4 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 4.4 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`[@Apad] 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 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 undocumented uses. 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 their 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 can help maintainers identify blind spots in their design.

4.6 Discussion

SUMs (Syntactic Usage Models) and SUFs (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.

4.6.1 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 IoC 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.

4.6.2 Compatibility and breaking changes

When libraries evolve, they sometimes introduce BCs (Breaking Changes) that affect clients and force them to adapt their code. How BCs 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 BCs. Some approaches attempt to measure the impact of potential BCs by analyzing client code. While some only rely on import declarations to determine if client code may be impacted [Xav+17], more recent

approaches gather usage information to improve the accuracy of impact detection [Och+22; ODF22]. 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. It could be recompiled whenever the library is updated, allowing the compiler to automatically check if any of the interactions break, signaling a BC. Using the compiler as ground truth for BC detection would address the accuracy issues of existing detection tools [JD17] and automatically cope with the evolution of programming language specifications and their implementation in compilers and virtual machines.

4.6.3 API evolution

Predicting the consequences of API changes can be challenging. When an API evolves, it can not only introduce BCs (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.

4.7 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 BCs (Breaking Changes) analysis.

4.7.1 API symbols analysis

Traditionally, the analysis of API usage has primarily been conducted at the level of individual symbols. That is, individually exported fields or methods, regardless of which kind of interaction is performed by the client on it.

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 [Sty+09; SB17; TX08].

Stylos et al. propose an approach that integrates "placeholders" into an API, enabling developers to more easily identify how to perform tasks that are not immediately obvious from the available methods [Sty+09]. For instance, when a developer searches for a "send" method that does not exist because it requires instantiating or interacting with another class, the placeholder guides

them toward the correct procedure directly within the API documentation. The authors also investigate adapting the visual prominence of documentation elements based on how frequently each API component is used, to improve navigation efficiency. These placeholders present the most common strategies for accomplishing specific tasks, thereby helping developers understand typical usage patterns. The effectiveness of the approach is examined through a user study measuring how quickly participants can determine how to implement required functionality using the system. They ultimately show the importance of discovering not just symbols of the API but also the different ways they can be correctly used regarding API over- or under-utilization.

Sawant and Bacchelli introduce a dataset-driven analysis of API usage by client projects [SB17]. The authors examine five popular libraries/APIs from GitHub and investigate clients' adoption delays for new releases, referred to as "lag time". They further evaluate the proportion of API features and symbols actually used by each client. Their findings reveal that only a small subset of the available features is regularly employed, and that the majority of commonly used symbols were introduced early in the API's evolution.

Thummalapenta and Xie present a tool designed to extract frequently used code snippets ("hotspots") from a given API or library by mining open-source projects on the web [TX08]. These recurring snippets indicate frequently invoked API elements and reveal common usage patterns. The extracted hotspots serve both as practical code examples illustrating how the API is used in real-world contexts and as a means to compensate for missing or incomplete documentation. This is particularly beneficial for complex libraries whose correct usage may not be immediately evident. The tool also provides recommendations for relevant code examples associated with specific API elements, effectively exposing underlying usage protocols. Overall, the hotspots highlight which API components developers are most likely to rely on, based on widespread usage across numerous projects.

A 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 [QLL16]. The authors focus on how APIs are actually used by clients and the extent to which these usages align with the intentions of the API designers. They measure the degree to which the core API is exercised–finding that it is not fully utilized–identify continued use of deprecated APIs in real-world projects, and present both hotspots and coldspots in API usage. They also consider the variety of ways in which a symbol may be invoked.

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 [Har+22]. They also show paradoxically that sufficiently large client populations tend to rely on the full breadth of an API. Their analysis focuses primarily on type-level interactions and does not incorporate the full spectrum of ways in which a symbol can be used.

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 [Har+22], source code and resolved ASTs [QLL16; LPS11; DLP13], or other sources [Sty+09].

Lämmel et al. leverage AST analysis to extract the usage footprint of a given API and compare it to the full set of available elements [LPS11]. The focus is limited to type usage, and the results are used to motivate potential API migration scenarios. The approach identifies usage by inspecting import references to API types and counting their occurrences. The authors also examine framework-like usage separately by analyzing the presence of `extends` relationships to determine which APIs exhibit framework characteristics.

De Roover et al. present a tool designed to infer usage characteristics of individual API elements within client applications [DLP13]. For instance, it distinguishes between trivial interactions (such as direct method calls) and non-trivial ones (such as extensions or subclassing) to provide a more nuanced understanding of real-world API usage. The tool offers a user interface that displays API elements alongside the usage patterns identified in client code, providing developers with insights into how their API is employed in practice. However, it does not enumerate all possible forms of symbol usage nor quantify how frequently each usage variant occurs.

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.

4.7.2 Usage patterns and protocols

Usage patterns and protocols are a specific series of interactions performed on one or multiple exported API symbols to achieve specific functionality. The series of interactions can be categorized as language patterns common between different APIs, or be seen as one unique protocol that one must follow to use the functionality offered by the library.

A variety of work in the literature addresses the problem of mining and recommending unordered or sequential API usage patterns and protocols to users [Ngu+19b; Rob+13; Zho+09; Wan+13; BW12].

Nguyen et al. present FOCUS, a tool that recommends API usage–both in terms of method invocations and broader usage patterns–by analyzing how similar open-source projects employ the same APIs [Ngu+19b]. The tool mines these projects to derive usage recommendations, which can then be integrated into the developer's existing code. It employs a similarity calculator to compare projects and declarations, constructs usage patterns or snippets via a code builder, and ranks relevant API function calls using an API generator. Static analysis is used to parse the code, relying on Rascal as the underlying modeling and analysis framework for extracting method invocations and declarations.

Zhong et al. present a tool, MAPO, that recommends code snippets related to the usage pattern the developer is attempting to implement [Zho+09]. These short snippets are displayed directly within the IDE (Integrated Development Environment) and are intended to be contextually relevant. The approach inductively derives API usage patterns from open-source repositories,

enabling the construction of usage models and protocols that guide developers toward correct and common practices.

Wang et al. propose an improved approach over MAPO for mining succinct and high-coverage usage patterns of API methods from source code [Wan+13]. The method aims to generate more precise and representative usage patterns, thereby enhancing the quality of pattern recommendations.

Buse and Weimer introduce an algorithm for synthesizing API usage examples using a combination of data-flow analysis, clustering, and pattern abstraction techniques [BW12]. The approach automatically generates representative and coherent usage examples from existing codebases.

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.

Other approaches even gather data from Q&A websites to infer and recommend usage patterns [Zha+18; UKR20].

Zhang et al. analyze API usage violations found in StackOverflow code snippets, which–despite being intended as helpful guidance–often contain misuse patterns such as incorrect call ordering, missing guard conditions, or omitted API calls [Zha+18]. The authors advocate for augmenting StackOverflow answers with additional API usage details to mitigate such issues.

Uddin et al. present an approach for mining API usage scenarios directly from StackOverflow [UKR20]. The method automatically retrieves relevant code snippets from posts to identify and characterize common usage patterns.

Other work empirically studies these usages in the wild.

Zhong and Mei examine how different categories of APIs are employed and evaluate the relative importance of various API elements, usage patterns, and pitfalls [ZM19]. The study investigates suitable formats for representing API usage, the roles of fields and static elements, differences between single-type and multi-type usage, and the overall frequency with which APIs are used in practice.

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 SUMs (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 SUMs (Syntactic Usage Models) and footprints can contribute to extracting more fine-grained usage patterns and protocols, thereby complementing existing approaches.

4.7.3 API evolution induced syntactic breaking changes

There is a wide range of literature on the nature and driving forces of API evolution [KTD20; Kul+18; MRK13]. These approaches try to investigate breaking changes without complete modeling of API symbol uses. We detail some of this work, which touches on the modeling of symbol usage without fully considering the various kinds of interactions. We also detail related work, which shows the importance of going beyond symbol name references for usage analysis.

Ketkar et al. analyze type changes in Java across a large collection of OSS projects [KTD20]. The authors highlight that type changes are surprisingly common–more so than renamings–despite receiving comparatively little prior research attention. The study demonstrates the prevalence and importance of type-level evolution in real-world software systems.

Kula et al. investigate how refactoring performed by library developers affects the subset of the API actually used by clients, with a particular focus on API breakages [Kul+18]. The study reveals that refactorings account for only 37% of the BCs affecting client-used API elements. The remaining breakages are predominantly motivated by other maintenance activities–such as feature additions and bug fixes–that tend to introduce more substantial modifications. These findings suggest that refactoring alone is not the primary driver of API breakage in practice.

McDonnell et al. examine how Android client applications adopt new API levels and how these adoption patterns correlate with API evolution [MRK13]. The authors observe that, although the Android API evolves rapidly, client applications update their API level comparatively slowly. They analyze changes across API levels, investigate which components evolve most frequently, and study the distribution of API levels used across clients. Their results indicate that client projects tend to adopt faster-evolving APIs more readily, and that these APIs become disproportionately more used over time. They also report that frequent API updates correlate strongly with bug fixes, and that codebases undergoing more API updates tend to experience more defects. The authors caution that some findings may have been influenced by the instability of the most recent Android releases at the time, which may have discouraged developers from upgrading.

There is also a wide range of literature on BCs (Breaking Changes) and their impact [Och+22; Xav+17; Bri+18b].

Ochoa et al. expand prior work on semantic versioning compliance in Maven Central by adding seven additional years of dependency history [Och+22]. The study evaluates whether libraries hosted on Maven Central adhere to semantic versioning conventions when introducing BCs. It relies on Maracas, a static analysis tool that detects BCs between successive versions of a library and identifies their corresponding impact locations in client code.

Xavier et al. conduct a large-scale empirical investigation of SyBCs in OSS projects [Xav+17]. The authors analyze the frequency of BCs, their client impact, and characteristics of libraries prone to generating BCs. They find that the number of BCs increases as libraries evolve and grow in size, and that BCs predominantly affect methods. While they hypothesize that developers may lose track of their API as it becomes more complex–contributing to accidental breakages–the

study also notes that the majority of BCs have limited practical impact on clients and are often linked to library evolution, growing adoption, and development activity.

Brito et al. explore the motivations behind SyBCs by contacting developers directly via email after detecting breakages using APIDiff [Bri+18b]. The survey asks whether developers were aware of the BC, their motivations for introducing it, whether they agreed it constituted a BC, whether deprecation was considered, and whether clients were notified. The study finds that developers often introduce BCs without realizing it, especially when the affected functionality is believed to be internal or unlikely to be used externally. Other reasons include attempts to simplify the API, which can be challenging without insight into client usage. Developers report that deprecating older methods increases maintenance burden, making refactoring or removal more appealing. A lack of visibility into client usage appears to complicate decision-making, leading to unintentional breakages.

Thus, we believe that UCov could complement these approaches by providing them with more accurate usage models.

Some studies focus on detecting BCs (Breaking Changes) through API usage patterns [Gao+21; Zai+22; LGS22; Ven+23; Ngu+19a; Foo+18; Har+22].

Gao et al. introduce a tool, APIFix, for synthesizing client-side adaptations required to migrate to newer API versions that include SyBCs [Gao+21]. APIFix uses examples provided by developers–including updated test cases–to infer transformations that update client code automatically. Because documentation does not always capture the full impact of SyBCs, developers may lack the necessary insight to anticipate migration difficulties. The paper illustrates this with an example from DbUp, where many clients remained on an older version (v3.3.5) because migration to v4.0 required adapting to a constructor that gained an additional parameter. This underscores the challenge developers face when anticipating the client impact of such changes.

Zaitsev et al. address questions not fully explored in earlier research on library evolution and BCs, focusing on developers' and clients' perceptions of library evolution, its impact, and the support needed to mitigate related issues [Zai+22]. By surveying both client developers and library maintainers, the authors analyze the challenges associated with updating APIs and the obstacles clients face during migration. Consistent with earlier findings (*e.g.*, [Bri+18b]), the most common reasons for introducing BCs are adding new features and simplifying the API. Clients report that insufficient or misleading documentation significantly complicates migration, whereas well-documented changes make the process considerably easier.

Venturini et al. study BCs in the npm ecosystem [Ven+23]. The authors investigate how BCs affect client packages and how clients recover from them. They report that most libraries introduce BCs in violation of semantic versioning, and that clients frequently cope by downgrading to a previous version. They further find that the most common form of BC arises from feature modifications rather than object-type changes. Additionally, they observe that BCs are typically documented informally, predominantly through issue reports.

Nguyen et al. present a tool designed to detect frequently occurring semantic code changes by leveraging change graphs and AST representations [Ngu+19a]. Although intended to identify

recurring change patterns in practice, the technique can also support API designers by revealing how client code is typically adapted to new features, thereby enabling the extraction of feedback from existing projects that have already undergone such transitions.

Foo et al. present another tool aimed at recommending dependency updates by statically analyzing changes between library versions and determining whether clients rely on modified code paths [Foo+18]. The authors conclude that semantic versioning is routinely violated and is often impractical to follow strictly. Based on a corpus of popular OSS projects, they show that their approach can suggest dependency updates for approximately 10% of libraries.

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 BC, for example, depends on both the symbol being used and the way it is used.

We believe that the rich information provided by SUMs (Syntactic Usage Models) and footprints can better inform BC (Breaking Change) detection tools [JD17] and improve their accuracy in assessing the impact of BCs on client code.

The works in question, by Jezek and Dietrich, compare state-of-the-art static analysis tools designed to assess whether module versions remain backwards compatible [JD17]. The authors construct a benchmark dataset intended to expose BCs and evaluate each tool's ability to detect them. Their results indicate that existing tools exhibit notable shortcomings, frequently failing to identify certain categories of API incompatibilities.

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 [BSV15; Bus+19].

The works in question, by Businge et al. study Eclipse API usage [BSV15]. Eclipse provides stable APIs (referred to as good APIs) and unstable APIs (referred to as bad APIs). The authors examine the extent to which projects rely on these interfaces and find that projects using bad APIs are typically larger and utilize more API elements. They also observe that developers remove dependencies on bad APIs either by re-implementing the needed functionality themselves, migrating to equivalent stable APIs, or eliminating the functionality that relies on unstable API elements.

Additionally, Businge et al. analyze Eclipse's unstable API elements (*i.e.*, bad APIs from the previous study) to assess their long-term stability and determine whether some can be promoted to stable status [Bus+19]. The authors apply clone detection techniques to measure how code snippets evolve and to identify which unstable elements exhibit consistent stability.

4.8 Conclusion

In this chapter, we introduced SUMs (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. This

static analysis tool 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 demonstrated how UCov can identify undocumented and untested interactions, as well as misalignments between the legal uses of the API and actual uses in client code.

5 Client–Library Compatibility Testing with API Interaction Snapshots

Modern software development heavily relies on third-party libraries to speed up development and enhance quality. As libraries evolve, they may break the tacit contract established with their clients by introducing BeBCs (Behavioral Breaking Changes) that alter run-time behavior and silently break client applications without being detected at compile time. Traditional regression tests on the client side often fail to detect such BeBCs, either due to limited library coverage or weak assertions that do not sufficiently exercise the library's expected behavior.

To address this issue, we propose a novel approach to client–library compatibility testing that leverages existing client tests in a novel way. Instead of relying on developer-written assertions, we propose recording the actual interactions at the API boundary during the execution of client tests (protocol, input and output values, exceptions, *etc.*). These sequences of API interactions are stored as *snapshots* which capture the exact contract expected by a client at a specific point in time. As the library evolves, we compare the original and new snapshots to identify perturbations in the contract, flag potential BeBCs, and notify clients.

We implement this technique in our prototype tool Gilesi, a Java framework that automatically instruments library APIs, records snapshots, and compares them. Through a case study on several client–library pairs with artificially seeded BeBCs, we show that Gilesi reliably detects BeBCs missed by client test suites.

Chapter Contents

This chapter in the literature

This chapter is initially published in the literature as

Published Conference Paper

Gustave Monce, Thomas Degueule, Jean-Rémy Falleri, and Romain Robbes. “Client–Library Compatibility Testing with API Interaction Snapshots”. In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME). Auckland, New Zealand: IEEE, 2025

and features artifacts published as

Published Artifacts

Gustave Monce, Thomas Degueule, Jean-Rémy Falleri, and Romain Robbes. *Artifacts for "Client–Library Compatibility Testing with API Interaction Snapshots"*. Version 1.0.0. July 24, 2025. DOI: 10.5281/ZENODO.16411966. URL: https://zenodo.org/doi/10.5281/zenodo.16411966

as well as a proof of concept implementation of the concepts presented, available as

Proof of Concept Implementation

[SW] Gustave Monce, Thomas Degueule, and Jean-Rémy Falleri, *Gilesi, Alien-Tools – GitHub*. URL: https://github.com/alien-tools/gilesi

This chapter extends the original publication with a more comprehensive presentation of the approach (Section 5.3) and a complete description of Gilesi implementation in (Section 5.4). Further detailed experimentation results are added additionally (in Section 5.5).

5.1 Introduction

Third-party libraries are a key component of modern software development (Section 3.1). The contract between the API of the library and the client can break due to the introduction of BCs (Breaking Changes) during library evolution (Section 3.3.1). These frictions at the boundary between clients and libraries have serious implications for both parties [LDP20; LGS22]. Clients must be warned of potential changes that could impact them and must protect themselves from vulnerabilities [Wu+23] and supply chain attacks [Lad+23]. Libraries must avoid carelessly breaking their clients to retain their trust and evolve gracefully [Bog+21]. While there has been significant work on the prevalence [Och+22; Jay+25; Bri+20], detection [Lat+25; Bri+18a; Foo+18], and remediation [Gao+21; Nav+23; ZM24] of syntactic changes, BeBCs remain a challenge [Zha+22a; MRW17]. Library tests—particularly regression tests—while well-suited to detect BeBCs [Liu+23], are written by library maintainers and may not accurately reflect the actual usage of the library in the wild, which can differ significantly from the maintainers' expectations [SAD22] (Chapter 4).

How can client projects ensure that changes introduced in a new library release will not affect them? Client developers naturally write regression tests to validate their own logic, which indirectly exercise the libraries upon which their logic is built. However, empirical evidence suggests that client tests are often too weak and ineffective in detecting BeBCs introduced in library releases [Jay+24; Gyo+18; HG22]. This can be caused by poor coverage of the library from the tests, weak assertions, and the significant distance between the tests and the exercised method in the library [NW19]. Approaches in the literature that tackle this problem either do not take into account how libraries are used in client code [Zha+22a; Che+20], suffer from false positives [HG22], or adopt a crowd-sourced approach that assumes the existence of already migrated clients [Che+20; Zhu+23].

In this chapter, we propose a novel approach to leverage existing client tests and increase their ability to detect library BeBCs. We propose to infer the behavioral contract between a client and a library by observing the interactions (protocol and values exchanged) at the API boundary between the client and the library while client tests are running. The sequence of interactions recorded during the execution of a test forms a *snapshot* that embodies the precise expectations of a healthy client towards the library at a given point in time. When the library is upgraded, the snapshots produced against the new library version can be compared against the original snapshots to detect differences in the interactions and report potential BeBCs. As snapshots are recorded directly at the API boundary, they are more sensitive to potential perturbations in the library's behavior. They are more likely to catch BeBCs than client tests.

We present Gilesi, a prototype implementation of our approach for Java libraries [SMDF]. Gilesi is applicable to two versions of a library presenting no SyBCs to highlight potentially breaking BCs. Gilesi seamlessly instruments libraries to record interaction snapshots during the execution of client tests and compare them when the library is upgraded. We showcase the capabilities of our approach and Gilesi on an exploratory case study involving the popular libraries `JSoup` and `Commons Lang 3`, as well as 27 of their clients. We find that Gilesi can detect artificial BeBCs introduced in the libraries that are missed by client test suites.

5.2 State of practice

BeBCs are a type of BCs (Section 3.3.1) that can affect the client of libraries during dependency updates. BeBCs unlike SyBCs can be hard to detect. Due to the rapid evolution of software libraries, BCs pose an increased risk to client. Zimmermann notes that for the Coq software project, version 8.5 caused enough BCs for some projects depending on Coq to take more than 3 years to migrate successfully from a previous version [TZim19]. Rapid evolution of software also hardens the task of migration as more releases get created with potentially more BC to tackle. While SyBCs have been extensively studied in the literature [ODF22; Jay+24], BeBCs have received little attention.

To mitigate the risks associated with BeBCs, clients naturally write tests to validate their own logic, which indirectly exercises the logic of the APIs they use. In particular, developers use *regression tests* to ensure that changes do not unexpectedly alter their software's behavior [Gyo+18; Liu+23]. When libraries are updated to newer versions, these tests can sometimes catch BeBCs introduced in the library and inform client developers [HG22; Jay+24]. Executing the clients' test suites is also beneficial to library maintainers, as it can exercise their library in ways that they have not necessarily anticipated and catch regressions unseen by their own regression tests [Gyo+18; TZim19]. For example, Coq leverages a set of community-built "contrib" projects to construct a compatibility benchmark. A limitation of this approach is that the list of projects needs to be maintained as the software evolves and the number of users evolves, which can be complicated due to discovery challenges and needs to cover a broad set of functionality to remain effective towards new features. Projects also need to be patched to work with newer versions of the library, which complicates their inclusion as part of CI. A popular implementation of the RDCT idea is GitHub's Dependabot, which adopts a crowd-sourced approach to run the CI and tests of client projects that have migrated to a new version and identify potentially unsafe upgrades [RCH24; He+23] to provide a compatibility score shown to users before accepting the pull request. Said compatibility score, however, may not reflect the actual compatibility with the client existing usage of the library. An alternative implementation, more library developer focused, is BreakBot [ODF22], providing a compatibility report of client making use of the developer's dependency, by employing static analysis to spot usage sites in client code impacted by SyBCs.

5.3 Approach

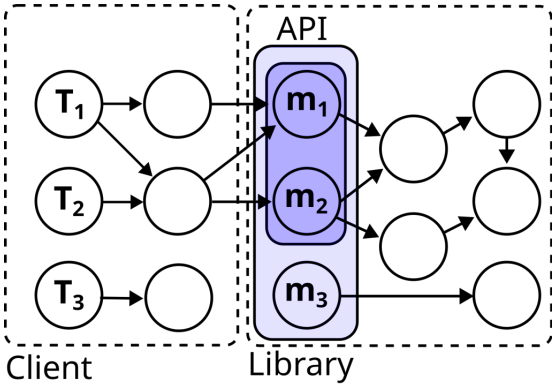


Fig. 5.1 Libraries expose their features through dedicated APIs. Clients use a subset of these APIs to implement their own features.

For the sake of readability, only methods are depicted. T_1, T_2 and T_3 represent the client tests. m_1 and m_2 represent the subset of the APIs used by the client. m_3 ends up unused by the client's tests.

We hypothesize that developers write test cases to exercise important features and execution paths in their code. When the implementation of these features relies on third-party code, however, developer tests may not be able to properly assert its behavior due to the distance between the test case and the method it tests [NW19] and the difficulty of writing assertions on code they do not own.

Our intuition is that BeBCs (Behavioral Breaking Changes) introduced in libraries primarily manifest as perturbations in the sequence of invocations (protocol) or values exchanged at the API boundary before and after the change (Figure 5.1). These include return values, in-out parameters, exceptions, or crashes. These perturbations reflect deviations from the library's expected behavior but may not always propagate far enough through client code to trigger existing assertions. Nevertheless, they can still affect the client's behavior and should be detected.

To address this challenge, we propose an approach that involves instrumenting and recording interactions at the API boundary during the execution of client tests. When running a client's test suite against a specific version of the library, we capture these interactions and record them as *snapshots*, *i.e.*, concrete traces that materialize the exact expectations of the client towards the library at a given time. Inspired by snapshot testing [GRV23] and production monitoring techniques such as PANKTI [Tiw+22], snapshots document the behavioral contract between a client and a library.

We then produce another set of snapshots when the dependency gets updated. By comparing those snapshots, we can detect changes in the behavioral contract between this dependency and the client, and alert the client on potential BeBCs affecting its code.

5.3.1 Motivating example

To illustrate the challenge and approach taken to resolve it, we show a motivating example that will be the basis of our explanations throughout the entire section in Figure 5.2. The example showcases an API element of a library, Commons Text, and a subset of a client making use of this library via one of its internal methods, accessed by one of its unit tests. In the example, two different versions of an API element are showcased; the newer version of it introduces a BeBCs that affects the client showcased.

```
public class StringTokenizer ... { public List<String> getTokenList() {
-   List<String> list = new ArrayList<>(tokens.length);
-   Collections.addAll(list, tokens);
-   return list;                                          // Returns ArrayList
+   return Arrays.asList(tokens);                         // Returns Arrays$ArrayList
  }}
```

Snp. (5.1) Diff between API `1.9` (compatible) and API `1.10` (breaking) for Figure 5.2.

```
static List<String> fetchProducts(
                      String products) {

  StringTokenizer st =
         new StringTokenizer(products);
  st.setDelimiterChar(',');
  return st.getTokenList(); }
```

Snp. (5.2) Client code for Figure 5.2.

```
@Test void test_fetchProducts() {
  String str = "apple,banana,cherry";
  List<String> fruits =
                  Products.fetchProducts(str);
  assertEquals(3, fruits.size());
  assertThat(fruits.
     contains("apple", "banana", "cherry")); }
```

Snp. (5.3) Client test for Figure 5.2.

Fig. 5.2 An example BeBC (Behavioral Breaking Change) introduced in Apache `Commons Text` and released in version `1.10`.

The regression was introduced in commit `2d1ab7`[1], identified in TEXT-219 and later fixed in commit `f9846b`[2].
Add and *remove* operations will throw `UnsupportedOperationException` upon update. The client test fails to detect this change.

[1] https://github.com/apache/commons-text/commit/2d1ab7ea72298949900df47f65b4f71d56411f0b
[2] https://github.com/apache/commons-text/commit/f9846b10b2365a36f95f63ff9f90e0f8847f901b

5.3.1.1 Interaction flow between a client test suite and a library dependency

In Figure 5.2 is shown a client interacting with the `Commons Text` library via one of its tests to turn a string representing fruits (`"apple,banana,cherry"`) into a token list. The client test ensures the string gets correctly tokenized into an array list of products. An internal client method `fetchProducts` is responsible for performing this task and makes use of the `Commons Text StringTokenizer` API elements to achieve this task. The client test therefore depends on an internal client defined method to split the string into an array of strings, and said internal method depends in turn on an API element of the `Commons Text` library. The client method `fetchProducts` first starts by initializing the instance of the `Commons Text StringTokenizer` class with an argument representing the string to tokenize, in this case `"apple,banana,cherry"`, stored in an internal variable of the object instance being constructed. It then calls the `setDelimiterChar` method, with an argument of value `','`, which sets an internal state in the `StringTokenizer` object instance to make the result of `getTokenList` split the string by the `','` character. When `getTokenList` is then called, it splits the provided string to return an array of value `"apple"`, `"banana"`, `"cherry"`. It is then returned to the test, and the test later compares it using the `assertEquals JUnit` API for its size, and the `assertThat JUnit` API for its content to ensure we have 3 elements, and their values contain `"apple"`, `"banana"` or `"cherry"`, regardless of the order.

5.3.1.2 Impact of API changes to a client test suite

As seen previously, the client depends on three API elements of the `Commons Text` library: `StringTokenizer`, `setDelimiterChar`, and `getTokenList`. The client behavior depends on the result of the `getTokenList` call. Furthermore, the result of the `getTokenList` call depends on the initialized internal variables set by the `StringTokenizer` constructor (for the string to tokenize) and the `setDelimiterChar` method, for the character to consider for the tokenization. These dependencies are on input and output values alone. A change in the behavior of these methods, for example, by adding a call or modifying another internal variable, would not impact the behavior of the client only using the return values. Only changes in the input and output values are subject to impacting the behavior, as it's what the client depends on. The changes can manifest themselves in future API element usage, like seen with the `setDelimiterChar` method impacting the result of `getTokenList`. A change in behavior of the `setDelimiterChar` method that does not set the internal variable that the `getTokenList` method uses would thus only be seen when the client makes use of `getTokenList` later. If the client does not use the `getTokenList` API element, then the change in `setDelimiterChar` would not be caught by the client, but would not impact it either. This means that some BeBCs can not be caught by clients, but for all of those changes, they do not impact the client unless they affect the input and output values that are transferred as part of the interaction contract between the client and the library at the API boundary. Some client tests may be lacking in quality and would not check the code fully. In this case, the client tests can miss BeBCs that affect the program behavior due to poor coverage of the client code. In the end, while API boundary changes are required for the BeBCs to impact the client behavior, they are not necessarily caught by client tests.

5.3.1.3 Impact of API value monitoring on behavioral breaking incompatibility testing

As discussed previously, if a BeBC occurs, it must manifest itself within the API boundary, as a change on input and/or output values (for both arguments, and return values, or exceptions). However, these changes are not always caught by client tests. Monitoring the API boundary for changes is therefore enabling the detection of those changes easily and works for detecting BeBC. In the previous example, thanks to the monitoring of the API elements, it is possible to catch the change in the type returned by the `getTokenList` method.

5.3.2 Interaction Snapshots

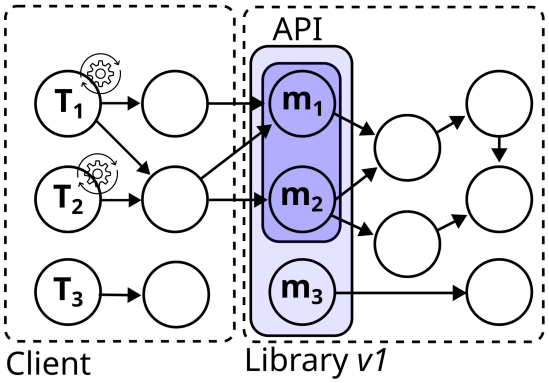


Fig. 5.3 Interaction snapshots are generated for a subset of API methods used by the client.

T_1, m_1 and T_2, m_2 are instrumented to obtain snapshots of the interactions performed on m_1 and m_2. T_3 is not instrumented due to it not consuming any API element of the library.

Each test case of the client ends up producing a snapshot S represented as a sequence of API interactions (Figure 5.3). An interaction $I = \langle m, o, \langle p_1, \ldots, p_n \rangle, r \rangle$ records, for a particular API method m, the receiver object o on which the method is invoked, the list of input values p_i passed as arguments and the (typed) result of its execution r—either a concrete value or an exception.

For example, in Figure 5.2, the initial execution of the test first invokes the client code depicted, which in turn invokes the API. All interactions depicted earlier in the motivating example are reflected in the snapshot, the instance creation with as its parameter `"apple", "banana", "cherry"`, and the subsequent `setDelimiter` and `getTokenList` calls. The resulting snapshot for this test consists of 3 API interactions depicted in Snp. 5.4. This snapshot materializes the exact interaction that happened when executing the test `test_fetchProducts` against version 1.9 of the library.

```
1 ⟨StrTokenizer.<init>, ∅, "apple,banana,cherry", o1⟩,
2 ⟨setDelimiter, o1, ",", ∅⟩,
3 ⟨getTokenList, o1, ∅, ArrayList("apple","banana","cherry")⟩
```

Snp. 5.4 Interaction snapshot (S_1) of a client of Apache `Commons Text` version 1.9, depicted as part of Figure 5.2

5.3.3 Detecting behavioral changes

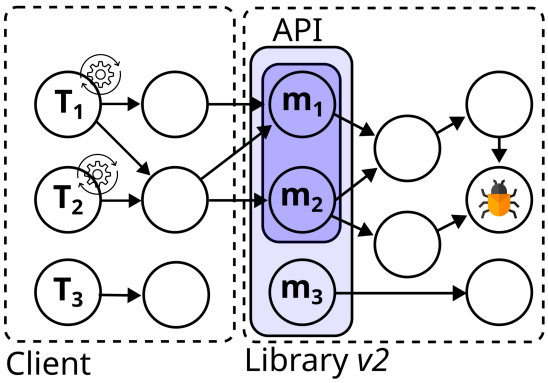


Fig. 5.4 A behavioral change has occurred in a library method accessible from the subset of API methods used by the client.

The change may propagate to the API boundary and cause a BeBC (Behavioral Breaking Change). T_1, m_1 and T_2, m_2 are instrumented to record snapshots.

When the library is upgraded, we run the client's test suite again and collect a new set of snapshots. The new snapshots can be compared to those recorded with the previous version to identify BeBCs. Any difference between the original and the new snapshots for a particular test case indicates a change in the library's behavior. These discrepancies may arise from altered usage protocols (when the order of method invocations differs) or different return values (when an API produces a different or exceptional result for the same inputs on the same object). Since input/output variables are now evaluated at the API boundary, our approach is more sensitive to subtle behavioral changes (Figure 5.4) that may not cause existing tests to fail, but still affect client logic. This enables clients to detect and review behavioral changes, even when those changes do not propagate to existing test assertions.

In Figure 5.2, when the API method is updated, the new snapshot (S_2) consists of 3 API interactions, among which the last one differs, because `getToList` returns a different value with a different run-time type, as depicted in Snp. 5.5. Because $S_1 \neq S_2$, the two snapshots differ, and a BeBC is reported for the method `getTokenList` in the new version. Comparing the snapshots involves first comparing the sequence of interactions to determine if they match in number and are ordered identically. Each interaction is compared by matching the symbol concerned, its input and output parameters, and whether instances of each object match with the other sequence. We detail further the comparison of two sequences of interactions in Section 5.4.3.

```
1 ⟨StrTokenizer.<init>, ∅, "apple,banana,cherry", o1⟩,
2 ⟨setDelimiter, o1, ",", ∅⟩,
3 ⟨getTokenList, o1,∅,Arrays$List("apple","banana","cherry")⟩
```

Snp. 5.5 Interaction snapshot (S_2) of a client of Apache `Commons Text` version 1.10, depicted as part of Figure 5.2

But are all behavioral changes necessarily breaking for the clients? Not all behavioral changes are breaking. For example, the returned type change depicted in Snp. 5.5 would not affect clients that do not leverage operations on the list, such as inserting new items. Without proper deep analysis of the program and without abundantly available test cases from the client capturing

the exact outcomes of the client, this is impossible to evaluate. This creates the possibility of potentially breaking BeBCs, but that could be compatible with certain types of use. While the behavioral changes do exist, we cannot affirm for sure they are going to be breaking for the client as it requires the client developer to evaluate. The approach, however, remains successful in detecting behavioral changes, and useful in alerting to potential BeBCs, more useful than tests that may not cover the code sufficiently, or catch all cases of a dependency API.

5.4 Gilesi: Java Client–Library Compatibility Testing

Gilesi is our proof-of-concept implementation of API interaction snapshots-based compatibility testing for Java client–library pairs. A binary compile of Gilesi alongside all artifacts discussed in this chapter is available on Zenodo [SMon+25a]. Gilesi implements a Java agent that records interaction sequences (*Snapshots*) for one execution of a client's test suite, and enables comparing it with other executions. When implementing such a tool, careful consideration needs to be applied to what data is recorded as part of those snapshots, how to compare said data, what to serialize, and how to instrument the library API boundary properly. In this section, we discuss the choices made when implementing Gilesi and detail what data is serialized and how to compare it between different runs of a client test suite. We also discuss the overall architecture of the tool, depicted in Figure 5.5.

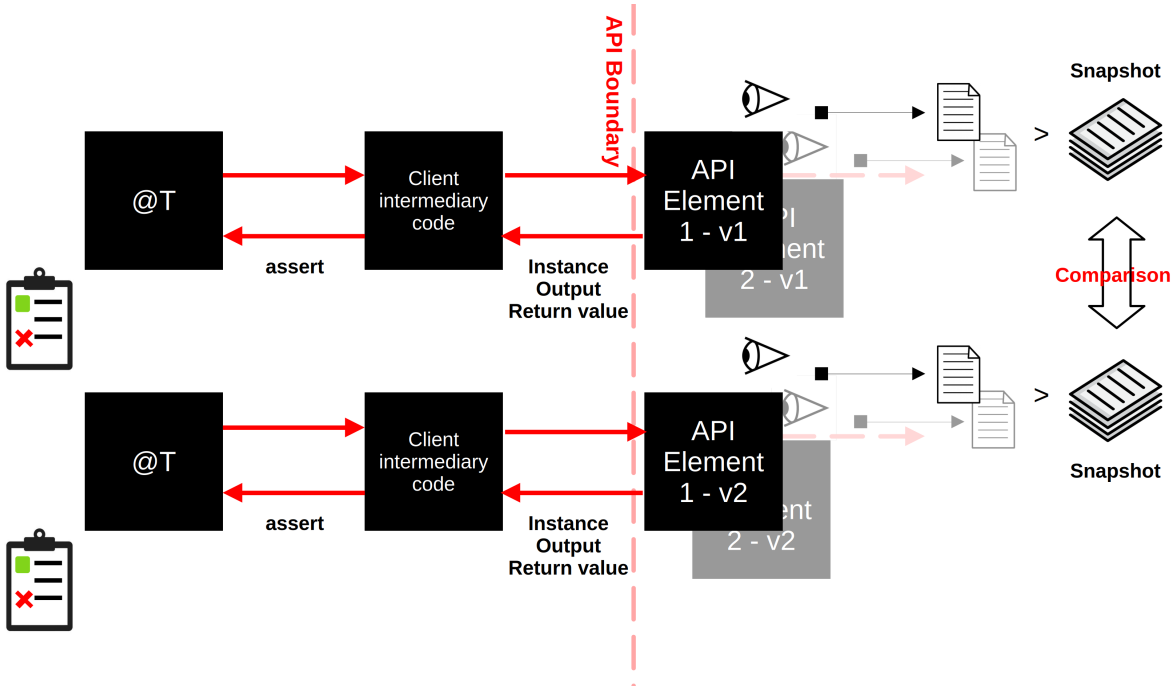


Fig. 5.5 Data collected by Gilesi during test execution. The version of the library dependency being evaluated for behavioral changes (v2) is compared with the previous version (v1). Gilesi compares snapshots created during runtime execution of client tests to determine the possible presence of BeBCs.

5.4.1 Instrumenting the API

As shown in Figure 5.1, only a subset of library code is typically exposed to clients through a dedicated API. In object-oriented languages like Java, this typically includes exported types, methods, and fields. BeBCs, whether introduced in public or internal code, must eventually surface at the API boundary. Thus, a first step is to delimit the API of the library and the subset used in a particular client.

To target only the relevant API used by a given client, we extract a client-specific SUF (Syntactic Usage Footprint) using UCov (Chapter 4). This enables us to focus on API symbols that can affect the client. These API symbols are then instrumented to log the values passed into and returned from each method during test execution. We use Java's instrumentation APIs in conjunction with ByteBuddy[3] to implement a run-time agent that intercepts class loading events, identifies relevant method signatures (as determined by UCov), and injects probes at their entry and exit points. These probes are responsible for capturing the receiver object, input arguments, and return values. Aside from the added probes, the method's behavior remains unchanged.

5.4.2 Recording Snapshots

Once instrumentation is in place, running the client's test suite triggers the probes on each direct or indirect API call. The main challenge lies in ensuring direct API symbol utilization from the client and not the library itself, and serializing the observed values into stable snapshots that can be compared across library versions.

5.4.2.1 Challenge in ensuring a direct symbol usage from the client

Software libraries feature an API designed to be consumed by clients. This API, however, for diverse reasons (simpler API symbol usage, internal code functionality) might also be used by the library itself. This creates a problem where a client may invoke one API symbol of the library, which itself will leverage another API symbol. This kind of usage is therefore considered to be "indirect" because it does not originate from the client code. We are only interested in capturing the interactions the client has towards the library's API, not how the library interacts with itself. Furthermore, if we included those indirect interactions, we would introduce errors into our snapshots. Indeed, to reproduce the indirect usage of the API, only reproducing the API symbol invocation originating from the client suffices to also reproduce the indirect invocation observed. For the same reasons, IoC scenarios created by the use of callbacks provided by the client that lead to additional API symbol invocations do not need to be captured. The callbacks are invoked upon invocation of another API symbol. We therefore need to filter API symbol invocations that are not directly originating from a client code.

To do this, Gilesi leverages the extracted API model provided by UCov to build a list of all API types. The probes inserted into the API symbols trigger Gilesi's capture of the stack trace that

[3] https://bytebuddy.net

led to this call. This stack trace is stored upon entry of the API symbol into a list, and removed upon exit. Additionally, a capture of the method invocation as an API interaction is made and stored in a list, for generating the snapshot later on.

When subsequent probes are triggered, it then looks for existing stack traces in said list. Suppose a match of the beginning of the stack trace is present. In that case, the recorded API interaction is considered to be the result of an indirect call not coming from the client, but resulting from an earlier direct call of the API. Indeed, suppose that a method present in the library's API `foo` is called directly by the client. The stack trace on entry of `foo` contains no element of the API of the library before `foo`, only client methods. Upon looking at the list of stack traces previously recorded, no stack trace exists that is contained within the `foo` stack trace on entry. Therefore `foo` is a direct invocation and is recorded as part of our snapshots. Its stack trace is saved in the list of previously recorded elements. Suppose now that `foo` calls another method, `bar`, also exported in the API of the library. On entry of `bar`, the stack trace is recorded. Upon reviewing the list of previously recorded stack traces, we find the `foo` recording, which matches the beginning of the stack trace on entry to `bar`. Indeed, `bar` was invoked directly by `foo`, not by the client. Therefore `bar` is an indirect invocation and is not recorded as part of our snapshots. Upon exit of `foo`, the stack trace previously recorded on entry is removed. Now the client decides to call `bar` directly. Because no stack trace previously recorded is found to be contained within the beginning of the entry stack trace collected for `bar`, `bar` is considered to be a direct invocation originating from the client and is recorded as part of our snapshots.

5.4.2.2 Serialization Challenges

While serialization is straightforward for standard JDK types (*e.g.*, primitives, collections), it becomes difficult for deeply nested or library-defined objects. We use the XStream framework to serialize standard types. For complex objects, we defer serialization until additional method calls on them expose simpler, serializable data. For example, when the library returns a library-defined object, we record the interaction but only keep track of the object's identifier without attempting to serialize it. Then, we wait until other API methods are invoked on it and record those as well. The intuition here is that when the new API returns a different complex object, it cannot affect the client until concrete services are invoked. For the standard JDK object types, we also stop at the public members of those types. Some objects contain members that are complex and known to vary from one execution to the next (like class loader information or byte buffers for network stream requests). Stopping serialization at the publicly exported JDK types should catch all possible ways a client may interact with this type and behave differently in case of differences. Limitations exist with IoC scenarios. Serialization frameworks such as XStream are unable to serialize lambdas or interfaces. This poses a problem for reconstructing the contract between the client and library because it prevents the successful capture of the interactions when the callbacks are being used.

5.4.2.3 Identifying object reuse

To capture objects being reused across API symbol uses, a unique instance identifier is collected. This is important because different methods can be invoked on the same objects. The JDK provides an API designed to collect a "system" hash code, designed to reflect the uniqueness of a specific object. This hash code is not guaranteed to be unique for executions leveraging a large amount of objects, but should remain stable for most projects being instrumented. Gilesi therefore calls into the Java's `java.lang.System` class exported `System.identityHashCode(Object)` method to collect this identifier. This identifier, in turn, enables us to distinguish receiving objects from one another.

5.4.2.4 Data collected as part of an API interaction

The probe inserted into the API symbol collects information relevant for creating the snapshot. Aside from the method signature encapsulated into the `methodReference` field of the schema depicted in Figure 5.6, the list of arguments alongside their values before and after the completion of the invocation is recorded in the `preCallArguments` and `postCallArguments` variables. Recording the state of the arguments post execution of the method is important for methods that rely on in-out semantics; *i.e.*, methods that modify passed-in argument states, and where the client makes use of that modified state later on. The return value is also recorded in the same manner. If an exception is thrown, no data is available for the `postCallArguments` variable, and instead the serialized data of the exception and its type information is recorded in `exceptionThrown`. The information about the instance when methods are invoked as part of an instance object is also recorded to know on which instance the call is being performed. This set of information represents the outcome of the methods being invoked. The remaining information is relevant for the tool implementation. The `confName` and `confDescriptor` fields each represent the symbol method fully qualified name and its signature for knowing which symbol has been invoked. The `stackTrace` variable holds the stack trace of the call invocation, which is used for filtering out indirect calls as explained previously. For each of the items serialized (instance, arguments, exception *etc.*), a unique value is recorded to know which unique object is being used, or reused across calls (without needing to re-create a new object) or to link multiple method invocations onto their instance. The serialized value is also stored as a string within the same structure. The information about the object type and its inherited subclasses and/or interfaces is also recorded for matching methods between snapshots. For methods, an additional piece of information is recorded to know whether the item being recorded is a constructor or not.

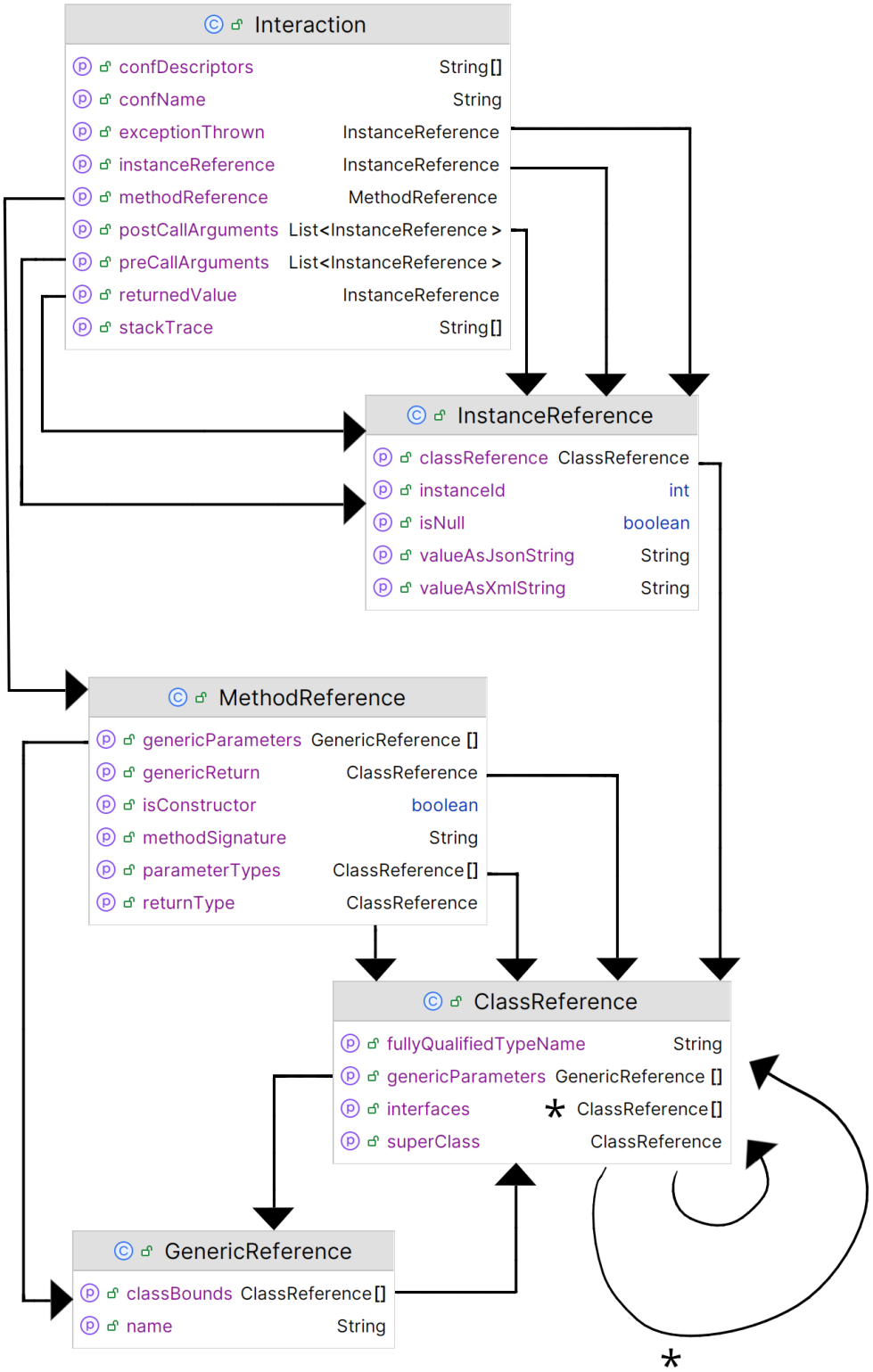


Fig. 5.6 Schema for the individual API interaction records collected by Gilesi during the instrumentation phase.

5.4.3 Comparing snapshots

Comparing snapshots first involves verifying that the order of interactions is identical between the old and new snapshots. When this is the case, interactions are compared pair-wise to verify that their nature and concerned symbol are the same. This is done by comparing the `methodReference` field containing the information on the symbol being used by the client. Then, a comparison is made to ensure the input and output values are the same. The input values are useful for identifying if the call is identical between snapshots; the output values are useful for identifying a difference in behavior. In the case of methods that leverage generics, the information present within the `methodReference` field enables accurate comparison of the types being used. A detailed map of the types and their inherited subtypes is present within the snapshot data itself to facilitate this.

To match interactions applied to specific objects, sometimes reused across invocations, all objects are assigned a unique identifier that enables tracing method invocations on the same objects throughout execution.

Because our approach relies on method instrumentation, it exhibits a limitation for all interactions concerning field reads and writes. However, we argue that those types of interactions are not standard within the Java programming style, especially in properly designed APIs, and developers instead make use of proper getter and setter methods, making this limitation not a major concern for its use on Java client–library pairs.

5.4.3.1 Alignment of identifiers

Snapshots contain unique identifiers for instances of Java classes. These identifiers are relevant for capturing the re-use of arguments across multiple method calls, or differentiating instance invocations of methods from one receiving object to another. These identifiers are collected by Gilesi using the Java's `java.lang.System` class exported `System.identityHashCode(Object)` method. However, the result of this method call is not reproducible across multiple JVM executions.

To compare snapshots and identify similarities in protocol interactions from one snapshot to the next, identifiers must be aligned with each other while retaining the same information about the use or reuse of objects across different method calls.

To achieve this alignment, we first normalize the identifiers for each snapshot. For each encountered identifier, we assign it a new unique value: 0 for the first encounter, 1 for the second, and so on. Suppose we encounter an already seen identifier prior. In that case, we assign it the same value as we previously assigned it, thanks to the use of a map, which stores the identifiers encountered alongside the new identifier we assigned to them. This implementation has the benefit of retaining the order of appearance, and if one object is a receiver for multiple method invocations encountered within the snapshot, while retaining all properties that the interaction sequences bring for capturing the semantic contract exercised by the client onto the library's API. We illustrate this transformation in Figure 5.7.

Comparing the snapshots afterwards becomes easier when differences in argument order, re-use, or instances are observed:

- *In the case of identical interaction snapshots*, the identifiers after alignment will be strictly equal.
- *In the case of interaction snapshots with a different number/order of arguments*, the identifiers would not align anymore due to the new argument(s) being inserted into the snapshot/the different order of arguments, leading to a different order of identifiers.
- *In the case of interaction snapshots where two different invocations on distinct instances get swapped with each other*, the identifier of those instances will not match within the same position.

```
{
 "interaction": {
  "preCallArguments": [
   { "instanceId": 1332208607, ... },
   { "instanceId": 1175662879, ... },
   { "instanceId": 1698746141, ... }
  ],
  "instanceReference": null,
  ...,
  "postCallArguments": [
   { "instanceId": 1332208607, ... },
   { "instanceId": 1175662879, ... },
   { "instanceId": 1698746141, ... }
  ],
  "returnedValue":
   { "instanceId": 1332208607, ... },
  ...
 },
 ...
}
```

Snp. (5.6) Before alignment.

```
{
 "interaction": {
  "preCallArguments": [
   { "instanceId": 0, ... },
   { "instanceId": 1, ... },
   { "instanceId": 2, ... }
  ],
  "instanceReference": null,
  ...,
  "postCallArguments": [
   { "instanceId": 0, ... },
   { "instanceId": 1, ... },
   { "instanceId": 2, ... }
  ],
  "returnedValue":
   { "instanceId": 0, ... },
  ...
 },
 ...
}
```

Snp. (5.7) After alignment.

Fig. 5.7 Example of identifier alignment applied on an API interaction record (original on the left).

5.4.4 Asserting Flakiness

Snapshots, like regular unit tests, can exhibit flakiness. Flakiness [Par+22] is the ability for tests to randomly fail or pass when re-run, even though no changes have been performed by the developer. These tests are therefore unreliable because in a stable execution environment, their outcome is unpredictable. This flakiness could impact the BeBCs detection results due to the unreliable nature of the output values being compared. To ensure that the detection results are not skewed and that detecting BeBCs is possible, it is required to ensure that the recorded snapshots are not flaky. Snapshots are flaky if a specific execution with an identical execution environment leads to different snapshots collected. Flakiness in snapshots is more relevant due to serialization. It is also possible, like with traditional unit tests, for flakiness to be observed due to the nondeterministic nature of the interactions exercised by the client onto the library's API. Because we want to detect BeBCs reliably, we need to assert that the instrumentation of a specific method does not lead to flaky snapshot results. Gilesi therefore automatically does this by performing twice its instrumentation (and overall execution of the program), which outputs two distinct snapshots. These snapshots are then compared strictly. If the snapshots match, the snapshot recording is not considered to be flaky. If they do not match, the snapshot recording linked to this specific execution is considered flaky and Gilesi is unable to be used to assert BeBCs for that particular test.

5.5 Case Study

In this section, we report on a preliminary case study of Gilesi's ability to detect BeBCs compared to client test suites. Client tests are primarily used by client developers to assess compatibility and proper functionality of their changes. Thanks to CI (Continuous Integration), these tests get executed every time the developer commits a change to their software. We argue, however, that reliance on these tests is not enough to protect against potential BeBCs. This is why we evaluate Gilesi's ability to detect more BeBCs compared to client tests as part of this preliminary case study.

5.5.1 Client–Library dataset

To evaluate our approach, we need a set of client–library pairs. In these pairs, the client depends on the library, via its project dependency declaration, but also in its code. We also want clients where the tests can reach the library API symbols. Our case study involves the popular Java libraries `JSoup` and 6 of its clients as well as `Commons Lang 3` and 21 of its clients. The data, notebooks, and source code discussed in this chapter are available on Zenodo [SMon+25a].

5.5.1.1 Libraries

To obtain a set of libraries, we first start with the DUETS dataset [DSB21]. DUETS is a dataset of Java client–library pairs obtained from the GitHub dependency graph that contain test suites built successfully, with enough popularity (large amount of stars on GitHub). We proceed to refresh this dataset to eliminate projects that are no longer obtainable or cannot be compiled in our environment. We then select projects we deem interesting due to their client list as well as their feature set and programming style (that can lead to various interactions for a given API symbol). In this case, we retain the following dependencies:

- Commons Lang 3
- JSoup

Commons Lang 3 is a library leveraging more the classical style, while JSoup leverages more the fluent style. This allows us to look into different types of interactions for a given symbol and to determine if we can detect changes in behaviors regardless of the type of interaction. Furthermore, both libraries contain a sufficient amount of clients compared to the others present in the DUETS dataset, allowing us to maximize the uses performed on the API.

5.5.1.2 Clients

To obtain a list of clients, we once again leverage the GitHub dependency graph to obtain 100 current consumers of those dependencies. We then proceed to attempt building each of them, after filtering non-Java projects as well as projects that declare a dependency on the library we selected, but do not end up utilizing it in their code. We then retain projects for which our approach can be successfully applied without technical issues (Maven crashes, or more generally Java compiler and JVM errors during execution) and remove any duplicate projects.

We end up with the following 6 clients of JSoup:

- `com.codeu.chatapp-chatapp`
- `com.github.beothorn-webGrude`
- `com.github.dhorions-boxable`
- `im.nll.data-extractor`
- `me.chyxion-table-to-xls`
- `us.codecraft-xsoup`

We end up with the following 21 clients of Commons Lang 3:

- `ch.zizka.csvcruncher-csv-cruncher`
- `com.amihaiemil.devops-comdor`
- `com.cognifide.aemrules-sonar-aemrules-plugin`
- `com.github.bingoohuang-delayqueue`
- `com.github.bingoohuang-excel2javabeans`
- `com.github.cschoell-junit-dynamicsuite`
- `com.github.daanvdh.javaforger-JavaForger`

- com.github.davidcarboni-restolino
- com.github.jonpereiradev-jfile-reader
- com.javacreed.examples-gson-typeadapter-example
- com.json.comparison-comparison-core
- com.omertron-API-OMDB
- com.yanceyzhang.tools-wx-chatbot
- de.pentabyte-springfox-enum-plugin
- edu.anadolu-FrequencyDistributionAnalysis
- io.prowave-chargify-webhook-java
- org.cyclopsgroup-jmxterm
- org.jvnet.hudson.plugins-warnings
- org.zalando-jzon
- org.zeroturnaround-zt-process-killer
- com.github.jknack.handlebars

5.5.2 Protocol

As an initial sanity check, we first run the test suites of client projects against the original version of the libraries twice. Then, we filter out any client with failing or flaky test suites. We also record and store which API methods are reached during the execution of the tests.

To simulate the introduction of BeBCs in these libraries, we implement another simple Java agent that dynamically introduces extreme mutations [NJW16] in API methods of interest. The agent removes all code in the original implementation and replaces it with a single return statement that returns a default value (`null` for reference types, zero for integers, *etc.*). This simulates a drastic behavior change that should typically be caught by both client test suites and Gilesi. This approach notably enables the detection of pseudo-tested methods [NJW16]. If the pseudo-tested methods are covered by a given test, it does not necessarily mean a drastic behavioral change (such as complete code removal) will be detected by the client test assertions, due to failure being too distant, or even, unchecked for. We argue that the failure in detecting complete code removal is an issue due to the importance of this kind of fault, and highlights a test failure to detect important functionality changes reliably. By leveraging extreme mutation operands, we enable the evaluation of pseudo-tested methods, and thus, evaluate the effectiveness of an approach towards detecting this type of BeBC. We provide the specification of our mutation operands in Table 5.1.

Then, we evaluate the ability of client test suites and Gilesi to detect the introduced BeBCs using a simple mutation score. First, we use Gilesi to record the set of snapshots produced when running the client test suites against the original version of the libraries. We repeat this operation twice to ensure that Gilesi produces stable snapshots and discard any test that yields flaky snapshots. Then, for each API method reached in client tests, we use the agent to insert the

extreme mutations, one at a time, and run the client test suites another time. If any of the tests fail after introducing the mutation, then we consider that the snapshots/Gilesi successfully kills the mutant and detects the BeBC. Suppose any of the snapshots extracted after the mutation is introduced differ from the corresponding snapshot extracted on the original version of the library. In that case, we consider that Gilesi successfully detects the BeBC and kills the mutant.

We further illustrate and detail each step of the protocol as part of Figure 5.8.

Value Type	Type	Mutant Body
Primitive **Object**	`void` `java.lang.Void`	`return;`

Primitive
 	
Object
	`byte`
	`short`
	`int`
	`java.lang.Byte`
	`java.lang.Short`
	`java.lang.Integer`
		`return 0;`

Primitive **Object**	`long` `java.lang.Long`	`return 0L;`
Primitive **Object**	`float` `java.lang.Float`	`return 0.0f;`
Primitive **Object**	`double` `java.lang.Double`	`return 0.0d;`
Primitive **Object**	`char` `java.lang.Character`	`return '\\u0000';`
Primitive **Object**	`boolean` `java.lang.Boolean`	`return false;`
Object	*Any other*	`return null;`

Primitive
 	
Object
	`byte[]`
	`short[]`
	`int[]`
	`long[]`
	`float[]`
	`double[]`
	`char[]`
	`boolean[]`
	`java.lang.Byte[]`
	`java.lang.Short[]`
	`java.lang.Integer[]`
	`java.lang.Long[]`
	`java.lang.Float[]`
	`java.lang.Double[]`
	`java.lang.Character[]`
	`java.lang.Boolean[]`
	Any other`[]`
		`return new T[] {};`

Tab. 5.1 Extreme mutation operators used as part of pseudo-tested method [NJW16] detection, used in Gilesi's case study experimentation process.

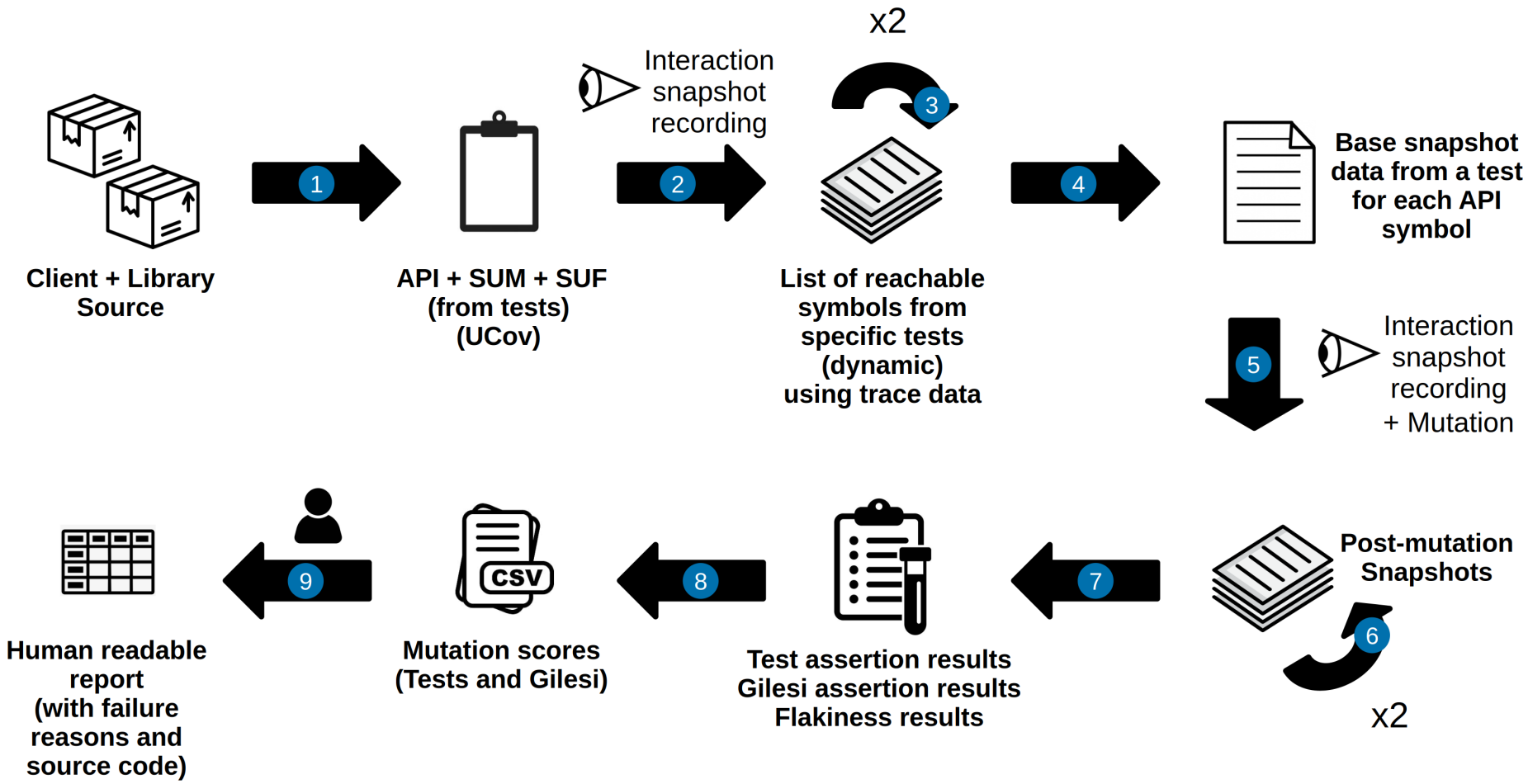


Fig. 5.8 Experimentation flow for the evaluation of Gilesi.

❶ Client test ⇔ API symbol extraction Using static analysis, we first extract the list of exported API symbols from the library source code. We then generate a series of call graphs from the tests, using them as entry points to keep only the paths that connect a client test to a library symbol. This process produces a mapping of which tests can potentially reach each API symbol. This step involves UCov (Chapter 4) to extract the API and usage models to deduce the above reachability list.

❷ Client test execution and snapshot collection From the list of tests obtained in the previous step, we execute each test using a custom Java agent that monitors calls to every API symbol in the library. When an API symbol is invoked, the agent records the execution. This allows us to build a list of symbols that were actually executed at runtime by each client test. During this step, we also collect detailed API interactions (Snapshots) for each API symbol call.

❸ Client test flakiness detection Flakiness may occur in both the client test assertions and the snapshot assertions. To eliminate flaky tests (in both clients and Gilesi), we re-run the identified tests and compare the results of the client assertions with the new ones to ensure consistency.

❹ Base snapshot generation per client test ⇔ API symbol and test result collection Using the collected snapshots, we generate a complete base snapshot for each test execution of a given API symbol. These snapshots capture the client's expected behavior toward specific API symbols during each test. They serve as a baseline for comparison in later runs.

❺ Post-mutation snapshot generation per client test ⇔ API symbol and test result collection For each reachable API symbol identified earlier, we apply extreme mutation. This approach is chosen because extreme mutations are easily detectable by client tests and introduce strong behavioral changes. We collect both snapshots and test results during the mutated runs, producing new snapshots for each API symbol and test pair. These post-mutation snapshots are then compared to the pre-mutation (base) snapshots to check whether the mutation was detected. We also compare the client test assertion results against the base run for the same purpose.

❻ Client test flakiness detection We re-run the tests with the same mutations in place to confirm that there is no flakiness in either the client or Gilesi assertions, just as with the base (non-mutated) tests. Only non-flaky test executions are retained.

❼ Base and post-mutation test result and snapshot comparison Using the collected data, we identify which tests passed or failed after applying the mutations. The test results indicate whether each test successfully detected (killed) the mutant. We then compare the post-mutation snapshots to their corresponding pre-mutation (base) snapshots. If differences are found, we conclude that the mutant was killed; if no differences are found, the mutant survived.

❽ Mutation score computation To compute the mutation score, we aggregate all client tests that can reach a specific API symbol and have been executed against a mutant. A mutant is considered "killed" if at least one test detects it, regardless of how many tests do so. The mutation score is calculated as the number of mutants killed by at least one client test divided by the total number of unique generated mutants. For Gilesi assertions, the same process is applied, based on whether the snapshot comparison detects the mutant.

❾ Report generation Finally, we generate a report showing which mutants were killed by the client tests or Gilesi assertions, which tests were responsible, and whether any results were flaky. The report also includes code snippets related to the executed mutants to support analysis and debugging.

5.5.3 Mutation Scores

To compare our new approach with existing techniques, we need a metric that can be used to measure its effectiveness, and that is comparable with others. The mutation score is typically used as a measure of test suite robustness and can be used to evaluate our approach versus others on an equal set of mutants. Specifically, it allows us to evaluate the ability of existing client tests and Gilesi to detect behavioral changes materialized as mutations.

In our evaluation, we consider that the comparison of 2 snapshots is a test.

We define the killed status of a mutant ($L_A(m)$) as the status indicating whether a mutant m was killed (value: 1) or survived (value: 0) during the execution of a test suite A. Given that approaches can kill the same mutant using different tests as entry points multiple times, we consider the death of a specific mutant only once. In other words, it is enough for the mutant to be killed once by one test to consider the approach effective towards detecting such drastic BeBC.

We define the $L_A(m)$ variable to be above status for a mutation m for the execution of a test suite (A) for this mutant in Equ. 5.1.

Let

$A = \{a^0 \dots a^n\}$: Every test present in the test suite A

$E_a(m)$: Execution of test a with the mutation m in place

0 indicates the test is failing

1 indicates the test is passing

$$L_A(m) = \begin{cases} 1 & \text{if } \exists\, a \in A,\ E_a(m) = 0 \\ 0 & \text{otherwise} \end{cases}$$

Equ. 5.1 Definition of a mutant killed status (L) for an approach's defined test suite (A) and a specific mutant (m).

Finally, we can define the mutation score (M_A) to be the ratio of all alive statuses of each generated mutant by the total number of mutants for a test suite A. The mutation score reflects the effectiveness of an approach towards detecting and killing the generated mutants. We define this variable in Equ. 5.2.

Let

m_{total} : The total amount of mutants generated for a client dependency

$Mutants = \{m_0 \ldots m_{m_{total}}\}$: All mutants generated for a client dependency

$$M_A = \frac{\sum_{m \in Mutants} L_A(m)}{m_{total}}$$

Equ. 5.2 Definition of the mutation score based on an approach's defined assertions (A) between an initial run and subsequent runs with mutants.

5.5.3.1 Client's test suite mutation score definition

When leveraging the client test suite to detect mutants, the tests assert that specific values conform to hardcoded developers' expected ones. These tests are already part of the client test suite already ($A_{client} = \{a^0_{client} \ldots a^n_{client}\}$) and can be reused as is, as shown in Equ. 5.3 for defining $M_{A_{client}}$.

$$E_{a_{client}}(m) = \begin{cases} 1 & \text{if the client test is passing with the mutation } m \text{ in place} \\ 0 & \text{otherwise} \end{cases}$$

Equ. 5.3 Definition of the execution of a test a_{client} on code with a mutant m generated using client tests.

5.5.3.2 Gilesi's mutation score definition

When leveraging Gilesi to detect mutants, the executions lead to the generation of snapshots (S) during execution. These snapshots are generated for every reachable API symbol. For every test capable of reaching an API symbol, we can define a test that asserts whether or not a change occurred between a base initial snapshot ($S_{initial}$) and a newer updated snapshot (S_{future}). By comparing these snapshots between runs, it is possible to define a test suite ($A_{gilesi} = \{a^0_{gilesi} \ldots a^n_{gilesi}\}$) based on an existing client test suite ($A_{client} = \{a^0_{client} \ldots a^n_{client}\}$) capable of detecting a change, as shown in Equ. 5.4 for defining $M_{A_{gilesi}}$. Comparing two snapshots S_0, S_1 is explained in Section 5.4.3.

Let

S_i : Snapshot of the interactions associated with a_{client} during the initial run

S_j : Snapshot of the interactions associated with a_{client} during the run against the mutant

$$E_{a_{gilesi}}(m) = \begin{cases} 1 & \text{if } S_i = S_j \\ 0 & \text{otherwise} \end{cases}$$

Equ. 5.4 Definition of the execution of a test a_{gilesi} associated with the execution of a client test (a_{client}) generated using Gilesi during two executions, i (on original code) and j (on code with a mutant m).

5.5.4 Overall results

Tables 5.2 and 5.3 presents the mutation scores obtained by client tests and Gilesi following our protocol. Overall, Gilesi detects 96% of the 158 mutants introduced in both libraries, while client test suites only detect 89% of them. All mutants killed by client tests are also killed by Gilesi. Conversely, Gilesi detects 10 extreme mutations missed by client tests. This shows that Gilesi is good for consistently killing mutants introduced in the library API boundary, more than the client test suite.

However, there are still seven mutants that are not detected by Gilesi. These cases stem from mutations that alter library-side I/O side effects without changing return values (*e.g.*, on `void` methods), or produce equivalent behavior because the original methods already return default values to all clients.

In this experiment, we rely on extreme mutations performed on the exported API methods to simulate the introduction of BeBCs. Client tests should be able to detect this type of change, due to the whole functionality being removed. We hypothesize that finer-grained, classical mutation operators would be less likely to trigger test failures on the client side. At the same time, Gilesi should retain much of its detection abilities, being closer to the seeded fault. Future experiments should thus further investigate how client tests and Gilesi handle finer-grained mutations (flipped addition/subtraction operands, different return values by default, *etc.*) like done in other programming languages like C [Agr+89] and real BeBCs sourced from a dataset of known BeBCs [Rey+24; DSB21]. Future experiments should also investigate how client tests and Gilesi handle mutations performed beyond the API boundary and see if being closer to the seeded fault by being at the boundary with the API instead of further away at the client test results in a greater coverage of BeBCs and if it reduces the impact on the client code.

Client	Mutants	Client kills	Gilesi kills
com.codeu.chatapp-chatapp	2	2	2
com.github.beothorn-webGrude	2	2	2
com.github.dhorions-boxable	7	5	7
im.nll.data-extractor	10	10	10
me.chyxion-table-to-xls	5	2	5
us.codecraft-xsoup	26	20	20
Total	52	41	46

Tab. 5.2 Mutation scores for JSoup clients as a result of Gilesi's case study/evaluation.

Client	Mutants	Client kills	Gilesi kills
ch.zizka.csvcruncher-csv-cruncher	5	5	5
com.amihaiemil.devops-comdor	3	2	3
com.cognifide.aemrules-sonar-aemrules-plugin	11	11	11
com.github.bingoohuang-delayqueue	7	7	7
com.github.bingoohuang-excel2javabeans	10	10	10
com.github.cschoell-junit-dynamicsuite	4	4	4
com.github.daanvdh.javaforger-JavaForger	5	5	5
com.github.davidcarboni-restolino	5	5	5
com.github.jonpereiradev-jfile-reader	5	5	5
com.javacreed.examples-gson-typeadapter-example	1	1	1
com.json.comparison-comparison-core	1	1	1
com.omertron-API-OMDB	2	1	2
com.yanceyzhang.tools-wx-chatbot	1	1	1
de.pentabyte-springfox-enum-plugin	1	1	1
edu.anadolu-FrequencyDistributionAnalysis	3	3	3
io.prowave-chargify-webhook-java	1	1	1
org.cyclopsgroup-jmxterm	14	12	14
org.jvnet.hudson.plugins-warnings	3	3	3
org.zalando-jzon	1	1	1
org.zeroturnaround-zt-process-killer	1	0	0
com.github.jknack.handlebars	22	21	22
Total	106	100	105

Tab. 5.3 Mutation scores for Commons Lang 3 clients as a result of Gilesi's case study/evaluation.

5.5.5 Illustrative Cases

As part of our case study, we review two noteworthy cases that highlight how Gilesi can help client developers protect themselves against library BeBCs, including for clients with strong test suites.

5.5.5.1 Case 1: Amihaiemil's Devops Comdor and Apache's Commons Lang 3

Amihaiemil's Devops Comdor features functionality designed to help create issues on the code hosting platform GitHub. Part of the test suite of Amihaiemil's Devops Comdor aims to test the creation of an issue when an exception is caught and to verify whether the created issue contains

accurate information about the exception. Information about the exception is extracted using the `getStackTrace` method of `Commons Lang 3` (Snps. 5.8 to 5.10).

With the original version of the library, the issue is successfully created with the correct title and body. When running the same test on the mutated library, `getStackTrace` no longer returns information about the exception. Thus, the created issue lacks crucial information. However, the corresponding test only asserts parts of the issue body and misses the introduced BeBC. The client is thus vulnerable to future evolutions of `Commons Lang 3` and may not detect potential regressions.

On the other hand, Gilesi manages to catch the regression automatically. The original snapshot for this test identifies that `getStackTrace` receives an `IOException` and produces a string describing the exception. The new snapshot produced after the mutation is introduced maps the same input to a different output, a `null` value. As the two snapshots differ, the BeBC is detected and successfully reported to the user.

5.5.5.2 Case 2: Chyxion's Table To Xls and JSoup

Chyxion's `Table To Xls` features functionality designed to turn an HTML document containing a table into an XLS file. This client leverages several `JSoup` features for parsing the HTML document, including `attr` to retrieve the string values of attributes (Snps. 5.11 to 5.13).

In this example, the test verifies that a predefined HTML document can be successfully converted into a spreadsheet. Running the test against the original version of the library successfully creates a new spreadsheet with appropriate rows and columns. Running the test against the mutated version of the library, however, returns a spreadsheet with missing rows and columns. Indeed, the test merely verifies that the conversion runs successfully without crashes or exceptions, without asserting the shape of the returned spreadsheet. Therefore, the client is vulnerable to potential regressions in the evolution of `JSoup`.

On the other hand, Gilesi manages to catch the regression automatically. The snapshots produced with the original and mutated versions differ on multiple aspects. The first interaction with the `attr` method produces a different outcome, and the number of interactions differs. Gilesi successfully catches the difference that can be reported to the client.

5.5.5.3 Code Snippets

```java
public class ExceptionUtils {
  ...
  // Input (IOException): <message>expected</message>
  //                      <stackTrace>
  //                        <trace>co.comdor. ...
  public static String getStackTrace(Throwable throwable) {
    StringWriter sw = new();
    PrintWriter pw = new(sw, true);

    throwable.printStackTrace(pw);

    return sw.getBuffer().toString();
  }
  // Original output: "IOException: expected at Vigilant ...
  // Mutant output  : null
  ...
}
```

Snp. 5.8 API showcasing a missed BeBC (Behavioral Breaking Change) in Amihaiemil's `Devops Comdor`.

```java
public class VigilantAction {
  ...

  public void perform() throws IOException {
    try {
      og.perform();
    }
    catch (IOException | RuntimeException ex) {
      String title = "...";

      String body = "... Exception:" + ExceptionUtils.getStackTrace(ex);

      Issue created = gh
        .repos()
        .get("amihaiemil/comdor")
        .issues()
        .create(title, body);
    }
  }

  ...
}
```

Snp 5.9 Client code showcasing a missed BeBC (Behavioral Breaking Change) in Amihaiemil's `Devops Comdor`.

```java
public class VigilantActionTestCase {
  @Test
  public void opensIssueOnIOException() throws Exception {
    Github gh = mockGithub();
    Action og = mockAction();

    doThrow(new IOException("expected")).when(og).perform();

    Action vigilant = new(og, gh);
    vigilant.perform();

    Issues all = gh.repos().get("amihaiemil/comdor").issues();

    assertThat(all, iterableWithSize(1));

    Issue op = all.get(1);
    String body = "... Exception:" + ExceptionUtils.getStackTrace(ex);

    assertThat(op.json().getString("body"), startsWith(body));
  }
}
```

Snp. 5.10 Client test showcasing a missed BeBC (Behavioral Breaking Change) in Amihaiemil's `Devops Comdor`.

```java
public class Node {
  ...
  // Input (string): colspan
  public String attr(String attributeKey) {
    Validate.notNull(attributeKey);

    if (attributes.hasKey(attributeKey)) {
        return attributes.get(attributeKey);
    } else if ... {
        ...
    }
    ...
  }
  // Output (string): 3
  // Mutant Output: null
  ...
}
```

Snp. 5.11 API showcasing a missed BeBC (Behavioral Breaking Change) in Chyxion's `Table To Xls`.

```
public class TableToXls { ...
    public static byte[] process(CharSequence html) {
        try {
            ByteArrayOutputStream baos = new(); ...
            for (Element table : Jsoup.parseBodyFragment(html).select("table")) {
                for (Element row : table.select("tr")) {
                    for (Element td : row.select("td, th")) { ...
                        int colSpan = 0;
                        String strColSpan = td.attr("colspan");
                        if (StringUtils.isNotBlank(strColSpan) &&
                                StringUtils.isNumeric(strColSpan)) { ...
                            colSpan = Integer.parseInt(strColSpan);
                        }
                        if ((colSpan > 1) && (rowSpan > 1)) {
                            ...
                        }
                    }
                }
            } ...
            return baos.toByteArray();
        } finally { ... }
    } ...
}
```

Snp. 5.12 Client code showcasing a missed BeBC (Behavioral Breaking Change) in Chyxion's `Table To Xls`.

```java
public class TestDriver {
    ...

    @Test
    public void run() throws Exception {
        StringBuilder html = new();
        Scanner s = new(getClass().getResourceAsStream("/sample.html"), "utf-8");

        ...

        FileOutputStream fout = new("target/data.xls");
        TableToXls.process(html, fout);

        ...
    }

    ...
}
```

Snp. 5.13 Client test showcasing a missed BeBC (Behavioral Breaking Change) in Chyxion's `Table To Xls`.

5.6 Related Work

In this section, we discuss related work on breaking change detection. Works exist towards the detection of SyBCs and BeBCs, especially in the context of the Android ecosystem and the Android API/ SDK. Works also exist towards the repair of some issues, but few target BeBCs, and are centered around a specific client use of a library dependency. In this section, we specifically review the existing tooling and studies, the extensive literature on BCs, and their applications towards preventing BeBCs on client–library pairs.

5.6.1 Regression Testing

Some approaches use client tests to detect BeBCs in third-party libraries, as part of regression testing. However, this may not be really effective due to the client tests being too weak for asserting changes in declared dependencies. We detail some of those regression testing-based approaches.

Jayasuriya et al. present a taxonomy of BeBCs, on their different types and reasons, as well as an analysis of the impact of dependency updates on client test suites [Jay+24]. In a large corpus of $8,086$ Maven artifacts with passing test suites, Jayasuriya et al. find that upgrading their dependencies to the latest available version triggers test failures in only 2.3% of cases [Jay+24]. Gyori et al. find that 26.9% of libraries that can be updated in client projects result in test failures [Gyo+18]. Hejderup and Gousios analyze 262 Java projects and find that client tests detect only 47% (resp. 35%) of artificial faults seeded in direct (resp. indirect) dependencies [HG22]. These empirical results suggest that existing client test suites are often too weak and ineffective in detecting BeBCs in libraries.

Regression tests can fail to detect BeBCs for two reasons: either they lack coverage and do not exercise the execution paths affected by BeBCs, or their assertions are too weak to observe the change. Indeed, for BeBCs to be detected, they must manifest at the API's boundary and propagate through client code up to a strong enough assertion that can observe the altered state. At each step, as the distance to change increases, lenient code can alter the observability of the change and reduce the likelihood of a failing assertion [NW19]. For instance, exception-swallowing code can prevent a newly raised exception from being detected by tests, or return values can be ignored. Subtle behavioral changes are particularly unlikely to be detected in this manner, leaving clients unaware of potential errors that may occur at run time. In Figure 5.2, although client tests accurately assert the behavior of the code relying on `StringTokenizer#getTokenList`, they do not detect the regression. Other parts of the client that would attempt to alter the returned list might break as a result, and the developers would not be informed before the issue manifests at run time.

Interestingly, Jayasuriya et al. note that, even when BeBCs do trigger test failures, the distance between the root cause of the error and the failing assertion severely hurts diagnosis and remediation [Jay+24]. To alleviate this issue, developers sometimes write tests specifically to detect changes and regressions in the behavior of the libraries they use. These tests directly

exercise the library's APIs—thus reducing distance and increasing observability—and focus on verifying the expected outputs and side effects. Regression tests are widespread for formatting or parsing libraries, but less common for other libraries. However, client tests are primarily concerned with validating their own logic, and such library-focused tests are relatively rare in practice.

5.6.2 Behavioral Breaking Change Detection Approaches

Only a subset of the API exposed by a library is typically used in a particular client [HG22; Har+22]. This subset, the syntactic footprint (Chapter 4), corresponds to the specific symbols of the API on which the client depends. BCs in these symbols can impact client code in diverse ways. Mostafa et al. study 296 BeBCs extracted from 68 version pairs of 15 popular Java libraries [MRW17]. They find that most BeBCs manifest as exceptions and return value changes, with side effects and environment effects rarely detected. Thus SyBCs affect the signature of these symbols and manifest as compilation or linking errors, while BeBCs affect their run-time behavior and manifest as changed program state and values, run-time exceptions, or crashes [MRW17; Rey+24; Jay+24]. SyBCs have been extensively studied in the literature [Och+22; Jay+25] and can be detected automatically with reasonable accuracy by static analysis tools [JD17; Lat+25]. Comparatively, BeBCs have received little attention and are particularly insidious as they cannot be reliably detected statically [Zha+22a].

Several approaches have attempted to detect BeBCs in libraries more reliably. They can be classified along three axes:

- Whether they consider the entire API of a library or only the subset used by a particular client
- Whether they exploit crowd-sourced knowledge from other clients or can be applied to a single client immediately after a release
- Whether they rely on test execution or static analysis of library changes.

The following 6 approaches can detect some types of BeBCs but are not primarily focused on BeBC detection and fail to capture the semantic contract exercised by the client, that is, the behavior the client expects from a series of invocations it makes.

Sun et al. introduce `JUnitTestGen`, a tool available on GitHub, that performs test augmentation by inferring Android SDK API symbol usage from applications statically, and generating test suites with randomized input values as parameters [Sun+22]. This approach is focused on the Android SDK API specifically, and does not capture the exact values used by the applications against it. The scope of the tool is on a subset of the Android API, specifically the part used by clients. It builds a crowd-sourced knowledge base of APIs being utilized, but does not leverage it for input and output values. As a result, the tool does not capture the semantic contract requirements the client imposes on the API and cannot be used to understand if, for the client's own requirements, a different version of the Android API level would be breaking. Despite this, the tool remains focused exclusively on testing the Android API and is not designed to detect BCs for a specific client. They, however, show that repair is also possible.

Zhao et al. present `RepairDroid`, a tool designed to perform program repair by leveraging hand-written templates that specify which usages should be fixed, and how [Zha+22b]. The tool extracts API usage to localize usage hotspots for repair at a specific API symbol.

Mezzetti et al. present an approach to detect type regressions causing BCs [MT19; MMT18]. Type regressions consist of changed types from one version of the API to the next in output or input values. In typed languages, this can occur when super types are used in the API and clients expect or cast to a different kind for their functionality. In non-typed languages, this can occur more freely; for example, changing the entire type from integer to string with no SyBCs, as the language lacks type information. The paper implements type regression testing for programming languages that lack type information (via, for example, annotations) like JavaScript in an already existing tool, named `noregretts`. The tool extracts the types used by the API and builds a model. It then compares this model with a newly generated one to ensure types remain consistent from one version to the next. This tool is library-centric and is not applied for a given client use of the API. The tool is available on GitHub. They evaluate its effectiveness on a dataset of Node.js libraries sourced from npm.

Mahmud et al. present `ACID`, a tool available on GitHub for detecting BCs in the Android SDK/API level [MCY23]. The tool leverages diffs of the APIs and checks the compatibility between those for invocations and callbacks. The diff generated leads to a list of suspicious methods identified to cause BCs potentially. The tool then checks for the usage of those APIs to alert on incompatibility issues for some specific API levels on clients. The tool is client-centric but is more oriented towards SyBCs. It also does not take into consideration the client usage first, and instead considers the entire API available.

Liu et al. present `AndroMevol`, a tool designed to harvest API incompatibility across different Android device manufacturers [Liu+24]. Due to customization done by the OEMs of Android devices, the APIs can vary from one device to the next, despite the Android "API level" value being identical. Furthermore, these APIs may feature OEM-specific additions that are lacking in another Android vendor's implementation. To address the issue brought about by customized frameworks, `AndroMevol` extracts differences between multiple vendors on both the SDK and the OEM additions. The tool is client-centric as it first extracts the symbols used by a specific client.

Wang et al. introduce Sensor, an automated testing technique to detect changes in behavior when dependencies conflict [Wan+22]. Conflicts with dependency versions can occur when multiple projects declare the same dependency but on different versions. The version picked depends on what the build system decides to include in the final software distribution and can have different behavior compared to the intended one for a specific component of the program. The approach first identifies conflicting pairs of dependencies (and their APIs), determines the difference between the two APIs, and generates tests to compare the behavior of both. In turn, a list of semantic conflict issues is given to the developer at the end. This approach can't be used for detecting behavioral changes in general and is more suited for conflicts with transitive dependency declarations. It, however, manages to highlight behavioral differences between conflicting declared versions of a given dependency by augmenting the test suite of the client to check for the behavior of both used APIs.

Sembid and DeBBI are two approaches aiming at detecting BeBCs introduced in new library versions.

Zhang et al. present Sembid, a tool that employs static analysis to compute semantic diffs between two API method versions and flag changes as potential BeBCs within libraries [Zha+22a]. It is library-centric, and leverages static analysis as well as runtime execution of the tests to detect BCs in one version of the library. The paper introduces an approach that statically infers semantic diffs between two versions of an API and checks whether existing inputs can effectively trigger them and propagate to clients. This allows identifying and filtering benign changes to avoid false positives. However, this approach is dependent on the client test suite's ability to detect such issues manifesting themselves on the API boundary with the library, as well as the change detected inside the library to manifest itself on the API boundary. In the case the change does not impact the usage of the client directly on the API boundary, the change cannot break for the client either. Because tests can be weak against library dependency changes, they may not assess whether or not a change is benign or impacting for the client functionality.

Chen et al. introduce DeBBI, a tool designed to detect BeBCs between two versions of a dependency [Che+20]. DeBBI works by analyzing the source code of the client to extract the used API symbols of the library. It then extracts the changes between the two versions of the library using static analysis techniques to build a knowledge base of symbols that were modified. Using the list of used symbols DeBBI is then able to alert the client on potential BeBCs by executing test cases of the client that are likely to detect the regressions.

DeBBI, compared to Sembid, adopts a crowd-sourced approach to aggregate the test suites of different clients using the same library [Che+20]. As executing the resulting tests can be costly, DeBBI uses information retrieval techniques to prioritize the execution of test cases that are more likely to detect regressions in a given upgrade. While these approaches are beneficial for library maintainers who wish to avoid breaking any of their clients, they do not account for the specificity of particular clients that may tolerate BeBCs that do not impact them.

Uppdatera and CompCheck are two approaches aiming at detecting BeBCs affecting a particular client.

Hejderup and Gousios introduce Uppdatera, a client-specific approach that classifies changes in the methods reached by a particular client as regressions by identifying suspicious changes in control flow and data flow using structural diffs between the two versions [HG22].

Zhu et al. introduce CompCheck, another client-specific approach that leverages crowd-sourced knowledge to generate incompatibility-revealing tests adapted from other clients using the library in a similar way [Zhu+23]. It is available on GitHub and enables detecting BeBCs for a specific client by using a knowledge base of compatibility issues crowd-sourced from other clients. Using static analysis, it attempts to match known compatibility issues with the knowledge base, and if found, tests the possibility of a BeBCs occurring. The tool requires mining knowledge before execution on other clients and does not enable discovering client unique issues. The tool provides a means for test augmentation using the knowledge base by generating a test case on a match of an issue present within the knowledge base.

While Uppdatera sometimes produces false positives since the diff matching rules are too strict, CompCheck is guaranteed to produce true positives only with an accompanying test case. However, it may still suffer from false negatives when the generated assertions are too weak and not sensitive enough to detect the difference.

5.7 Conclusion

In this chapter, we introduced a novel approach for detecting BeBCs in libraries by capturing API interaction snapshots during client test execution and comparing these snapshots across releases. This approach introduced interaction snapshots, capturing the interactions between the client and its library right at the API boundary. By being closer to the interface, we detected BeBCs reliably using client existing test suites as an entry point. Through a case study involving 2 real-world libraries (`Commons Lang 3` and `JSoup`) and clients of those libraries (21 and 6 respectively), we showed that Gilesi can identify multiple BeBCs missed by client test suites.

Interestingly, our approach addresses three challenges typically faced in the detection of BeBCs. First, it naturally captures reflective calls, as all method invocations are logged regardless of how they are dispatched. Second, it can detect changes in transitive dependencies as long as they propagate up to the API of a direct dependency to impact client code. More generally, it does not matter where the change is introduced as long as it propagates and can be observed at the API boundary manipulated by the client. Finally, snapshots explicitly document the exact behavior of the two library versions, making it easier to understand and debug BeBCs.

6 Conclusion and Perspectives

This chapter concludes the thesis. It first summarizes all our contributions, findings, and methodology and offers concluding remarks in Section 6.1. Lastly, it provides potential directions for future research based on the main findings presented in this thesis in Section 6.2.

Chapter Contents

6.1 Conclusion

Software libraries are an implementation of software modules relying on a library developer-defined API to define the functionality to be reused, and to govern the set of allowed uses and interactions between the client and the library. The API itself, despite its importance, is a source of friction between clients and libraries during software maintenance and evolution, due to the complexity and interplay of the language mechanisms that let a developer control which and how a symbol can be used. The evolution of the library being independent from the client work also further complicates friction between clients and libraries. BCs (Breaking Changes), in both semantic and behavioral forms affect clients.

In this thesis, we identified two main challenges to address gaps induced by the co-evolution of third-party libraries and clients. First, focusing on the library developer point of view, the modeling of uses permitted by the API alongside the set of interactions bound to each symbol is critical for library developers to understand not just coldspots or hotspots in their software (notably to guide design and evolution decision further), but also control the set of allowed uses and interactions as the library evolves, without causing BCs. Second, focusing on the client developer point of view, capturing the set of interactions and expectations between a client and a library via API interaction snapshots is important for detecting changes in behavior in library dependencies and thus combat potential future BeBCs that can go undetected by existing tooling and techniques, and ensuring program reliability and stability, all while reducing the cost associated with the maintenance of the project.

To tackle these challenges, we first introduced in Chapter 4 the notion of SUM (Syntactic Usage Model) and SUF (Syntactic Usage Footprint) to provide library developers a mean of accurately

modeling the permitted uses in their API and controlling which of these uses are used in practice in their documentation, samples, tests, or in clients. We evaluated SUM (Syntactic Usage Model) and SUF (Syntactic Usage Footprint) models on Java client–library pairs and found that our models expose uses likely unintended by library developers, and used in practice by clients, or even samples. We also showed that they can help identify under-tested and under-documented APIs.

Then in Chapter 5, we introduced API interaction snapshots for client–library compatibility testing to provide client developers with a mean to detect potential BeBCs (Behavioral Breaking Changes) occurring in their dependencies as part of a library update. We evaluated the effectiveness of our approach on client–library pairs and showed that our approach was more effective for detecting potential BeBCs simulated in library software compared to traditional CI (Continuous Integration) based and regression testing techniques.

Both of our contributions are useful for each side of the relation, library developers can get help designing their API or identifying hotspots and coldspots, or control their uses, spot lacking documentation, and catch misuses with UCov; client developers can prevent their software from being affected by BeBCs with Gilesi.

In conclusion, the contributions presented in this thesis address the two challenges presented in Chapter 1: the lack of understanding and modeling of possible API uses (by considering symbols and their interaction types); the capture of API interactions and detection of behavioral changes in underlying library dependencies for clients. UCov (Chapter 4) and Gilesi (Chapter 5) constitute implementations of our contributions for the Java programming language, applicable to client–library pairs. Our studies are accompanied by artifacts and the corresponding evaluation dataset, available freely on Zenodo [SMon+24a; SMon+25a]. Our implementations are available as open source projects written in Java on GitHub [SMon+][1][SMDF][2] for people to use, modify, or contribute to.

6.2 Perspectives

This thesis is a step towards the understanding of API usage, interactions, and compatibility testing in software ecosystems. Along the way, we identified two gaps in the library developer side and the client developer side, respectively, leading us to introduce novel modeling and compatibility testing techniques. Outside of our intended application of those, they can be further applied to other software engineering problems. In this section, we detail the long-term perspectives stemming from our contributions, going beyond understanding API usage and compatibility checking. We first outline the perspectives associated with our first contribution, focused on the library developer side: the API usage models defined in Chapter 4. We then outline the perspectives associated with our second contribution, focused on the client developer side: the interaction snapshots used for API compatibility testing in Chapter 5.

[1] https://github.com/alien-tools/ucov
[2] https://github.com/alien-tools/gilesi

6.2.1 Lightweight Syntactic API Usage Analysis

Using the SUMs and SUFs defined as part of Chapter 4, we can model the permitted uses in an API as well as the currently used interactions in consumers of the API. There are, however, multiple perspectives associated with the definition of SUMs and SUFs models.

First, to help library developers in creating their library, we could explore the idea of letting API designers define their own SUM as part of their design process. This SUM would then be used to verify, as the library is developed, that the API design aligns with the implementation of the library by comparing the designer-defined SUM with the extracted one during the development process. When unexpected uses and interactions are found, the library developer would be alerted and would prevent in time clients from consuming those unintended interactions. The SUM would therefore act as a software change contract guarding against any deviation from the designed API. This method would be more effective and expressive than what the programming language currently offers, enabling smarter verifications.

We can also expand the use of our models to existing downstream tasks. One task that could be explored is automatic documentation extraction. Using the SUM, we can deduce the full list of interactions permitted for a given symbol exported by the library's API. By leveraging consumers of the library, we can extract the SUFs from those clients, and then identify locations in their code where a given interaction type is used. We can then consider extracting a small portion of the client code to serve as a sample for that kind of usage. Continuing with every symbol of the API, this can then be used to build up documentation for the whole set of permitted uses in the API. This would, in fact, be akin to chrestomathy but for programming languages. It may also be possible to combine multiple symbol usages to build more advanced samples showcasing how to perform one task using multiple symbols and multiple interaction types offered by the API, like some approaches in the literature provide, albeit without considering the full scope of interactions permitted for a given symbol.

We can also enhance existing work designed to identify the most used symbols (hotspots) and the least used symbols to take into consideration their various interaction types, as represented in our models. Current work on symbol utilization statistics only considers the reference to a symbol. Analysis could be performed to understand the most used type of interaction by clients, or the most offered kind of interaction from the library. Wright's law was previously evaluated in a recent study by considering references to symbols. [Har+22] By leveraging our models, we could showcase whether the law remains true when taking into consideration the different kinds of interactions.

Furthermore, we can investigate the usefulness of the SUMs towards SyBCs detection. Notably, it is possible to generate models for two distinct versions of a library and analyze whenever or not interactions got removed, indicating a SyBCs.

6.2.2 Client–Library compatibility testing with API interaction snapshots

Using the interaction snapshots defined as part of Chapter 5, we can provide API compatibility testing between two versions of a library. There are, however, multiple perspectives associated with the use of interaction snapshots.

First, sticking with the initial goals and use case of interaction snapshots, towards API compatibility testing, further work needs to be completed in Gilesi. Gilesi should first be evaluated against finer-grained mutants to see how resilient the approach might be towards more subtle kinds of changes. Secondly, library's APIs can export fields. These fields can be read and written by clients in addition to the library itself. Let's consider the following example:

```java
public class Client {
    public void runTask() {
        CounterLib.value = 2;
        CounterLib.increment();

        if (CounterLib.value == 3) {
            System.out.println("Value is 3");
        } else {
            System.out.println("Value is not 3");
        }
    }
}
```

Snp. 6.1 Client code leveraging a field exported by the library's API.

```java
public class CounterLib {
    public static Integer value = 0;

    public static void increment() {
        value++;
    }
}
```

Snp. 6.2 Library code leveraging a field exported by the API (Version 1).

```java
public class CounterLib {
    public static Integer value = 0;

    public static void increment() {
        value--;
    }
}
```

Snp. 6.3 Library code leveraging a field exported by the API (Version 2).

In version 1 of the library (Snp. 6.2), the client (Snp. 6.1) gives the following output: "Value is 3". The library provides a counter functionality via the `CounterLib` class. It exports two symbols: a field named `value`, which stores the initial value of the counter–and a method named `increment`, which increases the counter and returns its value. The client first writes to the counter field to set the initial value and calls `increment` to increase it by one. It then retrieves the value of the initialization field `value`. However, using version 2 of the library (Snp. 6.3), the client (Snp. 6.1) gives the following output: "Value is not 3". The library was changed and now features a bug where the count decrements instead of incrementing the exported `value` field. The captured context around the `sum` method has not changed; the same returned value (void) is present in both versions of the library when the method is invoked. However, the value of the field has changed between those two versions. Such behavior change cannot currently be asserted by Gilesi. While exported fields are not a common practice in the Java programming language,

nothing prevents developers from exporting fields for the client to use. The above example showcases why capturing field writes (to capture the full set of client interactions) and field reads (to capture the behavior of the library) from the client is important to catch additional types of behavioral changes that could affect the client and remain to be studied and implemented. Field usage also constitutes another source of side effects that can occur in software. Even if the client does not make use of fields, monitoring exported static fields in objects can bring an additional layer of protection against changes in behaviors. The impact and usefulness of handling fields, as well as the extent to which field exports are widespread, require investigation using a dataset.

Snapshots can also be used to generate persistent regression tests, that is, concrete code snippets, written in the library's maintainer programming language and presented as unit tests in the project, which can be easily integrated into a library test suite generated from consumers making use of the API. The snapshots of Gilesi are JSON files, containing information about the API symbol invoked, with its context and input/output arguments. It is possible to turn this information into a Java code snippet that accurately reproduces the same series of invocations captured by the snapshot. The output arguments upon execution would be identical to the ones captured when executing the client. The invocation order and on which objects they're applied would also be preserved. The client behavior would thus be summarized into a dedicated code snippet. This would integrate into the existing developer's test workflow, and would not require using Gilesi anymore to compare the snapshots. Furthermore, as the library evolves and its behavior or syntax intentionally changes, the library developer can also maintain these regression tests to adjust for the new version of the library. It can also be a good lever for understanding how to migrate from one version to the next by training on real-world code sourced from clients, and integrating those migration steps into the documentation.

Furthermore, beyond BCs applications, we can leverage snapshots in code snippet generation techniques to provide them to developers, showcasing how the API may be used to achieve a specific goal. This kind of usage would be in line with various state-of-the-art approaches that try to infer usage samples from APIs automatically and could benefit from snapshots as a means to crowd-source usage data from real-world projects. Because snapshots contain the list of performed interactions from a client, including on which objects they're performed, for instance, invocations, it is possible to accurately reproduce code outside of the client project that reproduces the client behavior against that library. This would help generate valid code samples that are identical to the behavior of the analyzed client, to be used elsewhere. Because snapshots also contain information about the input and output values, we can also augment those snippets with not only sample input data showcasing a real-world use case of those API elements, but also their respective outcome, further enhancing the documentation to developers about what the API elements do.

Lastly, snapshots could be used to identify identical interactions within different software projects. Snapshots capture the interactions exercised by the client onto the library as well as the client expectations in terms of input/output values. These snapshots, therefore, abstract away the client code base only to retain the relation the client has against the library's API. Using said information, we can identify other projects maintained by another team of developers exercising

the same interactions as that client and highlight a frequently performed set of interactions. We could also identify specific programming patterns or styles by grouping similar interactions as part of a taxonomy of interactions. The comparison could be pushed further to identify if replacement libraries may not exist, implementing the same functionality the client asks for, and identifying the work required to migrate from one library to another using the data captured by the snapshots. One benefit of identifying a client that uses a library in the same manner is to find all clients that might be impacted by the same bug or vulnerability. This could also be used to automate the migration of dependencies from one vendor to another.

Bibliography

[Par72] David L. Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Communications of the ACM* 15.12 (Dec. 1972). Publisher: Association for Computing Machinery (ACM), pp. 1053–1058. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/361598.361623. URL: https://dl.acm.org/doi/10.1145/361598.361623 (cit. on pp. 2, 12, 23).

[Agr+89] Hiralal Agrawal, Richard A DeMillo, R_ Hathaway, et al. *Design of mutant operators for the C programming language*. Tech. rep. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue , 1989 (cit. on p. 119).

[TX08] 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*. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. L’Aquila, Italy: IEEE, Sept. 2008, pp. 327–336. DOI: 10.1109/ase.2008.43. URL: http://ieeexplore.ieee.org/document/4639336/ (cit. on pp. 4, 15, 35, 60, 82, 83).

[Rob09] Martin P. Robillard. “What Makes APIs Hard to Learn? Answers from Developers”. In: *IEEE Software* 26.6 (Nov. 2009). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 27–34. ISSN: 0740-7459. DOI: 10.1109/ms.2009.193. URL: http://ieeexplore.ieee.org/document/5287006/ (cit. on p. 59).

[Sty+09] 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)*. 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Corvallis, OR, USA: IEEE, Sept. 2009, pp. 119–126. DOI: 10.1109/vlhcc.2009.5295283. URL: http://ieeexplore.ieee.org/document/5295283/ (cit. on pp. 4, 15, 60, 82, 83).

[Zho+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. “MAPO: Mining and Recommending API Usage Patterns”. In: *Lecture Notes in Computer Science*. ISSN: 0302-9743, 1611-3349. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 318–343. ISBN: 978-3-642-03013-0. DOI: 10.1007/978-3-642-03013-0_15. URL: http://link.springer.com/10.1007/978-3-642-03013-0_15 (cit. on pp. 60, 84).

[Ngu+10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, et al. “A graph-based approach to API usage adaptation”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. SPLASH ’10: Systems Programming Languages and Applications: Software for Humanity. Reno/Tahoe Nevada USA: ACM, Oct. 17, 2010, pp. 302–321. DOI: 10.1145/1869459.1869486. URL: https://dl.acm.org/doi/10.1145/1869459.1869486 (cit. on pp. 1, 11, 24).

[LPS11] 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*. SAC’11: The 2011 ACM Symposium on Applied Computing. TaiChung Taiwan: ACM, Mar. 21, 2011, pp. 1317–1324. DOI: 10.1145/1982185.1982471. URL: https://dl.acm.org/doi/10.1145/1982185.1982471 (cit. on pp. 4, 15, 60, 83, 84).

[BW12] Raymond P. L. Buse and Westley Weimer. “Synthesizing API usage examples”. In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE). Zurich, Switzerland: IEEE, June 2012, pp. 782–792. DOI: 10.1109/icse.2012.6227140. URL: https://ieeexplore.ieee.org/document/6227140/ (cit. on pp. 84, 85).

[CW12] Jean-François Cossette and Robert J. Walker. “Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries”. In: *Proceedings of the 2012 ACM Conference on Software Engineering*. 2012, pp. 301–311. DOI: 10.1145/2337223.2337263. URL: https://doi.org/10.1145/2337223.2337263 (cit. on p. 26).

[DLP13] Coen De Roover, Ralf Lammel, and Ekaterina Pek. “Multi-dimensional exploration of API usage”. In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013 IEEE 21st International Conference on Program Comprehension (ICPC). San Francisco, CA, USA: IEEE, May 2013, pp. 152–161. DOI: 10.1109/icpc.2013.6613843. URL: http://ieeexplore.ieee.org/document/6613843/ (cit. on pp. 4, 15, 60, 83, 84).

[MRK13] 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*. 2013 IEEE International Conference on Software Maintenance (ICSM). Eindhoven, Netherlands: IEEE, Sept. 2013. DOI: 10.1109/icsm.2013.18. URL: http://ieeexplore.ieee.org/document/6676878/ (cit. on p. 86).

[Rob+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. “Automated API Property Inference Techniques”. In: *IEEE Transactions on Software Engineering* 39.5 (May 2013). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 613–637. ISSN: 0098-5589, 1939-3520. DOI: 10.1109/tse.2012.63. URL: http://ieeexplore.ieee.org/document/6311409/ (cit. on pp. 26, 84).

[Wan+13] Jue Wang, Yingnong Dang, Hongyu Zhang, et al. “Mining succinct and high-coverage API usage patterns from source code”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013 10th IEEE Working Conference on Mining Software Repositories (MSR 2013). San Francisco, CA, USA: IEEE, May 2013. DOI: 10.1109/msr.2013.6624045. URL: http://ieeexplore.ieee.org/document/6624045/ (cit. on pp. 60, 84, 85).

[KBM14] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. “Automatic mining of specifications from invocation traces and method invariants”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT/FSE’14: 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering. Hong Kong China: ACM, Nov. 11, 2014, pp. 178–189. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635890. URL: https://dl.acm.org/doi/10.1145/2635868.2635890 (cit. on p. 27).

[Lex+14] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. “UpSet: Visualization of Intersecting Sets”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 31, 2014). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 1983–1992. ISSN: 1077-2626. DOI: 10.1109/tvcg.2014.2346248. URL: http://ieeexplore.ieee.org/document/6876017/ (cit. on p. 78).

[BSV15] John Businge, Alexander Serebrenik, and Mark G. J. Van Den Brand. “Eclipse API usage: the good and the bad”. In: *Software Quality Journal* 23.1 (Mar. 2015). Publisher: Springer Science and Business Media LLC, pp. 107–141. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-013-9221-3. URL: http://link.springer.com/10.1007/s11219-013-9221-3 (cit. on p. 88).

[Cox+15] Joel Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. “Measuring Dependency Freshness in Software Systems”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). Florence, Italy: IEEE, May 2015. DOI: 10.1109/icse.2015.140. URL: http://ieeexplore.ieee.org/document/7202955/ (cit. on pp. 1, 11, 24).

[Hor+15] André Hora, Romain Robbes, Nicolas Anquetil, et al. “How Do Developers React to API Evolution? The Pharo Ecosystem Case”. In: *Empirical Software Engineering* 20.4 (2015), pp. 1114–1146. DOI: 10.1007/s11219-016-9344-4. URL: https://doi.org/10.1007/s11219-016-9344-4 (cit. on p. 26).

[NJW16] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. “Will My Tests Tell Me If I Break This Code?” In: *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. CSED ’16. event-place: Austin, Texas. New York, NY, USA: Association for Computing Machinery, 2016, pp. 23–29. ISBN: 978-1-4503-4157-8. DOI: 10.1145/2896941.2896944. URL: https://doi.org/10.1145/2896941.2896944 (cit. on pp. 111, 113).

[Paw+16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. “SPOON: A library for implementing analyses and transformations of Java source code”. In: *Software: Practice and Experience* 46.9 (Sept. 2016). Publisher: Wiley, pp. 1155–1179. ISSN: 0038-0644. DOI: 10.1002/spe.2346. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.2346 (cit. on p. 68).

[QLL16] Dong Qiu, Bixin Li, and Hareton Leung. “Understanding the API usage in Java”. In: *Information and Software Technology* 73 (May 2016). Publisher: Elsevier BV, pp. 81–100. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.01.011. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584916300027 (cit. on pp. 4, 15, 59, 60, 64, 67, 73, 83).

[JD17] Kamil Jezek and Jens Dietrich. “API Evolution and Compatibility: A Data Corpus and Tool Evaluation.” In: *The Journal of Object Technology* 16.4 (2017). Publisher: AITO - Association Internationale pour les Technologies Objets, 2:1. ISSN: 1660-1769. DOI: 10.5381/jot.2017.16.4.a2. URL: http://www.jot.fm/contents/issue_2017_04/article2.html (cit. on pp. 82, 88, 129).

[MRW17] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. “Experience paper: a study on behavioral backward incompatibilities of Java software libraries”. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’17: International Symposium on Software Testing and Analysis. Santa Barbara CA USA: ACM, July 10, 2017, pp. 215–225. DOI: 10.1145/3092703.3092721. URL: https://dl.acm.org/doi/10.1145/3092703.3092721 (cit. on pp. 93, 129).

[SB17] Anand Ashok Sawant and Alberto Bacchelli. “fine-GRAPE: fine-grained APi usage extractor – an approach and dataset to investigate API usage”. In: *Empirical Software Engineering* 22.3 (June 2017). Publisher: Springer Science and Business Media LLC, pp. 1348–1371. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-016-9444-6. URL: http://link.springer.com/10.1007/s10664-016-9444-6 (cit. on pp. 4, 15, 60, 64, 82, 83).

[Xav+17] 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)*. 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). Klagenfurt, Austria: IEEE, Feb. 2017. DOI: 10.1109/saner.2017.7884616. URL: http://ieeexplore.ieee.org/document/7884616/ (cit. on pp. 81, 86).

[Bri+18a] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. “APIDiff: Detecting API breaking changes”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). Campobasso: IEEE, Mar. 2018. DOI: 10.1109/saner.2018.8330249. URL: http://ieeexplore.ieee.org/document/8330249/ (cit. on pp. 47, 93).

[Bri+18b] 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)*. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). Campobasso: IEEE, Mar. 2018, pp. 255–265. DOI: 10.1109/saner.2018.8330214. URL: http://ieeexplore.ieee.org/document/8330214/ (cit. on pp. 86, 87).

[Foo+18] 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*. ESEC/FSE ’18: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista FL USA: ACM, Oct. 26, 2018, pp. 791–796. DOI: 10.1145/3236024.3275535. URL: https://dl.acm.org/doi/10.1145/3236024.3275535 (cit. on pp. 1, 12, 47, 87, 88, 93).

[Gyo+18] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. “Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem”. In: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE). Memphis, TN: IEEE, Oct. 2018, pp. 112–122. DOI: 10.1109/issre.2018.00022. URL: https://ieeexplore.ieee.org/document/8539074/ (cit. on pp. 5, 16, 93, 94, 128).

[Kul+18] 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”. In: *Information and Software Technology* 93 (Jan. 2018). Publisher: Elsevier BV, pp. 186–199. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2017.09.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584917304780 (cit. on p. 86).

[MMT18] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. “Type Regression Testing to Detect Breaking Changes in Node.js Libraries”. In: *LIPIcs, Volume 109, ECOOP 2018* 109 (2018). Ed. by Todd Millstein. Artwork Size: 24 pages, 757334 bytes ISBN: 9783959770798 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 7:1–7:24. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ECOOP.2018.7. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2018.7 (cit. on p. 130).

[Wan+18] Ying Wang, Ming Wen, Zhenwei Liu, et al. “Do the dependency conflicts in my project matter?” In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE ’18: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista FL USA: ACM, Oct. 26, 2018, pp. 319–330. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236056. URL: https://dl.acm.org/doi/10.1145/3236024.3236056 (cit. on p. 24).

[Zha+18] 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*. ICSE ’18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, May 27, 2018, pp. 886–896. DOI: 10.1145/3180155.3180260. URL: https://dl.acm.org/doi/10.1145/3180155.3180260 (cit. on p. 85).

[Bus+19] 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)*. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). Hangzhou, China: IEEE, Feb. 2019, pp. 117–127. DOI: 10.1109/saner.2019.8668018. URL: https://ieeexplore.ieee.org/document/8668018/ (cit. on p. 88).

[Cox19] Russ Cox. “Surviving software dependencies”. In: *Communications of the ACM* 62.9 (Aug. 21, 2019). Publisher: Association for Computing Machinery (ACM), pp. 36–43. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3347446. URL: https://dl.acm.org/doi/10.1145/3347446 (cit. on pp. 1, 11, 24).

[MT19] Anders Møller and Martin Toldam Torp. “Model-based testing of breaking changes in Node.js libraries”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE ’19: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn Estonia: ACM, Aug. 12, 2019, pp. 409–419. DOI: 10.1145/3338906.3338940. URL: https://dl.acm.org/doi/10.1145/3338906.3338940 (cit. on p. 130).

[Ngu+19a] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, et al. “Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Montreal, QC, Canada: IEEE, May 2019. DOI: 10.1109/icse.2019.00089. URL: https://ieeexplore.ieee.org/document/8812071/ (cit. on p. 87).

[Ngu+19b] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, et al. “FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Montreal, QC, Canada: IEEE, May 2019. DOI: 10.1109/icse.2019.00109. URL: https://ieeexplore.ieee.org/document/8812051/ (cit. on pp. 24, 84).

[NW19] Rainer Niedermayr and Stefan Wagner. “Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?” In: *Proceedings of the Evaluation and Assessment on Software Engineering*. EASE ’19: Evaluation and Assessment in Software Engineering. Copenhagen Denmark: ACM, Apr. 15, 2019, pp. 189–198. DOI: 10.1145/3319008.3319021. URL: https://dl.acm.org/doi/10.1145/3319008.3319021 (cit. on pp. 93, 95, 128).

[ZM19] Hao Zhong and Hong Mei. “An Empirical Study on API Usages”. In: *IEEE Transactions on Software Engineering* 45.4 (Apr. 1, 2019). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 319–334. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/tse.2017.2782280. URL: https://ieeexplore.ieee.org/document/8186224/ (cit. on p. 85).

[Bri+20] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. “You broke my code: understanding the motivations for breaking changes in APIs”. In: *Empirical Software Engineering* 25.2 (Mar. 2020). Publisher: Springer Science and Business Media LLC, pp. 1458–1492. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-019-09756-z. URL: http://link.springer.com/10.1007/s10664-019-09756-z (cit. on pp. 47, 93).

[Che+20] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. “Taming behavioral backward incompatibilities via cross-project testing and analysis”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20: 42nd International Conference on Software Engineering. Seoul South Korea: ACM, June 27, 2020, pp. 112–124. DOI: 10.1145/3377811.3380436. URL: https://dl.acm.org/doi/10.1145/3377811.3380436 (cit. on pp. 6, 17, 93, 131).

[Dan+20] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. “An approach and benchmark to detect behavioral changes of commits in continuous integration”. In: *Empirical Software Engineering* 25.4 (July 2020). Publisher: Springer Science and Business Media LLC, pp. 2379–2415. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-019-09794-7. URL: http://link.springer.com/10.1007/s10664-019-09794-7 (cit. on p. 54).

[KTD20] 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*. ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event USA: ACM, Nov. 8, 2020, pp. 629–641. DOI: 10.1145/3368089.3409725. URL: https://dl.acm.org/doi/10.1145/3368089.3409725 (cit. on p. 86).

[LDP20] Patrick Lam, Jens Dietrich, and David J. Pearce. “Putting the semantics into semantic versioning”. In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. SPLASH ’20: Conference on Systems, Programming, Languages, and Applications, Software for Humanity. Virtual USA: ACM, Nov. 18, 2020, pp. 157–179. DOI: 10.1145/3426428.3426922. URL: https://dl.acm.org/doi/10.1145/3426428.3426922 (cit. on p. 93).

[MNT20] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. “Detecting locations in JavaScript programs affected by breaking library changes”. In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 13, 2020). Publisher: Association for Computing Machinery (ACM), pp. 1–25. ISSN: 2475-1421. DOI: 10.1145/3428255. URL: https://dl.acm.org/doi/10.1145/3428255 (cit. on pp. 1, 12).

[UKR20] Gias Uddin, Foutse Khomh, and Chanchal K Roy. “Mining API usage scenarios from stack overflow”. In: *Information and Software Technology* 122 (June 2020). Publisher: Elsevier BV, p. 106277. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2020.106277. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584920300276 (cit. on p. 85).

[Bog+21] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. “When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems”. In: *ACM Transactions on Software Engineering and Methodology* 30.4 (Oct. 31, 2021). Publisher: Association for Computing Machinery (ACM), pp. 1–56. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3447245. URL: https://dl.acm.org/doi/10.1145/3447245 (cit. on pp. 53, 93).

[DSB21] Thomas Durieux, Cesar 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)*. 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). Madrid, Spain: IEEE, May 2021, pp. 545–549. DOI: 10.1109/msr52588.2021.00071. URL: https://ieeexplore.ieee.org/document/9463096/ (cit. on pp. 70, 110, 119).

[Gao+21] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, et al. “APIfix: output-oriented program synthesis for combating breaking changes in libraries”. In: *Proceedings of the ACM on Programming Languages* 5 (OOPSLA Oct. 20, 2021). Publisher: Association for Computing Machinery (ACM), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3485538. URL: https://dl.acm.org/doi/10.1145/3485538 (cit. on pp. 47, 87, 93).

[Gos+21] James Gosling, Bill Joy, Guy Steele, et al. *The Java language specification: Java SE 17 edition*. 2021 (cit. on p. 64).

[Ral21] Paul Ralph. “ACM SIGSOFT Empirical Standards Released”. In: *ACM SIGSOFT Software Engineering Notes* 46.1 (Jan. 29, 2021). Publisher: Association for Computing Machinery (ACM), pp. 19–19. ISSN: 0163-5948. DOI: 10.1145/3437479.3437483. URL: https://dl.acm.org/doi/10.1145/3437479.3437483 (cit. on p. 68).

[Har+22] Nicolas Harrand, Amine Benelallam, César Soto-Valero, et al. “API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages”. In: *Journal of Systems and Software* 184 (Feb. 2022). Publisher: Elsevier BV, p. 111134. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111134. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121221002314 (cit. on pp. 4, 15, 42, 45, 59, 60, 73, 74, 83, 87, 129, 135).

[HG22] Joseph Hejderup and Georgios Gousios. “Can we trust tests to automate dependency updates? A case study of Java Projects”. In: *Journal of Systems and Software* 183 (Jan. 2022). Publisher: Elsevier BV, p. 111097. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111097. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121221001941 (cit. on pp. 5, 6, 16, 17, 93, 94, 128, 129, 131).

[LGS22] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. “A Systematic Review of API Evolution Literature”. In: *ACM Computing Surveys* 54.8 (Nov. 30, 2022). Publisher: Association for Computing Machinery (ACM), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3470133. URL: https://dl.acm.org/doi/10.1145/3470133 (cit. on pp. 87, 93).

[ODF22] Lina Ochoa, Thomas Degueule, and Jean-Remy Falleri. “BreakBot: Analyzing the Impact of Breaking Changes to Assist Library Evolution”. In: *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). Pittsburgh, PA, USA: IEEE, May 2022, pp. 26–30. DOI: 10.1109/icse-nier55298.2022.9793524. URL: https://ieeexplore.ieee.org/document/9793524/ (cit. on pp. 55, 82, 94).

[Och+22] 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”. In: *Empirical Software Engineering* 27.3 (May 2022). Publisher: Springer Science and Business Media LLC. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-021-10052-y. URL: https://link.springer.com/10.1007/s10664-021-10052-y (cit. on pp. 2, 12, 26, 47, 59, 82, 86, 93, 129).

[Par+22] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. “A survey of flaky tests”. In: *ACM Transactions on Software Engineering and Methodology* 31.1 (Jan. 31, 2022), pp. 1–74. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3476105 (cit. on p. 109).

[SAD22] Igor Schittekat, Mehrdad Abdi, and Serge Demeyer. “Can We Increase the Test-coverage in Libraries using Dependent Projects’ Test-suites?” In: *The International Conference on Evaluation and Assessment in Software Engineering 2022*. EASE 2022: The International Conference on Evaluation and Assessment in Software Engineering 2022. Gothenburg Sweden: ACM, June 13, 2022, pp. 294–298. DOI: 10.1145/3530019.3535309. URL: https://dl.acm.org/doi/10.1145/3530019.3535309 (cit. on p. 93).

[Sun+22] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, et al. “Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22: 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester MI USA: ACM, Oct. 10, 2022, pp. 1–13. DOI: 10.1145/3551349.3561151. URL: https://dl.acm.org/doi/10.1145/3551349.3561151 (cit. on p. 129).

[Tiw+22] Deepika Tiwari, Long Zhang, Martin Monperrus, and Benoit Baudry. “Production Monitoring to Improve Test Suites”. In: *IEEE Transactions on Reliability* 71.3 (Sept. 2022). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 1381–1397. ISSN: 0018-9529, 1558-1721. DOI: 10.1109/tr.2021.3101318. URL: https://ieeexplore.ieee.org/document/9526340/ (cit. on p. 95).

[Wan+22] Ying Wang, Rongxin Wu, Chao Wang, et al. “Will Dependency Conflicts Affect My Program’s Semantics?” In: *IEEE Transactions on Software Engineering* 48.7 (July 1, 2022). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 2295–2316. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/tse.2021.3057767. URL: https://ieeexplore.ieee.org/document/9350237/ (cit. on p. 130).

[Wu+22] Jianyu Wu, Hao He, Wenxin Xiao, Kai Gao, and Minghui Zhou. “Demystifying software release note issues on GitHub”. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ICPC ’22: 30th International Conference on Program Comprehension. Virtual Event: ACM, May 16, 2022, pp. 602–613. ISBN: 978-1-4503-9298-3. DOI: 10.1145/3524610.3527919. URL: https://dl.acm.org/doi/10.1145/3524610.3527919 (cit. on pp. 5, 16, 53).

[Zai+22] Oleksandr Zaitsev, Stephane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. “How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community”. In: *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*. 2022 29th Asia-Pacific Software Engineering Conference (APSEC). Japan: IEEE, Dec. 2022, pp. 309–317. DOI: 10.1109/apsec57359.2022.00043. URL: https://ieeexplore.ieee.org/document/10043250/ (cit. on p. 87).

[Zha+22a] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, et al. “Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22: 37th IEEE/ACM International Conference on Automated Software Engineering. Rochester MI USA: ACM, Oct. 10, 2022, pp. 1–12. DOI: 10.1145/3551349.3556956. URL: https://dl.acm.org/doi/10.1145/3551349.3556956 (cit. on pp. 6, 17, 51, 93, 129, 131).

[Zha+22b] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. “Towards automatically repairing compatibility issues in published Android apps”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22: 44th International Conference on Software Engineering. Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 2142–2153. DOI: 10.1145/3510003.3510128. URL: https://dl.acm.org/doi/10.1145/3510003.3510128 (cit. on p. 130).

[GRV23] Victor Pezzi Gazzinelli Cruz, Henrique Rocha, and Marco Tulio Valente. “Snapshot testing in practice: Benefits and drawbacks”. In: *Journal of Systems and Software* 204 (Oct. 2023). Publisher: Elsevier BV, p. 111797. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111797. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121223001929 (cit. on p. 95).

[He+23] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. “Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot”. In: *IEEE Transactions on Software Engineering* 49.8 (Aug. 2023). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 4004–4022. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/tse.2023.3278129. URL: https://ieeexplore.ieee.org/document/10130588/ (cit. on pp. 53, 94).

[Lad+23] Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, and Olivier Barais. “Journey to the Center of Software Supply Chain Attacks”. In: *IEEE Security & Privacy* 21.6 (Nov. 2023). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 34–49. ISSN: 1540-7993, 1558-4046. DOI: 10.1109/msec.2023.3302066. URL: https://ieeexplore.ieee.org/document/10224821/ (cit. on p. 93).

[Liu+23] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. “More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle WA USA: ACM, July 12, 2023, pp. 664–676. DOI: 10.1145/3597926.3598086. URL: https://dl.acm.org/doi/10.1145/3597926.3598086 (cit. on pp. 93, 94).

[MCY23] Tarek Mahmud, Meiru Che, and Guowei Yang. “Detecting Android API Compatibility Issues With API Differences”. In: *IEEE Transactions on Software Engineering* 49.7 (July 2023). Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 3857–3871. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/tse.2023.3274153. URL: https://ieeexplore.ieee.org/document/10121651/ (cit. on p. 130).

[Nav+23] Nacho Navarro, Salwa Alamir, Petr Babkin, and Sameena Shah. “An Automated Code Update Tool For Python Packages”. In: *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME). Bogotá, Colombia: IEEE, Oct. 1, 2023, pp. 536–540. DOI: 10.1109/icsme58846.2023.00068. URL: https://ieeexplore.ieee.org/document/10336306/ (cit. on pp. 47, 93).

[Ven+23] 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”. In: *ACM Transactions on Software Engineering and Methodology* 32.4 (Oct. 31, 2023). Publisher: Association for Computing Machinery (ACM), pp. 1–26. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3576037. URL: https://dl.acm.org/doi/10.1145/3576037 (cit. on p. 87).

[Wu+23] Yulun Wu, Zeliang Yu, Ming Wen, et al. “Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem”. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE, May 2023, pp. 1046–1058. DOI: 10.1109/icse48619.2023.00095. URL: https://ieeexplore.ieee.org/document/10172868/ (cit. on p. 93).

[Zhu+23] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. “Client-Specific Upgrade Compatibility Checking via Knowledge-Guided Discovery”. In: *ACM Transactions on Software Engineering and Methodology* 32.4 (Oct. 31, 2023). Publisher: Association for Computing Machinery (ACM), pp. 1–31. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3582569. URL: https://dl.acm.org/doi/10.1145/3582569 (cit. on pp. 6, 17, 93, 131).

[Jai+24] Damien Jaime, Pascal Poizat, Joyce El Haddad, and Thomas Degueule. “Balancing the Quality and Cost of Updating Dependencies”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’24: 39th IEEE/ACM International Conference on Automated Software Engineering. Sacramento CA USA: ACM, Oct. 27, 2024, pp. 1834–1845. ISBN: 979-8-4007-1248-7. DOI: 10.1145/3691620.3695595. URL: https://dl.acm.org/doi/10.1145/3691620.3695595 (cit. on pp. 5, 16, 53).

[Jay+24] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. “Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications”. In: *Proceedings of the ACM on Software Engineering* 1 (FSE July 12, 2024). Publisher: Association for Computing Machinery (ACM), pp. 1238–1261. ISSN: 2994-970X. DOI: 10.1145/3643782. URL: https://dl.acm.org/doi/10.1145/3643782 (cit. on pp. 2, 5, 12, 16, 93, 94, 128, 129).

[Liu+24] Pei Liu, Yanjie Zhao, Mattia Fazzini, et al. “Automatically Detecting Incompatible Android APIs”. In: *ACM Transactions on Software Engineering and Methodology* 33.1 (Jan. 31, 2024). Publisher: Association for Computing Machinery (ACM), pp. 1–33. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3624737. URL: https://dl.acm.org/doi/10.1145/3624737 (cit. on p. 130).

[Rey+24] Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. “BUMP: A Benchmark of Reproducible Breaking Dependency Updates”. In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Rovaniemi, Finland: IEEE, Mar. 12, 2024, pp. 159–170. DOI: 10.1109/saner60148.2024.00024. URL: https://ieeexplore.ieee.org/document/10589737/ (cit. on pp. 119, 129).

[RCH24] Benjamin Rombaut, Filipe R. Cogo, and Ahmed E. Hassan. *Leveraging the Crowd for Dependency Management: An Empirical Study on the Dependabot Compatibility Score*. Version Number: 1. 2024. DOI: 10.48550/ARXIV.2403.09012. URL: https://arxiv.org/abs/2403.09012 (cit. on p. 94).

[ZM24] Hao Zhong and Na Meng. “Compiler-directed Migrating API Callsite of Client Code”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24: IEEE/ACM 46th International Conference on Software Engineering. Lisbon Portugal: ACM, Apr. 12, 2024, pp. 1–12. DOI: 10.1145/3597503.3639084. URL: https://dl.acm.org/doi/10.1145/3597503.3639084 (cit. on pp. 47, 93).

[HVK25] Hao He, Bogdan Vasilescu, and Christian Kästner. “Pinning is futile: you need more than local dependency versioning to defend against supply chain attacks”. In: *Proceedings of the ACM on Software Engineering* 2 (FSE June 19, 2025). Citations: 2 (Crossref) [2026-01-15] Citations: 8 (SemanticScholar) [2026-01-15], pp. 266–289. ISSN: 2994-970X. DOI: 10.1145/3715728. URL: https://dl.acm.org/doi/10.1145/3715728 (cit. on pp. 5, 16).

[Jay+25] Dhanushka Jayasuriya, Samuel Ou, Saakshi Hegde, et al. “An extended study of syntactic breaking changes in the wild”. In: *Empirical Software Engineering* 30.2 (Mar. 2025). Publisher: Springer Science and Business Media LLC. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-024-10563-4. URL: https://link.springer.com/10.1007/s10664-024-10563-4 (cit. on pp. 47, 93, 129).

[Lat+25] Corentin Latappy, Thomas Degueule, Jean-Rémy Falleri, Romain Robbes, and Lina Ochoa. “Roseau: Fast, Accurate, Source-based Breaking Change Analysis in Java”. In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME). Auckland, New Zealand: IEEE, 2025 (cit. on pp. 47, 93, 129).

Books

[BBlo08] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. 2nd ed. USA: Prentice Hall PTR, 2008. ISBN: 0321356683 (cit. on pp. 4, 14, 27, 59).

PhD Thesis

[TZim19] Théo Zimmermann. “Challenges in the collaborative evolution of a proof language and its ecosystem”. Université Paris Cité, Dec. 2019. URL: https://inria.hal.science/tel-02451322 (cit. on p. 94).

Webpages

[@Fow05] Martin Fowler. *Fluent Interface*. 2005. URL: https://www.martinfowler.com/bliki/FluentInterface.html (cit. on p. 69).

[@Wri17] Hyrum Wright. *Hyrum's Law*. 2017. URL: https://www.hyrumslaw.com/ (cit. on pp. 42, 74, 135).

[@Apaa] Apache. *About Apache Commons CLI*. URL: https://commons.apache.org/proper/commons-cli/ (cit. on p. 60).

[@Apab] Apache. *Commit 2d1ab7 - Commons TEXT Codebase on Github*. URL: https://github.com/apache/commons-text/commit/2d1ab7ea72298949900df47f65b4f71d56411f0b (cit. on p. 51).

[@Apac] Apache. *commons-text TEXT-219 issue on Apache's Jira bug tracket*. URL: https://issues.apache.org/jira/browse/TEXT-219 (cit. on p. 51).

[@Apad] Apache. *Documentation of Apache Commons CLI*. URL: https://commons.apache.org/proper/commons-cli/usage.html (cit. on pp. 70, 72, 80).

[@Apae] Apache. *Welcome to Apache Maven*. URL: https://maven.apache.org (cit. on p. 32).

[@Git] Github. *Understanding your software supply chain, about the dependency graph*. URL: https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph (cit. on p. 73).

[@JSoa] Authors of JSoup. *About JSoup*. URL: https://jsoup.org/ (cit. on p. 60).

[@JSob] Authors of JSoup. *Documentation of JSoup*. URL: https://jsoup.org/cookbook/ (cit. on p. 70).

[@JSoc] Authors of JSoup. *Documentation of JSoup - Load document from url*. URL: https://jsoup.org/cookbook/input/load-document-from-url (cit. on p. 72).

[@JSod] Authors of JSoup. *HtmlToPlainText.java - JSoup Codebase on Github*. URL: https://github.com/jhy/jsoup/blob/84fd4376699240105 7b73f740acec4b82f1e3dd6/src/main/java/org/jsoup/examples/HtmlToPlainText.java (cit. on p. 77).

[@Lê] Nhat Minh Lê. *Cofoja on Github*. URL: https://github.com/nhatminhle/cofoja (cit. on p. 27).

[@per] perwendel. *Spark template engines on Github*. URL: https://github.com/perwendel/spark-template-engines (cit. on p. 70).

[@Pre] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: https://semver.org/ (cit. on pp. 33, 55).

[@Spaa] Authors of Spark. *About Spark*. URL: https://sparkjava.com/ (cit. on p. 60).

[@Spab] Authors of Spark. *Application Structure tutorial for Spark*. URL: https://sparkjava.com/tutorials/application-structure (cit. on p. 70).

[@Spac] Authors of Spark. *Documentation of Spark*. URL: https://sparkjava.com/documentation (cit. on p. 70).

[@Spad] Authors of Spark. *Documentation of Spark - Routes*. URL: https://sparkjava.com/documentation%5C#routes (cit. on p. 72).

[@Spae] Authors of Spark. *Tutorials for Spark*. URL: https://sparkjava.com/tutorials (cit. on p. 70).

[@tip] tipsy. *Spark basic structure on Github*. URL: https://github.com/tipsy/spark-basic-structure (cit. on p. 70).

Own Work

[SMon+24a] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. *Artifacts for "Lightweight Syntactic API Usage Analysis with UCov"*. Version 1.0.0. Jan. 26, 2024. DOI: 10.5281/ZENODO.10571867. URL: https://zenodo.org/doi/10.5281/zenodo.10571867 (cit. on pp. 7, 18, 58, 60, 64, 68, 134).

[SMon+24b] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri. "Lightweight Syntactic API Usage Analysis with UCov". In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. ICPC '24: 32nd IEEE/ACM International Conference on Program Comprehension. Lisbon Portugal: ACM, Apr. 15, 2024, pp. 426–437. DOI: 10.1145/3643916.3644415. URL: https://dl.acm.org/doi/10.1145/3643916.3644415 (cit. on pp. 7, 18, 58).

[SMon+25a] Gustave Monce, Thomas Degueule, Jean-Rémy Falleri, and Romain Robbes. *Artifacts for "Client–Library Compatibility Testing with API Interaction Snapshots"*. Version 1.0.0. July 24, 2025. DOI: 10.5281/ZENODO.16411966. URL: https://zenodo.org/doi/10.5281/zenodo.16411966 (cit. on pp. 8, 19, 20, 92, 101, 109, 134).

[SMon+25b] Gustave Monce, Thomas Degueule, Jean-Rémy Falleri, and Romain Robbes. "Client–Library Compatibility Testing with API Interaction Snapshots". In: *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME). Auckland, New Zealand: IEEE, 2025 (cit. on pp. 8, 19, 92).

[SMon+] [SW] Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, and Jean-Rémy Falleri, *UCov, Alien-Tools – GitHub*. URL: https://github.com/alien-tools/ucov (cit. on pp. 7, 18, 19, 58, 64, 134).

[SMDF] [SW] Gustave Monce, Thomas Degueule, and Jean-Rémy Falleri, *Gilesi, Alien-Tools – GitHub*. URL: https://github.com/alien-tools/gilesi (cit. on pp. 8, 19, 20, 92, 93, 134).

Glossary

classical — Classical programming style *used on pp.* 62, 63, 69, 70, 72, 76, 110, 157

fluent — Fluent programming style *used on pp.* 3, 14, 59, 69, 70, 72, 76, 110, 157

framework — Framework programming style *used on pp.* 3, 14, 62, 63, 69, 70, 72, 76, 81, 91, 103, 157

java — The Java object oriented programming language *used on pp.* 1, 2, 4, 6–8, 11, 12, 14, 15, 18, 19, 23–27, 31–33, 35, 38, 40, 42, 47, 57, 59–62, 64–66, 68, 70, 76, 80, 81, 83, 86, 88, 89, 91, 93, 101, 102, 104, 106, 109–111, 115, 128, 129, 134, 137, 138, 155, 156

symbol — Symbol (or element) of an API *used on pp.* 2–4, 26, 27, 29, 31, 40–42, 45, 47, 49, 54, 59–65, 67, 68, 73–77, 80–86, 88, 89, 100, 102–104, 106, 109, 110, 115, 116, 118, 129–131, 133–135, 137, 138, 157

symbole — Symbole (ou élément) d'une API *used on pp.* 12–15

Acronyms

ABI — Application Binary Interface *used on pp.* 2, 49

ABI — Interface Binaire Applicative *used on p.* 12

API — Application Programming Interface *used on pp.* 2–9, 23–27, 29–31, 35, 41–43, 45, 47, 51, 52, 54, 55, 57, 59–69, 72–77, 79–89, 91, 93–106, 108–111, 115, 116, 118, 119, 122, 125, 128–138, 155–158, 160

API — Interface de Programmation Applicative *used on pp.* 12–19, 21

AST — Abstract Syntax Tree *used on pp.* 4, 15, 60, 68, 83, 84, 87

BC — Breaking Change *used on pp.* 2, 3, 5, 12–15, 23, 35, 46, 47, 49, 51, 53–55, 81, 82, 86–88, 93, 94, 128–131, 133, 138, 160

BeBC — Behavioral Breaking Change *used on pp.* 5–8, 16, 17, 19, 46, 51–53, 91, 93–102, 109, 111, 112, 117, 119–129, 131–134, 156–158

Bib — Bibliothèque *used on p.* 21

bib — bibliothèque *used on pp.* 11–19, 21, 155

CD — Continuous Development *used on p.* 54

CI — Continuous Integration *used on pp.* 53, 54, 94, 109, 134

Cli — Client *used on pp.* 5, 9, 21, 25, 34, 48, 50, 52, 53, 63, 69, 71, 75, 87, 93, 95, 97, 109, 115, 116, 119, 120, 123, 124, 126, 127, 137, 156–158

cli — client *used on pp.* 1–9, 11–19, 21, 26, 32, 34–36, 38, 39, 42, 45–47, 49, 51–57, 59–62, 64–66, 68–70, 73–89, 91, 93–104, 106, 109–112, 115–121, 128–139, 155–157, 160

GAV — Group, Artifact, Version *used on pp.* 33, 35, 155

HTML — HyperText Markup Language *used on pp.* 60, 68, 77, 121

IDE Integrated Development Environment *used on pp.* 42, 81, 84
IoC Inversion of Control *used on pp.* 3, 14, 28, 31, 59, 62, 69, 81, 102, 103

JAR Java ARchives *used on pp.* 31–34, 42, 49
JDK Java Development Kit *used on pp.* 61, 103, 104
JPMS Java Platform Module System *used on p.* 26
JRE Java Runtime Environment *used on p.* 40
JSON JavaScript Object Notation *used on pp.* 24, 25, 138
JVM Java Virtual Machine *used on pp.* 31, 47, 106, 110

Lib Library *used on pp.* 3, 9, 23, 24, 26, 30, 31, 36, 38, 42, 48, 50, 53, 54, 93, 95, 137, 156–158, 160
lib library *used on pp.* 1–9, 23–36, 38, 40–42, 44–50, 53–57, 59–62, 64, 65, 68–74, 76, 77, 79–89, 91, 93–96, 98–103, 106, 109–112, 115, 119–121, 128–139, 155–158, 160

NIER New Ideas and Emerging Results *used on pp.* 8, 19

POM Package Object Model *used on pp.* 33, 34, 37, 41, 42, 157

Q&A Questions and Answers *used on p.* 85

RDCT Reverse-Dependency Compatibility Testing *used on pp.* 55, 94

SDK Software Development Kit *used on pp.* 128–130
SemVer Semantic Versioning *used on pp.* 33, 55, 156
SUF Syntactic Usage Footprint *used on pp.* 57, 59, 61–65, 68, 73, 75–82, 102, 133–135, 155, 156
SUM Syntactic Usage Model *used on pp.* 57, 59, 61–65, 68, 73, 75, 76, 79, 81, 82, 85, 88, 133–135, 155, 156
SyBC Syntactic Breaking Change *used on pp.* 5, 16, 46–51, 86, 87, 93, 94, 128–130, 135, 156

XLS eXceL Spreadsheet *used on p.* 121
XML eXtensible Markup Language *used on pp.* 34, 35

Listings

List of Tables

List of Figures

List of Code Snippets

The End.

Diminuer les frictions entre les utilisateurs et les mainteneurs de bibliothèques logicielles

Résumé : Les bibliothèques logicielles sont largement utilisées dans le développement logiciel moderne par les développeurs afin d'intégrer dans une application des fonctionnalités déjà implémentées par d'autres développeurs. Une telle dépendance à du code tiers pose des problèmes de maintenance, de compatibilité et d'évolution, tant pour les développeurs de clients que pour ceux de bibliothèques. Dans cette thèse, nous analysons la relation et les frontières qui existent entre les bibliothèques et les clients. Une particularité des applications utilisant des bibliothèques est la présence et la consommation d'interfaces de programmation d'applications (APIs). Ces APIs constituent la principale frontière d'interaction que le développeur utilise pour consommer une fonctionnalité spécifique d'une bibliothèque. Cette frontière permet le transfert de données dans les deux sens, du client vers la bibliothèque, et de la bibliothèque vers le client. Les développeurs de bibliothèques manquent souvent de connaissances sur les usages réels de leurs bibliothèques, ce qui empêche une évolution correcte et sûre. À l'inverse, les développeurs de clients rencontrent souvent des difficultés à trouver un équilibre entre la mise à jour vers la dernière version de leurs dépendances et le maintien de la compatibilité sans casser leur code. En effet, la mise à jour des dépendances expose les clients au risque de changements cassants (BCs) provenant de leurs dépendances. Bien que certains travaux existent déjà sur certains types de problèmes de compatibilité, d'autres restent inexplorés. Nous proposons, en nous appuyant sur l'existence de telles frontières, de nouvelles méthodes pour aider les bibliothèques à comprendre leurs usages et les clients à comprendre les changements de comportement auxquels ils sont confrontés.

Mots-clés : Bibliothèques logicielles, Évolution logicielle, Génie logiciel

---

Reducing friction between clients and maintainers of software libraries

Abstract: Libraries are heavily used in modern software development by developers to bring in functionality already implemented by other developers into an application. Such dependence on third-party code brings issues regarding maintenance, compatibility, and evolution for both the client and library software developer. In this thesis, we analyze the relation and frontiers that exist between libraries and client software. A particularity of applications using libraries is the presence and consumption of APIs (Application Programming Interfaces). These APIs are the main interaction frontier the developer uses to consume a specific library feature. This frontier allows for data to be transferred in both directions from the client to the library, and from the library to the client. Library developers often lack usage knowledge on the uses of their libraries, preventing proper and safe evolution. In contrast, client developers often face difficulties finding a balance between updating to the latest version of their dependencies and retaining compatibility without breaking their code. Indeed, updating dependencies exposes clients to the risk of BCs (Breaking Changes) coming from their dependencies. While some work already exists on some types of compatibility issues, others remain unexplored. We propose, based on the existence of such frontiers, new methods to help libraries understand their uses, and clients to understand the behavioral changes they face.

Keywords: Libraries, Software evolution, Software engineering

---

LaBRI - Laboratoire bordelais de recherche en informatique
UMR 5800, 351, cours de la Libération F-33405 Talence cedex, France.