Project Number 732223

D2.7 Framework and API Analysis - Final Report

Version 1.3
29 June 2019
Final

Public Distribution

Centrum Wiskunde & Informatica (CWI)

Project Partners: **Athens University of Economics & Business**, **Bitergia**, **Castalia Solutions**, **Centrum Wiskunde & Informatica**, **Eclipse Foundation Europe**, **Edge Hill University**, **FrontEndART**, **OW2**, **SOFTEAM**, **The Open Group**, **University of L'Aquila**, **University of York**, **Unparallel Innovation**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the CROSSMINER Project Partners accept no liability for any error or omission in the same.

© 2019 Copyright in this document remains vested in the CROSSMINER Project Partners.

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L'Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	Initial outline	16 October 2018
0.2	API migration part	4 December 2018
0.3	First internal release	14 December 2018
1.0	Final release initial report	28 December 2018
1.1	Full update on MARACAS	29 May 2019
1.2	Release for internal review	24 June 2019
1.3	Final version	29 June 2019

Table of Contents

Executive Summary

This document reports on the final results obtained for **Task 2.3** and **Task 2.4**:

Task 2.3: Dependency Analysis This task will enable the inclusion of domain-specific knowledge about the API and its semantics of often used software frameworks, in particular at least for OSGi and the Eclipse plugin model. Since these frameworks are typically implemented using Java Reflection, static analysis will lead to inaccurate and therefore unusable results. The task is to improve the accuracy of facts extracted from client code written against these frameworks to a level where we can actually produce actionable results;

Task 2.4: API Analysis This task will produce analyses to identify API and API usage information. The goal is to learn from existing projects about typical usage to be able to concretely advise software engineers about the use of open-source software components. The resulting analyses should be amenable to bespoke extensions.

In the previous deliverables **D2.3 – Dependency Inference and Analysis – Final Report** and **D2.4 – Dependency Inference Components**, we have extensively addressed **Task 2.3: Dependency Analysis** by incorporating domain-specific knowledge about OSGi and Maven in the analysis of dependencies extracted from source code and project meta-data. In particular, we have shown how the singular use of OSGi within the Eclipse ecosystem, as exemplified by the Eclipse plug-in model, impacts the way dependencies are declared and managed. The metric providers we have developed in this context take this domain-specific knowledge into account to produce accurate results. A last update on the software produced in the context of **Task 2.3: Dependency Analysis** is available in **D2.5 – Dependency Analysis Components**.

In this deliverable, we focus specifically on **Task 2.4: API Analysis** and complete the initial contributions presented in **D2.6 – Framework and API Analysis – Initial Progress Report**. Driven by the specific needs of use cases, e.g., as described for FrontEndArt and Softeam in **D1.1 – Project Requirements** and **D1.2 – Evaluation Plan**, we put a particular emphasis on two main challenges related to API analysis, described below.

The first challenge is the recommendation of *API Function Calls and Usage Patterns*, which assists developers in understanding and using complex APIs, and is addressed in Part I. The approach we present emerges from a close collaboration with **WP6** and is extracted from a common paper that has ben published and presented at the *41st International Conference on Software Engineering* [64]:

FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns.
Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, Massimiliano Di Penta.
In *Proceedings of the 41st International Conference on Software Engineering* (ICSE 2019).

The second challenge is the support for *API Evolution and Migration*: how to keep up with the evolution of a library when its public API changes to accommodate new features or refactorings. We focus on this challenge in Part II. We present an extensive state of the art of the current approaches in the literature and in practice, focusing on the assumptions and limitations of each. Then, we introduce MARACAS, a new framework for automatic API migration which focuses on extensibility and integration with CROSSMINER through the use of M^3 models. We conduct several use cases and report on the ability of MARACAS to study API evolution and to support API migration. We refer the reader to the companion deliverable **D2.8 – API Analysis Components** for more information about the concrete CROSSMINER components we developed, and how they integrate with the CROSSMINER platform and IDE. Finally, we conclude by highlighting the alignment of the contributions and tools presented here with the project requirements in Part III.

*All the contributions introduced here, along with the analysis components presented in **D2.8** were created from scratch specifically for CROSSMINER and were not part of the previous OSSMETER project and platform.*

Part I

API Function Calls and Usage Patterns Recommendation

1 Introduction

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails the use of external libraries. Rather than implementing new systems from scratch, developers look for, and try to integrate into their projects, libraries that provide functionalities of interest. Libraries expose their functionality through Application Programming Interfaces (APIs) which govern the interaction between a client project and the libraries it uses.

Developers therefore often face the need to learn new APIs. The knowledge needed to manipulate an API can be extracted from various sources: the API source code itself, the official website and documentation, Q&A websites such as StackOverflow, forums and mailing lists, bug trackers, other projects using the same API, etc. However, an official documentation often merely reports the API description without providing non-trivial example usages. Besides, querying informal sources such as StackOverflow might become time-consuming and error-prone [81]. Also, API documentation may be ambiguous, incomplete, or erroneous [103], while API examples found on Q&A websites may be of poor quality [59].

Over the past decade, the problem of API learning has garnered considerable interest from the research community. Several techniques have been developed to automate the extraction of API *usage patterns* [82] in order to reduce developers' burden when manually searching these sources and to provide them with high-quality code examples. However, these techniques, based on clustering [68, 105, 114] or predictive modeling [34], still suffer from high redundancy [34] and—as we show later—poor run-time performance.

To cope with these limitations, we propose a new approach for API usage patterns mining that builds upon concepts emerging from collaborative-filtering recommender systems [87]. The fundamental idea of these systems is to recommend to users items that have been bought by similar users in similar contexts. By considering API methods as products and client code as customers, we reformulate the problem of usage pattern recommendation in terms of a collaborative-filtering recommender system. Informally, the question the proposed system can answer is:

> *"Which API methods should this piece of client code invoke, considering that it has already invoked these other API methods?"*

Implementing a collaborative-filtering recommender system requires to assess the similarity of two customers, i.e., two projects. Existing approaches assume that any two projects using an API of interest are equally valuable sources of knowledge. Instead, we postulate that not all projects are equal when it comes to recommending usage patterns: a project that is highly similar to the project currently being developed should provide higher quality patterns than a highly dissimilar one. Our recommender system attempts to narrow down the search scope by considering only the projects that are the most similar to the active project. Thus, methods that are typically used conjointly by similar projects in similar contexts tend to be recommended first.

We incorporate these ideas into a recommender system that mines OSS repositories to provide developers with API *FunctiOn Calls and USage patterns*: FOCUS. Our approach represents mutual relationships between projects using a 3D matrix and mines API usage from the most similar projects.

We evaluated FOCUS on different datasets comprising 610 Java projects from GitHub and 3,600 JAR archives from the Maven Central Repository. In the evaluation, we simulate different stages of a development process, by removing portions of client code and assessing how FOCUS can recommend snippets with API invocations to complete them. We find that FOCUS outperforms PAM, a state-of-the-art tool for API usage patterns mining [34], with regards to success rate, accuracy, and execution time.

This deliverable part is organized as follows. Section 2 introduces a motivating example and background notions. Our recommender system for API mining, FOCUS, is introduced in Section 3. The evaluation is presented in Section 4, with the key results being analyzed in Section 5. Section 6 discusses the threats to validity. In Section 7, we present related work and conclude in Section 8.

2 Background

This section presents a motivating example and the main components of the proposed solution. Then, we introduce the main notions underpinning our approach, mostly originating from Schafer et al. [88] and Chen [17].

2.1 Motivating Example

The typical setting considered in this document is as shown in Figure 1: a developer is implementing some method to satisfy the requirements of the system being developed. In the specific case shown in Figure 1 (b), the `findBoekrekeningen` method queries the available entities and retrieve those of type `Boekrekening`. To this end, the `Criteria API` library[1] is used as it provides useful interfaces for querying system entities according to defined criteria.

Figure 1 (a) depicts the situation where the development is at an early stage and the developer already used some methods of the chosen API to develop the required functionality. However, she is not sure how to proceed from this point. In such cases, different sources of information may be consulted, such as StackOverflow, video tutorials, API documentation, etc. We propose an approach aiming at providing developers with recommendations consisting of a list of API method calls that should be used next, and with usage patterns that can be used as a reference for completing the development of the method being defined (e.g., code snippets that could support developers to complete the method definition with the framed code in Figure 1 (b)).

2.2 API Function Calls and Usage Patterns

A *software project* is a standalone source code unit that performs a set of tasks. Furthermore, an *API* is an interface that abstracts the functionalities offered by a project by hiding its implementation details. This interface is meant to support reuse and modularity [71, 81]. An API X built in an object-oriented programming language (e.g., the `Criteria API` in Figure 1) consists of a set T_X of public types (e.g., `CriteriaBuilder` and `CriteriaQuery`). Each type in T_X consists of a set of public methods and fields that are available to client projects (e.g., see the method `createQuery` of the type `CriteriaQuery`).

A *method declaration* consists of a name, a (possibly empty) list of parameters, a return type, and a (possibly empty) body (e.g., the method `findBoekrekeningen` in Figure 1). Given a set of declarations D in a project P, an API *method invocation* i is a call made from a declaration $d \in D$ to another declaration m. Similarly, an API *field access* is an access to a field $f \in F$ from a declaration d in P. API method invocations MI and field

[1] https://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html

```
public List<Boekrekening> findBoekrekeningen() {
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Boekrekening> criteriaQueryBoekrekening = cb
                                                        .createQuery(
                                                         Boekrekening.class
                                                        );
 Root<BoekrekeningPO> boekrekeningFrom = criteriaQueryBoekrekening
                                          .from(BoekrekeningPO.class);




}
```

(a) Initial version

```
public List<Boekrekening> findBoekrekeningen() {
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Boekrekening> criteriaQueryBoekrekening = cb
                                                        .createQuery(
                                                         Boekrekening.class
                                                        );
 Root<BoekrekeningPO> boekrekeningFrom = criteriaQueryBoekrekening
                                          .from(BoekrekeningPO.class);

 criteriaQueryBoekrekening.select(boekrekeningFrom);
 criteriaQueryBoekrekening.orderBy(cb.asc(boekrekeningFrom
                                           .get(BoekrekeningPO_.rekeningnr)));
 return entityManager.createQuery(criteriaQueryBoekrekening).getResultList();
}
```

(b) Final version

Figure 1: Motivating example

accesses FA in P form the set of API usages U, i.e., $U = MI \cup FA$. Finally, an *API usage pattern* (or code snippet) is a sequence $(u_1, u_2, ..., u_n)$, $\forall u_k \in U$ [62].

2.3 Context-aware Collaborative Filtering

As stated by Schafer et al. [88] "*Collaborative Filtering* (CF) is the process of filtering or evaluating items through the opinions of other people." In a CF system, a *user* that buys or uses an *item* attributes a rating to it based on her experience and perceived value. Therefore, a *rating* is the association of a user and an item through a value in a given unit (usually in scalar, binary, or unary form). The set of all ratings of a given user is also known as a *user profile* [17]. Moreover, the set of all ratings given in a system by existing users can be represented in a so-called *rating matrix*, where a row represents a user and a column represents an item.

The expected outcome of a CF system is a set of predicted ratings (aka. *recommendations*) for a specific user and a subset of items [88]. The recommender system considers the most similar users (aka. *neighbors*) to the *active* user to suggest new ratings. A similarity function $sim_{usr}(u_a, u_j)$ is used to compute the *weight* of the active user profile u_a against each of the user profiles u_j in the system. Finally, to suggest a recommendation for an item i based on this subset of similar profiles, the CF system computes a weighted average $r(u_a, i)$ of the existing ratings, where $r(u_a, i)$ varies with the value of $sim_{usr}(u_a, u_j)$ obtained for all neighbors [17, 88].

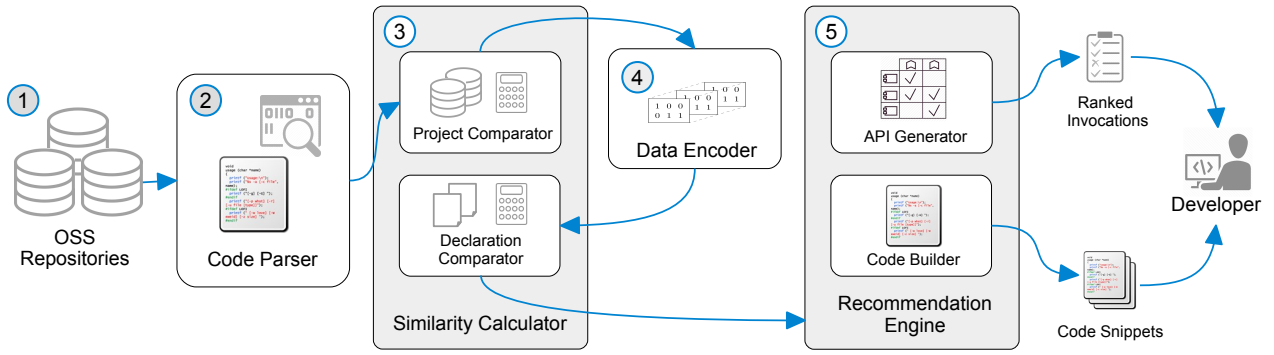

Figure 2: Overview of the FOCUS architecture

Besides, *context-aware CF* systems compute recommendations based not only on neighbors' profiles but also on the *context* where the recommendation is demanded. Each rating is associated with a context [17]. Therefore, for a tuple C modeling different contexts, a *context similarity* metric $sim_{ctx}(c_a, c_i)$, for $c_a, c_i \in C$ is computed to identify relevant ratings according to a given context. Then, the weighted average is reformulated as $r(u_a, i, c_a)$ [17].

3 Proposed Approach

To tackle the problem of recommending API function calls and usage patterns, we leverage the wisdom of the crowd and existing recommender system techniques. In particular, we hypothesize that API calls and usages can be mined from existing codebases, prioritizing the projects that are similar to the one from where the recommendation is demanded.

More specifically, FOCUS adopts a context-aware CF technique to search for invocations from closely relevant projects. This technique allows us to consider both project and declaration similarities to recommend API function calls and usage patterns. Following the terminology of recommender systems, we treat *projects* as the enclosing *contexts*, *method declarations* as *users*, and *method invocations* as *items*. Intuitively, we recommend a method invocation for a declaration in a given project, which is analogous to recommending an item to a user in a given context. For instance, the set of method invocations and the usage pattern (cf. framed code in Figure 1.b) recommended for the declaration `findBoekrekeningen` can be obtained from a set of similar projects and declarations in a codebase. The *collaborative* aspect of the approach enables to extract recommendations from the most similar projects, while the *context-awareness* aspect enables to narrow down the search space further to similar declarations.

3.1 Architecture

The FOCUS architecture is depicted in Figure 2. To provide its recommendations, FOCUS considers a set of *Open Source Software (OSS) Repositories* ①. The *Code Parser* ② component is in charge of extracting both method declarations and invocations from the source code of these projects. The *Project Comparator*, a subcomponent of the *Similarity Calculator* ③, computes the similarity between projects in the OSS repositories and the project under development. Using the set of projects and the information extracted by the Code Parser,

Listing 1: M^3 model excerpt.

```
1 m3.declarations = {
2 <|java+method://org/edr/services/impl/StandaardBoekrekeningService/findBoekrekeningen()|,
3 |file:///Users/[...]StandaardBoekrekeningService.java(501,531,<17,4>,<33,5>)|>, [...]}
4
5 m3.methodInvocation = {
6 <|java+method://org/edr/services/impl/StandaardBoekrekeningService/findBoekrekeningen()|,
7 |java+method://javax/persistence/EntityManager/getCriteriaBuilder()|>, [...]}
```

the *Data Encoder* component ④ computes rating matrices which are introduced later in this section. Afterwards, the *Declaration Comparator* computes the similarities between declarations. From the similarity scores, the *Recommendation Engine* ⑤ generates recommendations, either as a ranked list of API function calls using the *API Generator*, or as usage patterns using the *Code Builder*. Finally, the recommendations are presented to the developer. In the remainder of this section, we present in greater details each of these components.

3.1.1 Code Parser

FOCUS relies on Rascal M^3 [13] to extract method declarations and invocations from a set of OSS repositories, an intermediate model that performs static analysis on source code to extract facts about a given project. This model is an extensible and composable algebraic data type that captures both language agnostic and Java-specific facts in immutable binary relations. These relations represent program information such as existing *declarations*, *method invocations*, *field accesses*, *interface implementations*, *class extensions*, among others [13]. To gather relevant data, Rascal M^3 employs the Eclipse JDT Core Component [4] to create and traverse the abstract syntax trees of the target Java project.

In the context of FOCUS, we consider the data provided by the *declarations* and *methodInvocation* relations of the M^3 model. Both of them contain a set of pairs $\langle v_1, v_2 \rangle$, where v_1 and v_2 are values representing *locations* in the Rascal environment. These locations are uniform resource identifiers that represent artifact identities (aka. logical locations) or physical pointers on the file system to the corresponding artifacts (aka. physical locations). The *declarations* relation maps the logical location of an artifact (e.g., a method) to its physical location. The *methodInvocation* relations map the logical location of a *caller* to the logical location of a *callee*.

Listing 1 depicts an excerpt of the M^3 model extracted from the code presented in Figure 1.a. As illustrated, the *declarations* relation links the logical location of the method `findBoekrekeningen`, to its corresponding physical location in the file system. Besides, the *methodInvocation* relation states that the `getCriteriaBuilder` method of the `EntityManager` type is invoked by the `findBoekrekeningen` method in the current project.

Rascal M^3 follows a conservative approach when building the *methodInvocation* relation: method invocations are only included when the callee's owner type can be resolved. To mitigate this issue, the classpath of each project must be reconstructed carefully. Thus, given that a plethora of Java projects rely on dependency managers and modular systems such as Apache Maven [1] and OSGi [6], we use both Maven and Tycho [8] to build Maven and OSGi-based projects and extract their classpaths. Plain libraries included as JAR archives in the project are also considered. In the end, we discard method invocations that cannot be properly resolved.

When facing conditional or loop statements, Rascal M^3 adds to the *methodInvocation* relation method calls present in all branches of the control flow. In the presence of inheritance, Rascal M^3 first considers the methods

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{array} \begin{array}{c} \begin{array}{cccc} i_1 & i_2 & i_3 & i_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 3: Representation of declarations and invocations

of type T related to the invocation. If the corresponding method is only declared in T or if it overrides a superclass, the fully qualified name points to the method in T. Otherwise, if the method is inherited from a superclass T' and it is not overridden in T, then the relation is associated to T'. Besides, method invocations of a super class are also considered. When considering interfaces, if there is an object o of type T and T implements interface I, the method invocation through o points to I or T according to the static type of o. Finally, we do not consider method invocations through dynamic mechanisms such as Java reflection.

3.1.2 Data Encoder

Once method declarations and invocations are extracted, FOCUS represents the relationships among them using a rating matrix. For a given project, each row in the matrix represents a method declaration and each column represents a method invocation. A cell is set to 1 if the declaration in the corresponding row contains the invocation in the column, otherwise it is set to 0. For example, Figure 3 shows the ratings matrix of a project with 4 declarations $p_1 \ni (d_1, d_2, d_3, d_4)$ and 4 invocations (i_1, i_2, i_3, i_4).

To capture the intrinsic relationships among various projects, declarations, and invocations, we come up with a 3D context-based ratings matrix. In this matrix a third dimension is added to represent a project, which is analogous to the so-called context in context-aware CF techniques. For example, Figure 4 depicts a list of three OSS projects $P = (p_a, p_1, p_2)$ represented by three slices with 4 method declarations and 4 method invocations. Project p_1 has already been introduced in Figure 3 and for the sake of readability, the column and row labels are removed from all slices in Figure 4. There, p_a is the *active project* and it has an *active declaration*. *Active* here means the artifact (project or declaration), being considered or developed. Both p_1 and p_2 are complete projects and are strictly similar to the active project p_a. The former projects (i.e., p_1 and p_2) are also called *background data* since they are already available and serve as a base for the recommendation process. In practice, the higher the number of complete projects considered as background data, the higher the probability to recommend relevant invocations.

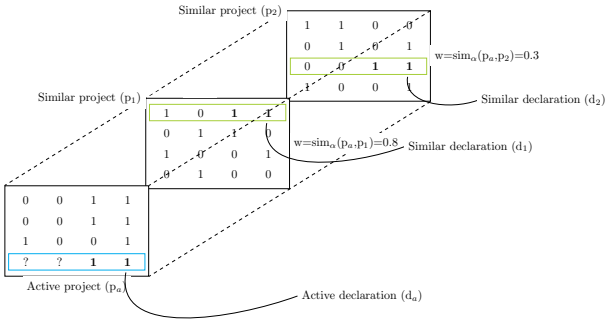

Figure 4: 3D context-based ratings matrix

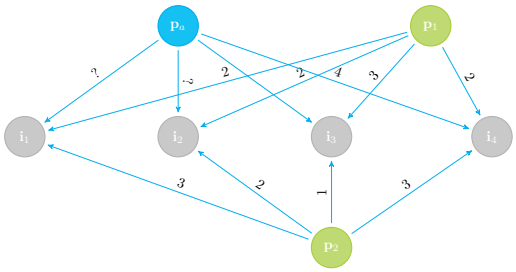

Figure 5: Graph representation of projects and invocations

3.1.3 Similarity Calculator

Exploiting the context-aware CF technique, the presence of additional invocations is deduced from similar declarations and projects. Given an active declaration in an active project, it is essential to find the subset of the most similar projects, and then the most similar declarations in that set of projects. To compute similarities, we employ a weighted directed graph that models the relationships among projects and invocations. Each node in the graph represents either a project or an invocation. If project p contains invocation i, then there is a directed edge from p to i. The weight of an edge $p \rightarrow i$ represents the number of times project p performs the invocation i. Figure 5 depicts the graph for the set of projects introduced in Figure 4. For instance, p_a has 4 declarations and all of them have i_4 as an invocation. As a result, the edge $p_a \rightarrow i_4$ has a weight of 4. In the graph, a question mark represents missing information, since for the active declaration in p_a, it is not clear if invocations i_1 and i_2 also belong to it or not.

The similarity between two project nodes p and q is computed by considering their feature sets [27]. Given that p has a set of neighbor nodes $(i_1, i_2, .., i_l)$, the feature set of p corresponds to the vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_k being the weight of node i_k. This weight is computed as the *term-frequency inverse document frequency* value, i.e., $\phi_k = f_{i_k} * log(\frac{|P|}{a_{i_k}})$, where f_{i_k} is the weight of the edge $p \rightarrow i_k$; $|P|$ is the number of all considered projects; and a_{i_k} is the number of projects connecting to i_k. Eventually, the similarity between p and q with their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_k\}_{k=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$ is:

$$sim_\alpha(p, q) = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n} (\phi_t)^2} \times \sqrt{\sum_{t=1}^{n} (\omega_t)^2}} \quad (1)$$

The similarities among method declarations are calculated using the Jaccard similarity index [43] as follows:

$$sim_\beta(d, e) = \frac{|\mathbb{F}(d) \bigcap \mathbb{F}(e)|}{|\mathbb{F}(d) \bigcup \mathbb{F}(e)|} \quad (2)$$

where $\mathbb{F}(d)$ and $\mathbb{F}(e)$ are the sets of invocations made from declarations d and e, respectively.

3.1.4 API Generator

This component, which is part of the *Recommendation Engine*, is in charge of generating a ranked list of API function calls. In Figure 4, the active project p_a already includes three declarations, and the developer is working on the fourth declaration, which corresponds to the last row of the slice. p_a has only two invocations, represented in the last two columns of the matrix (i.e., cells filled with 1). The first two cells are marked with a question mark (?), indicating that it is unclear whether these two invocations should also be incorporated into p_a.

The recommendation engine attempts to predict additional invocations for the active declaration by computing the missing ratings using the following formula [17]:

$$r_{d,i,p} = \overline{r}_d + \frac{\sum_{e \in topsim(d)} (R_{e,i,p} - \overline{r}_e) \cdot sim_\beta(d,e)}{\sum_{e \in topsim(d)} sim_\beta(d,e)} \tag{3}$$

Equation (3) is used to compute a score for the cell representing method invocation i, declaration d of project p, where $topsim(d)$ is the set of top similar declarations of d; $sim_\beta(d,e)$ is the similarity between d and a declaration e, computed using Equation (2); $\overline{r}_d$ and $\overline{r}_e$ are average ratings of d and e, respectively; and $R_{e,i,p}$ is the combined rating of declaration d for i in all similar projects, computed as follows [17]:

$$R_{e,i,p} = \frac{\sum_{q \in topsim(p)} r_{e,i,q} \cdot sim_\alpha(p,q)}{\sum_{q \in topsim(p)} sim_\alpha(p,q)} \tag{4}$$

where $topsim(p)$ is the set of top similar projects of p; and $sim_\alpha(p,q)$ is the similarity between p and a project q, computed using Equation (1). Equation (4) implies that a higher weight is given to projects with a higher similarity. In practice, it is reasonable since, given a project, its similar projects contain more relevant API calls than less similar projects. Using Equation (3) we compute all the missing ratings in the active declaration and get a ranked list of invocations with scores in descending order, which is then suggested to the developer.

3.1.5 Code Builder

This subcomponent is also part of the *Recommendation Engine*, and it is responsible of recommending usage patterns to developers. Thus, from the ranked list, *top-N* function calls are used as query to search the database for relevant declarations. To limit the search scope, only the most similar projects are considered by means of the *Similarity Calculator*. The Jaccard index is used to compute similarities among the selected invocations and a given declaration. For each query, we search for declarations that contain as many invocations of the query as possible. Once we identify the corresponding declarations we map back the identified methods to its original source code. To this aim, we build the corresponding Rascal M^3 logical location for each one of the identified methods, and we seek the corresponding physical location in the *declarations* relation. Afterwards, we rely on Rascal to extract and retrieve the code snippet of the declaration from the specified location, which is then recommended to the developer.

For the sake of illustration, we now present an example of how FOCUS suggests real code snippets, considering declaration `findBoekrekeningen` in Figure 1.a as input. The present invocations are used together with the other declarations in the current project as query to feed the *Recommendation Engine*. The final outcome is a ranked list of real code snippets. We choose the top one named `findByIdentifier` and depict it in Figure 6. By carefully examining this code and the original one in Figure 1.b, we see that although they are not exactly the same, they indeed share different function calls and a common intent: both exploit a `CriteriaBuilder` object to build, perform a query and eventually get back some results. Furthermore, the outcome of both declarations is of the `List` type. Interestingly, compared to the original one, the recommended code appears to be of higher quality since it includes a `try/catch` pair to handle possible exceptions. Thus, the recommended code, coupled with the corresponding list of function calls (i.e., `get`, `equal`, `where`, `select`, etc.), provides the developer with helpful directions on how to use the APIs at hand to implement the desired function.

```
public List<QuestionsStaged> findByIdentifier(String identifier) {
    log.fine("getting Session instance by identifier: " + identifier);
    try {
      CriteriaBuilder cb = entityManager.getCriteriaBuilder();
      CriteriaQuery<QuestionsStaged> criteria = cb.createQuery(QuestionsStaged.class);
      Root<QuestionsStaged> qs = criteria.from(QuestionsStaged.class);
      criteria.select(qs).where(cb.equal(qs.get("identifier"), identifier));
      log.fine("get identifier successful");
      return entityManager.createQuery(criteria).getResultList();

    } catch (RuntimeException re) {
      log.severe("get identifier failed" + re);
      throw re;
    }
  }
```

Figure 6: Real source code recommended by FOCUS

4 Evaluation

The *goal* of this evaluation is to assess FOCUS, and compare it with another state-of-the-art tool (PAM [34]), with the aim of assessing its capability to recommend API usage patterns to developers, while they are writing code. The *quality focus* is two-fold: studying the API recommendation accuracy and completeness, as well as the time required by FOCUS and PAM to provide a recommendation. The *context* consists of 610 Java open source projects, and 3,600 JAR archives from the Maven Central repository [5]. For the sake of reproducibility and ease of reference, all artifacts used in the evaluation, together with the developed tools, are available online [12]. We choose PAM as a baseline for comparison, because it has been shown to outperform [34] other similar tools such as MAPO [114] and UP-Miner [105]. To conduct the comparison with PAM, we leverage its original source code made available online by its authors [33].

In the following, we detail our research questions, the datasets we use, our evaluation methodology, and the evaluation metrics.

4.1 Research Questions

The evaluation's research questions are as follows:

RQ$_1$ *To what extent is FOCUS able to provide accurate and complete recommendations?* This research question relates to the capability of FOCUS to produce accurate and complete results. Having too many false positives would end up being counter-productive with developers, whereas having too many false negatives means that the tool would not be able to provide recommendations in many cases where this is needed;

RQ$_2$ *What are the timing performances of FOCUS in building its models and in providing recommendations?* This research question aims at assessing whether, from a timing point of view, FOCUS—compared to PAM—could be used in practice. We evaluate the time required by both tools to provide a recommendation. We mainly focus on the recommendation time because, while it is acceptable that the model training phase is relatively slow (i.e., the model could be built offline), the recommendation time has to be fast enough to make the tool applicable in practice.

RQ$_3$ *How does FOCUS perform compared with PAM?* Finally, this research question directly compares the recommendation capabilities of FOCUS and PAM.

4.2 Datasets

To answer the evaluation's questions, we relied on four different datasets.

The first dataset, $\mathtt{SH}_L$, has been assembled starting from $5,147$ randomly selected Java projects retrieved from GitHub via the Software Heritage archive [26]. To comply with the requirements of PAM, we first restricted the dataset to the list of projects that use at least one of the third-party libraries listed in Table 1. Most of them were used to assess the performance of PAM [34]. Each row in Table 1 lists a third-party library, the number of projects that depend on it, and the number of classes that invoke methods of this library. To comply with the requirements of FOCUS, we then restricted the dataset to the list of projects containing at least one *pom.xml*, as it eases the creation of the M^3 models. We thus obtained our first dataset consisting of 610 Java projects.

From $\mathtt{SH}_L$, we extracted a second dataset $\mathtt{SH}_S$ consisting of the 200 smallest (in size) projects of $\mathtt{SH}_L$.

As a third dataset, we randomly collected a set of $3,600$ JAR archives from the Maven Central repository, which we name $\mathtt{MV}_L$. Through a manual inspection of $\mathtt{MV}_L$, we noticed that many projects only differ in their version numbers (*ant-1.6.5.jar* and *ant-1.9.3.jar*, for instance, are two versions of the same project *ant*). These cases are interesting as we assume two versions of the same project share many functionalities [96]. The collaborative-filtering technique works well given that highly similar projects exist, since it just "copies" invocations from similar methods in the very similar projects (see Equation (3) and Equation (4)). However, a dataset containing too many similar projects may introduce a bias in the evaluation. Thus, we decided to populate one more dataset. Starting from $\mathtt{MV}_L$, we randomly selected one version for every project and filtered out the other versions. The removal resulted in a fourth dataset consisting of $1,600$ projects, which we name $\mathtt{MV}_S$.

Three datasets, i.e., $\mathtt{SH}_L$, $\mathtt{MV}_L$ and $\mathtt{MV}_S$ are used to assess the performance of FOCUS ($\mathbf{RQ}_1$). The smallest dataset $\mathtt{SH}_S$ is used to compare FOCUS and PAM ($\mathbf{RQ}_2$ and $\mathbf{RQ}_3$).

Eventually, the process of creating required metadata consists of the following main steps:

- for each project in the dataset the corresponding Rascal M^3 model is generated;
- for each M^3 model, the corresponding ARFF representations [2] are generated in order to be used as input for applying FOCUS and PAM during the actual evaluation steps discussed in the next sections.

4.3 Study Methodology

Performing a user study has been accepted as the standard method to validate an API usage recommendation tool [58, 114]. While user studies are valuable, they are limited in the size of the task a participant can conduct, and are highly susceptible to individual skills and subjectiveness. Instead, we decide to perform a different, offline evaluation, by simulating the behavior of a developer working at different stages of a development project on partial code snippets.

More specifically, we simulate different stages of a development process to study if FOCUS is applicable in real-world settings, by considering a programmer who is developing a project p. To this end, some parts of p are removed to mimic an actual development. Given an original project p, the total number of method declarations it contains is called Δ. However, only δ declarations ($\delta < \Delta$) are used as input for recommendation and the rest is discarded. In practice, this corresponds to the situation when the developer already finished δ declarations, and she is now working on the *active declaration* d_a. For d_a, the developer has just written π invocations. In practice, δ is small at an early stage, and increases over the course of time. Similarly, π is small when the developer just starts working on d_a. The two parameters δ, π are used to stimulate different development phases. In particular, we consider the following configurations.

Table 1: Fragment of the third-party libraries used by Dataset $\texttt{SH}_L$

Project Name	# of Client Projects	# of Client Classes
com.google.gson	51	337
io.netty	105	13,456
org.apache.camel	36	1,017
org.apache.hadoop	158	14,596
org.apache.lucene	15	397
org.apache.mahout	25	8541
org.apache.wicket	44	3,360
org.drools	27	886
org.glassfish.jersey	105	3811
org.hornetq	15	123
org.jboss.weld	39	1875
org.jooq	16	243
org.jsoup	23	55
org.neo4j	28	4,983
org.restlet	19	326
org.springside	16	821
twitter4j	45	597
	610	**55,425**

Configuration `C1.1` ($\delta = \Delta/2 - 1, \pi = 1$): Almost the first half of the declarations is used as testing data and the second half is removed. The last declaration of the first half is selected as the active declaration d_a. For d_a, only the *first* invocation is provided as a query, and the rest is used as ground-truth data which we call `GT(p)`. This configuration mimics a scenario where the developer is at an early stage of the development process and, therefore, only limited context data is available to feed the recommendation engine.

Configuration `C1.2` ($\delta = \Delta/2 - 1, \pi = 4$): Similarly to `C1.1`, almost the first half of the declarations is kept and the second half is discarded. d_a is the last declaration of the first half of declarations. For d_a, the first *four* invocations are provided as query, and the rest is used as `GT(p)`.

Configuration `C2.1` ($\delta = \Delta - 1, \pi = 1$): The last method declaration is selected as testing, i.e., d_a and all the remaining declarations are used as training data ($\Delta - 1$). In d_a, the *first* invocation is kept and all the others are taken out as ground-truth data `GT(p)`. This represents the stage where the developer almost finished implementing p.

Configuration `C2.2` ($\delta = \Delta - 1, \pi = 4$): Similar to `C2.1`, d_a is selected as the last method declaration, and all the remaining declarations are used as training data ($\Delta - 1$). The only difference with `C2.1` is that in d_a, the first *four* invocations are used as query and all the remaining ones are used as ground-truth data `GT(p)`.

When performing the experiments, we split a dataset into two independent parts, namely a *training set* and a *testing set*. In practice, the training set represents the OSS projects that have been collected a priori. They are available at developers' disposal, ready to be exploited for mining purposes. The testing set represents the project being developed, or *the active project*. This way, our evaluation mimics a real development scheme: *the recommender system should produce recommendations for a project based on the data available from a set of existing projects*. We opt for *k-fold cross validation* [49] as it is widely chosen to evaluate machine learning models. Depending on the availability of input data, the dataset with n elements is divided into k equal parts, so-called *folds*. For each validation round, one fold is used as testing data and the remaining $k - 1$ folds are

used as training data. For this evaluation, we select two values, i.e., $k = 10$ and $k = n$. The former corresponds to *ten-fold cross validation* and the latter corresponds to *leave-one-out cross validation* [107].

4.4 Evaluation Metrics

For a testing project p, the outcome of a recommendation process is a ranked list of invocations, i.e., `REC(p)`. It is our firm belief that the ability to provide accurate invocations is important in the context of software development. Thus, we are interested in how well a system can recommend API invocations that eventually match with those stored in `GT(p)`. To measure the performance of the recommender systems, i.e., PAM and FOCUS, we utilize two metrics, namely *success rate* and *accuracy* [27]. Given a ranked list of recommendations, a developer typically pays attention to the *top-N* items only. *Success rate* and *accuracy* are computed by using N as the *cut-off value*. Given that $REC_N(p)$ is the set of *top-N* items and $match_N(p) = GT(p) \bigcap REC_N(p)$ is the set of items in the *top-N* list that match with those in the ground-truth data, then the metrics are defined as follows.

Success rate: Given a set of *P* testing projects, this metric measures the rate at which a recommendation engine can return at least a match among *top-N* recommended items for every project $p \in P$.

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} \times 100\% \tag{5}$$

where *count()* counts the number of times the boolean expression given as parameter evaluates to *true*.

Accuracy: Precision and recall are employed to measure accuracy [27]. *Precision@N* is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

$$precision@N = \frac{|match_N(p)|}{N} \tag{6}$$

and *recall@N* is the ratio of the ground-truth items being found in the *top-N* items:

$$recall@N = \frac{|match_N(p)|}{|GT(p)|} \tag{7}$$

Recommendation time: As mentioned in $\mathbf{RQ}_2$, we measure the time needed by both PAM and FOCUS to perform a prediction on a given infrastructure, which is a laptop with Intel Core i5-7200U CPU @ 2.50GHz × 4, 8GB RAM, and Ubuntu 16.04.

5 Results

$\mathbf{RQ}_1$: *To what extent is FOCUS able to provide accurate and complete recommendations?*

To answer this research question, we use the dataset $\mathtt{SH}_L$ and vary the length of the input data for every testing project. Two main configurations are taken into account, with two sub-configurations for each as introduced in Section 4.3. Table 2 shows the success rate for all the configurations. For a small N, i.e., $N = 1$ (a developer expects a very brief list of items) FOCUS is still able to provide matches. For example, the success rates of `C1.1` and `C1.2` are 24.59% and 30.65%, respectively. When the cut-off value N is increased, the corresponding success rates improve linearly. For example, when $N = 20$, FOCUS obtains 40.98% success rate for `C1.1` and 47.70% for `C1.2`. By comparing the results obtained for `C1.1` and `C1.2`, we see that when more invocations

Table 2: Success rate for SH_L, N={1,5,10,15,20}

	SH_L			
N	C1.1	C1.2	C2.1	C2.2
1	24.59	30.65	23.44	29.83
5	31.96	40.00	31.31	39.01
10	35.90	43.77	35.73	43.77
15	39.34	47.21	37.70	45.57
20	40.98	47.70	39.34	46.88

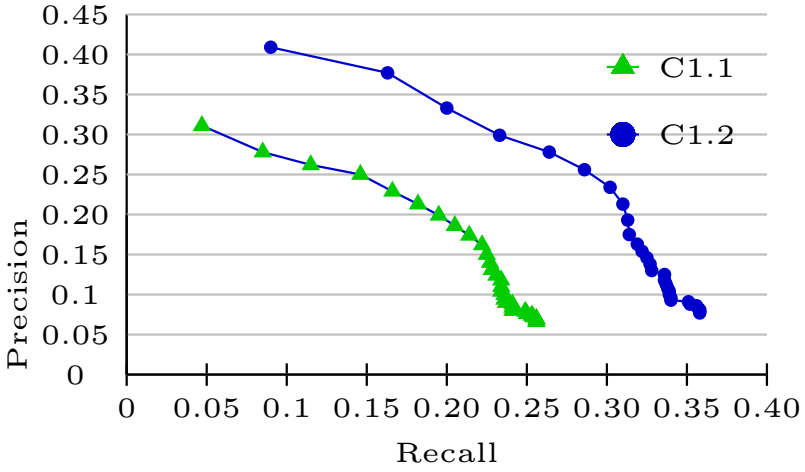

Figure 7: Precision and recall for C1.1 and C1.2 on SH_L

are incorporated into the query, FOCUS provides more precise recommendations. In practice, this means that the accuracy of recommendations improves with the maturity of the project.

We now consider the outcomes obtained for C2.1 and C2.2. In these configurations, more background data is available for a recommendation. For C2.1 ($\delta = \Delta - 1, \pi = 1$), the success rates for the smallest values of N, i.e., $N = 1$ and $N = 5$ are 23.44% and 31.31%, respectively. In other words, it improves with N. The same trend can be observed with other cut-off values, i.e., $N = 10, 15, 20$: the success rates for these settings increase correspondingly. We notice the same pattern considering C2.1 and C2.2 together, or C1.1 and C1.2 together: if more invocations are used as query, FOCUS suggests more accurate invocations.

Figure 7 and Figure 8 depict the precision and recall curves (PRCs) for the above mentioned configurations. In particular, Figure 7 represents the accuracy when almost the first half of the declarations ($\delta = \Delta/2 - 1$) together with one (C1.1) and four invocations (C1.2) from the testing declaration d_a are used as query. Since a PRC close to the upper right corner indicates a better accuracy [27], we see that the accuracy of C1.2 is superior to that of C1.1. Similarly with C2.1 and C2.2, as depicted in Figure 8, the accuracy improves substantially when the query contains more invocations. These facts further confirm that FOCUS is able to recommend more relevant invocations when the developer keeps coding. This improvement is obtained since the similarity between declarations can be better determined when more invocations are available as comprehended in Equation (4).

The results reported so far appear to be promising at the first sight. However, by considering Table 2, Figure 7, and Figure 8 together, we realize that both success rate and accuracy are considerably low: The best success rate is 47.70% for C1.2 when $N = 20$, which means that more than half of the queries got no match. In this sense,

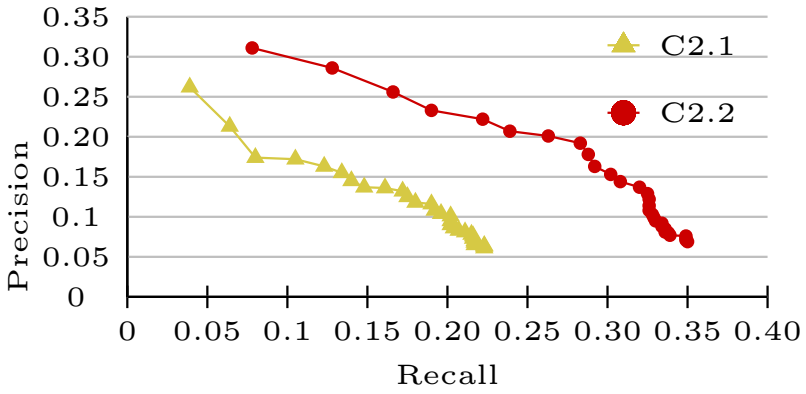

Figure 8: Precision and recall for `C2.1` and `C2.2` on $\mathtt{SH}_L$

it is necessary to ascertain the cause of this outcome: Is FOCUS only capable of generating such moderate recommendations, or is it because of the data? Our intuition is that $\mathtt{SH}_L$ is rather small in size, which means the background data available for the recommendation process is limited. Thus, to further validate the performance of FOCUS, we perform additional experiments by considering more data, using both $\mathtt{MV}_L$ and $\mathtt{MV}_S$. For this evaluation, we just consider the case when only one invocation together with other declarations are used as query, i.e., `C1.1` and `C2.1`. This aims at validating the performance of FOCUS, given that the developer just finished only one invocation in d_a.

Table 3 depicts the success rate obtained for different cut-off values using both datasets. The success rates for all configurations are much better than those of $\mathtt{SH}_L$. The scores are considerably high, even when $N = 1$, the success rate obtained by `C1.1` and `C2.1` are 72.30% and 72.80%, respectively. For $\mathtt{MV}_S$, the corresponding success rates are lower. However, this is understandable since the set has less data compared to $\mathtt{MV}_L$.

The PRCs for $\mathtt{MV}_L$ and $\mathtt{MV}_S$ are shown in Figure 9 and Figure 10, respectively. We see that for $\mathtt{MV}_L$, a superior performance is obtained by configuration `C2.1`, i.e., when more background data is available for recommendation in comparison to `C1.1`. For $\mathtt{MV}_S$, we witness the same trend as with success rate: the difference between the two configurations `C1.1` and `C2.1` is negligible. Considering both Figure 9 and Figure 10, we observe that the overall accuracy for $\mathtt{MV}_L$ is much better than that of $\mathtt{MV}_S$. The maximum precision and recall for $\mathtt{MV}_L$ are 0.75 and 0.62, respectively. Whereas, the maximum precision and recall for $\mathtt{MV}_S$ are 0.52 and 0.36, respectively. This further confirms the fact that with more similar projects, the system can provide better recommendations. Referring back to the outcomes of $\mathtt{SH}_L$, we observe that the performance on $\mathtt{MV}_L$ and $\mathtt{MV}_S$ is improved substantially.

Table 3: Success rate for $\mathtt{MV}_L$ and $\mathtt{MV}_S$, N={1,5,10,15,20}

	$\mathtt{MV}_L$		$\mathtt{MV}_S$	
N	`C1.1`	`C2.1`	`C1.1`	`C2.1`
1	72.30	72.80	49.40	50.10
5	82.80	82.70	64.60	65.40
10	86.40	86.40	69.30	70.10
15	88.10	87.90	71.60	72.20
20	89.20	89.00	73.30	74.30

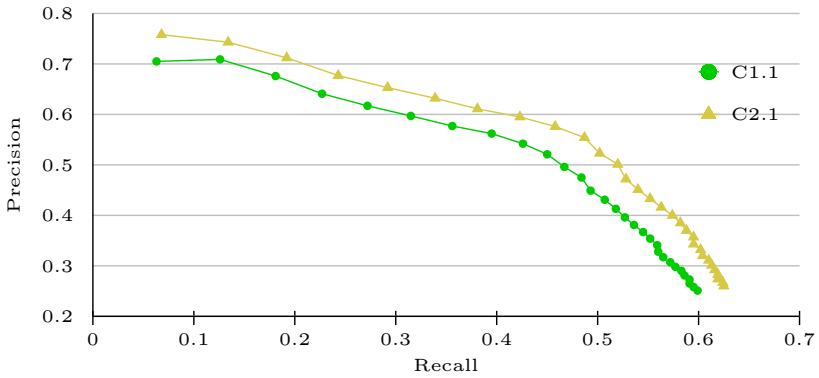

Figure 9: Precision and recall for `C1.1` and `C2.1` on $\texttt{MV}_L$

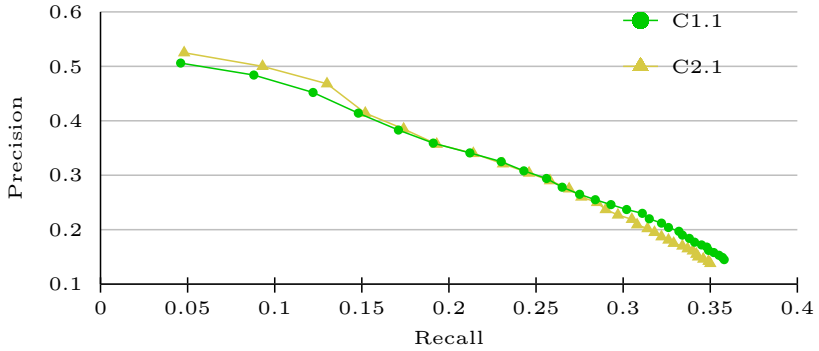

Figure 10: Precision and recall for `C1.1` and `C2.1` on $\texttt{MV}_S$

To sum up, we conclude that the performance of FOCUS relies on the availability of background data. The system works effectively given that more OSS projects are available for recommendation. In practice, it is expected that we can crawl as many projects as possible, and use them as background data for the recommendation process.

RQ_2: *What are the timing performances of FOCUS in building its models and in providing recommendations?*

To measure the execution time of PAM and FOCUS for the very first attempt, we ran both systems on the $\texttt{SH}_L$ dataset, consisting of 610 projects. With PAM, for each testing project, we combined the extracted query with all the other training projects to produce a single `ARFF` file provided as input for the recommendation process [34]. Nevertheless, we then realized that the execution of PAM is very time-consuming. For instance, for one fold containing 1 testing and 549 training projects (i.e., $610/10 \cdot 9$ training folds) with 80MB in size, PAM takes around 320 seconds to produce the final recommendations. Instead, the corresponding execution time by FOCUS is quite faster than PAM, around 1.80 seconds. Given the circumstances, it is not feasible to run PAM on a large dataset.

Therefore, we decided to use the $\texttt{SH}_S$ dataset (consisting of 200 projects) for this purpose. For the experiments, we opt for *leave-one-out cross-validation* [107], i.e., one project is used as test set, and all the remaining 199 projects are used for the training. The rationale behind the selection of this method instead of ten-fold cross-validation is that we want to exploit as much as possible the projects available as background data, given a testing project. The validation was executed 200 times, and we measured the time needed to finish the recommendation process. On average, PAM required 9 seconds to provide each recommendation while FOCUS just required 0.095 seconds, i.e., it is two orders of magnitude faster and suitable to be integrated into a development environment.

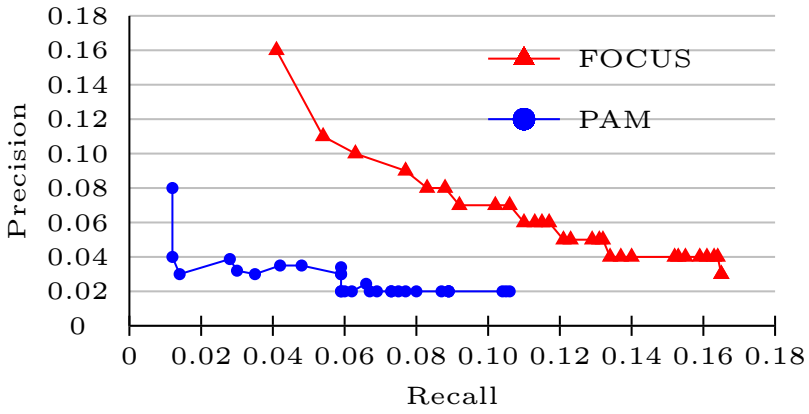

Figure 11: Precision and recall for PAM and FOCUS using $\mathtt{SH}_S$.

RQ$_3$: *How does FOCUS perform compared with PAM?*

For the reasons explained in **RQ**$_2$, the comparison between PAM and FOCUS has been performed on the $\mathtt{SH}_S$ dataset. FOCUS gains a better success rate than PAM does, i.e., 51.20% compared to 41.60%. Furthermore, as depicted in Figure 11, there is a big gap between the PRCs for PAM and FOCUS, with the one representing FOCUS closer to the upper right corner. This implies that the accuracy obtained by FOCUS is considerably superior to that of PAM.

A statistical comparison of PAM and FOCUS using Fisher's exact test [32] indicates that, for $1 \leq N \leq 20$, FOCUS always outperforms PAM: We achieved p-values < 0.001 (adjusted using the Holm's correction [41]) in all cases, with an Odds Ratio between 2.21 and 3.71, and equal to 2.54 for $N = 1$. In other words, FOCUS has over twice the odds of providing an accurate recommendation than PAM.

It is worth noting that the overall accuracy of FOCUS achieved and reported in this experiment is, although better than that of PAM, still considerably low. Following the experiments on $\mathtt{MV}_L$ and $\mathtt{MV}_S$ from **RQ**$_1$, we believe that this can be attributed to the limited background data available for the evaluation, since we only consider 200 projects.

In summary, by considering both **RQ**$_2$ and **RQ**$_3$, we come to the conclusion that FOCUS obtains a better performance in comparison to PAM with regards to success rate, accuracy and execution time. Lastly, since PAM takes considerable time to produce the final recommendations, it might be impractical to deploy in a development environment.

6 Threats to Validity

The main threat to *construct validity* concerns the simulated setting used to evaluate the approaches, as opposed to performing a user study. While our study has simulated the presence of incomplete code (with different degrees on incompleteness) for which a recommendation it is required, in a real development setting the order in which one writes statements might not fully reflect our simulation. This would make, in perspective, a further evaluation involving developers highly desirable.

Threats to *internal validity* concern factors internal to our study that could have influenced the results. One possible threat is represented by the achieved results for the datasets $\mathtt{SH}_L$ and $\mathtt{SH}_S$. As noted, these datasets exhibit lower precision/recall with respect to $\mathtt{MV}_L$ and $\mathtt{MV}_S$ mainly because of the limited training set used.

However, such datasets, and in particular SH_S, were created to allow us to compare FOCUS with PAM, otherwise less feasible on larger datasets.

Threats to *external validity* concern the generalization of our findings. Our tool is currently limited to Java programs, though in principle the approach could be implemented and evaluated with other programming languages.

7 Related Work

In this section, we summarize related work about API usage recommendation, and relate our contributions to the literature.

7.1 API Usage Pattern Recommendation

Acharya et al. [9] present a framework to extract API patterns as partial orders from client code. To this aim, control-flow-sensitive static API traces are extracted from source code and sequential patterns are computed. While this approach proposes a representation for API patterns, suggestions regarding API usage are still missing.

MAPO (Mining API usage Pattern from Open source repositories) is a tool that mines API usage patterns from client projects [114]. The system analyzes source files to collect API usage information and groups the API methods into clusters. It then mines API usage patterns from the clusters, ranks them according to their similarity with the current development context, and recommends code snippets to developers. Similarly, UP-Miner [105] mines API usage patterns by relying on *SeqSim*, a clustering strategy that reduces patterns redundancy and improves coverage. UP-Miner employs the BIDE algorithm [106] to mine API frequent closed call sequences. Differently from FOCUS, these approaches are based on clustering techniques, and consider all client projects in the mining regardless of their similarity with the current project.

Fowkes et al. introduce PAM (Probabilistic API Miner), a parameter-free probabilistic approach to mine API usage patterns [34]. PAM uses the structural Expectation-Maximization (EM) algorithm to infer the most probable API patterns from client code, which are then ranked according to their probability. PAM outperforms both MAPO and UP-Miner (lower redundancy and higher precision). We directly compare FOCUS to PAM in Section 4.

Niu et al. extract API usage patterns using API class or method names as queries [68]. They rely on the concept of object usage (method invocations on a given API class) to extract patterns. The approach of Niu et al. outperforms UP-Miner and Codota [3], a commercial recommendation engine, in terms of coverage, performance, and ranking relevance. In contrast, FOCUS relies on context-aware CF techniques—which favors recommendations from similar projects, and uses the whole development context to query API method calls.

The NCBUP-miner (Non Client-based Usage Patterns) [84] is a technique that identifies unordered API usage patterns from the API source code, based on both structural (methods that modify the same object) and semantic (methods that have the same vocabulary) relations. The same authors also propose MLUP [83], which is based on vector representation and clustering, but in this case client code is also considered.

DeepAPI [38] is a deep-learning method used to generate API usage sequences given a query in natural language. The learning problem is encoded as a machine translation problem, where queries are considered the source language and API sequences the target language. Only commented methods are considered during the search. The same authors [37] present CODEnn (COde-Description Embedding Neural Network), where, instead of

API sequences, code snippets are retrieved to the developer based on semantic aspects such as API sequences, comments, method names, and tokens.

With respect to the abovementioned approaches, FOCUS uses CF techniques to recommend and rank API method calls and usage patterns from a set of similar projects. In the end, not only relevant API invocations are recommended, but also code snippets are returned to the developer as usage examples.

7.2 API-Related Code Search Approaches

Strathcona [42] is a recommender system used to suggest API usage. It is an Eclipse plug-in that extracts the structural context of code and uses it as a query to request a set of code examples from a remote repository. Six heuristics (associated to class inheritance, method calls, and field types) are defined to perform the match. Similarly, Buse and Weimer [15] propose a technique for synthesizing API usage examples for a given data type. An algorithm based on data-flow analysis, k-medoids clustering, and pattern abstraction is designed. Its outcome is a set of syntactically correct and well-typed code snippets where example length, exception handling, variables initialization and naming, and abstract uses are considered.

Moreno et al. [58] introduce MUSE (Method USage Examples), an approach designed for recommending code examples related to a given API method. MUSE extracts API usages from client code, simplifies code examples with static slicing, and detects clones to group similar snippets. It also ranks examples according to certain properties (i.e., reusability, understandability, and popularity) and documents them.

SWIM (Synthesizing What I Mean) [77] seeks API structured call sequences (control and data-flows are considered), and then synthesizes API-related code snippets according to a query in natural language. The underlying learning model is also built with the EM algorithm. Similarly, Raychev et al. [79] propose a code completion approach based on natural language processing, which receives as input a partial program and outputs a set of API call sequences filling the gaps of the input. Both invocations and invocation arguments are synthesized considering multiple types of an API.

Thummalapenta and Xie propose SpotWeb [98], an approach that provides starting points (hotspots) for understanding a framework, and highlights where examples finding could be more challenging (coldspots). McMillan et al. [54] propose Portfolio, a tool that finds relevant functions implementing high-level requirements. Other tools exploit StackOverflow discussions to suggest context-specific code snippets and documentation [21, 72, 73, 74, 78, 80, 95, 99].

8 Conclusion

In this deliverable part, we introduced FOCUS, a context-aware collaborative-filtering system to assist developers in selecting suitable API function calls and usage patterns. To validate the performance of FOCUS, we conducted a thorough evaluation on different datasets consisting of GitHub and Maven open source projects. The evaluation was twofold. First, we examined whether FOCUS is applicable to real-world settings by providing developers with useful recommendations as they are programming. Second, we compared FOCUS with a well-established baseline, i.e., PAM, with the aim of showcasing the superiority of our proposed approach. Our results show that FOCUS recommends API calls with high success rates and accuracy. Compared to PAM, FOCUS works both effectively and efficiently as it can produce more accurate recommendations in a shorter time. The main advantage of FOCUS is that it can recommend real code snippets that match well with the development context. In contrast with several existing approaches, FOCUS does not depend on any specific set of libraries and just needs OSS projects as background data to generate API function calls. Lastly, FOCUS also scales well with

large datasets by using the collaborative-filtering technique that helps sweep irrelevant items, thus improving efficiency. With these advantages, we believe that FOCUS is suitable for supporting developers in real-world settings.

*We refer the reader to the companion deliverable **D2.8 – API Analysis Components** for more information on the implementation of FOCUS and its integration with the overall architecture of CROSSMINER.*

Part II
API Evolution and Migration

9 Introduction

Modularity and reuse in software engineering enable software engineers to rely on external libraries providing specific functionalities into the system they develop. The features of a library are exposed through an Application Programming Interface (API) which specifies what are the features that can be accessed and how to access them.

However, libraries are software themselves and, just like any software, they evolve to incorporate new features, to fix bugs, or simply to refactor their source code. When a library evolves, it is likely that its public API also evolves. Depending on the kind of changes introduced in the public API, it may be required to *migrate* the client code using this API if the developers want to benefit from the latest version of the library.

In this deliverable part, we first study this problem through an extensive state of the art in Section 10. Then, we present our framework to support API migration, MARACAS, in Section 11 and evaluate it on several case studies in Section 12. We conclude in Section 13.

10 State of the Art

10.1 API Switching and Upgrading

Since the early 2000's, researchers have invested time and effort on tackling the API switching and upgrading problem. It was only later in 2012 that Meng et al. [56] introduced two main categories to classify existing work: operation-based and matching-based approaches. We aim at extending this classification, as well as referencing and classifying new work in the field. Thus, we identify three main categories in the API switching and upgrading field. First, *record-and-replay* approaches [108], also referred to as *operation-based* approaches, leverage Integrated Development Environments (IDE) to record changes made by developers during API migration. These changes are stored as a list of operations that are then replayed by other developers who are facing a similar migration context [56]. Second, *matching-based* approaches derive API mappings based on source code differences between two APIs (including two versions of the same API). Third, *wisdom-of-the-crowd* approaches consider a code base of pairs of projects that already went through a migration process. They extract API mappings from the code base, assuming that studying how developers managed to go from one API to the other provides valuable information.

In the following sections, we dive into the state of the art of each of these approaches. Table 4 and Table 5 present a high-level overview of the studied approaches, together with details regarding their publication year, expected inputs and outputs, type of mappings (i.e., one-to-one, one-to-many, many-to-one, many-to-many, and API usage), assumptions, and availability of a tool.

10.1.1 Record-and-replay Approaches

Record-and-replay approaches are pioneers in the API migration field. The main idea behind them is that developers' actions can be recorded during a migration task to be replayed afterwards in a similar context.

Table 4: API switching and upgrading approaches.

Approach	Year	Input(s)	Output(s)	Mappings	Assumptions	Tool
JBuilder [44]	2005	API operations (trace file), and client source code	Modified client source code (by replaying operations)	Traces	• Developers and users rely on the same IDE. • API changes are directly mapped to API uses.	–
CatchUp! [40]	2005	API operations (trace file), and client source code	Modified client source code (by replaying operations)	Traces	• Developers and users rely on the same IDE. • API changes are directly mapped to API uses.	–
Diff-CatchUp [110]	2007	Source code of two API versions, and client source code	API changes, plausible API replacements, and API usage examples[2]	One-to-one and API usage	• All API migration issues are reported by the compiler. • API changes are reflected within the API itself. • Deprecated and visibility-restricted entities can be treated as removed.	–
SemDiff [24]	2008	API source code repository and client source code	API changes and API replacement (method invocations and confidence value)	API usage	• Modified API methods are not root methods[3]. • Calls to removed methods are replaced by one or more method invocations. • Confidence value threshold is known.	[25]
AURA [108]	2010	Source code of two API versions	API migration rules	One-to-many and many-to-one	• A removed method is replaced by one or more methods. • One or more removed methods are replaced by one method. • Only API method evolution is considered.	[109]
HiMa [56]	2012	Revisions between two releases of an API	API migration rules	Many-to-many	• Revision comments contain information about API migration.	[55]

Table 5: API switching and upgrading approaches.

Approach	Year	Input(s)	Output(s)	Mappings	Assumptions	Tool
MathFinder [86]	2014	API-independent mathematical expression	Pseudo-code comprised of API methods	API usage	• APIs and client programs count with complete and valid test suites. • API developers provide mappings between API and API-independent data types.	[85]
Schäfer et al. [89]	2008	Client codebase	API migration rules	One-to-one	• Learning thresholds can be computed based on parameter analysis. • There is an existing client codebase with API migrations.	–
LibSync [63]	2010	Source code of two API versions, client source code, and client codebase	Locations of affected client code and edit operations	API usage	• An API is only accessed via method invocations and inheritance. • There is an existing client codebase with API migrations. • Similarity thresholds are known.	–
Rosetta [36]	2013	Client codebase	API migration rules	Many-to-many	• Up to two methods are considered per mapping. • Client programs must be executed in similar ways. • Inference thresholds are known.	–
Teyton et al. [97]	2013	Revisions of two API versions and client codebase	API migration rules	One-to-many	• "There are few migration segments in a project history, with short lengths." • It is unlikely to have more than one API migration in a short time span. • Rollback of APIs do not happen frequently.	–

We highlight two main approaches in this category: JBuilder and CatchUp! JBuilder [44] is a development environment that supports the creation of Java applications, applets, servlets, and JavaBeans. It is able to record the refactoring operations performed in a given project. These traces are stored so developers can later reuse them, upgrading their own projects to the new API version. JBuilder provides a dialog box where local and global refactorings can be automatically applied to the client code. If the client code still reports errors after applying refactoring operations, JBuilder provides error insight capabilities to help in the migration process. For instance, if a method has been renamed and no automatic refactoring is available, then the developer can identify the broken code, check method alternatives, and choose the one that replaces the old method. This refactoring is then applied to all occurrences of the renamed method, and the action is stored for future use. This refactoring pipeline is known as *distributed refactoring*.

CatchUp! is an approach that captures API refactoring operations to replay them in other client projects that also need to migrate [40]. The case of deprecated methods is a particular case of API migration that must be considered. Henkel and Diwan account for this situation and propose ways to tackle this scenario. In fact, they estimate that around 59% of deprecated methods can be easily replaced with refactoring operations. With CatchUp!, refactorings made by API developers are recorded in a trace file by the IDE. Then, this file is shipped to the client developers, so they can replay the operations in their own code. With this strategy, deprecated methods related to refactoring operations can be deleted from the API, given that client code will automatically evolve according to the operations within the trace file. Although, there is no validation of the CatchUp! tool in the article that introduces it, the authors do show an application example where a developer performs a set of refactorings to the BCEL library. These operations are then captured by the CatchUp! tool in the Eclipse IDE, and an explanation of how developers can upgrade their own code is given.

The main advantage of these approaches is the possibility to directly capture the API refactoring operations. Consequently, precision and recall are positively affected. Nonetheless, these approaches are highly reliant on the IDE capabilities: both API developers and API users become dependent on the underlying technology. To favour portability, new IDE-independent techniques should be provided. We explain some of these techniques in the following sections.

10.1.2 Matching-based Approaches

When thinking about supporting API migration, it comes naturally to check the differences between two API versions. The difference between them results in the set of operations that must be considered by API users when migrating to a new version (or a similar API). These operations or migration rules are derived by using different type of heuristics based on call-dependency analysis, text similarity, structure similarity, and various other metrics [108]. In this section, we present some of these approaches [24, 56, 86, 108, 110], reviewing the employed techniques, their strengths, and drawbacks.

Xing and Stroulia [110] present Diff-CatchUp. As its name suggests, Diff-CatchUp is a tool that detects changes between two API versions based on model differencing. What API changes have been made? What are their plausible replacements in the new version? Are there any examples? These are the main questions that Xing and Stroulia want to answer with their tool. With this goal in mind, Diff-CatchUp considers three main phases. First, to compute API differences, they use UMLDiff [111]. API differences may refer to renamings, moves, removals, and additions of certain API elements (i.e., subsystems, packages, classes, fields, methods); attribute changes (e.g., deprecation, visibility); and changes in containment and inheritance relations. Second, when a migration problem is detected in the client code, Diff-CatchUp creates a set of suggestions. These suggestions include the origin of the change, possible replacements, and a set of usage examples (taken from the API code history itself). Third, these suggestions are ordered and displayed to the developer according to four sets of heuristics (i.e., name, inheritance, usage, and association) and the similarity between API entities' names.

When deprecating API elements or restricting their visibility, Diff-CatchUp treats them as removed elements. Diff-CatchUp is evaluated using two case studies. The tool results are compared against a set of manual API mappings, identified by a group of developers. If the manual change appears in the top-ten suggestions of Diff-CatchUp, it is considered as a successful recommendation. In the end, the authors report a success rate of 92.7% and 76.9% for the two case studies.

The idea of considering migration examples from within the target API, is also adopted by Dagenais and Robillard [24]. They introduce SemDiff, a recommendation system that suggests API upgrading mappings. To this aim, SemDiff analyses how the API adapts to its own changes. The generation of recommendations is triggered when a broken method invocation is found within client source code. Similar to Diff-CatchUp, each recommendation is accompanied by a replacement method, a confidence value, and usage examples. The confidence value is a key aspect for ordering the recommendations retrieved to the API user. A threshold over the recommendation's confidence value is set to filter false positives. To evaluate SemDiff, the authors consider a case study with one evolving API (Eclipse Java Development Tool) and three client-projects requiring API adaptations. Automatically derived mappings are compared against the manual migration performed for each one of the client projects. As main results, SemDiff found useful recommendations for 97% of broken methods. In addition, SemDiff's results are compared against results obtained with other two state-of-the-art tools (RefactoringCrawler [28] and Kim et al. [46] tool) for detecting code changes. Instead of improving the results of the latter, SemDiff provides complementary information mostly related to the identification of non-refactoring changes.

Wu et al. [108] propose AURA (AUtomatic change Rule Assistant), an approach that identifies API migration rules by combining call dependency analysis and text similarity. Research is also focused on identifying one-to-many, many-to-one and simply deleted mappings, which according to their own evaluation cover 8.08% of migrated methods of four real-world systems. In the overall approach, AURA first identifies method invocations through call dependency analysis. Afterwards, method signatures are tokenized and the similarity of any two methods is computed using first their signatures, then their Levenshtein distance (which indicates how different the methods are), and lastly, longest common subsequence (which indicates how common the methods are). It is important to note that a mapping may have multiple target methods as candidates. The final mappings are derived based on a confidence value computation, which –contrary to SemDiff [24]– does not imply the specification of additional thresholds. To evaluate the approach, authors use AURA to migrate five Java APIs. They compare AURA's behaviour against results obtained with Kim et al. [46] and Schäfer et al. [89] approaches, as well as against SemDiff [24]. In the first two cases, there is a relative recall improvement of 53.07%, and precision tends to be the same (relative difference of -0.1%). In the latter case, AURA has a precision of 92.86% whilst SemDiff reaches a 100% precision.

Meng et al. [56] also conduct an API migration research based on call-dependency analysis. Nevertheless, they take a step further: they do consider whole revisions in the version control system of an API. All revisions between two releases of an API are studied and aggregated. Then, both comments and source code are analysed. The main result of this work is enclosed in the HiMa (History-based Matching) tool. This solution derives API evolution rules after matching pairs of consecutive revisions of the corresponding library. Comments of the target revisions are parsed, and raw migration rules are extracted. These rules are related to an action (i.e., addition, deletion, modification) and a set of API entities (i.e., class, method). After their identification, rules are validated against source code according to a set of heuristics. Many-to many mappings, as well as simply-deleted methods can be identified with HiMa. HiMa is compared against AURA when migrating three Java projects. In most cases, HiMa presents better precision and recall values, however it shows a high computational cost.

Lastly, we consider a more specific approach that tackles the problem of discovering and migrating math APIs. Both API switching and upgrading are considered. The name of the approach is MathFinder and is introduced

by Santhiar et al. [86]. In this section, we will only highlight the API migration capabilities offered by this approach. In general, MathFinder heavily relies on unit tests, to such an extent that it uses tests as a description of method semantics. These unit tests are mined to identify methods that can solve a user problem. To this aim, a user specifies a math expression in an API-independent way. This expression is evaluated by a math language interpreter, working as a user query. MathFinder then extracts subexpressions from this query and mines unit tests of APIs. A mapping among variables in the math subexpressions and method parameters is performed (aka. actuals-to-formals mappings). To successfully perform this task, the API developer must provide code that maps the interpreter data types into the API data types. Afterwards, methods are ranked according to the number of times the corresponding unit test provides the same results as the ones obtained with the math interpreter. Lastly, pseudo-code is synthetized and retrieved to the developer. To test their approach, Santhiar et al. use MathFinder on a collection of math expressions. They observe a precision of 98% when synthesizing pseudo-code. They also migrate an internal API of Weka, a machine learning library for Java programs. In 94% of the cases they get a valid migration suggestion.

The aforementioned approaches are promising, given that they not depend on IDE environments or big source code corpora. However, reaching high precision and recall is one of the biggest challenges they face. Arbitrary or complex heuristics are required to enhance the derived results. Moreover, as developers, we need to somehow guarantee that the suggested API mappings or migrated code are valid. Some further steps are needed to provide practical solutions. Improving the way tools are evaluated and the way API mappings are verified are relevant aspects to current research. More recent approaches leverage version control systems and big codebases to obtain mappings based on the wisdom of the crowd. They are explored in the next subsection.

10.1.3 Wisdom-of-the-crowd Approaches

The wisdom-of-the-crowd category is an apparently new group that was actually mentioned by Meng et al. [56] in their HiMa article. They referred to it as a sub-category of the matching-based approaches. Their differentiating factor is that these solutions infer API mappings from client programs (aka. instantiation code) instead of only considering two API versions. Given the increasing popularity of software repositories mining, and the need to leverage the so-called wisdom of the crowd, we extract this new category and we consider it in isolation. As its name suggests, the main goal of the solutions listed in this category is to extract knowledge from the collective intelligence of a group [94]. In this section, we present wisdom-of-the-crowd approaches [36, 63, 89, 97] appearing in the early 2010's (apart from a few exceptions) until our days.

Schäfer et al. [89] approach can be considered as the pathfinder of the wisdom-of-the-crowd solutions. They aim at mining API migration rules from a codebase containing already migrated versions of client code. They go beyond identification of refactoring operations, noticing that between 10% and 34% of changes within three APIs, are related to what they call conceptual changes. The authors are especially aware of complex software evolution situations, where for instance, outdated code is not removed immediately but it is first marked as deprecated [28]. This is also the case when one code unit is affected by multiple modifications (e.g., renaming and moving). The whole approach is based on association rule mining [10]. The authors consider structural context to filter API usages in clients' code. They also define four change patterns to avoid the creation of unwanted migration rules. Finally, they ignore unchanged API usages to avoid the introduction of noise in the data. To improve the interestingness of the derived rules, minimum support (i.e., number of transactions that contain all items in the rule) and confidence (i.e., percentage of transactions that contain the antecedent of the rule and the whole consequence) thresholds are required. The approach is evaluated with three Java projects. A qualitative comparison with RefactoringCrawler [28], is also performed. They show that results obtained with both techniques are complementary. The reported precision of the tool is 86.7%.

As in the case of Diff-CatchUp, Nguyen et al. [63] introduce a model-differencing approach to identify changes between API versions. However, this technique is classified as a wisdom-of-the-crowd approach given that it considers a set of client programs that have already been migrated to a different API version. The approach is called LibSync and it considers four main phases. First, the two API versions are represented as trees, so they can be aligned. The alignment is done based on text similarity and a mapping algorithm based on UMLDiff [111]. Addition, deletion, renaming, moving, and modification actions might be identified. Second, API usages are extracted from the client codebase. The authors assume two different kinds of usages: method invocations (aka. API i-usage) and class inheritance (aka. API x-usage). These usages are then represented as Groum (GRaph based Object Usage Model) graphs [66]. A Groum is a graph representation of API usages. Each node in the graph represents an action (e.g., method invocation, field access) or a control point (e.g., conditional, loop). In addition, edges represent existing control or data dependencies among nodes. Third, API usages are aligned through text-similarity and graph-alignment algorithms. Finally, LibSync recommends a set of locations in the client code that should be modified, as well as the corresponding edit operations. LibSync is evaluated on three open source projects, and authors report a precision that oscillates between 97% and 100%. The tool is also compared against Kim et al. approach [46], claiming that LibSync is more accurate.

Gokhale et al. [36] introduce Rosetta. This approach aims at supporting the automatic creation of a database where mappings between any two target and source APIs are stored. The approach considers four main steps. First, Rosetta collects application pairs. This means that a database of applications written in both source and target APIs is built. Second, each application pair is executed in similar ways, using similar features. A log of traces, mainly including method invocations, is recorded in parallel. The main output of this phase is a database with trace pairs related to the target applications. Third, traces are analysed, and an inference algorithm is used to derive mappings from the target traces. A probability related to the likelihood of having such mappings is also retrieved. The inference algorithm considers various attributes to compute probabilities, namely call frequency, call position, call context (i.e., neighbour methods), and method names. The latter uses the Levenshtein edit distance to compute the similarity. Fourth, the inference of multiple traces is combined using the weighted average of the probabilities. The more data there is regarding an inferred mapping, the stronger the confidence in that mapping. The approach is evaluated with 21 JavaME and Android pairs. These pairs were independently developed. In 70% of cases Rosetta returned at least one valid mapping in its top-ten list. Lastly, in 40% of the cases the top-ranked result was a valid mapping.

The last approach presented in this section corresponds to the research done by Teyton et al. [97]. In this solution, API mappings are extracted from versions of projects that already underwent a migration process. There are three main tasks performed by this approach. First, API migration is identified in a given project history. This approach considers chunks, which are added, removed, or removed and added lines of code, instead of considering method scopes. Second, code changed during the migration segment is identified by means of using textual differencing. Only the commits that handle source and target API methods are studied to generate candidate API mappings. Third, a filtering method is used to improve the precision of the solution. To test the approach, a corpus of 11,598 open-source Java projects is considered. These are non-empty projects downloaded from GitHub, GoogleCode, and SourceForge. The main goal is to detect API mappings between four pairs of Java APIs (i.e., commons.io-guava.io, commons.lang-guava.lang, org.json-gson, jmock-mockito). The tool detected 135 correct mappings, having a precision of about 50% and a recall of 85%. Moreover, 33% of the mappings contained in this set are one-to-many mappings. The approach is also compared against Schäfer et al. approach [89], outperforming their results.

To wrap up the section, we observe that the main strength of wisdom-of-the-crowd approaches resides on leveraging collective knowledge from codebases. Effort and time spent by multiple developers in API migration tasks are mined from the built repositories. Yet, these approaches also need to explore different solutions to

Table 6: Cross-language API migration approaches.

Approach	Year	Input(s)	Output(s)	Mappings	Assumptions	Tool
TMAP [70]	2015	Documentation of two APIs	API migration rules	One-to-one	• API documentation is available and complete.	[69]
DeepAM [39]	2017	Bilingual client codebase	API migration rules	One-to-one	• Source code documentation is valid and complete.	–
MAM [113]	2010	Migrated client codebase	API migration rules	Many-to-many	• There is an already migrated client codebase. • Semantics of API entities can be inferred from names. • Similarity thresholds are known.	–
StaMiner [62]	2014	Migrated client codebase	API migration rules	Many-to-many	• There is an already migrated client codebase. • Only methods with similar qualified names can be mapped. • Confidence thresholds are known.	–

reach high precision and recall as it is the case in matching-based solutions. They heavily rely on the assumption of having a stable client codebase with multiple projects migrated to different API versions.

10.2 Cross-language API Migration

Often, programs need to be ported to different programming languages or technological platforms, so they can reach their target users. This is why cross-language API migration is required. However, this is not a trivial task: researchers in the field must account for several variables such as the syntactic differences between the involved programming languages, and semantic differences between equivalent APIs developed by different vendors. There is a close similarity between the cross-language API migration and the API switching and upgrading problems. This similarity must be understood in terms of common goals and techniques. Consequently, it is handy to adopt some of the categories exposed in Section 10.1 to classify existing approaches. We only consider *matching-based* and *wisdom-of-the-crowd* solutions. The *record-and-replay* category is kept aside, mainly because of the absence of research responding to the category description. The following sections introduce some of the existing approaches in both categories. Table 6 and Table 7 show the studied solutions. Publication year, expected inputs and outputs, type of mappings (i.e., one-to-one, one-to-many, many-to-one, many-to-many, and API usage), assumptions, and availability of presented tools are also shown per approach.

Table 7: Cross-language API migration approaches.

Approach	Year	Input(s)	Output(s)	Mappings	Assumptions	Tool
mppSMT [60]	2015	Migrated client codebase	API migration rules	One-to-one	• There is an already migrated client codebase. • Migrated programs keep a similar directory structure and class/method names. • Involved programming languages share the same paradigm.	–
codeSMT [61]	2016	Migrated client codebase	API migration rules	One-to-one	• There is an already migrated client codebase. • Five (semantic) features are chosen to describe code context. • Selected features combination is sufficient.	–
Nguyen et al. [65]	2016	Migrated client codebase	API migration rules	One-to-one	• Context is important when mining API mappings. • APIs are frequently use in API usages.	–

10.2.1 Matching-based Approaches

After some of the first matching-based approaches were introduced for solving API switching and upgrading problems, researchers started to question how they could support API migration in a cross-language environment. Some of the underlying techniques of matching-based approaches targeting one programming language can be adopted. Despite the promising results that might be obtained with these techniques, aligning and mapping API and language entities that are semantically equivalent requires additional information that cannot be inferred by existing solutions. In this section, we describe some solutions [39, 70] that attempt to reuse matching techniques by just considering API documentation.

Pandita et al. [70] present TMAP (Text Mining based approach to discover likely API mappings). TMAP discovers mappings between alternative APIs usually written in different programming languages. The approach relies on the similarity of natural language of the APIs' method descriptions. TMAP maps target API descriptions into a vector space model. Certain text mining challenges such as confounding effects (managing terms that are too generic), weights (related to the importance of terms), and structure of documents are considered to produce good quality results. Then, the source API is used to query this model and find all possible mappings between the two APIs. Queries are automatically generated following text mining techniques. All matching documents are then retrieved and ranked according to the cosine similarity measure [93]. In the end, the developer accepts or refuses suggested API mappings. The approach is validated by discovering mappings for 10 classes when migrating from Java to C# APIs, and 5 classes when migrating from Java ME to Android APIs. Results are compared against the ones obtained with Rosetta [36] and StaMiner [62]. On average, TMAP finds relevant mappings for 57% more methods than the referenced solutions. No precision, recall, or performance statistics are provided.

Gu et al. [39] present a system based on deep learning that supports API migration between Java and C#. The system is called DeepAM (Deep API Migration). Although DeepAM relies on a client codebase, it is not classified as a wisdom-of-the-crowd approach. This decision is taken mainly because it does not rely on migrated projects to detect API mappings. The rationale behind this decision is that only 15 out of 11K studied projects have a manual translation to C#. However, they do rely on a codebase with projects written in Java and C# that do not necessarily have a direct translation to the other language. In general, DeepAM extracts API sequences (i.e., method invocations) per function in the codebase. Each API sequence is related to a description in natural language taken from the code comments. Then, sequence-to-sequence learning [19] is used to embed and correlate the set of pairs into a set of fixed-length vectors. These vectors are then translated into their corresponding description. Related API sequences are identified based on their semantic vectors. Lastly, SMT is employed to extract the cross-language API mappings between aligned sequences. The authors downloaded 442,928 Java projects and 182,313 C# projects with at least one star from GitHub. In the end, they extracted a dataset consisting of 9,880,169 API sequence-description pairs. To validate their approach, the authors use Java2CSharp API mappings as ground truth. The tool has a recall of 82.6% and 82.3% and a precision of 82.7% and 71.9% for class and method migrations, respectively. DeepAM is also compared against StaMiner [62] and TMAP [70]. DeepAM outperforms StaMiner in terms of recall and provides a similar precision. Sequence alignment is shown to be better than the one provided by TMAP.

As it can be seen, cross-language approaches based on matching leverage existing documentation to map API entities. However, they suffer from strong assumptions regarding the quality and completeness of documentation. Notwithstanding, these research contributions are worthy for future research, particularly when exploring hybrid approaches; semantics extracted from natural language can complement the findings derived from source code and program behaviour.

10.2.2 Wisdom-of-the-crowd Approaches

Most of the studied approaches in the subfield of cross-language API migration are classified in the wisdom-of-the-crowd category. This is mainly due to the hurdle that inferring API and language semantics supposes. Taking advantage of the collective intelligence and effort made by developers, when migrating programs from one language to another, seems reasonable and promising. In the remainder of the section, we present some of the approaches [62, 60, 65, 113] tackling the cross-language API migration problem, while relying on client codebases where API migration already happened.

The first approach that we consider is this section is MAM (Mining API Mapping) [113]. This approach mines API mappings between two target languages. MAM follows three phases to mine API mappings. First, it aligns classes and methods between the two implementations in different languages of an API. To this aim, they rely on name similarity between the API entities. Second, it mines mappings among API classes by considering the similarity between classes, fields, and method parameters names. The Levenshtein measure is used for this purpose. Third, MAM aligns API methods, which given its complexity (e.g., one method in an API could be mapped to more than one method in the other version), cannot just rely on text similarity. To solve this problem, MAM builds an API Transformation Graph (ATG) for aligning methods between two projects written in Java and C#. An ATG is a directed graph that captures information related to method inputs, outputs, and functionality (inferred from the method name). To validate the approach, MAM considers 15 open source projects written in both Java and C#. From the evaluation, authors show that MAM is able to mine 25,805 API mappings, with an accuracy of more than 80%. They also compare the set of mined mappings against the ones written for the Java2CSharp tool, which are used as oracle. They show a precision of 68.8% and 84.6%, and a recall of 77.9% and 73.9% for class and method relations, respectively. During the evaluation they discover new mappings that were not part of the chosen oracle.

Nguyen et al. [62] design StaMiner, a data-driven approach that mines mappings between APIs written in Java and C#, using a corpus of client projects written in both languages. They focus on the identification of API usage mappings, meaning that they can map code snippets with more than one line of code. StaMiner creates a Groum [66] per method in a client project (cf. Diff-CatchUp [63]). As an important remark, labels assigned to nodes in the Groum graph are created from the names of involved classes, methods, and control structures. StaMiner extracts sub-Groums in the method's graph, and then creates a sequence of symbols based on the label of each node (i.e., usage symbols). Sub-Groums are selected based on the number of involved variables to avoid exponential complexity. Afterwards, all sequences are put together to create the sentence of the method. These sentences are used in the symbol-to-symbol alignment algorithm, which uses Expectation-Maximization (EM) [48]. Finally, aligned symbols with an alignment score greater than a given threshold, are selected as API mappings. To build the client projects corpus, StaMiner considers nine open source projects written in both Java and C#. After performing an empirical evaluation, StaMiner shows an accuracy of 87.1% when mining API usages. Moreover, it outperforms MAM [113], having a relative improvement of 17.1% and 28.6% in precision and recall, respectively.

Nguyen et al. [60] also introduce a technique to support cross-language API migration, but this time using phrase-based Statistical Machine Translation (SMT) in three phases. The approach is called mppSMT (multi-phase, phrase-based SMT). SMT is a paradigm that supports the translation among languages based on statistical models created from a codebase [48]. The overall approach works in the following way. First, syntactic units are extracted from source code by traversing the program's Abstract Syntax Tree (AST). These units are then represented as syntactic symbols (aka. syntaxemes) and are put together in a sequence. SMT is used to align syntaxeme sequences between two target languages (e.g., Java and C#). Second, mppSMT processes lexical tokens within identified syntactic units. Each lexical token is annotated with its corresponding token and data type (aka. sememe). Third, lexical tokens are aligned between the target languages with SMT (aka. lexeme). In

the case of deep and complex syntactic structures (e.g., inner classes), mppSMT uses placeholders (a type of syntaxeme) to produce an independent alignment. For the migration process, syntaxeme sequences are migrated first, then sememe sequences, and finally lexical tokens. To evaluate the accuracy of mppSMT, the authors considered nine open source projects written in Java and then manually ported to C#. The evaluation of the approach shows that between 84.8% and 97.9% of the migrated methods are syntactically correct, and between 70% and 83% are semantically correct. Results were also compared against Java2CSharp tool, showing that mppSMT outperforms it in terms of semantic accuracy.

To improve mppSMT and semantic correctness in its derived results, Nguyen et al. [61] present codeSMT. The authors want to know if incorporating tokens context in the phrase-based SMT approach can improve its accuracy. To this aim, they study a subset of semantic relations surrounding code tokens, namely: co-occurrence association, data and control dependencies, visibility constraints, consistency among declarations, and entities accesses (similar to lexical cohesion). These features are integrated into the SMT process using the Direct Maximum Entropy (DME) approach. To test the impact of this approach, they consider eight open source projects written both in Java and C#. Afterwards, they conduct a first experiment where each feature is applied in isolation. As main result, they observe that co-occurrence association and data-control dependencies generate an improvement in semantic correctness of 18.3% and 18.5%, respectively. Then, they perform a second experiment where multiple features are applied at the same time. The authors detect that the combination of co-occurrence association, data-control dependencies, and visibility constraints causes an improvement of 19.1% and 26.4% in syntactic and semantic correctness. They conclude that considering code tokens context in SMT approaches does improve results accuracy.

After using a SMT-based approach, Nguyen et al. [65] introduce a statistical approach to mine API mappings. This approach uses Word2Vec vector representation [57] to characterize an API element based on its context or co-occurrence relations (aka. usage relations). Moreover, it aims at identifying similar structures in different APIs that reveal similar roles related to API elements. Similar geometrical arrangements between two APIs in the vector space show similar functionality and usage relations. Then, while considering a transformation method we can learn the projections between two different vector spaces (e.g., Java and C#). To test the approach, the authors considered an existing dataset of Java projects [11] and they downloaded 7,724 C# projects with +10 stars from GitHub. The main purpose corresponds to create a Word2Vec model for the JDK and .NET APIs. They also use Java2CSharp as an oracle do identify their approach accuracy. The approach derives correct API mappings in 42.8% of the cases when considering top-one suggestions, and up to 73.2% of the cases when considering top-five suggestions.

In addition to approaches directly tackling the cross-language API migration problem, other researchers have started to study factors that can improve existing solutions. For instance, Zhong et al. [112] study behavioural differences between equivalent APIs written in different programming languages. They highlight eight findings that should be taken into account when building migration tools. First, methods from different APIs handle null inputs in different ways. Second, string values may be constructed in a different way. Third, equivalent methods can have different input domains, so minimum and maximum values of target data types should be taken into account. Fourth, there are differences on the implementation and interpretation of a feature or functionality. Fifth, exceptions are handled in a different way. Sixth, constants may have different values. Seventh, there might be different inheritance hierarchies. Eighth, valid method sequences may become invalid after migration. All these concerns should be taken into account in future research.

Besides the already identified challenges of wisdom-of-the-crowd approaches (cf. Section 10.1), approaches designed for cross-language contexts should consider an additional quest: the ability to derive semantic equivalence among API and programming languages. Furthermore, completely relying on client codebases is a risky move, especially when there are just a few programs that are ported into different languages. To make it

Table 8: Refactoring detection approaches.

Approach	Year	Input(s)	Output(s)	Ref. Types [4]	Assumptions	Tool
RefactoringCrawler [28]	2006	Source code of two API versions	Log of refactoring operations	7	• Refactorings have a partial ordering relation. • Similarity is computed on API entity bodies[5]. • Similarity thresholds are known.	[29]
Ref-Finder [76]	2010	Source code of two API versions	List of refactoring operations	63	• Refactorings have a partial ordering relation. • Similarity thresholds are known.	[75]
RefDiff [92]	2017	Revisions of two API versions	List of refactoring operations	13	• Non-modified source code is not relevant in the analysis. • Similarity thresholds are known.	[91]
RMiner [100]	2018	Revisions of two API versions	List of refactoring operations	15	• Non-modified source code is not relevant in the analysis. • Two elements in two revisions that share the same signature represent the same code entity.	[101]

even more challenging, from this small set we only get information of a small group of APIs. Many questions remain open in this field, and more research effort is needed to provide useful support to developers.

10.3 Refactoring Detection

Finding a section tackling refactoring detection approaches might seem unnatural to the reader. Yet, we consider that it is important to have an overview of refactoring detection solutions that have been considered by the API migration state of the art. Some of the approaches that are mentioned in this section, are actually considered as part of the development or validation of solutions explored in Section 10.1 and Section 10.2. Moreover, detection of refactoring operations becomes relevant, especially for solutions that include some sort of model-differencing technique. It is also important to recall that 80% of API evolution changes are related to refactoring operations [30]. In the remainder of this section, we present four approaches [28, 76, 92, 100] used to detect refactorings between two versions of a project. Table 8 keeps track of the mentioned solutions, gathering information related to their publication year, inputs and outputs, number of detected refactoring types, assumptions, and availability of the solution as a tool.

We first introduce RefactoringCrawler, a tool developed by Dig et al. [28]. RefactoringCrawler detects refactoring operations between two versions of an API. The main goal of this tool is to produce a log where identified refactorings are added and can later be replayed in co-evolving client programs (cf. CatchUp! [40]). To deal with scalability and performance concerns, RefactoringCrawler relies on syntactic and semantic analyses. On the one hand, syntactic analysis detects changes between API versions. It is based on the Shingles encoding [14], an information-retrieval technique that generates a fingerprint per text component (e.g., method) and finds similarities among them. On the other hand, semantic analysis is used to select refactoring candidates obtained with the syntactic analysis. The former is based on reference graphs that show dependencies among API entities. The authors also address a set of challenges such as noise introduced when preserving backwards compatibility, and identification of complex refactorings consisting of more than one operation. Multiple thresholds (w.r.t. Shingles encoding and text similarity) must be specified by the user. Using three different Java projects, authors show that RefactoringCrawler achieves a precision that oscillates between 90% and 100%, and a recall between 86% and 100%. To compute this metric, they considered a set of manually identified refactorings as an oracle.

Prete et al. [45, 76] introduce Ref-Finder. This tool identifies refactorings between two versions of a given program. To detect refactoring operations, Ref-Finder defines a set of template logic rules per refactoring type. Composite refactorings set other logic rules as pre-requisites. A partial ordering relationship among refactorings should also be defined. Then, a logic programming engine is employed to detect refactoring occurrences in source code. Each program version is represented with a set of logic predicates, where code elements (e.g., packages, classes), containment relations, and structural dependencies (e.g., method invocation, subtyping) are specified. Ref-Finder computes the set of added and deleted elements between the two target versions. Some refactorings are identified based on text similarity using the longest common subsequence algorithm. To identify cases where a refactoring is applied, the antecedent of each logic rule is used as a query in a facts database where all changes are stored. Retrieved entities are labelled as refactoring instances. Ref-Finder is tested against two case studies. The first one considers Fowler's code examples [35]. In this case study, authors report a precision of 97% and a recall of 93.7%. The second evaluation considers version-pairs of three open-source Java projects. In this case, authors report a precision of 74% and a recall of 96% when identifying refactoring operations.

Aiming to provide valuable information during software evolution tasks, Silva et al. [92] introduce RefDiff. This tool identifies refactoring operations between two project revisions taken from a version control system. RefDiff considers two phases, that combine static analysis and code similarity techniques. First, a *source code analysis* phase builds a factual model representing modified code entities (e.g., classes, methods, fields). Second, the approach considers a *relationship analysis* phase where relations between modified entities are found. They distinguish between matching relations (e.g., entity renaming or moving), and non-matching relations (e.g., method extraction). In the case of matching relations, if there is more than one candidate for a given refactoring operation (e.g., a method could be renamed to two other different methods), then a ranking is applied based on text similarity; the candidate with the highest similarity is then chosen. To compute this similarity, code is represented as a bag or multiset of tokens, and the weighted Jaccard coefficient [18] is then calculated. The detection of non-matching relations is easier because API entity cannot be involved in multiple operations, then there is no need to solve conflicts. To evaluate the approach, the authors build an oracle consisting of 448 refactorings extracted from seven Java projects. Refactoring operations are artificially introduced in the corpus. After evaluating the tool, authors claim a precision of 100% and a recall of 87.7%. RefDiff is also compared against other three refactoring detection tools, namely Refactoring Miner [90], RefactoringCrawler [28], and Ref-Finder [45]. In this experiment, RefDiff outperforms all the other tools.

The last approach we present in this section is Tsantalis et al. [100] tool, RMiner (Refactoring Miner). This approach is able to detect 15 refactoring types. RMiner takes as input two revisions of a project (i.e., a commit and its parent commit from the git-based system), and outputs a list of refactoring operations between the two versions. It analyses only the delta between the two versions. RMiner aims at solving two state-of-the-art

issues: dependence on similarity thresholds and project building for analysis. Most refactoring tools assume that projects should be fully compiled, but as the authors pointed out, only 37% of the history of software projects can be successfully build [102]. The target technique uses an AST-based matching algorithm to detect refactoring candidates. No similarity threshold needs to be specified. Abstraction and argumentization pre-processing techniques are introduced to match more complex refactoring cases such as inline and extract method. To test the approach, the authors consider a dataset [90] reviewed by means of using a triangulation technique. The dataset considers 3,188 refactorings from 185 Java projects downloaded from GitHub. Its precision is of 98% and its recall of 87%. They compare their tool against RefDiff [92]; RMiner outperforms the target tool.

We have seen that refactoring detection approaches have reached important research milestones. This is a handy situation for researchers working in the API migration, mainly because part of their work can rely on the aforementioned results. Still, some improvements should be presented to enhance existing solutions. The number of considered refactoring types can be increased as well as the precision to detect both ordinary and complex refactoring types.

10.4 Program Transformation Languages

Some practitioners tackle the API migration problem, by manually specifying their own migration rules in a transformation language that helps automating the process. We consider that it is worthy to mention some of the existing transformation languages that have been used for that purpose. In fact, we believe that some of the underlying techniques, design decisions, and even challenges might be taken into consideration by future research. This section presents four transformation languages [22, 53, 67, 104] that are frequently mentioned in the state of the art. Considering that API mappings are known and can be specified in each language, is however the strong assumption underpinning all solutions.

The first programming language that we introduce is TXL (Turing eXtender Language). This language is presented by Cordy [22] as a programming language that supports rapid prototyping of notations and features of other programming languages. TXL was introduced in the early 1980's to avoid rebuilding all the compiler phases to add lexical, syntactic, semantic, and code generation elements related to new features of the language. This is achieved by using source-to-source transformations (aka. grammar overriding). The main idea behind TXL is to define a grammar, some syntactic modifications over it, and then implement a prototype by transforming the original language according to the defined modifications. In TXL terms, there is a base grammar, whose non-terminals can be modified by means of using grammar redefinitions. These redefinitions replace the original non-terminal definitions with a new ones. Developers can also provide pre- and post-extensions to the previous definition of the non-terminal. Patterns related to these extensions are specified in the concrete syntax of the target language, following the native patterns directives. TXL supports the definition of rules, functions, guards, lexical control, and global variables (usually used to represent symbol tables).

More recent and specialized approaches start to appear with Twinning. Nita and Notkin [67] introduce this language as a way to specify mappings used for migrating programs from a source to a target API. The authors identify two types of API adaptations. The first type is known as *shallow adaptation*. In this case, a client program that uses a source API is modified to directly use a target API. The second adaptation type is known as *deep adaptation*. In this context, client programs are modified to use a more abstract interface that has two different implementations: one pointing to the source API and another pointing to the target API. The authors claim that the tradeoffs between one and another are similar to the ones found when dealing with duplicated code and abstraction; the first one is easier to write but hard to maintain and vice versa. In both cases, the developer that aims to migrate client code needs to specify a one-to-one mapping between code snippets. Each mapping specifies source (aka. domain) and target (aka. range) return types, list of formals, and bodies. Finally,

appearances of the domain body in the program are replaced with the range body specified in the mapping. A limitation of the approach is that data or control flow is not taken into account when matching the mappings. A big challenge identified by the authors is to handle exceptions, especially when facing asymmetric mappings where the source and target bodies handle a different amount of exceptions. To manage this issue, Twinning allows the overlapping of exception type replacements. Twinning is applied to two case studies (i.e., API switching from Crimson to Dom4J and from Twitter to Facebook), proving itself useful but limited.

SWIN (Safe tWINning) is the successor of Twinning [67]. It is designed by Li et al. [53] as a transformation language for migrating Java programs between alternative APIs. SWIN enhances Twinning with more flexible transformation rules, formal semantics, and the type-safeness guarantee.[6] Additionally, SWIN is *convergent*, meaning that it only considers terminating and confluent transformations (the order in which transformation rules are applied matters). To support this, SWIN does not allow the introduction of two transformation rules with the same source pattern. In SWIN, developers specify a transformation rule set, where each rule has a set of metavariable declarations, source patterns and target patterns. A *metavariable* declaration considers the target and source types of involved variables; a mapping is then defined between the types of source and target APIs. To guarantee type safety, four conditions must be fulfilled. First, each pattern of a transformation rule must be well-typed [67]. Second, type mappings must form a function (i.e., one-to-one relation) [67]. Third, subtyping relations must be preserved. Fourth, transformation rules must cover all API changes. Moreover, they focus on supporting one-to-many mappings given its frequency in the API migration landscape, and its importance when thinking on a more general scenario (i.e., many-to-many mappings) [23]. To validate the language expressiveness, SWIN is used within three case studies (Crimson to Dom4j migration, Twitter4J to Sina Weibo Java migration, and Google Calendar API upgrading). The authors wrote 94 migration rules for the three scenarios, covering 97% of the methods that needed a transformation. They face problems when dealing with API entities splitting.

Lastly, Wang et al. [104] introduce PATL (PAtch-like Transformation Language). As in the case of SWIN [53], PATL is a declarative program transformation language that supports migration of Java projects between APIs. This language is part of a migration approach that considers many-to-many mappings, def-use relations preservation, and program normalization. This guided-normalization solution aims at changing (aka. *normalizing*) programs to a semantically equivalent basic form. Thus, this is the only program that developers should take into account when writing transformation rules. The solution also performs a set of semantics-preserving transformations (e.g., alias renaming, swapping) until a syntactic substitution is made to support the migration. In addition, a static checker is provided to preserve def-use relations. In PATL, developers specify a transformation rule sequence, where each rule has a set of metavariable declarations, source patterns, and target patterns. The authors apply their approach to three case studies (i.e., updating Google Calendar API, migration from JDom to Dom4j, and migration from Swing to SWT). To check the correctness of the transformation, they consider functional and performance tests, exceptions, and functional behaviour of the original and transformed program. In the end, they write 236 rules to cover 66 classes and 204 methods included in the case studies. 97.3% of target lines are successfully transformed with PATL, the rest require manual resolution.

The abovementioned approaches support the automatic migration of client programs when API mappings are known. These solutions can be combined with approaches presented in previous subsections, so a complete and practical system can be handed in to developers.

[6] The type-safety property implies both correctness (all transformed pieces in the code are well-typed), and completeness (unchanged sections are still well-typed after API migration).

10.5 Discussion

We have considered approaches that contribute to the API switching and upgrading problem, the cross-language API migration problem, the refactoring detection problem, and the program transformation problem. In the first two cases, we identify studies pertaining to three main categories: *record-and-replay*, *matching-based*, and *wisdom-of-the-crowd* approaches. The main advantage of *record-and-replay* approaches is that they reach a high precision when migrating APIs. This is mainly due to the storage and reuse of changes performed by developers during manual migrations. However, the presence of manually migrated code is a strong assumption that supposes significant effort from developers. Other drawbacks of these approaches are the dependence on the underlying IDE and the need of including additional resources in the packed API, which affects the portability of solutions.

Aware of the challenges faced by *record-and-replay* studies, researchers consider alternatives that focus on automated API analysis (*matching-based* approaches) and client codebases where manual migrations have been performed (*wisdom-of-the-crowd* approaches). In both cases, the idea of relying on developers' IDEs is discarded. Instead, *matching-based* approaches only consider the target APIs code to identify mappings. This removes the need for existing client codebases, and requires instead ad-hoc heuristics and source code analysis and comparison techniques. *Wisdom-of-the-crowd* approaches, on the other hand, require a large codebase of client code for a given API, which might not exist in all cases. Heuristics and code or text comparison methods are also used within these studies. In both cases, the goal is to reach high precision and recall by tweaking the heuristics, thresholds, and source code comparison techniques. Moreover, the evaluation and validation of the derived API mappings is rarely addessed by the aforementioned approaches. Identifying differences when analyzing source code and bytecode, as well as identifying non-trivial changes related to certain programming styles such as Inversion of Control (IoC) are still open research topics.

Concerning *refactoring detection* approaches, researchers have presented promising solutions where several refactoring types are identified. Well-defined conditions to spot certain refactorings are included in the current state of the art. Nevertheless, not all refactoring types as reported by Fowler [35] are considered by these studies. Identification of nested refactorings is also a complex task. Lastly, in regards to *program transformation languages*, some languages support the migration of client code to support a new API while preserving semantics and guaranteeing type-safeness. These studies are useful once API mappings are identified, so they can be included as part of the migration process.

So far, we discern the need to integrate capabilities offered by *API migration*, *refactoring detection*, and *program transformation* approaches to provide functional tools that help developers during API evolution and client code migration. Specifically, researchers must address corner cases such as supporting migration when facing APIs with certain programming patterns or styles (e.g., IoC). Many client projects do not have access to the source code of the APIs they use. Instead, they rely on JARs directly or through dependency management systems such as Apache Maven and OSGi. This requires inferring API mappings not only from source code as it is usually done, but also from bytecode.

To address these concerns, we introduce in the next section MARACAS, the framework for API analysis and migration that we develop in the context of CROSSMINER.

11 MARACAS: A Framework for API Analysis and Migration

MARACAS is a framework written in Rascal that aims at supporting automatic migration of Java client code following changes in Java APIs. Essentially, MARACAS can be classified as am *API upgrading* approach. That is, it tackles the API-client co-evolution problem by supporting developers of client projects in API migration

processes. It also incorporates theories and approaches coming from related fields such as *refactoring detection* and *program transformation*. This allows the construction of more holistic and practical tools when migrating code due to API evolution. Roughly, the MARACAS framework supports the following tasks: (i) identify both breaking and non-breaking changes between two versions of a Java API from both either its source code or its bytecode; (ii) generate mappings from one version to the other; (ii) detect affected Java source code in client projects; and (iv) evaluate if the suggested API mappings are considered in already migrated client projects.

The problem of automatically migrating the client code to comply with the updated version of an API is not addressed in this document. We refer the reader to the companion deliverables of **WP3** and **WP6** for more information on how the information computed by MARACAS is then used by other tools to recommend actual code snippets and documentation supporting the migration of client code.

Hereafter, we present MARACAS as a standalone framework and tool. The current version of MARACAS is hosted at `https://github.com/crossminer/maracas` under the CROSSMINER organization umbrella on GitHub. MARACAS is also integrated into CROSSMINER. Details about this integration are presented in **D2.8 – API Analysis Components**. For further information regarding API migration assitance, refer to deliverables **D3.5 – Mining Documentation and Code Snippets** and **D6.5 – The CROSSMINER knowledge base – Final Report**.

11.1 API-client Co-evolution

In this section, we first introduce a motivating example to better understand the API-client co-evolution problem (cf. Section 11.1.1). We use this motivating example to illustrate the problem of API evolution and migration (cf. Section 11.1.2). We end this section by presenting a high-level overview of our approach (cf. Section 11.1.3).

11.1.1 Motivating Example

We consider a simple Java API, API_{v1}, depicted in Listing 2, consisting of a single class `FileManager` which exposes a single method `fileExists(String)`. Later, as depicted in Listing 3, this API evolves into API_{v2} and the method `fileExists(String)` is renamed to `isFile(String)`.

```
package api;

public class FileManager {
  public boolean fileExists(String path) {
    return ...;
  }
}
```

Listing 2: FileManager API_{v1}.

```
package api;

public class FileManager {
  public boolean isFile(String path) {
    return ...;
  }
}
```

Listing 3: FileManager API_{v2}.

Now, let us consider a client project that is currently using API_{v1}. As depicted in Listing 4, the `Client` class creates a `FileManager` object (Line 7) and directly invokes the `fileExists` method (Line 8) declared in API_{v1}. The client project developers then decide to upgrade their dependencies, migrating from API_{v1} to API_{v2}. Afterwards, the client code does not compile anymore. When facing this small and highly scoped scenario a manual migration is trivial. However, when facing tens or hundreds of API modifications, changes might become untraceable. The lack of awareness of the API modifications might also result in a new layer of complexity when migrating client code. This is especially evident when trying to identify the proper replacement of

a given API member, like is the case in our renamed method scenario.

```
1  package client;
2
3  import api.FileManager;
4
5  public class Client {
6    public void foo() {
7      FileManager fm = new FileManager();
8      boolean exists = fm.fileExists("/etc/passwd");
9    }
10 }
11
```

Listing 4: Client code using the FileManager API$_v1$.

11.1.2 Problem Description

As stated by Lehman [50], the need for change is intrinsic to any software artefact. This property leads to what we currently know as *software evolution*; a phenomenon that implies a set of continual progressive changes in a set of software attributes [51]. These changes appear as a direct consequence of a changing environment. Software artefacts need then to adapt so they can deliver their expected value, while keeping or even improving their quality [51]. Considering our motivating example, we can observe that the evolution of APIs used by client code supposes the evolution of the client software itself. Of course, this is true just only when the client project needs to leverage security patches, features, and other improvements introduced in the newer version of the API. This problem is accurately defined as the API upgrading or API-client co-evolution problem.

Figure 12 presents an overview of the API-client co-evolution problem. This scenario shows a client project Client$_{v1}$ that uses API$_{v1}$. As time passes, the API evolves into API$_{v2}$. At some point, developers of the client code identify their own need to evolve so they can benefit from the new or modified features included in API$_{v2}$. Their main goal is then to migrate from Client$_{v1}$ to Client$_{v2}$, the latter using API$_{v2}$ (see grey area in the figure). Nevertheless, this migration process might result in a cumbersome task depending on the number and complexity of the changes introduced in the API. What are the changes introduced in the API? Which features have been removed? Are there planned replacements to these removed declarations? What changes can break the client code? What changes do not break the client code but can be tackled in a proactive way? These are some of the questions that developers relying on an evolving API are faced with.

Our main concern within the CROSSMINER project is to detect API changes and possible replacements of removed declarations. This actionable information is then provided to developers of client projects, so they are actively supported during their own migration process.

11.1.3 Approach Overview

To tackle the API-client co-evolution problem, we borrow some ideas related to models co-evolution from the Model-Driven Engineering (MDE) field.

MDE is a software methodology that treats models as first-class citizens, changing the focus from code to models that capture the underlying domain concepts. The main idea behind MDE consists in extracting and representing domain concepts and the relationships between them as *metamodels*. Instances of these metamodels

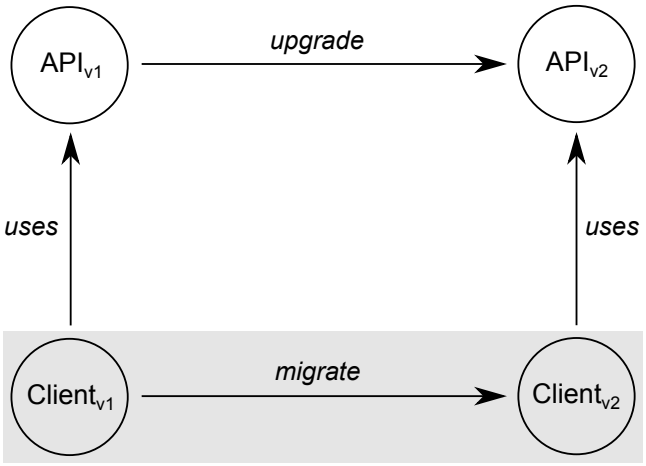

Figure 12: API-client co-evolution problem.

are then represented as *models* which are artefacts that conform to the corresponding metamodel. Constraints specified at the meta-level must be satisfied during the instantiation process [18].

As software artefacts themselves, models are also amenable to change. That is why recent approaches have aimed to support the evolution of metamodels and their instance models [20]. This process is known as models co-evolution or co-adaptation. Given two versions of an evolved metamodel, a difference model that captures all introduced changes is computed automatically. In the case of Cicchetti et al. [20], both breaking changes (changes that break the conformance of models) and non-breaking changes (changes that do not break the conformance of models) are identified. Afterwards, higher-order transformations generate model transformations composed of co-evolution actions that migrate the affected models.

To better understand and analyse the existing relations among elements in the API-client co-evolution problem (i.e., *uses*, *upgrade*, *migrate*), we aim at modelling them. Intuitively, if we consider a program as a model that represents a domain or universe of discourse [50], the application of MDE terminology in program evolution is straightforward. Therefore, we borrow some of the MDE ideas for API-client co-evolution problem. Figure 13 gives an overview of the artefacts and phases that must be considered in a typical API migration scenario. The main artefacts in the figure are introduced below:

API Projects. API projects (denoted $\texttt{API}_{\texttt{vX}}$ in Figure 13) are software projects that expose a public API meant to be used by client projects. The public API of such projects is the set of *access points* it exposes, e.g., the set of public classes, fields, and methods in a Java project that are accessible from the outside;

Client projects. Client projects (denoted $\texttt{Client}_{\texttt{vX}}$ in Figure 13) are users of API projects. They contain code that invokes part of the public API of an API project (its access points);

Usage model (❶). A usage model describes which part of a library's public API is being used in a client project. More specifically, it pinpoints both which access points of the API project are used in the client project, and which portion of the client project uses which access points;

Δ-model (❷). A Δ-model specifies which parts of the API have changed in between two target versions, including breaking and non-breaking changes. Thus, modifiers modifications of API access points, changes in types and list of parameters, added and removed superclasses and interfaces, as well as renamed, moved, and deprecated API members should be reported within the model;

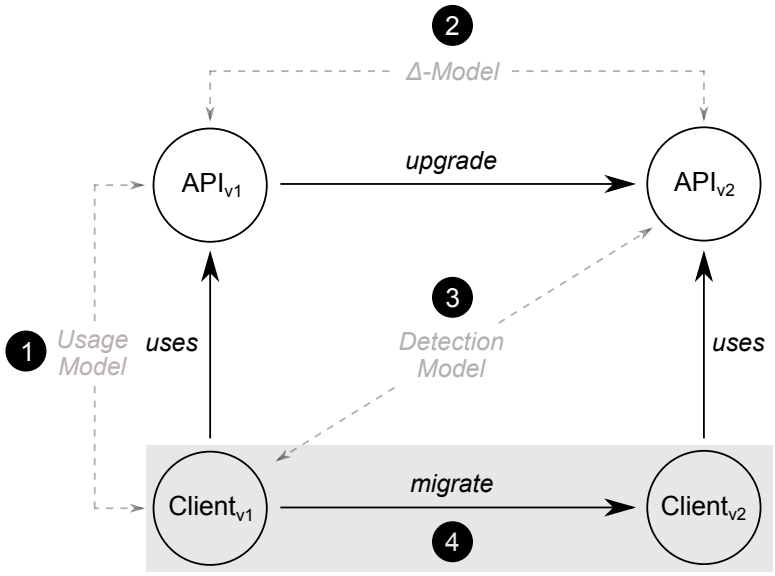

Figure 13: API-cliento co-evolution process and artefacts.

Detection model (❸). A detection model pinpoints which locations in a client project identified by a usage model are affected by the changes reported in a Δ-model; put another way, it detects which parts of client code are affected by the API evolution and must thus be migrated;

Migration (❹). Finally, the migration phase uses information extracted in the usage model, Δ-model, and detection model to (i) formulate recommendations regarding the migration of the locations in the client code affected by changes in the API and, whenever possible, (ii) automatically migrate these affected locations by patching the code.

Support for API migration can be classified in three ascending degrees of support and automation:

1. Detection. Enumerating the list of API uses in a client project which have to be migrated to comply with the updated version of an API;

2. Assistance. Providing developers with information extracted from heterogeneous sources (Q&A websites, forums, bugtrackers, commits, etc.) to assist them in the manual migration of their code base;

3. Migration. Automatically migrating the code base from version n of the API to version $n + 1$.

MARACAS is the backbone of API migration and analysis support in CROSSMINER. This framework supports the *detection* phase within the API migration process. Other partners in the project provide useful recommendations based on analysing textual sources related to OSS (cf. **D3.5 – Mining Documentation and Code Snippets**) and client codebases (cf. **D6.5 – The CROSSMINER knowledge base – Final Report**). These approaches cover the *assistance* phase of the migration process. The *migration* phase is out of scope for this project.

11.2 MARACAS Architecture

MARACAS is a tool and framework intended to support developers during the migration of a client project to a newer version of a given API. In other words, MARACAS is meant to support the API-client co-evolution as described in Section 11.1. Our approach is fully implemented in Rascal, a functional programming language that supports code analysis, transformation, and generation [47]. Aligned with the approach overview presented in Section 11.1.3, software artefacts are manipulated as M^3 models [13] which are automatically extracted from Java bytecode or source code with Rascal. A M^3 model captures language-agnostic and Java-specific facts from a project (cf. *Usage* model in Section 11.2.1). MARACAS specifies a set of different models that similarly captures the information required during model evolution scenarios. These models are implemented as Algebraic Data Types (ADTs), and different components are included in MARACAS' architecture to manage the corresponding models. The migration process is still opened to extension. In particular, this allows us to experiment with other tools of the literature (cf. Section 10) to evaluate their factual strengths and shortcomings.

We introduce the main models and corresponding ADTs that support our API-client co-evolution approach (cf. Section 11.2.1); as well as the main architectural components of our MARACAS framework (cf. Section 11.2.2). At the end of the section we also revisit our motivating example (cf. Section 11.2.3).

11.2.1 Models

In this section we introduce the main models of our API-client co-evolution approach (cf. Section 11.1).

Usage model

A *Usage* model specifies which parts of an API is used in a given client project. In our approach, *Usage* models are M^3 models which describe facts about source code in the form of a set of relations between source code locations.

Listing 5 presents the `M3` data type implementing the M^3 model. All elements within the `M3` datatype are relations, except for the `messages` attribute, which stores error and warning messages produced during the construction of the model. Some of these relations are language agnostic, specifically the first six relations, namely `declarations`, `types`, `uses`, `containment`, `names`, and `documentation`. The remainder are Java-specific relations storing information regarding `extends`, `implements`, `methodInvocation`, `fieldAccess`, `typeDependency`, `methodOverrides`, and `annotations`. Details on the semantics of each relation are presented by Basten et al. [13].

The main idea behind *Usage* models is to gather all tuples from the client M^3 models which use one or more API members. To get the corresponding information we consider the `typeDependency`, `methodInvocation`, `fieldAccess`, `implements`, `extends`, and `annotations` relations from the client M^3 model. Then we check if API members appear in the range (i.e., the right-hand side) of the abovementioned relations. That is, we cross-check the corresponding ranges against the set of API members that constitute the domain of the `declarations` relation in the M^3 model of the considered API. As of now, MARACAS considers fields, methods, constructors, classes, interfaces, and enumerations as API members. These elements can be identified by checking the scheme of the corresponding logical location: `java+field`, `java+method`, `java+constructor`, `java+class`, `java+interface`, and `java+enum`, respectively.

```
data M3 (
  rel[loc name, loc src] declarations = {},
  rel[loc name, TypeSymbol typ] types = {},
```

```
  rel[loc src, loc name] uses = {},
  rel[loc from, loc to] containment = {},
  list[Message] messages = [],
  rel[str simpleName, loc qualifiedName] names = {},
  rel[loc definition, loc comments] documentation = {},
  rel[loc definition, Modifier modifier] modifiers = {},
  rel[loc from, loc to] extends = {},
  rel[loc from, loc to] implements = {},
  rel[loc from, loc to] methodInvocation = {},
  rel[loc from, loc to] fieldAccess = {},
  rel[loc from, loc to] typeDependency = {},
  rel[loc from, loc to] methodOverrides = {},
  rel[loc declaration, loc annotation] annotations = {} )
  = m3 (loc id);
```

Listing 5: M3 data type.

Δ-model

Listing 6 depicts the `Delta` datatype that implements our notion of Δ-model. This model stores all API changes, including breaking and non-breaking changes. A filtering feature is provided to retrieve only API breaking modifications (cf. **D2.8 – API Analysis Components**). In essence, the model has one constructor `delta(tuple [loc from, loc to] id)`, which receives an `id` pointing to the physical location of the old and new versions of the analysed API. To capture API evolution information, we define a group of relations of type `rel[loc elem, Mapping[&T] mapping]`. `Mapping[&T]` is an alias for a tuple that has an original element `from` of type `&T`, a modified element `to` of the same type `&T` that maps to the former, a confidence score `conf`, and a similarity method or function `method`. The confidence score and similarity function are of particular interest when MARACAS faces mappings that cannot be directly derived from the bytecode or source code. Instead, a comparison between code snippets should be performed to infer possible replacements in the code base. For this purpose, we use similarity and distance metrics such as Levenshtein [52] and Jaccard [43], and plan to implement the GumTree algorithm [31] in the future. MARACAS uses these similarity methods to compute renamed, moved, and deprecated API mappings.

```
data Delta (
  rel[loc elem, Mapping[Modifier] mapping] accessModifiers = {},
  rel[loc elem, Mapping[Modifier] mapping] finalModifiers = {},
  rel[loc elem, Mapping[Modifier] mapping] staticModifiers = {},
  rel[loc elem, Mapping[Modifier] mapping] abstractModifiers = {},
  rel[loc elem, Mapping[list[TypeSymbol]] mapping] paramLists = {},
  rel[loc elem, Mapping[TypeSymbol] mapping] types = {},
  rel[loc elem, Mapping[loc] mapping] extends = {},
  rel[loc elem, Mapping[set[loc]] mapping] implements = {},
  rel[loc elem, Mapping[loc] mapping] deprecated = {},
  rel[loc elem, Mapping[loc] mapping] renamed = {},
  rel[loc elem, Mapping[loc] mapping] moved = {},
  rel[loc elem, Mapping[loc] mapping] removed = {},
  rel[loc elem, Mapping[loc] mapping] added = {} )
  = delta (tuple[loc from, loc to] id);

alias Mapping[&T]
  = tuple[
```

```
    &T from,
    &T to,
    real conf,
    str method
  ];
```

Listing 6: `Delta` data type.

Each relation within the `Delta` datatype represents a certain type of modification in the API evolution. Changes to access, final, static, and abstract modifiers are stored in the `accessModifiers`, `finalModifiers`, `staticModifiers`, and `abstractModifiers` relations, respectively. In all these cases, the mapping considers two elements of type `Modifier`[7] where we specify the initial modifier of the member and its new modifier. For instance, a method can undertake a change from `public` to `private`, or from `static` to non-static.

Changes in the parameter list of method members or the type of both methods and fields are registered in the `paramLists` and `types` relations, respectively. The mappings of both relations rely on the `TypeSymbol` datatype.[8] In the case of the `paramLists` relation, the mapping considers a **`list`** of type symbols, each one representing the type of a method parameter in the corresponding position. In the case of the `types` relation, the mapping stores the return type and type of both methods and fields.

Concerning classes, we represent class extension and interface implementation through the `extends` and `implements` relations. The `extends` mappings only consider replacements from one type to another, since Java does not support multi-inheritance. Contrary to the former case, the `implements` mappings consider sets of interfaces, which is important if the client extends types from the API.

Similarity functions and data representations mentioned in Section 11.2.2 are of great importance when building the `deprecated`, `renamed`, and `moved` relations. Mappings stored in these relations refer to origin and target locations that, once again, are the unique identifier of an API member. An overlap between the `deprecated` and the `renamed` and `moved` relations can be seen if new API elements have a high resemblance with both deprecated and renamed or moved members. This overlap is materialized as a set of new API elements appearing in different relation as target API members. Moreover, there might be multiple tuples suggesting different mappings for a same element. This happens due to the use of different similarity functions in a same round of computations, or because multiple elements have a similiarity score above the underlying similarity threshold. We keep this overlap so developers of client projects can use their own criteria to decide which mapping suits them best.

Finally, the `renamed` and `added` relations report all the removed and added declarations between the two versions of the API. It is expected that many of the domain elements of these relations appear as origin or target members in mappings of the previously described relations.

Detection model

Listing 7 presents the implementation of the `Detection` model in Rascal. The model has one constructor, namely `detection(loc elem, loc used, Mapping[&T] mapping, DeltaType typ)`. This model considers a client member `elem` that is currently using an old version of an API member. This old member is represented with the logical location `used`. The model also keeps a copy of the mapping identified during the construction of the Δ-model to provide more information to the developer during the detection phase. Additionally, given that the type of change is of clear importance, we add a `typ` attribute that specifies if the

[7] The `Modifier` data type is declared within the Rascal library as part of the M^3 `AST` module.
[8] The `TypeSymbol` data type is declared in the M^3 `TypeSymbol` module of the Rascal library.

developer is addressing a change in access modifiers, final modifiers, static modifiers, etc. The `typ` attribute considers each of the relations presented in the Δ-model through a type definition in the `DeltaType` data type (e.g., `accessModifiers()`, `implements()`, `renamed()`).

Even though the Δ-model captures all changes within an API (including internal changes that are not meant to be accessed by client code), we only detect changes of the public API. However, we distinguish between the so-called factual API and the intended API. A *factual API* is the list of API members that are actually being used by a set of clients. In contrast, an *intended API* is the set of all API members that were planned to serve as API access points by the API developers. In the case of the *Detection* model, we identify the factual API used by one client. Due to these definitions, it is expected that the factual API inferred by detection models identify a subset of the elements intended to be exposed by developers in the intended API. This information might be valuable to API developers, so they can have a sense of the impact of certain API changes. They can also know which parts of the API are not being used and might not deserve more maintenance, and to which extent the API is used according to their design.

```
data Detection = detection (
    loc elem,
    loc used,
    Mapping[&T] mapping,
    DeltaType typ
  );

data DeltaType
  = accessModifiers()
  | finalModifiers()
  | staticModifiers()
  | abstractModifiers()
  | paramLists()
  | types()
  | extends()
  | implements()
  | deprecated()
  | renamed()
  | moved()
  | removed()
  ;
```

Listing 7: *Detection* model data type.

Migration model

Listing 8 shows the datatype definition of the *Migration* model. As mentioned above, the *Migration* model considers two versions of a client's source code: before the manual migration, and after the manual migration. The *Migration* model has only one constructor `migration(loc oldClient, loc newClient, Detection d )`, which receives the physical location of the old and new versions of the client code and the *Detection* model related to the target API. First of all, the model stores the `oldDecl` and `newDecl` attributes, which point to the logical location of a member declaration within the old and new client code, respectively. This helps us to highlight the declaration of the client code that has been affected by an API change, as well as the corresponding declaration on the migrated client code that has been manually migrated by developers. The `oldUsed` and `newUsed` attributes are pointers to the API member that was previously being used by the client, and its expected new version.

Lastly, the `oldUses` and `newUses` sets are employed to gather all the API elements used within the `oldDecl` and `newDecl` client declarations. This data can be used to cross-check the mappings inferred by MARACAS in Δ-models with what the developers have done to migrate their code manually. For instance, if a Δ-model specifies that the method `a1` has been renamed to `a2`, we expect the *Migration* model to contain `a1` as part of the `oldUses` relation, and `a2` as part of the `newUses` relation. This enables us to check whether the developers' work matches what has been inferred by MARACAS.

```
data Migration (
  loc oldDecl = |unknown:///|,
  loc newDecl = |unknown:///|,
  loc oldUsed = |unknown:///|,
  loc newUsed = |unknown:///|,
  set[loc] oldUses = {},
  set[loc] newUses = {} )
  = migration (loc oldClient, loc newClient, Detection d);
```

Listing 8: *Migration* model data type.

11.2.2 Components

The MARACAS framework comprises seven different components. Figure 14 depicts the architecture of MARACAS, showing its main components and the data dependencies between them. These components are in charge of creating, managing, or visualizing the API-client co-evolution models used within MARACAS (cf. Section 11.2.1). We dive into each one of these components, explaining what their inputs and outputs are, as well as their interaction with other parts of the architecture.

API. The API component is the main access point to MARACAS features. All public features are declared within this component. The main responsibility of the component is to support the interaction with the `DeltaBuilder`, `DetectionBuilder`, and `MigrationBuilder` components. That is, the API is responsible of orchestrating the computation of and retrieving the Δ-model, the *Detection* model, and the *Migration* model required by an end user. This functionality is declared in the `delta(loc oldAPI, loc newAPI)`, `detections(loc oldClient, Delta delta)`, and `migrations(loc newClient, set[Detection] detections)` functions. Moreover, to support the analysis of changes based on the type of API affected members, we provide a `Delta` filtering functionality. This feature is reachable through the `classDelta(Delta delta)`, `methodDelta(Delta delta)`, and `fieldDelta(Delta delta)` functions, which–as their names suggest–retrieve a `Delta` model restricted to class or interface, method, and field members, respectively.

Delta builder. The Delta Builder component is one of the core components in MARACAS. It is responsible of creating a Δ-model between two versions of an API, given two M^3 models representing both versions of the library. To do so, the component interacts with the `M3DiffBuilder` and the `Matcher` components. The former is responsible of retrieving a M^3Diff model that captures the differences (both additions and removals) of the input M^3 models. The latter is in charge of creating a set of mappings to identify possible replacements of removed API members (e.g., identifying renamed or moved methods).

M3Diff builder. The M^3Diff builder component is in charge of computing differences between two input M^3 models. It inspects each relation of the input values to identify both added and removed tuples. To this aim, it computes the set difference for each M^3 relation. Let O and N be two relations of the M^3 models representing the old and new API version, respectively. Both of them contain the same M^3 relations

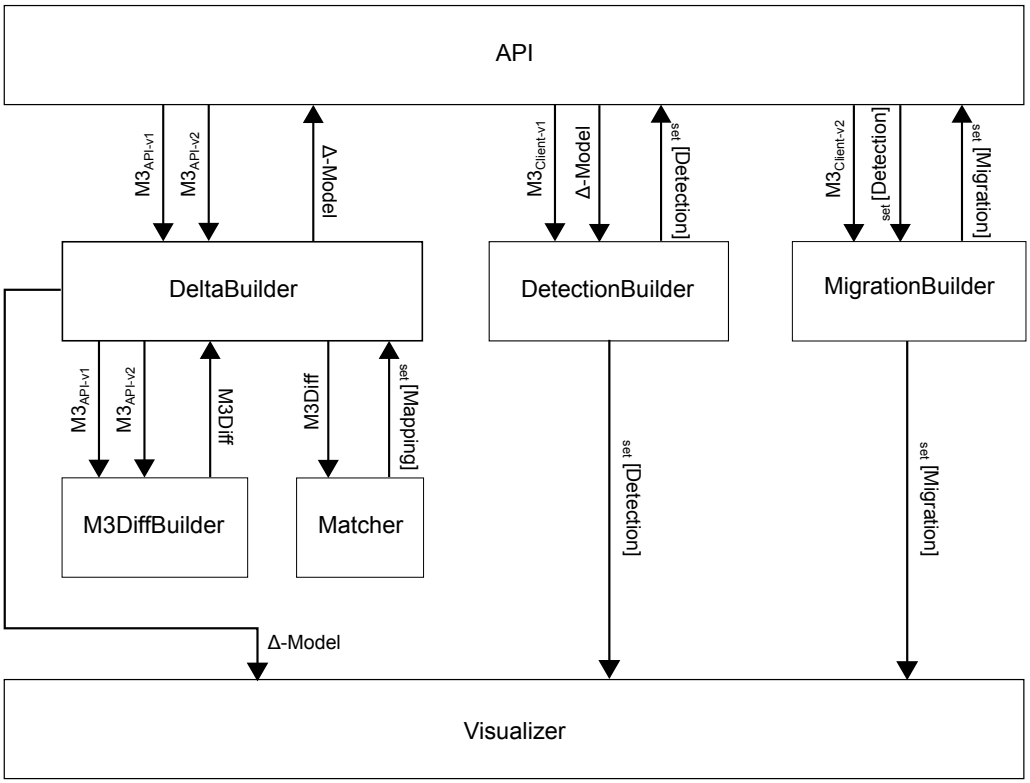

Figure 14: Maracas architecture.

in their corresponding models (e.g., `methodInvocation`). To compute the set of newly added tuples we compute the set difference $N - O$; to identify removed tuples we use the set difference $O - N$. Additionally, the component computes the sets of newly introduced and removed declarations according to each member unique logical location. The resulting M^{3}Diff is the baseline for populating the whole Δ-model between the two versions of the target API.

Matcher. The Matcher component is mainly used to compute the `deprecated`, `renamed`, and `moved` Δ-model relations. Its main input is a M^{3}Diff that specifies the members that must be compared and possibly matched. A similarity function and data representation are specified aside. This design is aligned with software similarity and classification theory. In order to compare two snippets of code, developers need to select a data representation (e.g., text, vectors, sets, graphs), and then a similarity function that operates on the chosen data type (e.g., Levenshtein on text, or Jaccard on sets) [16]. Therefore, plugging in various matchers that build on MARACAS infrastructure is pretty straightforward. End users can then specify their preferred similarity function and threshold (if any) within a configuration file parsed by MARACAS at run time.

Detection builder. The Detection builder component takes the M^3 model of the client code that needs to be migrated, as well as the Δ-model generated by the `DeltaBuilder`. The computed *Detection* model identifies the declarations within the client code that are affected by the API evolution stored in the Δ-model. We consider field access, method invocation, class subtyping, interface implementation, and annotation use as the main ways of accessing an API.

Migration builder. The Migration builder component is used to identify the real changes that a client underwent after manually migrating to the newer version of the target API. This model is needed to verify to which

extent a client follows the mapping suggestions computed by MARACAS in the Δ-model, and also to validate the correctness of the tool. In other words, the *Migration* model is helpful to both analyse existing migrations and validate the Δ-model data. In any case, it always serves as a feedback model to improve the accuracy and correctness of the tool.

Visualizer. The visualization of models computed by the aforementioned components is provided by the Visualizer component. This visualization support is included within MARACAS to further assist developers during their code migration. This goal is achieved by showing the Δ-models, *Detection* models, and *Migration* models in a user-friendly HTML-based interface. In the case of Δ-models, we display the content of each relation in independent HTML tables. Each table shows the logical location of the API member that was modified as well as its corresponding mapping, which considers the original and modified element, mapping score (in the case of `deprecated`, `renamed`, and `moved` relations), and similarity function if any (e.g., Levensthein, Jaccard).

11.2.3 Revisiting our Motivating Example

In this section, we describe the models inferred by MARACAS on our motivating example (cf. Section 11.1.1). As already mentioned, the function `delta` takes as input pointers to the old (cf.`|file:///home/ ... /FileManagerv1|`) and new (cf.`|file:///home/ ... /FileManagerv2|`) versions of the source code of the API and produces the corresponding Δ-model. Functions `classDelta`, `methodDelta`, and `fieldDelta` are offered to filter the derived Δ-model according to the corresponding API member type. In this example, we are interested in the `methodDelta` output. which is depicted in Listing 9.

```
1  Delta: delta(
2    <
3      |file:///home/ ... /FileManagerv1|,
4      |file:///home/ ... /FileManagerv2|
5    >,
6    removed={ <
7      |java+method:///api/FileManager/fileExists(java.lang.String)|,
8      <
9        |java+method:///api/FileManager/fileExists(java.lang.String)|,
10       |unknown:///|
11     >,
12     ...
13   >},
14   added={ <
15     |java+method:///api/FileManager/isFile(java.lang.String)|,
16     <
17       |unknown:///|,
18       |java+method:///api/FileManager/isFile(java.lang.String)|
19     >,
20     ...
21   >},
22   renamed={<
23     |java+method:///api/FileManager/fileExists(java.lang.String)|,
24     <
25       |java+method:///api/FileManager/fileExists(java.lang.String)|,
26       |java+method:///api/FileManager/isFile(java.lang.String)|
```

```
27      >,
28      0.87,
29      "Jaccard"
30    >},
31    moved={},
32    deprecated={},
33    paramLists={},
34    types={},
35    implements={},
36    extends={},
37    accessModifiers={},
38    abstractModifiers={},
39    staticModifiers={},
40    finalModifiers={},
41  )
```

Listing 9: Method-level Δ-model of the motivating example.

First, the Δ-model outputs pointers to the two versions of the analysed API: `FileManagerv1` and `FileManager v2` (Lines 2-5). Then, it specifies a list of API changes between the two versions. In our case, the `renamed` field is populated with a reference to the `api.FileManager.fileExists(String)` method (Line 23), specifying that it has been renamed to `api.FileManager.isFile(String)` in the new version of the API (Lines 24-27). This mapping is accompanied by a similarity score or confidence that in this case corresponds to `0.87` (Line 28), and the `Jaccard` similarity method (Line 29). Additionally, the `removed` and `added` fields are populated with references to the `api.FileManager.fileExists(String)` (Lines 6-13) and the `api.FileManager.isFile(String)` methods (Lines 14-21), respectively. All removals and additions detected between two versions of an API should be reported in these two relations, regardless of any further classification of a member element in one of the fields of the Δ-model (as it is the case in our renaming example).

This Δ-model can then be sent as input to the `detections(loc, Delta)` function, together with a pointer to the source code of the client code, which computes a *Detection* model pointing to the locations in the client code that must be migrated. Specifically, the location on Line 3 of Listing 10 points to the Line 8, characters 22 to 32, of the file `Client.java` (cf. Listing 4), which is the exact location of the method call that must be migrated. The *Detection* model presents the mapping identified with the Δ-model (Lines 4-12), and the type of API change, which in this case is `renamed()` (Line 13).

```
1  list[Detection]: [
2    detection (
3      |file:///home/ ... /FileManagerClient/src/client/Client.java|(145,10,<8,22>,<8,32>),
4      <
5        |java+method:///api/FileManager/fileExists(java.lang.String)|,
6        <
7          |java+method:///api/FileManager/fileExists(java.lang.String)|,
8          |java+method:///api/FileManager/isFile(java.lang.String)|
9        >,
10       0.87,
11       "Jaccard",
12     >
13     renamed() )
14 ]
```

Listing 10: *Detection* model of the motivating example.

With the provided information a client project developer can pinpoint the affected code and migrate it according to the proposed mapping. This manual (and in this case, trivial) migration is presented on Line 8 of Listing 11. The client project relying on the `FileManagerv2` now compiles and offers the expected functionality. Developers of the client project now make use of newly introduced features in the `FileManager` class. Specifically, they invoke the newly introduced method `api.FileManager.toURI(String)` (Line 10).

```
1  package client;
2
3  import api.FileManager;
4  import api.URI;
5
6  public class Client {
7    public void foo() {
8      FileManager fm = new FileManager();
9      boolean exists = fm.isFile("/etc/passwd");
10     URI uri = toURI("/etc/passwd");
11   }
12 }
13
```

Listing 11: Migrated client code using the FileManager API_{v2}.

We can now create a *Migration* model from the existing artefacts. In this particular example, the creation of the *Migration* model does not serve any particular purpose, but it shows how it should look like to have a better understanding of the artefact. Listing 12 depicts the *Migration* model obtained from analysing the two versions of the API and the two versions of the client project. In this case we can see that the `oldDecl` and `newDecl` point to the same API member, which corresponds to `client.Client.foo()` (Lines 3-4). The model also tells us that the `oldUsed` API member `api.FileManager.fileExists(java.lang.String)` (Line 5) is effectively replaced by the `newUsed` member `api.FileManager.isFile(java.lang.String)` (Line 6). The `oldUses` set considers the `api.FileManager.fileExists(java.lang.String)` method, which was the only API access point that was being used within the `client.Client.foo()` declaration (Line 7). Finally, the `newUses` set considers the `api.FileManager.isFile(java.lang.String)` and the `api.FileManager.toURI(java.lang.String)` methods, both of them being used within the new version of the `client.Client.foo()` declaration (Lines 8-11).

```
1  list[Migration]: [
2    migration (
3      oldDecl = |java+method:///client/Client/foo()|,
4      newDecl = |java+method:///client/Client/foo()|,
5      oldUsed = |java+method:///api/FileManager/fileExists(java.lang.String)|,
6      newUsed = |java+method:///api/FileManager/isFile(java.lang.String)|,
7      oldUses = { |java+method:///api/FileManager/fileExists(java.lang.String)| },
8      newUses = {
9        |java+method:///api/FileManager/isFile(java.lang.String)| ,
10       |java+method:///api/FileManager/toURI(java.lang.String)|
11     } )
12 ]
```

Listing 12: *Migration* model of the motivating example.

As of now, MARACAS only points to the precise locations of the code elements that must be migrated to comply with updated versions of an API. Further assistance covering API mappings, code snippets, and additional documentation is provided by CROSSMINER partners in deliverables **D3.5 – Mining Documentation and Code Snippets** and **D6.5 – The CROSSMINER knowledge base – Final Report**.

12 Case Studies

In this section, we evaluate the capability of MARACAS to reason about API evolution and migration using two complementary case studies. The first case study targets Google Guava, a mature and widely-used Java library primarily developed by Google. The second case study targets the plug-in API of SonarQube which allows developers to implement their own metrics and dashboard atop the SonarQube infrastructure, and which is at the heart of FrontEndArt's use case (cf. *Use Case 3: Software API Coupled Evolution* in **D1.2 – Evaluation Plan**). We choose these two case studies because the way their developers handle their evolution as well as the way users of these APIs interact with them is fundamentally different. While Guava is primarily used by instantiating types of the API and invoking their methods, the SonarQube plug-in API is primarily used by defining new types that extend or implement types of the API, enabling the SonarQube framework to automatically invoke the methods defined in client code (following the *inversion of control* style).

12.1 A Brief Historical Analysis of Google Guava

Guava is a collection of APIs for the Java programming language which provides support for basic utilities, collections, graphs manipulation, concurrency, I/O, reflection utilities, etc. Guava is currently the 4th most used library on Maven Central.[9] While Guava is an open-source framework, it was originally intended to be used mostly in Google's internal software projects. As a result, Guava's roadmap is set by Google developers, and developers rarely accept external contributions which helps ensuring the consistency and stability of the library throughout its evolution.[10] Guava was first released publicly on Maven Central in September 2011 with release 10.0; the latest version published on Maven Central is 27.1, released in March 2019.

In this case study, we study the course of evolution of Guava over the past 8 years, analyzing the 17 *major* revisions released over this period of time. For every major revision, we use MARACAS to parse and analyze the corresponding JAR file downloaded from Maven Central and, when appropriate, to compare two JARs corresponding to two subsequent versions. This case study allows us to showcase examples of analyses that can be conducted using MARACAS. As described in the companion deliverable **D2.8 – API Analysis Components**, these analyses are meant to be integrated in a dedicated API dashboard to help developers and deciders evaluate the maturity and stability of OSS projects and libraries.

Number and nature of declarations Figure 15 depicts the number of declarations (*all* declarations, including `private` and `protected` declarations that are not accessible from the outside) in Guava over 17 major versions. Unsurprisingly, most of the declarations are methods, followed by field declarations and types (classes,

[9] https://mvnrepository.com/popular
[10] https://github.com/google/guava/wiki/HowToContribute

interfaces, and enumerations). We observe that the number of declarations in Guava has steadily increased over the years with an average rate of growth per major version of ~3%, for a total of ~63% increase over 8 years. To compute this data, we simply rely on the M^3 models computed by Rascal to count the number of occurence of each kind of declaration.

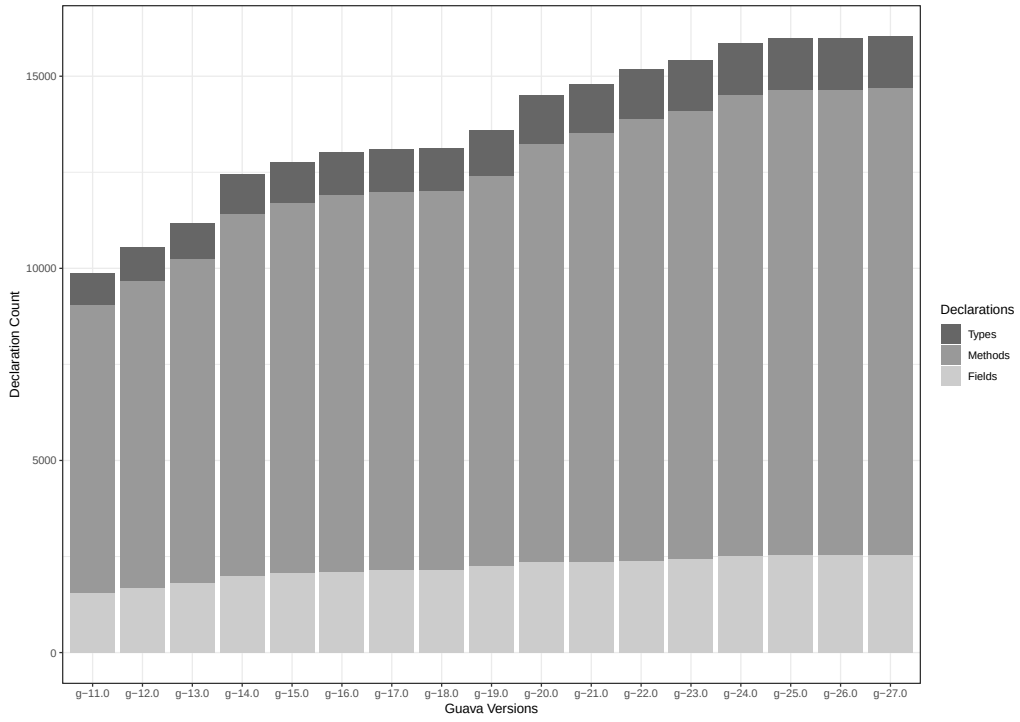

Figure 15: Type (class, interface, enumeration), method, and field declarations in Guava over time

Figure 16 depicts the evolution of the number of **`public`** declarations (classes, interfaces, enumerations, methods, fields) in the Guava API over time. This evolution naturally follows a similar trend as the total number of declarations in Guava (cf. Figure 15).

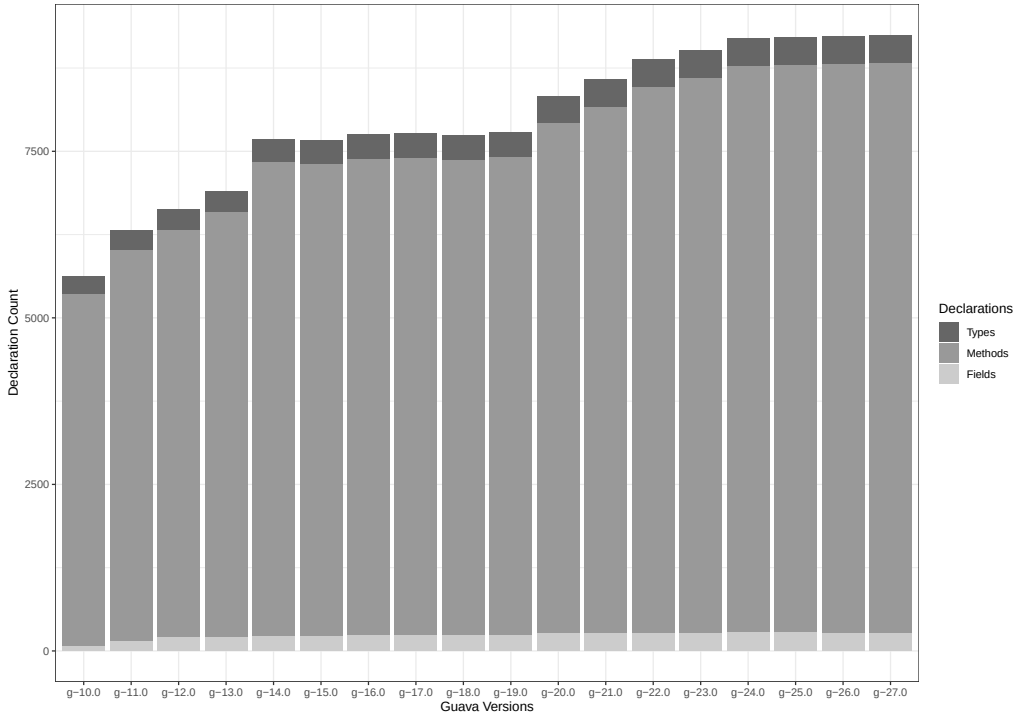

Figure 16: Public declarations in Guava over time

Breaking versus non-breaking changes For each couple of JAR representing version N and version $N + 1$, we use MARACAS to compute the Δ-model between these versions. We use the filtering function or MARACAS to compute the set of all changes, non-breaking changes, and breaking changes. The proportion of breaking changes related to the total number of changes varies from version to version and averages ~7% for each major version. This indicates that the bulk of changes (i) happens in protected parts of the library that are not exposed through the API or (ii) does not break the source and binary compatibility of the API. This can be explained by the particular attention that Google developers pay to not breaking their API, as well as the maturity and stability that the library has reached over the years. While the codebase and library keep evolving to incorporate new feature, bug fixes, and various other improvement, it does not impact the interface of the library significantly, nor its clients.

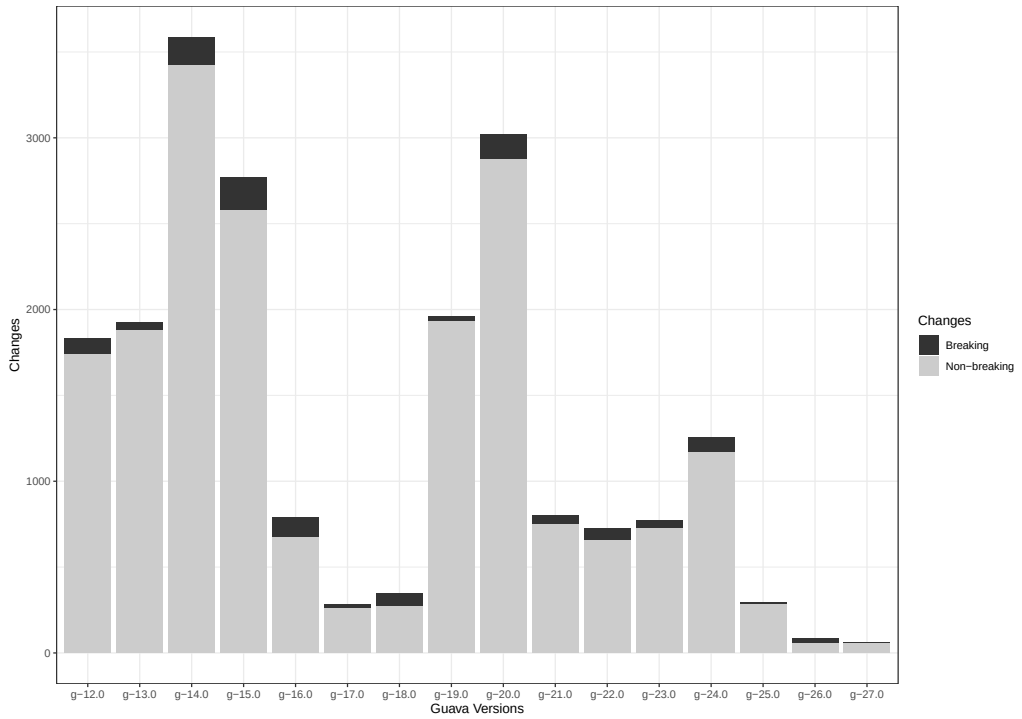

Figure 17: Breaking and non-breaking changes per release

Nature of breaking changes Unsurprisingly, and as most of the Java declarations in Guava are methods (cf. Figure 15 and Figure 16), the majority of breaking changes over time are related to methods, followed by types and then fields, as depicted in Figure 18. This indicates that support for migrating client code using the Guava API should primarily focus on the changes occurring at the method level, such as changes in modifiers, parameters lists, and return types.

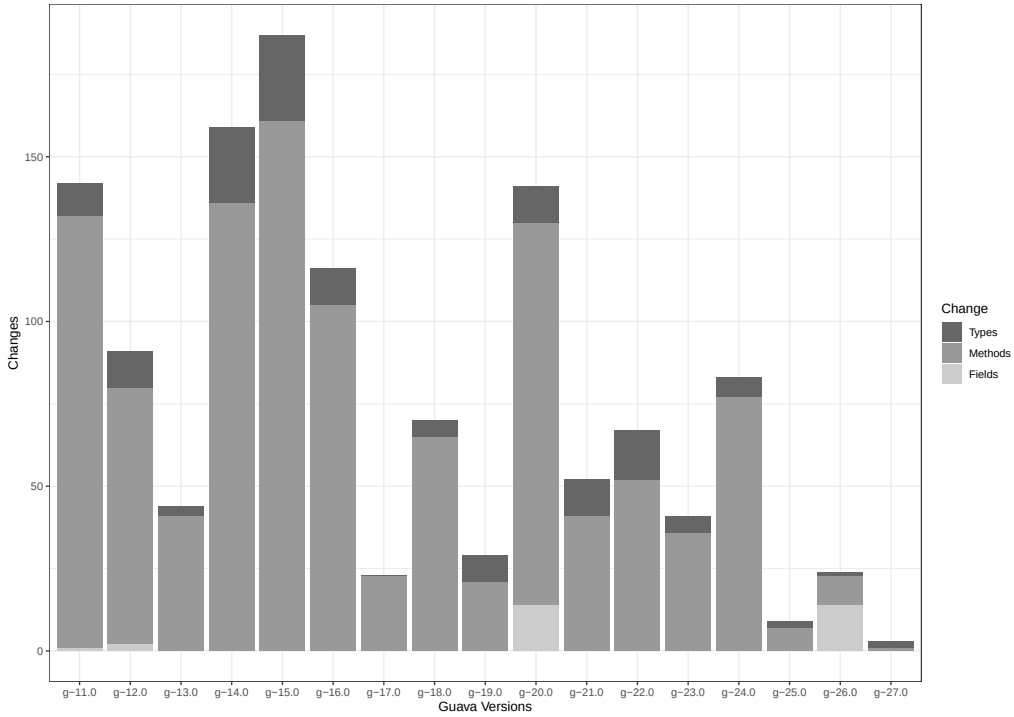

Figure 18: Nature of breaking changes over time

As shown in Figure 19, added and removed elements are the most common. While it may seem counter-intuitive that new elements (e.g., methods) introduced in an API may break it, remember that adding new methods in an abstract class that is being extended in client code forces the client code to provide an implementation of the new method. MARACAS is able to detect such scenarios and to report potentially breaking changes. Similarly, a large number of breaking changes are due to the introduction of new `implements` relations between classes and interfaces in the API: similarly to the previous case, classes in the client code that extends these classes should provide an implementation for the newly inherited methods if the class in the API does not provide a default implementation.

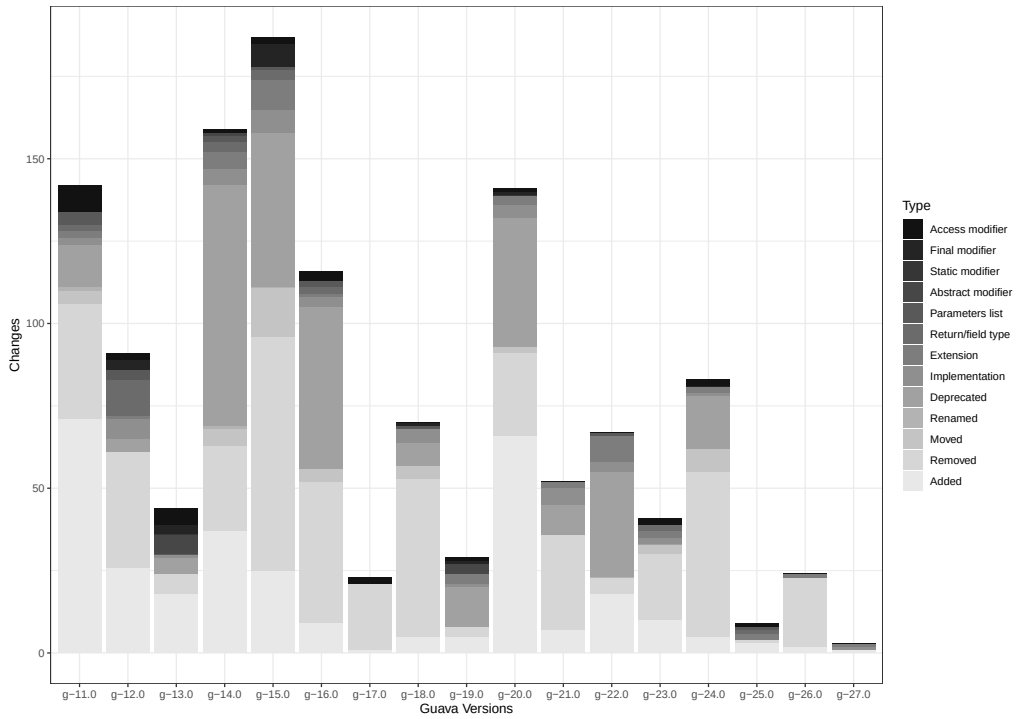

Figure 19: Types of breaking changes over time

In this first case study, we only study the evolution of changes in the API of Guava independently from the way it is used in client code. In constrast, the second case study introduced below focuses on the use of the SonarQube API by its clients.

12.2 SonarQube Plug-in API

SonarQube [7] is an open-source tool for supporting continuous inspection of source code developed by SonarSource. Its main features entail detection of bugs, vulnerabilities, and code smells by means of using code static analysis. Developers can leverage the SonarQube framework to implement their own metrics. The CROSSMINER partner FrontEndART develops SourceMeter and SourceMeter-based plugins atop the SonarQube infrastructure. All these plugins are offered to end users of the open-source platform. However, their main concern is the constant need to upgrade SourceMeter plugins when the API of the SonarQube framework evolves. SonarQube releases a major version yearly, and minor versions every two to three months. FrontEndART developers are also interested in the migration of deprecated API members, even though they do not necessarily break the plugins code.

In this section, we use MARACAS to explore the evolution of the SonarQube API between versions 4.2 and 6.7, for which SourceMeter developers have already manually migrated their plugins. Different change types including both breaking and non-breaking changes, are stored in our Δ-model. Afterwards, we detect how this evolution affects the SourceMeter plug-in for Java version 8.2. To select the abovementioned versions for both SonarQube and SourceMeter, we check available releases in the SourceMeter GitHub repository found at `https://github.com/FrontEndART/SonarQube-plug-in`. We then identify the latest SonarQube dependency version–that is, 6.7–and we then identify the two releases in which there is a change in the SonarQube dependency version in the POM file. In the end, we detect a SonarQube version change between SourceMeter version

8.2 and version 8.2v6.7, as labeled by FrontEndART developers. We wrap up the section by describing the manual migration performed by FrontEndART developers, and how it aligns with our approach. We included a SourceMeter pipeline in our tool to replicate the current analysis, which can be found at https://github.com/crossminer/maracas/tree/master/maracas/src/org/maracas/pipeline/sonarqube/sourcemeter.

The work described here is a first attempt at supporting concrete partners of the CROSSMINER project in their needs regarding API migration. In the remainder of the project, we will collaborate closely with FrontEndART and dive deeper into the specificities of SonarQube and their tool SourceMeter to assess precisely how MARACAS helps.

Breaking versus non-breaking changes. As mentioned above, we compute the Δ-model between versions 4.2 and 6.7 of SonarQube JARs taken from Maven Central. We then filter the resulting model to breaking changes only. Both breaking and non-breaking changes are plotted in Figure 20. The Δ-model shows 76.190 changes in the SonarQube framework between the two target versions. Contrary to Guava, 44% of changes in SonarQube are considered breaking changes. Nevertheless, it is important to note that we are not comparing two consecutive versions of the framework, instead there are two major versions and several minor versions in between the studied JARs. In any case, these number of changes are a real concern for FrontEndART developers.

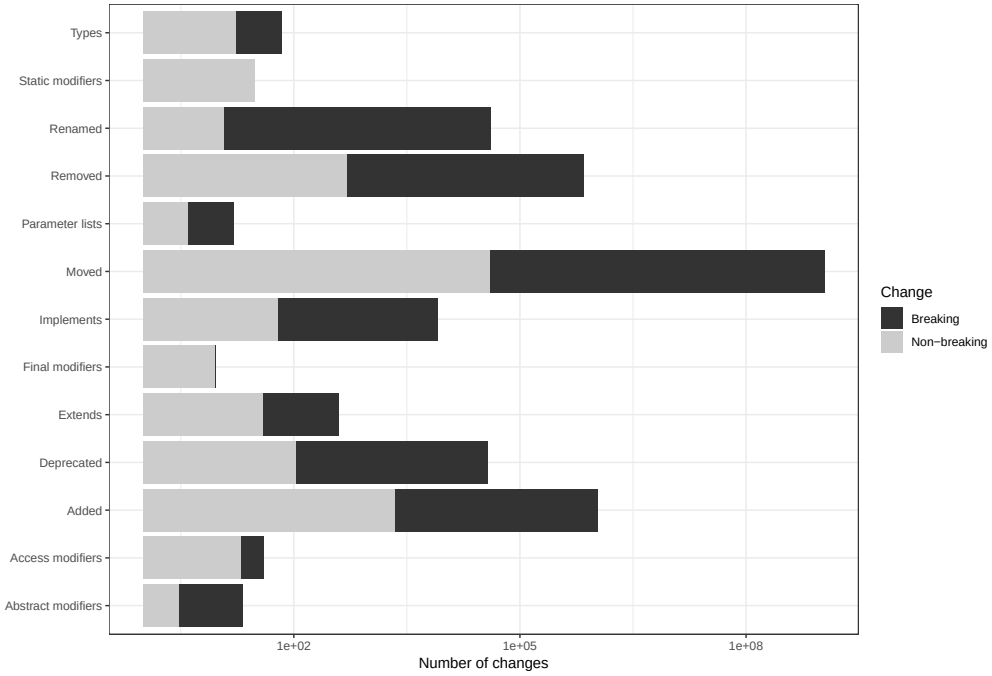

Figure 20: SonarQube changes between versions 4.2 and 6.7

Breaking change detections. We consider the core and Java plug-ins from SourceMeter version 8.2 as client projects to compute the set of breaking change detections. Thus, after computing the SonarQube Δ-model we compute the corresponding *Detection* models. On the one hand, we get 150 detection of breaking changes in the core SourceMeter plug-in. Figure 21 depicts all detections based on the change type (e.g., implements, deprecated) and on the the type of the affected API member (i.e., type, method, field). As it can be seen, the client project is using API types whose interfaces have changed or they have been deprecated. Given that SonarQube clients are supposed to follow an inversion of control style, the *implements* change detections are

expected. Some methods are also labelled as deprecated and one method has been added to an API type. The latter should be consider if one class in the client code is extending or implementing the API type providing the added method. 99% of detections are related to the use of API affected types, the remaining 1% concern method changes.

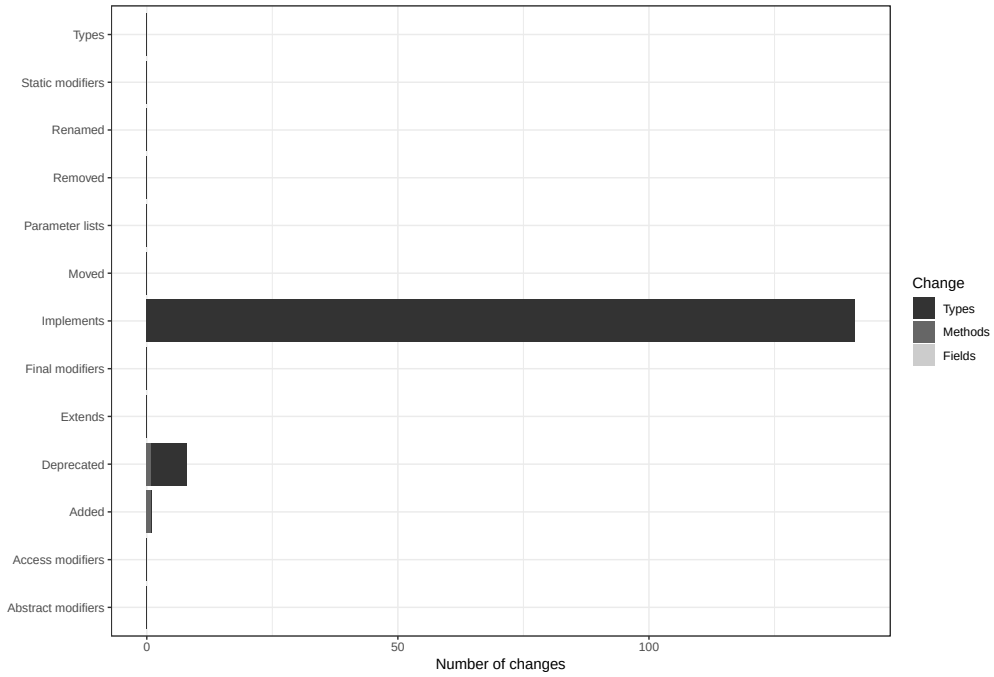

Figure 21: Breaking change detections in the core plug-in of SourceMeter version 8.2

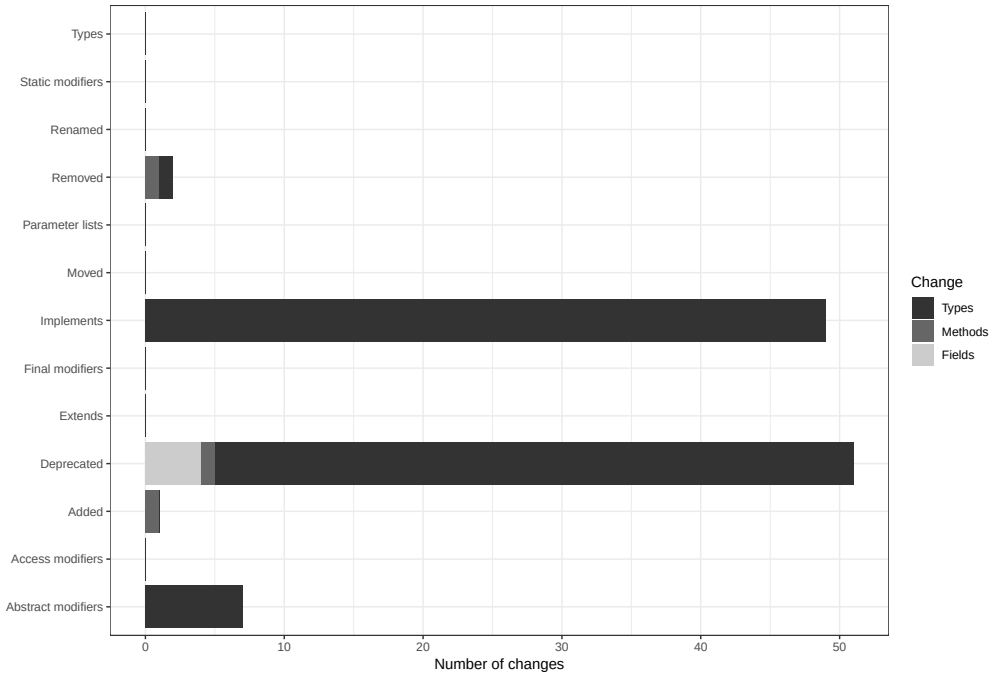

Figure 22: Breaking change detections in the Java plug-in of SourceMeter version 8.2

On the other hand, we spot 110 breaking change detections in the SourceMeter plug-in for Java. Figure 22 also shows detections in the Java plug-in based on change and API member types. In this case, 93% of detections are related to type changes, 3% to method changes, and 4% to field changes. Once again, there are several *implements* changes in the *Detection* model, that is 44% of all detections. However, the most represented type of detected change corresponds to the *deprecated* type present in 46% of detections. Besides from these two popular scenarios, the Java plug-in has some code affected by *removed*, *added*, and *abstract modifier* changes. As in the case of the core plug-in, most of the changes are related to API type modifications.

13 Conclusion

In this part, we have extensively surveyed the state of the art on API evolution and migration and the problem of API-client co-evolution. We have identified a number of limitations that make existing approaches and tools too limited to address the requirements of CROSSMINER partners and use cases.

We build on the state of the art to present our new approach and tool, MARACAS, which enables us to address the problem of API-client co-evolution by reifying three kinds of models: *usage models*, *delta models*, and *detection models*. We evaluate the capability of Maracas to analyze projects regarding API evolution and migration using two case studies: a widely-used mature OOS library (Google Guava), and a mature library used by our partners in their use case (SonarQube).

In the remainder of the project, we will collaborate with our partners closely to put MARACAS in their hands and support their evaluation scenarios.

Part III

Satisfaction of CROSSMINER Requirements

In this section, we present the alignment of the work described in this document with the requirements of CROSSMINER use cases and technologies extracted from **D1.1 – Project Requirements**. Specifically, we refer to the requirements listed in *Section 17: Consolidated Requirements and Mapping* for **WP2: Mining Source Code** *related to API analysis* and the use case requirements.

Req. No.	Requirement	Priority	Status
D6	Source code mining shall be able to classify source code changes (commits) as API or non API changes	SHALL	Full: MARACAS computes changes and classify them as breaking and non-breaking API changes. MARACAS can flexibly compute changes between two major versions of a library, two minor versions, two JARs, or even between two commits.
D7	Source code mining shall provide metrics for patterns and anti-patterns related to a commit	SHALL	Partial: MARACAS partially addresses this requirements by enabling developers to detect whether a given commit introduces breaking changes in the API of a project which, in certain settings, can be considered an anti-pattern. More precisely, it computes the number of breaking and non-breaking changes, which allows developers to look for problematic code introduced in a commit. Other metrics of the CROSSMINER platform, which are not described in this document, also partially address this requirement.
D12	Source code mining shall be able to detect the use of a 3rd-party API function from the source code of a project	SHALL	Full: The *usage models* computed by MARACAS precisely infer which parts of a 3rd-party API is invoked from a given piece of code in clients.
D13	Source code mining shall be able to extract the list of changed/deprecated 3rd-party API methods from the source code of the third-party API	SHALL	Full: MARACAS is able to compute (in the form of Δ-models) the list of changed and deprecated methods from the source code or bytecode of two versions of an API.

D14	Source code mining shall be able to detect from the configuration settings of a project if a new version of a used 3rd-party library is available and determine migration pattern from two (or more) code snippets when one of them uses the old API and the other one the new API	SHALL	Full: The first part of this requirement ("if a new version [...] is available") is addressed in **WP6**). The *migration model* introduced in this document allows to infer, from the source code of a client that has been migrated, migration patterns that can be re-applied on other projects.
D15	Source code mining shall be able to analyse the API documentation (when available) and determine if it matches the API of the library	SHOULD	None.
D17	Source code mining shall be able to identify the public API of a library, including the number (and which) functions are exposed in the API	SHALL	Full: As introduced in Part II, the models computed by MARACAS precisely specify which types/methods/fields are publicly exposed in the API of a library.
U70	Able to identify the list of changed third-party API methods from the source code of the third-party API	SHALL	cf. D13.
U71	Able to identify the list of deprecated third-party API methods from the source code of the third-party API	SHALL	cf. D13.
U72	Able to determine migration pattern from two (or more) code snippets when one of them uses the old third-party API and the other uses the new third-party API	SHALL	cf. D14
U73	Able to identify the part of the API that the developer is currently using to provide code snippets in relation with current development activity	SHALL	Full: The FOCUS tool described in Part I addresses this requirement.
U98	Classify source code changes (commits) as API or non API changes	SHALL	cf.D6
U100	Able to identify the public API provided by projects	SHOULD	cf. D17.

Table 9: Satisfaction of CROSSMINER requirements extracted from **D1.1 – Project Requirements**.

References

[1] Apache Maven. https://maven.apache.org. last access 24.08.2018.

[2] Attribute-Relation File Format (ARFF). https://www.cs.waikato.ac.nz/ml/weka/arff.html. last access 24.08.2018.

[3] Codota. https://www.codota.com/. last access 24.08.2018.

[4] Eclipse JDT Core. https://www.eclipse.org/jdt/core. last access 24.08.2018.

[5] Maven Central Repository. https://mvnrepository.com. last access 24.08.2018.

[6] OSGi. https://www.osgi.org. last access 24.08.2018.

[7] SonarQube. https://www.sonarqube.org/. last access 28.06.2019.

[8] Tycho. https://www.eclipse.org/tycho. last access 24.08.2018.

[9] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34, New York, 2007. ACM.

[10] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining Association Rules Between Sets of Items in Large Databases. In *International Conference on Management of Data*, pages 207–216, New York, 1993. ACM.

[11] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *10th Working Conference on Mining Software Repositories*, pages 207–216, Piscataway, 2013. IEEE.

[12] Anonymous. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns - Online Appendix. https://github.com/icse19-focus/FOCUS. last access 24.08.2018.

[13] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A General Model for Code Analytics in Rascal. In *1st International Workshop on Software Analytics*, pages 25–28, Piscataway, 2015. IEEE.

[14] Andrei Broder. On the Resemblance and Containment of Documents. In *Compression and Complexity of Sequences*, pages 21–29, Washington, 1997. IEEE.

[15] Raymond P. L. Buse and Westley Weimer. Synthesizing API Usage Examples. In *34th International Conference on Software Engineering*, pages 782–792, Piscataway, 2012. IEEE.

[16] Silvio Cesare and Yang Xiang. *Software Similarity and Classification*. Springer, London, 2012.

[17] Annie Chen. Context-Aware Collaborative Filtering System: Predicting the User's Preference in the Ubiquitous Computing Environment. In *First International Conference on Location- and Context-Awareness*, pages 244–253, Berlin, Heidelberg, 2005. Springer.

[18] Flavio Chierichetti, Ravi Kumar, Sandeep Pandey, and Sergei Vassilvitskii. Finding the Jaccard Median. In *21st Symposium on Discrete Algorithms*, pages 293–311, Philadelphia, 2010. Society for Industrial and Applied Mathematics.

[19] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Empirical Methods in Natural Language Processing*, pages 1724–1734, Stroudsburg, 2014. ACL.

[20] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, 2008. IEEE.

[21] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. Context-Based Recommendation to Support Problem Solving in Software Development. In *Third International Workshop on Recommendation Systems for Software Engineering*, pages 85–89, Piscataway, 2012. IEEE.

[22] James R. Cordy. The TXL Source Transformation Language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[23] Bradley E. Cossette and Robert J. Walker. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *20th International Symposium on the Foundations of Software Engineering*, pages 55:1–55:11, New York, 2012. ACM.

[24] Barthélémy Dagenais and Martin P. Robillard. Recommending Adaptive Changes for Framework Evolution. In *30th International Conference on Software Engineering*, pages 481–490, New York, 2008. ACM.

[25] Barthélémy Dagenais and Martin P. Robillard. SemDiff: Recommending Adaptive Changes for Framework Evolution. https://www.cs.mcgill.ca/~swevo/semdiff/, 2008.

[26] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *14th International Conference on Digital Preservation*, pages 1–10, Kyoto, 2017.

[27] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. Linked Open Data to Support Content-based Recommender Systems. In *8th International Conference on Semantic Systems*, pages 1–8, New York, 2012. ACM.

[28] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In *20th European Conference on Object-Oriented Programming*, pages 404–428, Berlin, Heidelberg, 2006. Springer.

[29] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. RefactoringCrawler. http://dig.cs.illinois.edu/tools/RefactoringCrawler/download.html, 2006.

[30] Danny Dig and Ralph Johnson. The Role of Refactorings in API Evolution. In *21st International Conference on Software Maintenance*, pages 389–398, Washington, 2005. IEEE.

[31] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[32] R. A. Fisher. Confidence limits for a cross-product ratio. *Australian Journal of Statistics*, 1962.

[33] Jaroslav Fowkes and Charles Sutton. PAM: Probabilistic API Miner. https://github.com/mast-group/api-mining. last access 24.08.2018.

[34] Jaroslav Fowkes and Charles Sutton. Parameter-free Probabilistic API Mining Across GitHub. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 254–265, New York, 2016. ACM.

[35] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[36] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. Inferring Likely Mappings Between APIs. In *International Conference on Software Engineering*, pages 82–91, Piscataway, 2013. IEEE.

[37] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *40th International Conference on Software Engineering*, pages 933–944, New York, 2018. ACM.

[38] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API Learning. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, New York, 2016. ACM.

[39] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *26th International Joint Conference on Artificial Intelligence*, pages 3675–3681. AAAI, 2017.

[40] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *27th International Conference on Software Engineering*, pages 274–283, New York, 2005. ACM.

[41] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 1979.

[42] Reid Holmes and Gail C. Murphy. Using Structural Context to Recommend Source Code Examples. In *27th International Conference on Software Engineering*, pages 117–125, New York, 2005. ACM.

[43] Paul Jaccard. The Distribution of the Flora in the Alpine Zone. *New Phytologist*, 11(2):37–50, 1912.

[44] C. Kemper and C. Overbeck. What's New with JBuilder. In *JavaOne Sun's Worldwide Java Developer Conference*, San Francisco, 2005. Sun Microsystems.

[45] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates. In *18th International Symposium on Foundations of Software Engineering*, pages 371–372, New York, 2010. ACM.

[46] Miryung Kim, David Notkin, and Dan Grossman. Automatic Inference of Structural Changes for Matching Across Program Versions. In *29th International Conference on Software Engineering*, pages 333–343, Washington, 2007. IEEE.

[47] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering*, pages 222–289, Berlin, Heidelberg, 2011. Springer.

[48] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, Cambridge, 2009.

[49] Ron Kohavi. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *14th International Joint Conference on Artificial Intelligence*, pages 1137–1143, San Francisco, 1995. Morgan Kaufmann Publishers Inc.

[50] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[51] Meir M. Lehman and Juan F. Ramil. Software Evolution: Background, Theory, Practice. *Inf. Process. Lett.*, 88(1-2):33–44, 2003.

[52] VIadimir Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[53] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. SWIN: Towards Type-Safe Java Program Adaptation Between APIs. In *Workshop on Partial Evaluation and Program Manipulation*, pages 91–102, New York, 2015. ACM.

[54] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding Relevant Functions and Their Usage. In *33rd International Conference on Software Engineering*, pages 111–120, New York, 2011. ACM.

[55] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. HiMa. https://github.com/lebuitienduy/hima, 2008.

[56] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A History-based Matching Approach to Identification of Framework Evolution. In *34th International Conference on Software Engineering*, pages 353–363, Piscataway, 2012. IEEE.

[57] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and Their Compositionality. In *26th International Conference on Neural Information Processing Systems*, pages 3111–3119, USA, 2013. Curran Associates Inc.

[58] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method? In *37th International Conference on Software Engineering*, pages 880–890, Piscataway, 2015. IEEE.

[59] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *28th IEEE International Conference on Software Maintenance*, pages 25–34, Piscataway, 2012. IEEE.

[60] Anh T. Nguyen, Tung T. Nguyen, and Tien N. Nguyen. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code. In *30th International Conference on Automated Software Engineering*, pages 585–596, Washington, 2015. IEEE.

[61] Anh T. Nguyen, Zhaopeng Tu, and Tien N. Nguyen. Do Contexts Help in Phrase-Based, Statistical Source Code Migration? In *International Conference on Software Maintenance and Evolution*, pages 155–165, Piscataway, 2016. IEEE.

[62] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical Learning Approach for Mining API Usage Mappings for Code Migration. In *29th ACM/IEEE International Conference on Automated Software Engineering*, pages 457–468, New York, 2014. ACM.

[63] Hoan A. Nguyen, Tung T. Nguyen, Gary Wilson, Jr., Anh T. Nguyen, Miryung Kim, and Tien N. Nguyen. A Graph-based Approach to API Usage Adaptation. In *International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, New York, 2010. ACM.

[64] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. FOCUS: a recommender system for mining API function calls and usage patterns. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1050–1060, 2019.

[65] Trong D. Nguyen, Anh T. Nguyen, and Tien N. Nguyen. Mapping API Elements for Code Migration with Vector Representations. In *38th International Conference on Software Engineering*, pages 756–758, New York, 2016. ACM.

[66] Tung T. Nguyen, Hoan A. Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *7th Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, pages 383–392, New York, 2009. ACM.

[67] Marius Nita and David Notkin. Using Twinning to Adapt Programs to Alternative APIs. In *32nd International Conference on Software Engineering*, pages 205–214, New York, 2010. ACM.

[68] Haoran Niu, Iman Keivanloo, and Ying Zou. API Usage Pattern Recommendation for Software Development. *Journal of Systems and Software*, 129(C):127–139, 2017.

[69] Rahul Pandita, Raoul P. Jetley, Sithu D. Sudarsan, and Laurie Williams. APISIM: Discovering Likely Mappings between APIs using Text Mining. https://sites.google.com/a/ncsu.edu/apisim/, 2015.

[70] Rahul Pandita, Raoul P. Jetley, Sithu D. Sudarsan, and Laurie Williams. Discovering Likely Mappings between APIs Using Text Mining. In *15th International Working Conference on Source Code Analysis and Manipulation*, pages 231–240, Piscataway, 2015. IEEE.

[71] David L. Parnas. Information Distribution Aspects of Design Methodology. Technical report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1971.

[72] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Leveraging Crowd Knowledge for Software Comprehension and Development. In *17th European Conference on Software Maintenance and Reengineering*, pages 57–66, Washington, 2013. IEEE.

[73] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *11th Working Conference on Mining Software Repositories*, pages 102–111, New York, 2014. ACM.

[74] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting Software Developers with a Holistic Recommender System. In *39th International Conference on Software Engineering*, pages 94–105, Piscataway, 2017. IEEE.

[75] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Ref-Finder. https://github.com/SEAL-UCLA/Ref-Finder/, 2010.

[76] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based Reconstruction of Complex Refactorings. In *International Conference on Software Maintenance*, pages 1–10, Washington, 2010. IEEE.

[77] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *38th International Conference on Software Engineering*, pages 357–367, New York, 2016. ACM.

[78] Mohammad Rahman, Shamima Yeasmin, and Chanchal Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 194–203, Piscataway, 2014. IEEE.

[79] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, New York, 2014. ACM.

[80] Peter C. Rigby and Martin P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *35th International Conference on Software Engineering*, pages 832–841, Piscataway, 2013. IEEE.

[81] Martin P Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE software*, 26(6):27–34, 2009.

[82] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[83] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining Multi-level API Usage Patterns. In *22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 23–32, Piscataway, 2015. IEEE.

[84] Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari Sahraoui. Could We Infer Unordered API Usage Patterns Only Using the Library Source Code? In *23rd International Conference on Program Comprehension*, pages 71–81, Piscataway, 2015. IEEE.

[85] Anirudh Santhiar, Omesh Pandita, and Aditya Kanade. MathFinder: Math API Discovery and Migration. http://www.iisc-seal.net/mathfinder/, 2014.

[86] Anirudh Santhiar, Omesh Pandita, and Aditya Kanade. Mining Unit Tests for Discovery and Migration of Math APIs. *ACM Trans. Softw. Eng. Methodol.*, 24(1):4:1–4:33, 2014.

[87] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based Collaborative Filtering Recommendation Algorithms. In *10th International Conference on World Wide Web*, pages 285–295, New York, 2001. ACM.

[88] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. The Adaptive Web: Methods and Strategies of Web Personalization. chapter Collaborative Filtering Recommender Systems, pages 291–324. Springer, Berlin, Heidelberg, 2007.

[89] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining Framework Usage Changes from Instantiation Code. In *30th International Conference on Software Engineering*, pages 471–480, New York, 2008. ACM.

[90] Danilo Silva, Nikolaos Tsantalis, and Marco T. Valente. Why We Refactor? Confessions of GitHub Contributors. In *24th International Symposium on Foundations of Software Engineering*, pages 858–870, New York, 2016. ACM.

[91] Danilo Silva and Marco T. Valente. RefDiff. https://github.com/aserg-ufmg/RefDiff/, 2017.

[92] Danilo Silva and Marco T. Valente. RefDiff: Detecting Refactorings in Version Histories. In *14th International Conference on Mining Software Repositories*, pages 269–279, Piscataway, 2017. IEEE.

[93] Amit Singhal. Modern Information Retrieval: A Brief Overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[94] James Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

[95] Watanabe Takuya and Hidehiko Masuhara. A Spontaneous Code Recommendation Tool Based on Associative Search. In *3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 17–20, New York, 2011. ACM.

[96] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Automatic discovery of function mappings between similar libraries. In *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 192–201, 2013.

[97] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. Automatic Discovery of Function Mappings between Similar Libraries. In *20th Working Conference on Reverse Engineering*, pages 192–201, Piscataway, 2013. IEEE.

[98] S. Thummalapenta and Tao Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, Washington, 2008. IEEE.

[99] Christoph Treude and Martin P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *38th International Conference on Software Engineering*, pages 392–403, New York, 2016. ACM.

[100] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and Efficient Refactoring Detection in Commit History. In *40th International Conference on Software Engineering*, pages 483–494, New York, 2018. ACM.

[101] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. RefactoringMiner. https://github.com/tsantalis/RefactoringMiner/, 2018.

[102] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and Back Again: Can You Compile That Snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2016.

[103] Gias Uddin and Martin P Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, 2015.

[104] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. Transforming Programs between APIs with Many-to-Many Mappings. In *30th European Conference on Object-Oriented Programming*, pages 25:1–25:26, Dagstuhl, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[105] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-coverage API Usage Patterns from Source Code. In *10th Working Conference on Mining Software Repositories*, pages 319–328, Piscataway, 2013. IEEE.

[106] Jianyong Wang and Jiawei Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *20th International Conference on Data Engineering*, pages 79–90, Washington, 2004. IEEE.

[107] Tzu-Tsung Wong. Performance Evaluation of Classification Algorithms by K-fold and Leave-one-out Cross Validation. *Pattern Recognition*, 48(9):2839–2846, 2015.

[108] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: A Hybrid Approach to Identify Framework Evolution. In *32nd International Conference on Software Engineering*, pages 325–334, New York, 2010. ACM.

[109] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: A Hybrid Approach to Identify Framework Evolution. www.ptidej.net/downloads/experiments/icse10b/, 2010.

[110] Zhenchang Xing and Eleni Stroulia. API-Evolution Support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.

[111] Zhenchang Xing and Eleni Stroulia. Differencing Logical UML Models. *Automated Software Engg.*, 14(2):215–259, 2007.

[112] Hao Zhong, Suresh Thummalapenta, and Tao Xie. Exposing Behavioral Differences in Cross-Language API Mapping Relations. In *16th International Conference on Fundamental Approaches to Software Engineering*, pages 130–145, Berlin, Heidelberg, 2013. Springer.

[113] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API Mapping for Language Migration. In *32nd International Conference on Software Engineering*, pages 195–204, New York, 2010. ACM.

[114] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *23rd European Conference on Object-Oriented Programming*, pages 318–343, Berlin, Heidelberg, 2009. Springer.

FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns

Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio
Università degli Studi dell'Aquila
L'Aquila, Italy
{firstname.lastname}@univaq.it

Lina Ochoa, Thomas Degueule
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
{firstname.lastname}@cwi.nl

Massimiliano Di Penta
Università degli Studi del Sannio
Benevento, Italy
dipenta@unisannio.it

***Abstract*—Software developers interact with APIs on a daily basis and, therefore, often face the need to learn how to use new APIs suitable for their purposes. Previous work has shown that recommending *usage patterns* to developers facilitates the learning process. Current approaches to usage pattern recommendation, however, still suffer from high redundancy and poor run-time performance. In this paper, we reformulate the problem of usage pattern recommendation in terms of a collaborative-filtering recommender system. We present a new tool, FOCUS, which mines open-source project repositories to recommend API method invocations and usage patterns by analyzing how APIs are used in projects similar to the current project. We evaluate FOCUS on a large number of Java projects extracted from GitHub and Maven Central and find that it outperforms the state-of-the-art approach PAM with regards to success rate, accuracy, and execution time. Results indicate the suitability of context-aware collaborative-filtering recommender systems to provide API usage patterns.**

I. Introduction

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails the use of external libraries. Rather than implementing new systems from scratch, developers look for, and try to integrate into their projects, libraries that provide functionalities of interest. Libraries expose their functionality through Application Programming Interfaces (APIs) which govern the interaction between a client project and the libraries it uses.

Developers therefore often face the need to learn new APIs. The knowledge needed to manipulate an API can be extracted from various sources: the API source code itself, the official website and documentation, Q&A websites such as StackOverflow, forums and mailing lists, bug trackers, other projects using the same API, etc. However, official documentation often merely reports the API description without providing non-trivial example usages. Besides, querying informal sources such as StackOverflow might become time-consuming and error-prone [32]. Also, API documentation may be ambiguous, incomplete, or erroneous [42], while API examples found on Q&A websites may be of poor quality [18].

Over the past decade, the problem of API learning has garnered considerable interest from the research community. Several techniques have been developed to automate the extraction of API *usage patterns* [33] in order to reduce developers' burden when manually searching these sources and to provide them with high-quality code examples. However, these techniques, based on clustering [23], [43], [45] or predictive modeling [10], still suffer from high redundancy [10] and—as we show later in the paper—poor run-time performance.

To cope with these limitations, we propose a new approach for API usage patterns mining that builds upon concepts emerging from collaborative-filtering recommender systems [36]. The fundamental idea of these systems is to recommend to users items that have been bought by similar users in similar contexts. By considering API methods as products and client code as customers, we reformulate the problem of usage pattern recommendation in terms of a collaborative-filtering recommender system. Informally, the question the proposed system can answer is:

> *"Which API methods should this piece of client code invoke, considering that it has already invoked these other API methods?"*

Implementing a collaborative-filtering recommender system requires to assess the similarity of two customers, i.e., two projects. Existing approaches assume that any two projects using an API of interest are equally valuable sources of knowledge. Instead, we postulate that not all projects are equal when it comes to recommending usage patterns: a project that is highly similar to the project currently being developed should provide higher quality patterns than a highly dissimilar one. Our recommender system attempts to narrow down the search scope by considering only the projects that are the most similar to the active project. Thus, methods that are typically used conjointly by similar projects in similar contexts tend to be recommended first.

We incorporate these ideas into a recommender system that mines open-source software (OSS) repositories to provide developers with API *FunctiOn Calls and USage patterns*: FOCUS. Our approach represents mutual relationships between projects using a 3D matrix and mines API usage from the most similar projects.

We evaluated FOCUS on different datasets comprising 610 Java projects from GitHub and 3,600 JAR archives from the Maven Central Repository. In the evaluation, we simulate different stages of a development process, by removing portions of client code and assessing how FOCUS can recommend snippets with API invocations to complete them. We find that

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00109

```
public List<Boekrekening> findBoekrekeningen() {
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Boekrekening> criteriaQueryBoekrekening = cb
                                                        .createQuery(
                                                         Boekrekening.class
                                                        );
 Root<BoekrekeningPO> boekrekeningFrom = criteriaQueryBoekrekening
                                        .from(BoekrekeningPO.class);




}
```

(a) Initial version

```
public List<Boekrekening> findBoekrekeningen() {
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Boekrekening> criteriaQueryBoekrekening = cb
                                                        .createQuery(
                                                         Boekrekening.class
                                                        );
 Root<BoekrekeningPO> boekrekeningFrom = criteriaQueryBoekrekening
                                        .from(BoekrekeningPO.class);

 criteriaQueryBoekrekening.select(boekrekeningFrom);
 criteriaQueryBoekrekening.orderBy(cb.asc(boekrekeningFrom
                                         .get(BoekrekeningPO_.rekeningnr)));
 return entityManager.createQuery(criteriaQueryBoekrekening).getResultList();
}
```

(b) Final version

Fig. 1. Motivating example

FOCUS outperforms PAM, a state-of-the-art tool for API usage patterns mining [10], with regards to success rate, accuracy, and execution time.

This paper is organized as follows. Section II introduces a motivating example and background notions. Our recommender system for API mining, FOCUS, is introduced in Section III. The evaluation is presented in Section IV, with the key results being analyzed in Section V. Section VI discusses the threats to validity. In Section VII, we present related work and conclude the paper in Section VIII.

II. Background

This section presents a motivating example for introducing the problem addressed by this paper and the main components of the proposed solution. Then, we introduce the main notions underpinning our approach, mostly originating from Schafer et al. [37] and Chen [4].

A. Motivating Example

The typical setting considered in the paper is as shown in Fig. 1: (a) developer is implementing some method to satisfy the requirements of the system being developed. In the specific case shown in Fig. 1 (b), the `findBoekrekeningen` method queries the available entities and retrieve those of type `Boekrekening`. To this end, the `Criteria API` library[1] is used.

Fig. 1 (a) depicts the situation where the development is at an early stage and the developer already used some methods of the chosen API to develop the required functionality. However, she is not sure how to proceed from this point. In such cases, different sources of information may be consulted, such as StackOverflow, video tutorials, API documentation, etc. In this paper, we propose an approach aiming at providing developers with recommendations consisting of a list of API method calls that should be used next, and with usage patterns that can be used as a reference for completing the development of the method being defined (e.g., code snippets that could support developers in completing the method definition with the framed code in Fig. 1 (b)).

[1] https://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html

B. API Function Calls and Usage Patterns

A *software project* is a standalone source code unit that performs a set of tasks. Furthermore, an *API* is an interface that abstracts the functionalities offered by a project by hiding its implementation details. This interface is meant to support reuse and modularity [24], [32]. An API X built in an object-oriented programming language (e.g., the `Criteria API` in Fig. 1) consists of a set T_X of public types (e.g., `CriteriaBuilder` and `CriteriaQuery`). Each type in T_X consists of a set of public methods and fields that are available to client projects (e.g., the method `createQuery` of the type `CriteriaQuery`).

A *method declaration* consists of a name, a (possibly empty) list of parameters, a return type, and a (possibly empty) body (e.g., the method `findBoekrekeningen` in Fig. 1). Given a set of declarations D in a project P, an API *method invocation* i is a call made from a declaration $d \in D$ to another declaration m. Similarly, an API *field access* is an access to a field $f \in F$ from a declaration d in P. API method invocations MI and field accesses FA in P form the set of API usages $U = MI \cup FA$. Finally, an *API usage pattern* (or code snippet) is a sequence $(u_1, u_2, ..., u_n)$, $\forall u_k \in U$ [19].

C. Context-aware Collaborative Filtering

As stated by Schafer et al. [37] "*Collaborative Filtering* (CF) is the process of filtering or evaluating items through the opinions of other people." In a CF system, a *user* who buys or uses an *item* attributes a rating to it based on her experience and perceived value. Therefore, a *rating* is the association of a user and an item through a value in a given unit (usually in scalar, binary, or unary form). The set of all ratings of a given user is also known as a *user profile* [4]. Moreover, the set of all ratings given in a system by existing users can be represented in a so-called *rating matrix*, where a row represents a user and a column represents an item.

The expected outcome of a CF system is a set of predicted ratings (aka. *recommendations*) for a specific user and a subset of items [37]. The recommender system considers the most similar users (aka. *neighbors*) to the *active* user to suggest new ratings. A similarity function $sim_{usr}(u_a, u_j)$ computes the *weight* of the active user profile u_a against each of the user profiles u_j in the system. Finally, to suggest a recommendation for an item i based on this subset of similar profiles, the CF system computes a weighted average $r(u_a, i)$ of the existing ratings, where $r(u_a, i)$ varies with the value of $sim_{usr}(u_a, u_j)$ obtained for all neighbors [4], [37].

Context-aware CF systems compute recommendations based not only on neighbors' profiles but also on the *context* where the recommendation is demanded. Each rating is associated with a context [4]. Therefore, for a tuple C modeling different contexts, a *context similarity* metric $sim_{ctx}(c_a, c_i)$, for $c_a, c_i \in C$ is computed to identify relevant ratings according to a given context. Then, the weighted average is reformulated as $r(u_a, i, c_a)$ [4].

III. Proposed Approach

To tackle the problem of recommending API function calls and usage patterns, we leverage the wisdom of the crowd and existing recommender system techniques. In particular, we hypothesize that API calls and usages can be mined from existing codebases, prioritizing the projects that are similar to the one from where the recommendation is demanded.

More specifically, our tool FOCUS adopts a context-aware CF technique to search for invocations from closely relevant projects. This technique allows us to consider both project and declaration similarities to recommend API function calls and usage patterns. Following the terminology of recommender systems, we treat *projects* as the enclosing *contexts*, *method declarations* as *users*, and *method invocations* as *items*. Intuitively, we recommend a method invocation for a declaration in a given project, which is analogous to recommending an item to a user in a given context. For instance, the set of method invocations and the usage pattern (cf. framed code in Fig. 1 (b)) recommended for the declaration `findBoekrekeningen` can be obtained from a set of similar projects and declarations in a codebase. The *collaborative* aspect of the approach enables to extract recommendations from the most similar projects, while the *context-awareness* aspect enables to narrow down the search space further to similar declarations.

A. Architecture

The architecture of FOCUS is depicted in Fig. 2. To provide its recommendations, FOCUS considers a set of *OSS Repositories* ①. The *Code Parser* ② component extracts method declarations and invocations from the source code or bytecode of these projects. The *Project Comparator*, a subcomponent of the *Similarity Calculator* ③, computes the similarity between projects in the repositories and the project under development. Using the set of projects and the information extracted by the *Code Parser*, the *Data Encoder* ④ component computes rating matrices which are introduced later in this section. Afterwards, the *Declaration Comparator* computes the similarities between declarations. From the similarity scores, the *Recommendation Engine* ⑤ generates recommendations, either as a ranked list of API function calls using the *API Generator*, or as usage patterns using the *Code Builder*, which are presented to the developer. In the remainder of this section, we present in greater details each of these components.

1) Code Parser: FOCUS relies on Rascal M^3 [2], an intermediate model that performs static analysis on the source code, to extract method declarations and invocations from a set of projects. This model is an extensible and composable algebraic data type that captures both language-agnostic and Java-specific facts in immutable binary relations. These relations represent program information such as existing *declarations*, *method invocations*, *field accesses*, *interface implementations*, *class extensions*, among others [2]. To gather relevant data, Rascal M^3 leverages the Eclipse JDT Core Component[2] to build and traverse the abstract syntax trees of the target Java projects.

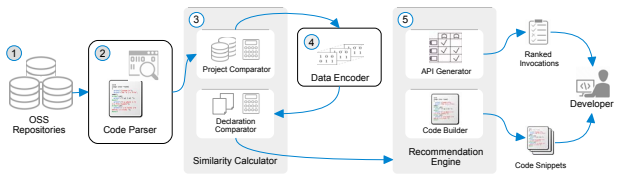

Fig. 2. Overview of the FOCUS architecture

In the context of FOCUS, we consider the data provided by the *declarations* and *methodInvocation* relations of the M^3 model. Both of them contain a set of pairs $\langle v_1, v_2 \rangle$, where v_1 and v_2 are values representing *locations*. These locations are uniform resource identifiers that represent artifact identities (aka. logical locations) or physical pointers on the file system to the corresponding artifacts (aka. physical locations). The *declarations* relation maps the logical location of an artifact (e.g., a method) to its physical location. The *methodInvocation* relation maps the logical location of a *caller* to the logical location of a *callee*. We refer the reader to a dedicated paper for the technical details of the inference of Java M^3 models [2].

Listing 1. Excerpt of the M^3 model extracted from Fig. 1

```
m3.declarations = {
<|java+method://StandaardBoekrekeningService/findBoekrekeningen|,
|file:// ... /StandaardBoekrekeningService.java
     (501,531,<17,4>,<33,5>)|>,
%[...]}
m3.methodInvocation = {
<|java+method://StandaardBoekrekeningService/findBoekrekeningen|,
|java+method://EntityManager/getCriteriaBuilder|>, [...]}
```

Listing 1 depicts an excerpt of the M^3 model extracted from the code presented in Fig. 1 (a). The *declarations* relation links the logical location of the method `findBoekrekeningen`, to its corresponding physical location in the file system. The *methodInvocation* relation states that the `getCriteriaBuilder` method of the `EntityManager` type is invoked by the `findBoekrekeningen` method in the current project.

2) Data Encoder: Once method declarations and invocations are extracted, FOCUS represents the relationships among them using a rating matrix. For a given project, each row in the matrix represents a method declaration and each column represents a method invocation. A cell is set to 1 if the declaration in the corresponding row contains the invocation in the column, otherwise it is set to 0. For example, Fig. 3 shows the rating matrix of a project with four declarations $p_1 \ni (d_1, d_2, d_3, d_4)$ and four invocations (i_1, i_2, i_3, i_4).

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{array} \begin{array}{c} \begin{array}{cccc} i_1 & i_2 & i_3 & i_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 3. Rating matrix for a project with 4 declarations and 4 invocations

To capture the intrinsic relationships among various projects, declarations, and invocations, we come up with a 3D context-based rating matrix [21]. The third dimension of this matrix

[2]https://www.eclipse.org/jdt/core/

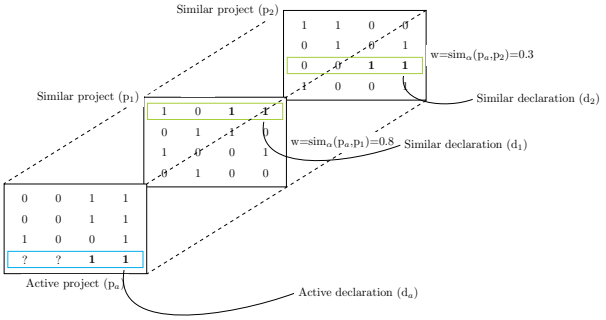

Fig. 4. 3D context-based rating matrix

represents a project, which is analogous to the so-called context in context-aware CF systems. For example, Fig. 4 depicts three projects $P = (p_a, p_1, p_2)$ represented by three slices with four method declarations and four method invocations. Project p_1 has already been introduced in Fig. 3 and for the sake of readability, the column and row labels are removed from all slices in Fig. 4. There, p_a is the *active project* and it has an *active declaration*. *Active* here means the artifact (project or declaration), being considered or developed. Both p_1 and p_2 are complete projects similar to the active project p_a. The former projects (i.e., p_1 and p_2) are also called *background data* since they are already available and serve as a base for the recommendation process. In practice, the higher the number of complete projects considered as background data, the higher the probability to recommend relevant invocations.

3) Similarity Calculator: Exploiting the context-aware CF technique, the presence of additional invocations is deduced from similar declarations and projects. Given an active declaration in an active project, it is essential to find the subset of the most similar projects, and then the most similar declarations in that set of projects. To compute similarities, we derive from [20] a weighted directed graph that models the relationships among projects and invocations. Each node in the graph represents either a project or an invocation. If project p contains invocation i, then there is a directed edge from p to i. The weight of an edge $p \rightarrow i$ represents the number of times a project p performs the invocation i. Fig. 5 depicts the graph for the set of projects introduced in Fig. 4. For instance, p_a has four declarations and all of them invoke i_4. As a result, the edge $p_a \rightarrow i_4$ has a weight of 4. In the graph, a question mark represents missing information. For the active declaration in p_a, it is not known yet whether invocations i_1 and i_2 should be included.

The similarity between two project nodes p and q is computed by considering their feature sets [7]. Given that p has a set of neighbor nodes $(i_1, i_2, .., i_l)$, the feature set of p is the vector $\overrightarrow{\phi} = (\phi_1, \phi_2, .., \phi_l)$, with ϕ_k being the weight of node i_k. This weight is computed as the *term-frequency inverse document frequency* value, i.e., $\phi_k = f_{i_k} * log(\frac{|P|}{a_{i_k}})$, where f_{i_k} is the weight of the edge $p \rightarrow i_k$; $|P|$ is the number

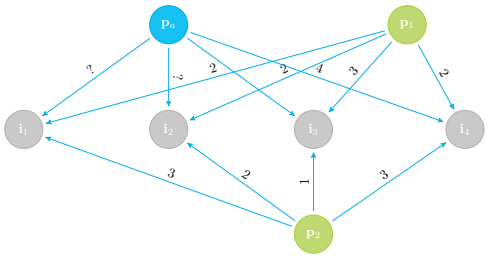

Fig. 5. Graph representation of projects and invocations

of all considered projects; and a_{i_k} is the number of projects connected to i_k. Eventually, the similarity between p and q with their corresponding feature vectors $\overrightarrow{\phi} = \{\phi_k\}_{k=1,..,l}$ and $\overrightarrow{\omega} = \{\omega_j\}_{j=1,..,m}$ is:

$$sim_\alpha(p,q) = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n}(\phi_t)^2} \times \sqrt{\sum_{t=1}^{n}(\omega_t)^2}} \quad (1)$$

The similarities among method declarations are calculated using the Jaccard similarity index [15] as follows:

$$sim_\beta(d,e) = \frac{|\mathbb{F}(d) \bigcap \mathbb{F}(e)|}{|\mathbb{F}(d) \bigcup \mathbb{F}(e)|} \quad (2)$$

where $\mathbb{F}(d)$ and $\mathbb{F}(e)$ are the sets of invocations made from declarations d and e, respectively.

4) API Generator: This component, which is part of the *Recommendation Engine*, is in charge of generating a ranked list of API function calls. In Fig. 4, the active project p_a already includes three declarations, and the developer is working on the fourth declaration, which corresponds to the last row of the slice. p_a has only two invocations, represented in the last two columns of the matrix (i.e., cells filled with 1). The first two cells are marked with a question mark (?), indicating that it is unclear whether these two invocations should also be added into p_a. The recommendation engine attempts to predict additional invocations for the active declaration by computing the missing ratings using the following formula [4]:

$$r_{d,i,p} = \overline{r}_d + \frac{\sum_{e \in topsim(d)} (R_{e,i,p} - \overline{r}_e) \cdot sim_\beta(d,e)}{\sum_{e \in topsim(d)} sim_\beta(d,e)} \quad (3)$$

Eq. 3 is used to compute a score for the cell representing method invocation i, declaration d of project p, where $topsim(d)$ is the set of top similar declarations of d; $sim_\beta(d,e)$ is the similarity between d and a declaration e, computed using Eq. 2; $\overline{r}_d$ and $\overline{r}_e$ are the mean ratings of d and e, respectively; and $R_{e,i,p}$ is the combined rating of declaration d for i in all similar projects, computed as follows [4]:

$$R_{e,i,p} = \frac{\sum_{q \in topsim(p)} r_{e,i,q} \cdot sim_\alpha(p,q)}{\sum_{q \in topsim(p)} sim_\alpha(p,q)} \quad (4)$$

where $topsim(p)$ is the set of top similar projects of p; and $sim_\alpha(p,q)$ is the similarity between p and a project q, computed using Eq. 1. Eq. 4 implies that a higher weight is given to projects with higher similarity. In practice, it is reasonable since, given a project, its similar projects contain

```
public List<QuestionsStaged> findByIdentifier(String identifier) {
    log.fine("getting Session instance by identifier: " + identifier);
    try {
      CriteriaBuilder cb = entityManager.getCriteriaBuilder();
      CriteriaQuery<QuestionsStaged> criteria = cb.createQuery(QuestionsStaged.class);
      Root<QuestionsStaged> qs = criteria.from(QuestionsStaged.class);
      criteria.select(qs).where(cb.equal(qs.get("identifier"), identifier));
      log.fine("get identifier successful");
      return entityManager.createQuery(criteria).getResultList();

    } catch (RuntimeException re) {
      log.severe("get identifier failed" + re);
      throw re;
    }
  }
```

Fig. 6. Real source code recommended by FOCUS

more relevant API calls than less similar projects. Using Eq. 3 we compute all the missing ratings in the active declaration and get a ranked list of invocations with scores in descending order, which is then suggested to the developer.

5) Code Builder: This subcomponent is also part of the *Recommendation Engine*, and it is responsible for recommending usage patterns to developers. From the ranked list, *top-N* method invocations are used as a query to search the database for relevant declarations. To limit the search scope, only the most similar projects are considered. The Jaccard index is used to compute similarities between the selected invocations and a given declaration. For each query, we search for declarations that contain as many invocations of the query as possible. Once we identify the corresponding declarations we retrieve their source code using the *declarations* relation of the Rascal M^3 model. The resulting code snippet is then recommended to the developer.

For the sake of illustration, we now present an example of how FOCUS suggests real code snippets, considering the declaration `findBoekrekeningen` in Fig. 1 (a) as input. The invocations it contains are used together with the other declarations in the current project as query to feed the *Recommendation Engine*. The final outcome is a ranked list of real code snippets. The top one, named `findByIdentifier`, is depicted in Fig. 6. By carefully examining this code and the original one in Fig. 1 (b), we see that although they are not exactly the same, they indeed share several method calls and a common intent: both exploit a `CriteriaBuilder` object to build, perform a query and eventually get back some results. Furthermore, the outcome of both declarations is of the `List` type. Interestingly, compared to the original one, the recommended code appears to be of higher quality since it includes a `try/catch` construct to handle possible exceptions. Thus, the recommended code, coupled with the corresponding list of function calls (i.e., `get`, `equal`, `where`, `select`, etc.), provides the developer with helpful directions on how to use the API at hand to implement the desired functionality.

IV. Evaluation

The *goal* of this study is to evaluate FOCUS, and compare it with another state-of-the-art tool (PAM [10]), with the aim of assessing its capability to recommend API usage patterns to developers, while they are writing code. The *quality focus* is twofold: studying the API recommendation accuracy and completeness, as well as the time required by FOCUS and PAM to provide a recommendation. The *context* consists of 610 Java open source projects, and 3,600 JAR archives from the Maven Central repository.[3] For the sake of reproducibility and ease of reference, all artifacts used in the evaluation, together with the tools are available online [22]. We choose PAM as a baseline for comparison, as it has been shown to outperform [10] other similar tools such as MAPO [45] and UP-Miner [43]. To conduct the comparison with PAM, we leverage its original source code made available online by its authors [9].

In the following, we detail our research questions, datasets, evaluation methodology, and metrics.

A. Research Questions

Our research questions are as follows:

RQ$_1$ *To what extent is FOCUS able to provide accurate and complete recommendations?* This research question relates to the capability of FOCUS to produce accurate and complete results. Having too many false positives would end up being counterproductive, whereas having too many false negatives would mean that the tool is not able to provide recommendations in many cases where this is needed.

RQ$_2$ *What are the timing performances of FOCUS in building its models and in providing recommendations?* This research question aims at assessing whether, from a timing point of view, FOCUS—compared to PAM—could be used in practice. We evaluate the time required by both tools to provide a recommendation. We mainly focus on the recommendation time because, while it is acceptable that the model training phase is relatively slow (i.e., the model could be built offline), the recommendation time has to be fast enough to make the tool applicable in practice.

RQ$_3$ *How does FOCUS perform compared with PAM?* Finally, this research question directly compares the recommendation capabilities of FOCUS and PAM.

B. Datasets

To answer our research questions, we relied on four different datasets. The first dataset, $\texttt{SH}_L$, has been assembled starting from 5,147 randomly selected Java projects retrieved from GitHub via the Software Heritage archive [6]. To comply with the requirements of PAM, we first restricted the dataset to the list of projects that use at least one of the third-party libraries listed in Table I. Most of them were used to assess the performance of PAM [10]. Each row in Table I lists a third-party library, the number of projects that depend on it, and the number of classes that invoke methods of this library. To comply with the requirements of FOCUS, we then restricted the dataset to the list of projects containing at least one *pom.xml*, as it eases the creation of the M^3 models. We thus obtained our first dataset consisting of 610 Java projects.

From $\texttt{SH}_L$, we extracted a second dataset $\texttt{SH}_S$ consisting of the 200 smallest (in size) projects of $\texttt{SH}_L$.

As a third dataset, we randomly collected a set of 3,600 JAR archives from the Maven Central repository, which we name $\texttt{MV}_L$. Through a manual inspection of $\texttt{MV}_L$, we noticed

[3] https://mvnrepository.com

TABLE I
EXCERPT OF THE THIRD-PARTY LIBRARIES USED BY DATASET $\texttt{SH}_L$

Project Name	# of Client Projects	# of Client Classes
com.google.gson	51	337
io.netty	105	13,456
org.apache.camel	36	1,017
org.apache.hadoop	158	14,596
org.apache.lucene	15	397
org.apache.mahout	25	8,541
org.apache.wicket	44	3,360
org.drools	27	886
org.glassfish.jersey	105	3,811
org.hornetq	15	123
org.jboss.weld	39	1,875
org.jooq	16	243
org.jsoup	23	55
org.neo4j	28	4,983
org.restlet	19	326
org.springside	16	821
twitter4j	45	597
	610	**55,425**

that many projects only differ in their version numbers (*ant-1.6.5.jar* and *ant-1.9.3.jar*, for instance, are two versions of the same project *ant*). These cases are interesting as we assume two versions of the same project share many functionalities [39]. The collaborative-filtering technique works well given that highly similar projects exist, since it just "copies" invocations from similar methods in the very similar projects (see Eq. 3 and Eq. 4). However, a dataset containing too many similar projects may introduce a bias in the evaluation. Thus, we decided to populate one more dataset. Starting from $\texttt{MV}_L$, we randomly selected one version for every project and filtered out the other versions. The removal resulted in a fourth dataset consisting of $1,600$ projects, which we name $\texttt{MV}_S$.

Three datasets, i.e., $\texttt{SH}_L$, $\texttt{MV}_L$ and $\texttt{MV}_S$ are used to assess the performance of FOCUS ($\mathbf{RQ}_1$). The smallest dataset $\texttt{SH}_S$ is used to compare FOCUS and PAM ($\mathbf{RQ}_2$ and $\mathbf{RQ}_3$).

Eventually, the process of creating required metadata consists of the following main steps:

- for each project in the dataset the corresponding Rascal M^3 model is generated;
- for each M^3 model, the corresponding ARFF representations[4] are generated in order to be used as input for applying FOCUS and PAM during the actual evaluation steps discussed in the next sections.

C. Study Methodology

Performing a user study has been accepted as the standard method to validate an API usage recommendation tool [17], [45]. While user studies are valuable, they are limited in the size of the task a participant can conduct and are highly susceptible to individual skills and subjectiveness. In this paper, to study if FOCUS is applicable in real-world settings we perform a different, offline evaluation, by simulating the behavior of a developer working at different stages of a development project on partial code snippets.

[4]https://www.cs.waikato.ac.nz/ml/weka/arff.html

More specifically, we consider a programmer who is developing a project p. To this end, some parts of p are removed to mimic an actual development. Given an original project p, the total number of declarations it contains is called Δ. However, only δ declarations ($\delta < \Delta$) are used as input for recommendation and the rest is discarded. In practice, this corresponds to the situation when the developer already finished δ declarations, and she is now working on the *active declaration* d_a. For d_a, originally there are Π invocations, however only the first π invocations ($\pi < \Pi$) are selected as query and the rest is removed and saved as ground-truth data for future comparison. In practice, δ is small at an early stage and increases over the course of time. Similarly, π is small when the developer just starts working on d_a. The two parameters δ, π are used to stimulate different development phases. In particular, we consider the following configurations.

Configuration $\texttt{C1.1}$ ($\delta = \Delta/2 - 1, \pi = 1$): Almost the first half of the declarations is used as testing data and the second half is removed. The last declaration of the first half is selected as the active declaration d_a. For d_a, only the *first* invocation is provided as a query, and the rest is used as ground-truth data which we call $\texttt{GT(p)}$. This configuration mimics a scenario where the developer is at an early stage of the development process and, therefore, only limited context data is available to feed the recommendation engine.

Configuration $\texttt{C1.2}$ ($\delta = \Delta/2 - 1, \pi = 4$): Similarly to $\texttt{C1.1}$, almost the first half of the declarations is kept and the second half is discarded. d_a is the last declaration of the first half of declarations. For d_a, the first *four* invocations are provided as query, and the rest is used as $\texttt{GT(p)}$.

Configuration $\texttt{C2.1}$ ($\delta = \Delta - 1, \pi = 1$): The last method declaration is selected as testing, i.e., d_a and all the remaining declarations are used as training data ($\Delta - 1$). In d_a, the *first* invocation is kept and all the others are taken out as ground-truth data $\texttt{GT(p)}$. This represents the stage where the developer almost finished implementing p.

Configuration $\texttt{C2.2}$ ($\delta = \Delta - 1, \pi = 4$): Similar to $\texttt{C2.1}$, d_a is selected as the last method declaration, and all the remaining declarations are used as training data ($\Delta - 1$). The only difference with $\texttt{C2.1}$ is that in d_a, the first *four* invocations are used as query and all the remaining ones are used as $\texttt{GT(p)}$.

When performing the experiments, we split a dataset into two independent parts, namely a *training set* and a *testing set*. In practice, the training set represents the OSS projects that have been collected a priori. They are available at developers' disposal, ready to be exploited for mining purposes. The testing set represents the project being developed, or *the active project*. This way, our evaluation mimics a real development scheme: *the system should produce recommendations for the active project based on the data from a set of existing projects*. We opt for *k-fold cross validation* [16] as it is widely chosen to evaluate machine learning models. Depending on the availability of input data, the dataset with n elements is divided into k equal parts, so-called *folds*. For each validation round, one fold is used as testing data and the remaining $k - 1$ folds are used as training data. For our evaluation, we select two values,

i.e., $k = 10$ and $k = n$. The former corresponds to *ten-fold cross validation* and the latter corresponds to *leave-one-out cross validation* [44].

D. Evaluation Metrics

For a testing project p, the outcome of a recommendation process is a ranked list of invocations, i.e., `REC(p)`. It is our firm belief that the ability to provide accurate invocations is important in the context of software development. Thus, we are interested in how well a system can recommend API invocations that eventually match with those stored in `GT(p)`. To measure the performance of the recommender systems, i.e., PAM and FOCUS, we utilize two metrics, namely *success rate* and *accuracy* [7]. Given a ranked list of recommendations, a developer typically pays attention to the *top-N* items only. *Success rate* and *accuracy* are computed by using N as the *cut-off value*. Given that $REC_N(p)$ is the set of *top-N* items and $match_N(p) = GT(p) \bigcap REC_N(p)$ is the set of items in the *top-N* list that match with those in the ground-truth data, then the metrics are defined as follows.

Success rate: Given a set of P testing projects, this metric measures the rate at which a recommendation engine can return at least a match among *top-N* recommended items for every project $p \in P$.

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} \times 100\% \tag{5}$$

where *count()* counts the number of times the boolean expression given as parameter evaluates to *true*.

Accuracy: Precision and recall are employed to measure accuracy [7]. *Precision@N* is the ratio of the *top-N* recommended items belonging to the ground-truth dataset:

$$precision@N = \frac{|match_N(p)|}{N} \tag{6}$$

and *recall@N* is the ratio of the ground-truth items being found in the *top-N* items:

$$recall@N = \frac{|match_N(p)|}{|GT(p)|} \tag{7}$$

Recommendation time: As mentioned in $\mathbf{RQ}_2$, we measure the time needed by both PAM and FOCUS to perform a prediction on a given infrastructure, which is a laptop with Intel Core i5-7200U CPU @ 2.50GHz × 4, 8GB RAM, and Ubuntu 16.04.

V. Results

$\mathbf{RQ}_1$: *To what extent is FOCUS able to provide accurate and complete recommendations?*

To answer this research question, we use the dataset $\mathtt{SH}_L$ and vary the length of the input data for every testing project. Two main configurations are taken into account, with two sub-configurations for each as introduced in Section IV-C. Table II shows the success rate for all the configurations. For a small N, i.e., $N = 1$ (the developer expects a very brief list of items) FOCUS is still able to provide matches. For example, the success rates of `C1.1` and `C1.2` are 24.59% and 30.65%, respectively. When the cut-off value N is increased, the corresponding success rates improve linearly. For example, when $N = 20$, FOCUS obtains 40.98% success rate for `C1.1` and 47.70% for `C1.2`. By comparing the results obtained for `C1.1` and `C1.2`, we see that when more invocations are incorporated into the query, FOCUS provides more precise recommendations. In practice, this means that the accuracy of recommendations improves with the maturity of the project.

TABLE II
SUCCESS RATE FOR $\mathtt{SH}_L$, $N = \{1, 5, 10, 15, 20\}$

N	$\mathtt{SH}_L$			
	C1.1	C1.2	C2.1	C2.2
1	24.59	30.65	23.44	29.83
5	31.96	40.00	31.31	39.01
10	35.90	43.77	35.73	43.77
15	39.34	47.21	37.70	45.57
20	40.98	47.70	39.34	46.88

We now consider the outcomes obtained for `C2.1` and `C2.2`. In these configurations, more background data is available for recommendation. For `C2.1` ($\delta = \Delta - 1, \pi = 1$), the success rates for the smallest values of N, i.e., $N = 1$ and $N = 5$ are 23.44% and 31.31%, respectively. In other words, it improves with N. The same trend can be observed with other cut-off values, i.e., $N = 10, 15, 20$: the success rates for these settings increase correspondingly. We notice the same pattern considering `C2.1` and `C2.2` together, or `C1.1` and `C1.2` together: if more invocations are used as query, FOCUS suggests more accurate invocations.

Fig. 7 and Fig. 8 depict the precision and recall curves (PRCs) for the above mentioned configurations by varying N from 1 to 30. In particular, Fig. 7 represents the accuracy when almost the first half of the declarations ($\delta = \Delta/2 - 1$) together with one (`C1.1`) and four invocations (`C1.2`) from the testing declaration d_a are used as query. As a PRC close to the upper right corner indicates a better accuracy [7], we see that the accuracy of `C1.2` is superior to that of `C1.1`. Similarly with `C2.1` and `C2.2`, as depicted in Fig. 8, the accuracy improves substantially when the query contains more invocations. These facts further confirm that FOCUS is able to recommend more relevant invocations when the developer keeps coding. This improvement is obtained since the similarity between declarations can be better determined when more invocations are available as comprehended in Eq. 4.

The results reported so far appear to be promising at the first sight. However, by considering Table II, Fig. 7, and Fig. 8 together, we realize that both success rate and accuracy are considerably low: The best success rate is 47.70% for `C1.2` when $N = 20$, which means that more than half of the queries do not get any matches at all. In this sense, it is necessary to ascertain the cause of this outcome: Is FOCUS only capable of generating such moderate recommendations, or is it because of the data? Our intuition is that $\mathtt{SH}_L$ is rather small in size, which means the background data available for the recommendation process is limited. Thus, to further validate the performance of FOCUS, we perform additional experiments by considering more data, using both $\mathtt{MV}_L$ and $\mathtt{MV}_S$. For this evaluation, we

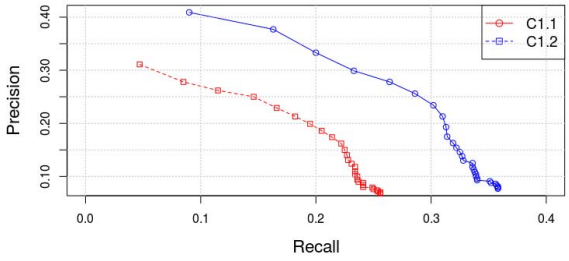

Fig. 7. Precision and recall for C1.1 and C1.2 on SH_L

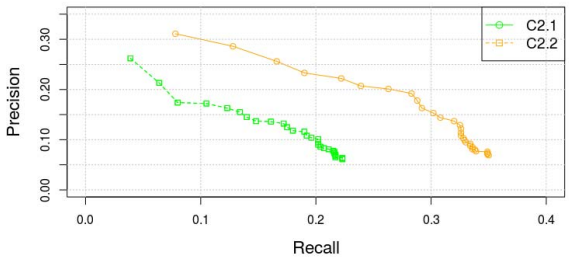

Fig. 8. Precision and recall for C2.1 and C2.2 on SH_L

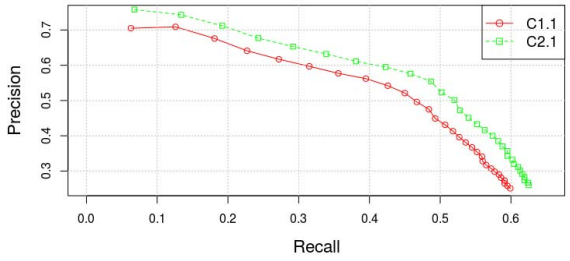

Fig. 9. Precision and recall for C1.1 and C2.1 on MV_L

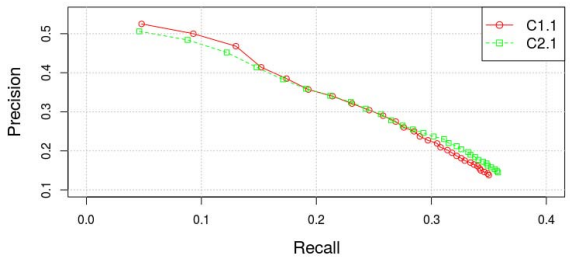

Fig. 10. Precision and recall for C1.1 and C2.1 on MV_S

just consider the case when only one invocation together with other declarations are used as query, i.e., C1.1 and C2.1. This aims at validating the performance of FOCUS, given that the developer just finished only one invocation in d_a.

Table III depicts the success rate obtained for different cut-off values using both datasets. The success rates for all configurations are much better than those of SH_L. The scores are considerably high, even when $N = 1$, the success rates obtained by C1.1 and C2.1 are 72.30% and 72.80%, respectively. For MV_S, the corresponding success rates are lower. However, this is understandable since the set has less data compared to MV_L.

The PRCs for MV_L and MV_S are shown in Fig. 9 and Fig. 10, respectively. We see that for MV_L, a superior performance is obtained by configuration C2.1, i.e., when more background data is available for recommendation compared to C1.1. For MV_S, we witness the same trend as with success rate: the difference between C1.1 and C2.1 is negligible. Considering both Fig. 9 and Fig. 10, we observe that the overall accuracy for MV_L is much better than that of MV_S. The maximum precision and recall for MV_L are 0.75 and 0.62, respectively. Whereas, the maximum precision and recall for MV_S are 0.52 and 0.36, respectively. This further confirms the fact that with more similar projects, FOCUS can provide better recommendations. Referring back to the outcomes of SH_L, we see that the performance on MV_L and MV_S is improved substantially.

TABLE III
SUCCESS RATE FOR MV_L AND MV_S, $N = \{1, 5, 10, 15, 20\}$

N	MV_L		MV_S	
	C1.1	C2.1	C1.1	C2.1
1	72.30	72.80	49.40	50.10
5	82.80	82.70	64.60	65.40
10	86.40	86.40	69.30	70.10
15	88.10	87.90	71.60	72.20
20	89.20	89.00	73.30	74.30

To sum up, we conclude that the performance of FOCUS relies on the availability of background data. The system works effectively given that more OSS projects are available for recommendation. In practice, it is expected that we can crawl as many projects as possible, and use them as background data for the recommendation process.

RQ_2: *What are the timing performances of FOCUS in building its models and in providing recommendations?*

To measure the execution time of PAM and FOCUS, for the very first attempt we ran both systems on the SH_L dataset, consisting of 610 projects. With PAM, for each testing project, we combined the extracted query with all the other training projects to produce a single ARFF file provided as input for the recommendation process [10]. Nevertheless, we then realized that the execution of PAM is very time-consuming. For instance, for one fold containing 1 testing and 549 training projects (i.e., $610/10 \times 9$ training folds) with 80MB in size, PAM takes around 320 seconds to produce the final recommendations. Instead, the corresponding execution time by FOCUS is quite faster than PAM, around 1.80 seconds. Given the circumstances, it is not feasible to run PAM on a large dataset.

Therefore, we decided to use the SH_S dataset (consisting of 200 projects) for this purpose. For the experiments, we opt for *leave-one-out cross-validation* [44], i.e., one project is used

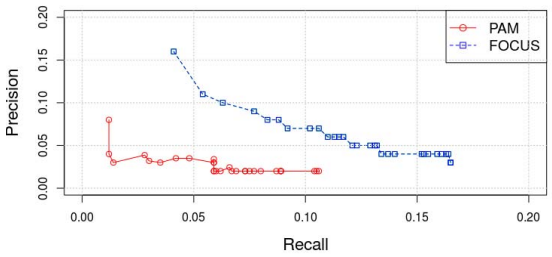

Fig. 11. Precision and recall for PAM and FOCUS using $\texttt{SH}_S$.

as testing, and all the remaining 199 projects are used for the training. The rationale behind the selection of this method instead of ten-fold cross-validation is that we want to exploit as much as possible the projects available as background data, given a testing project. The validation was executed 200 times, and we measured the time needed to finish the recommendation process. On average, PAM requires 9 seconds to provide each recommendation while FOCUS just needs 0.095 seconds, i.e., it is two orders of magnitude faster and suitable to be integrated into a development environment.

$\mathbf{RQ}_3$: *How does FOCUS perform compared with PAM?*

For the reasons explained in $\mathbf{RQ}_2$, the comparison between PAM and FOCUS has been performed on the $\texttt{SH}_S$ dataset. FOCUS gains a better success rate than PAM does, i.e., 51.20% compared to 41.60%. Furthermore, as depicted in Fig. 11, there is a big gap between the PRCs for PAM and FOCUS, with the one representing FOCUS closer to the upper right corner. This implies that the accuracy obtained by FOCUS is considerably superior to that of PAM.

A statistical comparison of PAM and FOCUS using Fisher's exact test [8] indicates that, for $1 \leq N \leq 20$, FOCUS always outperforms PAM: We achieved p-values < 0.001 (adjusted using the Holm's correction [13]) in all cases, with an Odds Ratio between 2.21 and 3.71, and equal to 2.54 for $N = 1$. In other words, FOCUS has over twice the odds of providing an accurate recommendation than PAM.

It is worth noting that the overall accuracy of FOCUS achieved and reported in this experiment is, although better than that of PAM, still considerably low. Following the experiments on $\texttt{MV}_L$ and $\texttt{MV}_S$ from $\mathbf{RQ}_1$, we believe that this attributes to the limited background data available for the evaluation, since we only consider 200 projects.

In summary, by considering both $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we come to the conclusion that FOCUS obtains a better performance in comparison to PAM with regards to success rate, accuracy and execution time. Lastly, since PAM takes considerable time to produce the final recommendations, it might be impractical to deploy PAM in a development environment.

VI. Threats to Validity

The main threat to *construct validity* concerns the simulated setting used to evaluate the approaches, as opposed to performing a user study. We mitigated this threat by introducing four configurations that simulate different stages of the development process. In a real development setting, however, the order in which one writes statements might not fully reflect our simulation. Also, in a real setting, there may be cases in which a recommender is more useful, and cases (obvious code completion) where it is less useful. This makes a further evaluation involving developers highly desirable.

Threats to *internal validity* concern factors internal to our study that could have influenced the results. One possible threat can be seen through the results obtained for the datasets $\texttt{SH}_L$ and $\texttt{SH}_S$. As noted, these datasets exhibit lower precision/recall with respect to $\texttt{MV}_L$ and $\texttt{MV}_S$ due to the limited size of the training sets. However, these datasets were needed to compare FOCUS and PAM due to the limited scalability of PAM.

The main threat to *external validity* is that FOCUS is currently limited to Java programs. As stated in Section III, however, FOCUS makes few assumptions on the underlying language and only requires information about method declarations and invocations to build the 3D rating matrix. This information could be extracted from programs written in any object-oriented programming language, and we wish to generalize FOCUS to other languages in the future.

VII. Related Work

In this section, we summarize related work about API usage recommendation and relate our contributions to the literature.

A. API Usage Pattern Recommendation

Acharya et al. [1] present a framework to extract API patterns as partial orders from client code. While this approach proposes a representation for API patterns, suggestions regarding API usage are still missing.

MAPO (Mining API usage Pattern from Open source repositories) is a tool that mines API usage patterns from client projects [45]. MAPO collect API usages from source files, groups API methods into clusters. Then, it mines API usage patterns from the clusters, ranks them according to their similarity with the current development context, and recommends code snippets to developers. Similarly, UP-Miner [43] mines API usage patterns by relying on *SeqSim*, a clustering strategy that reduces patterns redundancy and improves coverage. Differently from FOCUS, these approaches are based on clustering techniques, and consider all client projects in the mining regardless of their similarity with the current project.

Fowkes et al. introduce PAM (Probabilistic API Miner), a parameter-free probabilistic approach to mine API usage patterns [10]. PAM uses the structural Expectation-Maximization (EM) algorithm to infer the most probable API patterns from client code, which are then ranked according to their probability. PAM outperforms both MAPO and UP-Miner (lower redundancy and higher precision). We directly compare FOCUS to PAM in Section IV.

Niu et al. extract API usage patterns using API class or method names as queries [23]. They rely on the concept of object usage (method invocations on a given API class) to

extract patterns. The approach of Niu et al. outperforms UP-Miner and Codota,[5], a commercial recommendation engine, in terms of coverage, performance, and ranking relevance. In contrast, FOCUS relies on context-aware CF techniques—which favors recommendations from similar projects and uses the whole development context to query API method calls.

The NCBUP-miner (Non Client-based Usage Patterns) [35] is a technique that identifies unordered API usage patterns from the API source code, based on both structural (methods that modify the same object) and semantic (methods that have the same vocabulary) relations. The same authors also propose MLUP [34], which is based on vector representation and clustering, but in this case client code is also considered.

DeepAPI [12] is a deep-learning method used to generate API usage sequences given a query in natural language. The learning problem is encoded as a machine translation problem, where queries are considered the source language and API sequences the target language. Only commented methods are considered during the search. The same authors [11] present CODEnn (COde-Description Embedding Neural Network), where, instead of API sequences, code snippets are retrieved to the developer based on semantic aspects such as API sequences, comments, method names, and tokens.

With respect to the aforementioned approaches, FOCUS uses CF techniques to recommend and rank API method calls and usage patterns from a set of similar projects. In the end, not only relevant API invocations are recommended, but also code snippets are returned to the developer as usage examples.

B. API-Related Code Search Approaches

Strathcona [14] is a recommender system used to suggest API usage. It is an Eclipse plug-in that extracts the structural context of code and uses it as a query to request a set of code examples from a remote repository. Six heuristics (associated to class inheritance, method calls, and field types) are defined to perform the match. Similarly, Buse and Weimer [3] propose a technique for synthesizing API usage examples for a given data type. An algorithm based on data-flow analysis, k-medoids clustering and pattern abstraction is designed. Its outcome is a set of syntactically correct and well-typed code snippets where example length, exception handling, variables initialization and naming, and abstract uses are considered.

Moreno et al. [17] introduce MUSE (Method USage Examples), an approach designed for recommending code examples related to a given API method. MUSE extracts API usages from client code, simplifies code examples with static slicing, and detects clones to group similar snippets. It also ranks examples according to certain properties (i.e., reusability, understandability, and popularity) and documents them.

SWIM (Synthesizing What I Mean) [28] seeks API structured call sequences (control and data-flows are considered), and then synthesizes API-related code snippets according to a query in natural language. The underlying learning model is also built with the EM algorithm. Similarly, Raychev et al. [30] propose a code completion approach based on natural language processing, which receives as input a partial program and outputs a set of API call sequences filling the gaps of the input. Both invocations and invocation arguments are synthesized considering multiple types of an API.

Thummalapenta and Xie propose SpotWeb [40], an approach that provides starting points (hotspots) for understanding a framework, and highlights where examples finding could be more challenging (coldspots). Other tools exploit StackOverflow discussions to suggest context-specific code snippets and documentation [5], [25], [26], [27], [29], [31], [38], [41].

[5]https://www.codota.com/

VIII. Conclusions

In this paper, we introduced FOCUS, a context-aware collaborative-filtering system to assist developers in selecting suitable API function calls and usage patterns. To validate the performance of FOCUS, we conducted a thorough evaluation on different datasets consisting of GitHub and Maven open source projects. The evaluation was twofold. First, we examined whether the system is applicable to real-world settings by providing developers with useful recommendations as they are programming. Second, we compared FOCUS with a well-established baseline, i.e., PAM, with the aim of showcasing the superiority of our proposed approach. Our results show that FOCUS recommends API calls with high success rates and accuracy. Compared to PAM, FOCUS works both effectively and efficiently as it can produce more accurate recommendations in a shorter time. The main advantage of FOCUS is that it can recommend real code snippets that match well with the development context. In contrast with several existing approaches, FOCUS does not depend on any specific set of libraries and just needs OSS projects as background data to generate API function calls. Lastly, FOCUS also scales well with large datasets by using the collaborative-filtering technique that helps sweep irrelevant items, thus improving efficiency. With these advantages, we believe that FOCUS is suitable for supporting developers in real-world settings. For future work, we plan to conduct a user study to thoroughly study the system's performance. Moreover, we will embed FOCUS directly into the Eclipse IDE.

Acknowledgment

The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223. Moreover, the authors would like to thank Claudio Di Sipio for his hard work on supporting the evaluation of FOCUS, and Morane Gruenpeter for her hard work on collecting the dataset from the Software Heritage archive.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications," in *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York: ACM, 2007, pp. 25–34.

[2] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju, "M3: A General Model for Code Analytics in Rascal," in *1st International Workshop on Software Analytics*. Piscataway: IEEE, 2015, pp. 25–28.

[3] R. P. L. Buse and W. Weimer, "Synthesizing API Usage Examples," in *34th International Conference on Software Engineering*. Piscataway: IEEE, 2012, pp. 782–792.

[4] A. Chen, "Context-Aware Collaborative Filtering System: Predicting the User's Preference in the Ubiquitous Computing Environment," in *First International Conference on Location- and Context-Awareness*. Berlin, Heidelberg: Springer, 2005, pp. 244–253.

[5] J. Cordeiro, B. Antunes, and P. Gomes, "Context-Based Recommendation to Support Problem Solving in Software Development," in *Third International Workshop on Recommendation Systems for Software Engineering*. Piscataway: IEEE, 2012, pp. 85–89.

[6] R. Di Cosmo and S. Zacchiroli, "Software Heritage: Why and How to Preserve Software Source Code," in *14th International Conference on Digital Preservation*, Kyoto, 2017, pp. 1–10.

[7] T. Di Noia, R. Mirizzi, V. C. Ostuni, D. Romito, and M. Zanker, "Linked Open Data to Support Content-based Recommender Systems," in *8th International Conference on Semantic Systems*. New York: ACM, 2012, pp. 1–8.

[8] R. A. Fisher, "Confidence limits for a cross-product ratio," *Australian Journal of Statistics*, 1962.

[9] J. Fowkes and C. Sutton, "PAM: Probabilistic API Miner," https://github.com/mast-group/api-mining, last access 24.08.2018.

[10] ——, "Parameter-free Probabilistic API Mining Across GitHub," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York: ACM, 2016, pp. 254–265.

[11] X. Gu, H. Zhang, and S. Kim, "Deep Code Search," in *40th International Conference on Software Engineering*. New York: ACM, 2018, pp. 933–944.

[12] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API Learning," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York: ACM, 2016, pp. 631–642.

[13] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, 1979.

[14] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," in *27th International Conference on Software Engineering*. New York: ACM, 2005, pp. 117–125.

[15] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[16] R. Kohavi, "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection," in *14th International Joint Conference on Artificial Intelligence*. San Francisco: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[17] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?" in *37th International Conference on Software Engineering*. Piscataway: IEEE, 2015, pp. 880–890.

[18] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow," in *28th IEEE International Conference on Software Maintenance*. Piscataway: IEEE, 2012, pp. 25–34.

[19] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical Learning Approach for Mining API Usage Mappings for Code Migration," in *29th ACM/IEEE International Conference on Automated Software Engineering*. New York: ACM, 2014, pp. 457–468.

[20] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio, "CrossSim: Exploiting Mutual Relationships to Detect Similar OSS Projects," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2018, pp. 388–395.

[21] P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, "Knowledge-aware Recommender System for Software Development," in *Proceedings of the Workshop on Knowledge-aware and Conversational Recommender Systems 2018 co-located with 12th ACM RecSys, KaRS@RecSys 2018, Vancouver, Canada, October 7, 2018.*, 2018, pp. 16–22.

[22] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "crossminer/focus: Icse19-artifact-evaluation," Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2550379

[23] H. Niu, I. Keivanloo, and Y. Zou, "API Usage Pattern Recommendation for Software Development," *Journal of Systems and Software*, vol. 129, no. C, pp. 127–139, 2017.

[24] D. L. Parnas, "Information Distribution Aspects of Design Methodology," Departement of Computer Science, Carnegie Mellon University, Pittsburgh, Tech. Rep., 1971.

[25] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging Crowd Knowledge for Software Comprehension and Development," in *17th European Conference on Software Maintenance and Reengineering*. Washington: IEEE, 2013, pp. 57–66.

[26] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter," in *11th Working Conference on Mining Software Repositories*. New York: ACM, 2014, pp. 102–111.

[27] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting Software Developers with a Holistic Recommender System," in *39th International Conference on Software Engineering*. Piscataway: IEEE, 2017, pp. 94–105.

[28] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis," in *38th International Conference on Software Engineering*. New York: ACM, 2016, pp. 357–367.

[29] M. Rahman, S. Yeasmin, and C. Roy, "Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions," in *Conference on Software Maintenance, Reengineering, and Reverse Engineering*. Piscataway: IEEE, 2014, pp. 194–203.

[30] V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York: ACM, 2014, pp. 419–428.

[31] P. C. Rigby and M. P. Robillard, "Discovering Essential Code Elements in Informal Documentation," in *35th International Conference on Software Engineering*. Piscataway: IEEE, 2013, pp. 832–841.

[32] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[33] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API Property Inference Techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[34] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining Multi-level API Usage Patterns," in *22nd International Conference on Software Analysis, Evolution, and Reengineering*. Piscataway: IEEE, 2015, pp. 23–32.

[35] M. A. Saied, H. Abdeen, O. Benomar, and H. Sahraoui, "Could We Infer Unordered API Usage Patterns Only Using the Library Source Code?" in *23rd International Conference on Program Comprehension*. Piscataway: IEEE, 2015, pp. 71–81.

[36] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based Collaborative Filtering Recommendation Algorithms," in *10th International Conference on World Wide Web*. New York: ACM, 2001, pp. 285–295.

[37] J. B. Schafer, D. Frankowski, J. L. Herlocker, and S. Sen, "Collaborative filtering recommender systems," in *The Adaptive Web, Methods and Strategies of Web Personalization*, 2007, pp. 291–324.

[38] W. Takuya and H. Masuhara, "A Spontaneous Code Recommendation Tool Based on Associative Search," in *3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. New York: ACM, 2011, pp. 17–20.

[39] C. Teyton, J. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 192–201.

[40] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web," in *23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington: IEEE, 2008, pp. 327–336.

[41] C. Treude and M. P. Robillard, "Augmenting API Documentation with Insights from Stack Overflow," in *38th International Conference on Software Engineering*. New York: ACM, 2016, pp. 392–403.

[42] G. Uddin and M. P. Robillard, "How API Documentation Fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[43] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining Succinct and High-coverage API Usage Patterns from Source Code," in *10th Working Conference on Mining Software Repositories*. Piscataway: IEEE, 2013, pp. 319–328.

[44] T.-T. Wong, "Performance Evaluation of Classification Algorithms by K-fold and Leave-one-out Cross Validation," *Pattern Recognition*, vol. 48, no. 9, pp. 2839–2846, 2015.

[45] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *23rd European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2009, pp. 318–343.