UNIVERSITY OF L'AQUILA

MASTER THESIS

Supporting the development of complex software systems by means of API function call recommendations

Author:
Claudio DI SIPIO

Supervisor:
Dr. Davide DI RUSCIO
Co-supervisor:
Dr. Juri Di Rocco

Corso di Laurea Magistrale in Informatica

Department of Information Engineering Computer Science and Mathematics

Academic Year 2017/2018

Acknowledgements

First, I would like to express my special thank to my advisor Prof. Dr. Davide Di Ruscio for his kind support and encouragement. He has been always open for any discussions, thus helping me advance my research.

I am grateful to Dr. Juri Di Rocco and Dr. Phuong T. Nguyen for helping me during the time I performed the research as well as for proofreading my thesis. They have been responsive and supportive of me.

Finally, I want to express my very profound gratitude to my beloved family and to my friends who provide me with unconditional support and continuous encouragement throughout the years of my study. This work would not have been possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Summary

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 [23]. Also, API documentation may be ambiguous, incomplete, or erroneous [28], while API examples found on Q&A websites may be of poor quality [17].

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* [24] 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 [20, 31, 34] or predictive modeling [7], still suffer from high redundancy [7] and—as we show later in the thesis—poor run-time performance.

To cope with these limitations, a new approach is proposed in this thesis to recommend to developers items that have been bought by similar users in similar contexts. 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?"*

A real big issue is how to perform a good enough recommendation in this context, balancing possible bias and putting the proper hints for the developer. Moreover, the form of the recommendation is also important because, in general, there are variety of possible suggestion such as code snippet, patterns for the methods, enhance documentation and all things that make a recommendation really usable for the current project.

This work is developed within the European H2020 CROSSMINER project [9] that aims at conceiving techniques and tools for developing new software systems by reusing existing open source components. Nowadays, the complex software system are really big and it is not so easy to select and deploy a component in the right

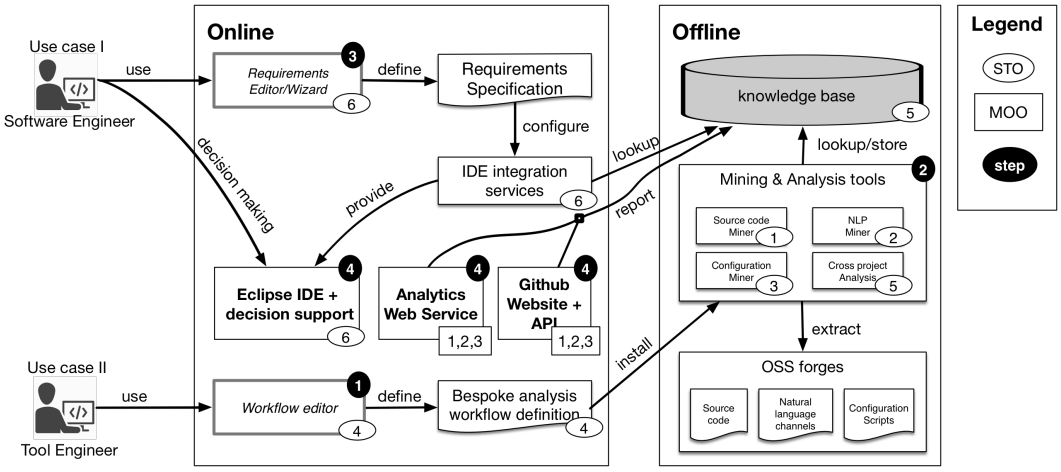

FIGURE 1.1: The CROSSMINER project at work

way. The planned CROSSMINER technical offering consists of the following analysis tools:

- *Source code analysis tools* to extract and store knowledge from the source code of a collection of open-source projects;
- *Natural language analysis tools* to extract quality metrics related to the communication channels, and bug tracking systems of OSS projects by using Natural Language Processing and text mining techniques;
- *System configuration analysis tools* to collect and analyse system configuration artefacts;
- *Workflow-based knowledge extractors* that simplify the analysis of a complex software system;
- *Cross-project relationship analysis tools* to manage a wider range of open source project relationships, such as dependencies and conflicts, based on user-defined similarity measures and the creation of project clusters;
- *Advanced integrated development environments* that will allow developers to adopt the knowledge base and analysis tools directly from the development environment, that providing alerts, recommendations, and user feedback which will help developers to improve their productivity.

Figure 1.1 shows an overview of the CROSSMINER approach at work. In such a context, the work presented in this thesis wants to propose a novel tool that perform API function call recommendations in the context of Java projects. It is integrated in the CROSSMINER knowledge base component in a flexible way.

1.2 Research Objectives

Figure 1.2 shows an overview of the CROSSMINER knowledge base underpinning the whole recommendation mechanism; the proposed approach gives support for the `APIrecommender` subcomponent in the picture. In particular, we combine the concepts of code cloning and patterns to retrieve real code snippets that show a concrete usage of the libraries used by the developer. We choose this approach because

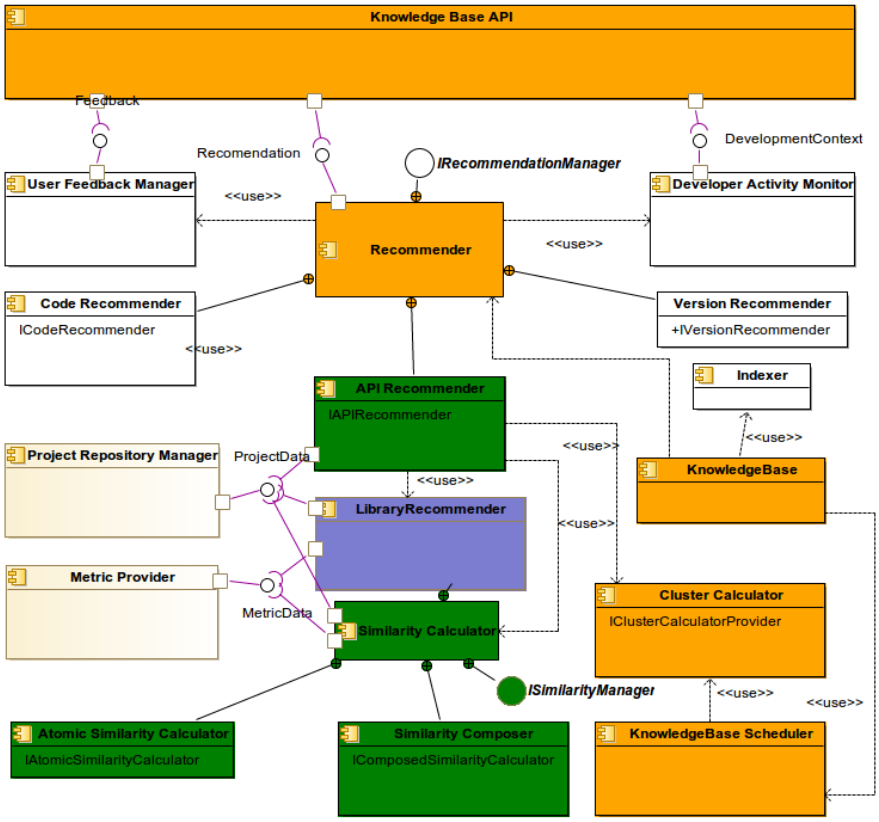

FIGURE 1.2: An overview of the CROSSMINER knowledge base

code snippets represent immediate hints in the developing context, as a concrete usage of a method or class is more relevant rather than a JavaDocumentation description or the list of imports. We exploit also the code cloning technique in order to search and retrieve possible suggestion.

1.3 Thesis Structure

The thesis is organized in the following chapters:

- Chapter 2 introduces the context of this thesis. The notions of recommendation systems and API mining are introduced. Code cloning techniques are also overviewed as underpinning the API recommendation approach proposed in Chapter 4 of this thesis.
- Chapter 3 presents an overview of existing approaches able to mine APIs. A comparative table is presented at the end of the chapter with respect to peculiar features.
- Chapter 4 presents the proposed approach, which relies on the combined adoption of the existing CLAMS and Simian tools presented in Chapter 3. In particular, CLAMs is used to create snippets of code representing recurring patterns in the analyzed APIs. Subsequently, Simian is used to analyze the developer's context, that contains what she is developing and, in particular, the fragment of code on which she would like to get recommendations. As final outcome, the proposed approach is able to provide developers with novel patterns in the

form of snippets of code that contains method invocation and all the variables that are needed to execute them in the proper way.

- Chapter 5 presents an evaluation of the proposed recommendation approach. The evaluation is performed by considering four metrics: *precision*, *recall*, *success rate* and *F-measure*.
- Chapter 6 concludes the thesis and performs an analysis of possible future works.

Chapter 2

Background

2.1 Recommendation Systems in Software Engineering

Software development is a challenging and knowledge-intensive activity. It requires mastering several programming languages, frameworks, design patterns, technology trends (among other aspects) under the pressure of ever-increasing arrays of external libraries and resources [25]. Consequently, software developers are continuously spending time and effort to understand new third-party libraries, existing code or how to properly implement a new feature. The time spent on discovering useful information can have a dramatic impact on productivity [6].

Over the last few years, a lot of effort has been spent on data mining and knowledge inference techniques to develop methods and tools able to provide automated assistance to developers in navigating large information spaces and giving recommendations that might be helpful to solve the particular development problem at hand. The main intuition is to bring to the domain of software development the notion of recommendation systems that are typically used for popular e-commerce systems to present users with interesting items previously unknown to them [22].

Robillard and colleagues define a *Recommendation System in Software Engineering (RSSE)* as "... a software application that provides information items estimated to be valuable for a software engineering task in a given context" [25]. In particular, when developers join a new project, they have to typically master a huge number of information sources [5] (often at a short time). In such a context, the problem is not the lack of information but instead an information overload coming from heterogeneous and rapidly evolving sources. Thus, RSSEs aim at giving developers recommendations, which can consist of different items including code examples, issue reports, reusable source code, possible third-party components that might used, documentation, etc.

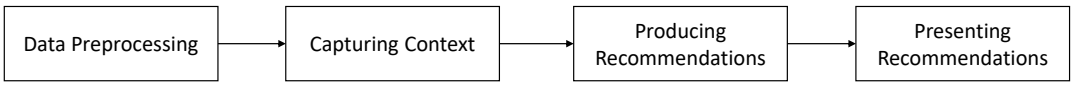

FIGURE 2.1: Main RSSEs challenges

The design and development of a RSSE has to take into account a number of challenges including those shown in Fig. 2.1 and described below [25]:

– *Data Preprocessing:* all the data sources that might contain valuable information for the developer have to be preprocessed in order to be subsequently analyzed. For instance, erroneous data needs to be removed, source code has to be parsed, bug tracker issues have to be analyzed, dependency graphs have to be derived etc.

- *Capturing Context:* to conceive recommendations that are really valuable for the developer, it is necessary to capture the user profile, and to properly represent the task the developer is working on. The context can be explicitly specified by the developer or implicitly inferred by the development environment e.g., by analysing the source code being edited, the third-party components used, the issue report that a user is reading, etc.

- *Producing Recommendations:* once data sources have been preprocessed and the developer context has been captured, the recommendation algorithms can be executed.

- *Presenting Recommendations:* produced recommendations should be sent to developers and presented in a timely and appropriate way. This means that depending on the task the developer is working on, recommendations can be represented as potential lists of issue reports, source code suggestions, links to Stack Overflow posts, etc. Beyond the recommendation items, developers should be provided also with explanations related to the suggested artifacts, which might be also ranked according to some criteria.

CROSSMINER can be seen as a recommendation system aimed at supporting developers while producing new software by integrating existing open source components. In particular, the *data preprocessing* challenge is addressed by the CROSSMINER analysis tools targeting source code, communication channels, and configuration specifications. The developer context is captured by an Eclipse-based IDE, which also produces recommendations that do not require particular and expensive data analysis. For more elaborated recommendations, preprocessed data available in a dedicated knowledge base is used. Both the IDE and Web based dashboards will be used to present the produced recommendations to the developer.

2.2 Learning APIs: issues and solutions

An API (Application Programming Interface) is defined as a set of procedures, protocols and objects that gives to the developers the necessary building blocks to implement a specific functionality. Depending on the context, these building blocks can be classes, interfaces and methods properly declared and used or intermediate software that acts as middleware in different situations as well as in the hardware context. The concept of API is strongly related to libraries that a developer uses and the kind of application that he is developing.

APIs can be complex and consequently their usability can be affected. In [15] authors analyse the difficulties that can raise when developers want to learn and use a given API. In particular, in [15] authors considered professional developers at Microsoft and try to understand what are the main issues in the API context. As starting point, the author of the article asks to a population composed by 30,000 people among engineers, developers and program manager in the Microsoft context. The survey starts with some questions about the developer's skills, to asses the knowledge of the participants. Additional questions are defined to characterize the problems on learning an API, strongly related to the context, familiarity with the application domain, the obstacles faced during the learning phase and so on. From the initial population, 83 developers were selected and distinguished as expert, and junior. From the answers of such developers five obstacle categories were identified with the aim of identifying the main problems that the considered developers

Code example category	Description
Code Snippet	It gives just a flavour of basic concepts of the API and their possible use
Tutorial	It is a quite complete example of a possible use of API, with multiple methods and functions usage
Application	It gives a well structured example of the API, given also the development context

TABLE 2.1: Main categories of code examples available with APIs

encountered during the learning phase of the used APIs. The category with the more answers is related to the obstacles due to the absence or inadequate resources in the learning phase, such as not suitable code example, partial information about the content of APIs, no reference about the task to perform, inadequate resources format, and insufficient high-level design and not well-formed structure of the API. Other problems are related to the structure, especially for debugging and runtime phases, the developer's background, technical issues and problems that arise during the runtime execution. In order to mitigate these issues, the documentation of an API must include good examples for the functions and code to develop, the support for case study scenarios, good organization of the relevant design elements.

In general, APIs follow the so called low barrier to entry, that consists to give just few information and key concept about the APIs structure and design, sometimes involving basic code examples. Although it is a very spread techniques, the survey underlines that is not enough to give a concrete help during the learning phase. Many developers involved in the survey, in facts, claim that they are often confused about multiple uses of a certain function, because there are different approaches to implement a feature but they not understand what is the proper one, considering also the context in they are. This issue affects also the general structure of the APIs, like the design that often generate confusion if it is not well specified and the survey respondents want also to understand what are the rationale behind the API.

Although the abstract design and the overall structure are important when we talk about APIs usage, the most immediate and concrete hints are related to code example [15]. In particular, the survey shows that the Microsoft developers not always understand the examples provided in the documentation. Table 2.1 shows the main three categories of code examples that could be provide by an online documentation or in a general *readme* file of a given API.

Looking at these categories, code snippets provide immediate hints but they are more related to specific issues, such as opening a connection or initialize a particular object. Tutorials provide more complete examples, with different methods, often followed by a textual explanation with the aim to show the rationale behind the code. The code example coming from the applications, instead, wants to give a general overview of the API features and it includes demonstration samples and complete open source projects. However, the Microsoft developers involved in the survey denoted that usually the code snippet doesn't provide how to put together all the small pieces. Another problem is that the code examples available from the Web are out-of-date and the maintenance of them is still an opening question. In [15], authors list possible improvements related to the code examples, by providing best practices in a certain situation, by giving the design behind the code example and by showing in a clear way how the considered API works in practice. All this information should be inserted in the documentation provided with the API. In [15], authors claim also that the behaviour of a given API can diverge from that described

in the documentation or from the code examples.

In [4] authors identify some issues and challenges that developers have to deal with to use and understand an API, such as the already mentioned inadequate documentation or the inappropriate abstraction in the overall design. Additionally, in [4] authors suggest possible workarounds for the different situations that the API designers should take into consideration. The ideal design flow when a API designer should follow includes gathering information from the stakeholders (mainly other developers that want to reuse the API functionalities), map the requested features to the proper components and set up the glue code to make an usable and understandable platform. However, this implies a well specified design flow and a lot of time, and sometimes the timing constraints on the publication of a new API do not allow to have a complete and exhaustive documentation. Even *Hello World* examples might not be enough especially if the target user is not an expert of the considered application domain. Another potential problem is represented by the orthogonal functionalities, also called *internal couplings*. They refer to situations when a method is strongly dependent with another one or its behaviour can affect other parts of the system. To avoid these situations, API designers have to keep the overall platform as simpler as possible, by limiting the exponential growing of the system.

Concerning the abstraction problem, the users are in the situation in which the requirements do not match the proper methods or interfaces provided by the API. Such a situation goes beyond a lack of information in the documentation, because it is a problem that affects the initial design of the API. A designer should set abstractions for each user's requirement, in order to maintain the proper mapping and to avoid the loss of functionalities. Another possible solution is to use the facade pattern to make more accessible the API itself. The last main issue underlined in [4] is the external dependencies, called *assumptions*, that an API could require to perform a certain operation. For the designer a possible solution can be the limitation of external calls to another API, maybe by reusing the internal API features.

All the solutions that authors provide in [4] are inspired by three key concepts: *i)* making the addressed problem as smaller as possible, by splitting the initial one to little problems that can be solved in less time; *ii)* approximating the final solution, by looking first at all to the user's requirements with the aim of satisfying them; *iii)* if the feature or the functionality to implement is really complex, an API's designer should choose an approach that is optimal in the average case.

As said before, usability plays an important role in API learning. In [14], authors focus their attention on this problem by performing a human survey. A cognitive framework is proposed to evaluate the human reaction that take place when a user is implementing a particular feature by means of a given API. This indicator wants to measure what is expected during the development and what really happens. With this technique, the authors gather the reactions and so the implicit feedback coming from the programmers avoiding in this way the bias led by subjective perceptions. Of course, the cognitive analysis is combined with the classical research question, about the difficulties to learn a new API, the understandability of the usage and the abstraction of the overall platform. Usability tokens are also extracted from the interviews to measure different developer's behaviours in different situations. Here below there are the list of tokens used in the questionnaire:

- *surprise*: it measures the unexpected behaviour of a particular component of the API, that seemed different at the beginning;

- *choice*: it evaluates the capacity of the developer to understand what is the optimal solution for solving a particular problem, like use the proper data structure;
- *missed*: it belongs to the abstraction problem, when the developer loses something because she does not understand the API design;
- *incorrect*: in this case, the developer uses in the wrong way the provided function or classes;
- *unexpected*: this token describes the situation in which the user takes decisions that are not documented by the API's designer.

The usability tokens are strongly related to the cognitive dimensions and they can be combined to depict a particular situation in which some obstacles occur at the same time. The experiment is overtaken on ABEL, an object oriented library for storing and retrieving huge amount of information. The results of this empirical study, that involves heterogeneous group of developers, show that the critical issue is to understand the relations between types and classes, not always understood by the participants. Other minors issues are the incorrect usage of the provided interfaces or the treatment of constructors without arguments, but in those cases the developers are able to overshoot these issues.

Overall, according to the studies mentioned above, understanding and using third-party APIs can be very challenging. Consequently, advanced techniques and tools are needed to automatically mine APIs with the aim of supporting developers during their adoption as presented in the next chapter.

Chapter 3

Mining APIs: an overview of existing approaches

Considering the state of the art, there are several approaches that face the problem described in previous section. The key point is to find a simple and effective way to extract API functions call and build a recommendation system able to provide developers with very suitable recommendations with respect to the developer context. Even though several techniques and tools have been proposed over the last decade, most of them share the main activities shown in Fig. 3.1. In particular, the developer context consisting on source code files is analysed to obtain corresponding ASTs. The retrieved abstract syntax trees are taken as input by a similarity analysis step in order to retrieve from the available knowledge the cluster of API client source codes that share some similarity with the code the developer is working on. Subsequently, a ranking activity is performed in order to finally recommend the developer with source code fragments that on one hand are similar to the developer context and on the other hand permits to grasp how other existing projects use the considered APIs. In the following, representative approaches implementing the process shown in Fig. 3.1 are presented and compared at the end with respect to a set of specific characteristics.

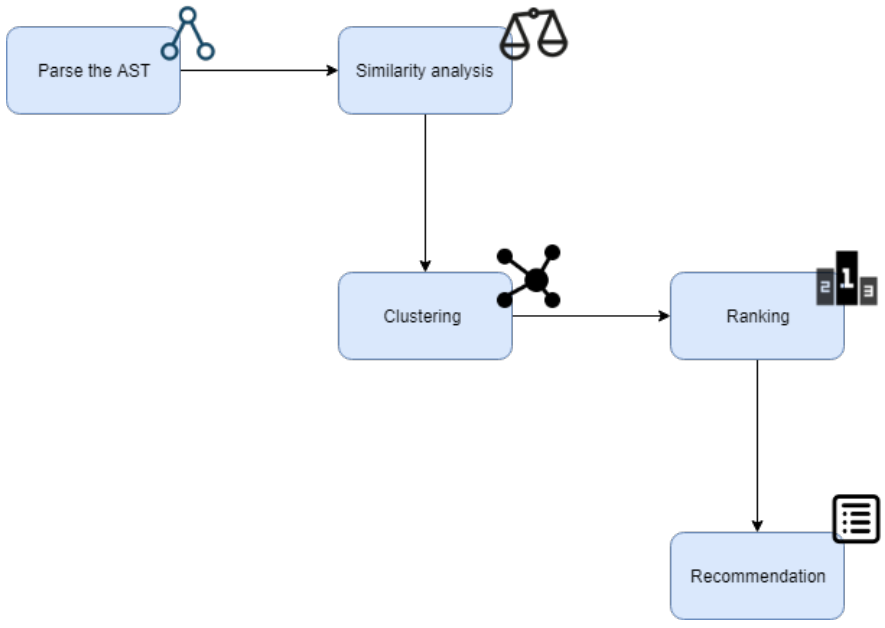

FIGURE 3.1: Main activities shared by existing techniques to provide developers with API recommendations

3.1 MAPO – API mining framework

In [35] authors propose the MAPO approach, which is able to perform method extraction using data mining techniques, to cluster them to have a more representative data, and to build the actual recommendations. Thus, the process consists of three main steps: source code analysis, API mining, and API recommender. First of all, MAPO parses the source code coming from Java Github projects that use Eclipse Graphical Editor Framework (GEF), by using JDT parser utilities that allow to analyze in deep a Java file and extract classes, interfaces, method invocations, and method declarations. Concerning the recommendations, MAPO considers as API method suitable for the recommender only those belonging to third-parts libraries, considering the GEF framework as external, class cast and creation associated to external classes and the method call that belongs to external classes. In order words, the authors ignore the libraries and so the internal API of the JDK.

Once MAPO have all method calls, the next step is mining their sequences to produce recommendations. To this end, MAPO uses similarity metrics, based on the name and the usage of methods to identify similar sequences, which are then clustered by using data-driven hierarchical clustering. In particular, the authors consider three levels of similarity given by class names, method names and called API methods and obtain in this way the similar sequence. Then, by using the Mathlab tool, MAPO performs classical hierarchical clustering to obtain the ranked list of representative sequences, transform them into transactions and put them into a database. Finally, the actual API recommender is an Eclipse plugin that allows developer to click on the method of interest, and to execute a query on DB to show possible uses by using sample code. The developer can also see details on the proper window tab in which the API methods are highlighted.

To validate this approach, the authors use a dataset consisting of 20 projects that used Eclipse GEF, run the tool and show the effectiveness of their approach by a quantitively comparison. To reinforce the validity, they also conduct an empirical study by considering a set of tasks to do and by involving a group of Java developers to solve those tasks.

3.2 UP-Miner – Mining succinct API usage patterns

A similar approach is proposed in [32] where authors presents the UP-Miner tool with the aim of improving MAPO in terms of accuracy. In particular, the aim of the authors is to achieve the succinctness but also the effectiveness of mined patterns that represent an enhancement of the previous approach. Once the authors define API usage mining, that is the optimal number of patterns under a given threshold, they propose a clustering technique based on the BIDE algorithm [30]. As first step, UP-Miner extracts API pattern sequences using the Roslyn[1] AST parser from the projects that compose the dataset. Then, they apply the SeqSim n-gram technique that takes two sequences and computes the similarity that is based on the shared items between them in term of objects and classes used and on the longer and consecutive sub-sequences rather than the shorter ones. This phase produces weighted results that are used for clustering objects: the maximum distance between two clusters is the maximum distance between two elements of those clusters.

As previously mentioned, UP-Miner uses the BIDE algorithm, which relies on the key concept of frequent sequence: a general sequence becomes frequent if its

[1] https://docs.microsoft.com/it-it/dotnet/csharp/roslyn-sdk/

super-sequences (namely the sequences that contains the considered sequence) is greater or less of the given threshold. The BIDE algorithm can extract the longest common sequences that are useful to divide the results into different clusters, even though this is not sufficient because at this point there may be some redundant cluster. So, it is necessary to apply once again the previous phase considering the cluster as usage pattern and this two-step clustering grants an improvement in term of redundancy considering also two different thresholds (one for pre-BIDE and another for post-BIDE application). Until this, UP-Miner addresses the problem of coverage but the aim of the authors is to reach also the succinctness of patterns. To do this, UP-Miner uses a dissimilarity metric that measures the diversity of a usage pattern from another and an utility function to maximize in order to obtain the better results, decreasing the threshold at each step of the algorithm implemented for this task.

Once UP-Miner computes the correct and more succinctness as possible usage API pattern, the authors show them to the developer by building a probabilistic graph in which each node is a usage pattern and the edge is weighted with certain probability. To produce this prototype and make some experiments that involves the developers in a similar way as we see in MAPO, the authors use a very large C# dataset as input files and compare their results with MAPO ones.

3.3 CLAMS – Summarizing API with clustering techniques

CLAMS is a tool proposed in [13] to produce as API recommendations snippets of code that represent a pattern for a certain library. As usual, the preprocessing phase is done by analyzing the AST of the source code (in case of CLAMS, projects related to 5 popular Java libraries) with a depth-first search by using JDT. This phase produces snippet of code that brings all information about API implemented in the code. The similarity technique is based on Longest Common Subsequence (LCS) and is more effective rather than an analysis at source code level. By using this technique, CLAMS creates a distance matrix that is used as input by the clustering module of CLAMS, that implements the hierarchical version of DBSCAN algorithm, called HDBSCAN, plus a post-clustering processing to eliminate the sequences that are identical to snippets for each cluster obtained. The aim of HDBSCAN is to isolate the less representative methods and, more in general, points into a distribution that are really far from the rest of the dataset. To this end, the algorithm uses a *core distance* to draw a circle on the points to exclude and then computes the mutual reachability distance that reduces the presence of sparse nodes in the dataset. Then, by applying Prim's algorithm, HDBSCAN calculates the minimum spanning tree among the closer nodes and finally sorts them by a hierarchical clustering (this phase is not present on the original DBSCAN algorithm).

There is a useful Python library used in CLAMS that provides utility functions to make all the previous steps in a very understandable way. The core module of CLAMS is represented by the snippet generator, that performs six main steps in order to obtain the final snippets that represent patterns. First of all, CLAMS replaces all literals present in the code with their abstract type by using srcML, a tool that produce XML files starting from source codes, and removes all comments. At the end of this step, we have code with the same structure of the original source file but more abstract. Then, CLAMS identifies API calls in that code and what are not related to them and creates two lists. In the next step, the authors identify all variable in the scope of the API sequences. To finish the process, the non-API statements

are removed and they put on top of the code snippet being generated, variable declarations related to APIs, plus of course the snippet of code related to those APIs. Thus, the snippet of code that is produced in this way is composed by a sequence of variables related to API classes and a possible use of them, with considering also the statements present in the original code (CLAMS retained also the structure of the code). The authors put a comment *Do Something* in the section of code in which the founded variables may be used.

Once CLAMS has these results, the snippet selector module finds for each cluster the most representative one by giving them a score. To this end, the authors use another algorithm that works on AST, called AP-TED that creates a distance matrix between two clusters and from this, calculate the similarity. Finally, CLAMS ranks all representative snippets using the definition of *snippet support* defined as follow: a snippet is supported if there is a file that contains a super-sequence of it. An example of extracted pattern is shown below and it is related to *Twitter4j*, a library that permits to create and manage connections with Twitter.

```
{
AccessToken accessToken;
final String CALLBACKURL;
RequestToken requestToken;
SharedPreferences prefs;
Twitter twitter;
twitter = new TwitterFactory().getInstance();
twitter.setOAuthConsumer(OAuthConsumer.CONSUMER_KEY,
    OAuthConsumer.CONSUMER_SECRET);

if(!prefs.contains(string) | !prefs.contains(string)) {
try {
requestToken = twitter.getOAuthRequestToken(CALLBACKURL);
String authUrl = requestToken.getAuthorizationURL();
// Do something with authUrl
} catch (TwitterException e) {
Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();
Log.e(string, e.getMessage());
}
} else{
accessToken = new AccessToken(prefs.getString(string, string),
    prefs.getString(string, string));
twitter.setOAuthAccessToken(accessToken);
}
}
```

Moreover, for human readability reasons, CLAMS beautifies code snippets by using the A-style tool, which removes useless spacing and fixes the indentation of the final snippets. For the evaluation task, the authors use 5 popular Java libraries coming from Github projects and they are *Apache Camel*, *Drools*, *Restlet framework*, *Twitter4j*, *Project Wonder* and *Apache Wicket*. A user survey is performed about the utility and real support given by CLAMS patterns. All this information are available at the CLAMS official site, that contains also the Github project, the original dataset and the instruction to set up the environment to launch CLAMS as standalone platform on Linux machines. As claimed by authors, the proposed approach is not dependent on the considered specific program language. The only constrain is related to the srcML tool to produce the XML files related to API patterns, but it is not a big issue from the adaptation point of view, as we can see later.

3.4 APIrec – API recommendation with statistical learning

APIrec tool [18] uses statistical techniques to keep track of the context in which a developer writes its own code, by analyzing the co-occurrences, and fine-grained code changes. Differently from the previous approaches, the authors keep track of the context in which the API method call may be useful. As usual, to extract the source code from the 50 Java projects randomly selected from Github, APIrec navigates the AST of the source code using the GumTree tool. It is important to remark that, in [18] authors use the term API to refer both external and internal method calls and APIrec performs recommendations only if the context of API is correct for a certain situation. Regarding the AST, a collection of atomic changes is called transaction and is stored in a bag, a particular data structure that we commonly find in the AST definitions. An important concept that is a key definition for the APIrec implementation is the code context, represented by code tokens related to the API that the developer is using. Taking into account the tokens, the authors consider both the distance and the order of this, as an API method calls have a specific call order and they are really effectiveness only if there are called in the proper way. To remark this feature, APIrec gives also a weight based on how near the token is by considering the distance matrix. Once they define this set of metric and concept, the authors show the inference model based on likehood scores by taking into account both change of the context and code.

The entire model is based on the correlation between a code token and another, expressed as we said in term of atomic changes in the AST and transactions. In particular, APIrec evaluates the probability that certain events, namely the transactions, occurs given another. This score is called association score and it is used to calculate the distance between two changes into the code as well as new methods that will take a part in the final recommendation. At the end of these computations, all possible scores and weights are calculated but it is not enough to perform the recommendations. In fact, it is necessary to apply machine learning techniques in order to train APIrec with the code changes and context. To do this, the authors use hill-climbing adaptive learning filled with three parameters: numbers of co-occurrence founded with fine-grained atomic changes, numbers of co-occurrences related to changes tokens and two weights. With this process, the necessary scores are calculated and APIrec is able to perform the recommendation by considering the most probable method calls for a certain API. The methods are also ranked by highest probability of usage but also distance, scope and dependency are considered in the ranking phase. To support their tool, the authors conducts several experiments over a very large dataset, including also analysis regarding code change context, user empirical studies, evaluation of accuracy, and predictions.

3.5 Buse-Weimer algorithm – Synthesizing API usage examples

In [2] authors propose an approach to produce automatically documentation for the project at hand in a human readable format. After a mining phase of the considered source code, the proposed algorithm extract API patterns and rearrange them in a more readable and effectiveness format. The focus is on Java documentation that provides useful hints and suggestions when a developer is implementing an API functionality. Although in general the Java doc helps in this kind of activity, it often lacks in something, as the examples are too general or it is not able to give a concrete

hint for the problem that the developer is trying to solve. So, the aim of the approach is to produce an enhanced version of the documentation that is really useful for the developers, by starting from retrieving API patterns, defined as the sequence of function call for a certain API class, called target class. Once this first step is done, the algorithm tries to produce a human-written documentation for the API, taking into account some characteristics such as the lines of code, that are 11 on average but 5 for the median. Moreover, the authors consider also the abstract initialization, abstract usage and exception handling. All this information is extracted using JDK utilities and they are validated by the authors, by putting more effort on the human aspect.

To validate the results, authors involve over 150 developers and collect their answers about the proposed results. By analyzing these statistics, a typical developer wants multiple uses for a certain class or API method and not only one, that may be not useful for his particular context. Another key point is the conciseness of the suggested snippet of codes, as well as the readability and variables names that must be related to the context, also including temporary ones to improve the readability. So, from this survey, we can identify four key points to achieve the human-written documentation: size of the code, readability, representativeness and concreteness. To reach these goals, the approach proposed in [2] consists of four main steps: the path enumeration, predicate generation, clustering and finally the output documentation. Starting from the path identification, they scan the code and identify acyclic part of the code that represent a path for the target class. Notice that this approach can led some bias but it happens in practice and the authors choose to stay close with respect to real implementation. As in the case of the the previous approaches, they parse the code and create clusters of the significant results with a distance matrix in order to build the related documentation. As the purpose is different from previous works, they are adding some clustering also on predicate and represent the abstract pattern as a graph. Then they compute symbolic execution over these paths in order to produce inter-procedural path predicates that are logical formulas used to represent the code and in particular, if a certain statement is reachable or not. From this abstraction, the algorithm computes user seeds that are local instantiations of fields, objects and whatever is related to the target class. Finally, from these it produces concrete uses, the real hints about the target class, that are stored in graph form in which edges keep trace about what happens before and so, in this way, we have the context as well as the chain of methods necessary to implements the features related to the target class.

To have better results, however, it is not enough to produce usage examples, as we seen so far with the other related works. Clustering is necessary to avoid heavy computations, and the proposed algorithm exploits the well-known k-medoids algorithm with some modifications, as the original algorithm is not suitable to detect distance about objects. The distance matrix that is taking into account by the k-medoids algorithm is based on the happens before relation obtained from the graph. So, at the end of this computation, the algorithm obtains the cluster and summarize them into abstract uses, represented once again in graph form.

The final step of the proposed algorithm is to produce the documentation. Starting from the abstract uses, it uses a topological approach to avoid the cycle and branches that appear in the code, as the final recommended documentation about the target class must be a flat file. Using this approach, the authors are able to retrieve a Java documentation related to the target class and respect also the Java syntax, so they avoid malformed documentation. Moreover, with this approach, the algorithm handles the exception treatment with try catch clauses, that are put always in the

correct order. As said before, the focus of this approach is on readability of the recommendation from a human perspective, so the authors set up a very big evaluation framework composed by 47 SDK classes ads dataset, the eXaDOC tool as concurrent approach and over 150 people as tester. The threats rise up from this evaluation are related to the validity of dataset (it may be not indicative and it doesn't represent all possible situation) and the background of the developers chosen for the evaluation (not expert in the field). However, this approach is useful to understand the concept of readability in the API recommendation domain.

3.6 APIMiner – Mining patterns for Android API

In [1] the authors extend their previous work in [16] to specifically target Android projects. The overall architecture of the approach consists of a pre-processing module based on slicing algorithm, and a ranker module that classifies the summarized methods considering all the lines of code (source code metric), the number of commits in the original repositories (called process metric), and the number of download (called usage metric). A Java Weaver tool is then applied to build and retrieve automatically the Java documentation related to the extracted methods calls. About the slicing algorithm, that represents the core of the system, it works as follows: first, it takes as inputs the API method call that the developer wants to analyze and all the body statements in which th method was found. Such a kind of analysis is performed off-line to avoid loss of time. At each iteration, the algorithm looks for similarity by analysing the list of variables that are present in the body statement of each method. If it founds some similarities, it puts the method in a list that represents the final recommendation, as in this list there are the most relevant methods. The slicing is performed both backward and forward; the former looks the writing variables while the latter analyzes the reading ones.

The API extraction is performed by considering the FP-Growth association as main index of similarity and run it by using the Weka tool that considers also the relation between two API calls, that are defined as sequence of methods that implement specific functions. The results are evaluated by defining two main metrics: *support*, defined as the number of patterns that include the method, and *confidence*, the probability of the method in the antecedent transaction. The final recommendation is displayed in the JavaDoc window in Eclipse and Android Studio, although the results is related to Android API functions. With respect to original APIMiner work, the authors also extend the graphic interface in which the recommendations are showed; in particular, the tool shows the complete chain of method calls related to the client method. An important drawback of this approach is that it works only for Android projects and it is no tested at the moment for other kinds of APIs.

3.7 API usage pattern recommendations with object usages

In [21] authors propose an approach based on clustering graph-based representation of Android applications. Starting from a graph representation of the so called object usage, a social network based on the co-existing relation among nodes is built. The aim of the work is to cover the less frequent API patterns in Android context. Before going in deep to the proposed implementation, the authors point out some definitions that are used in the approach. First of all, taking a general code snippet, an object usage is a list of methods that belongs to an API class that are used in the part of the considered code; in general, a fragment of code can contain more objects usage

and this feature is represented by a co-existence relation among objects usage. Thus, an object usage becomes a node in the graph and, if there is a co-existence relation, we put an edge between them and the weight are the number of co-occurrences. A usage pattern, instead, is the sequence of object usages that belong to different API classes. Once they define this to key concepts, in [21] authors underline the challenges underlying the API mining and propose an approach quite different those we have seen so far. They represent the object usage, and more in general usage patterns, with a graph; then, they define the co-existence relation and method call similarity, that are the baseline to define a similarity score among API calls. Regarding the co-existence relation, it is represented as a weight in the graph and represent the number of occurrences of the object usage in an API class. Basically, objects usage that are included in a particular API class are connected by a co-existence relation.

Authors propose two levels of clustering, one related to the co-existence relation and the other one is based on method calls similarity. For the first level, they propose a modularity index applied to community structures, defined as subnets of node densely connected. So, exploiting the graph format, they apply a greedy algorithm that calculates time by time the modularity, with the function goal that tries to achieve the maximum one. By running this algorithm, they get the optimum cluster and perform the first level of clustering. For the second level, they focus on the method call similarity and propose the Gamma index, based on consistent and inconsistent comparisons. A comparison is consistent if the distance of two object usage that belongs to different clusters is smaller than another pair that belongs to the same cluster. By applying these two techniques, they obtain a good coverage of usage patterns and avoid the redundancy by using abundance metric that describes how many times an object usage appears in the corpus. By means of this metric, the authors retrieve the most popular object usage but it is not enough to perform the recommendations. The last step is needed to map the objects usage into usage patterns in form of real code snippets that support the developer during the API implementation.

3.8 Jira platform – Automatic recommendations from feature requests

In [27], authors propose a tool for mining APIs and make recommendation within the JIRA platform. The main idea of the work is to analyze the pre-change and post-changed files and, in this way, find recommendations starting from a textual description of the input. The first step is the preprocessing of the input, necessary to clean the code and to give it a proper representation for the algorithm used in next steps. The textual preprocessing is done by taking into account two issues: *tokenization* and *stemming*. The first one involves the process of breaking into smaller pieces of code the entire document using delimiters as frontier and put them in a word token structures (also called bag of words). Stemming, instead, is related to the root of the word and transform it in stem word: in this way, authors summarize multiple words to avoid bias during the analysis. Once this preprocessing is done, the algorithm uses a term frequency indicator to count the number of times that a word appears in the document and consequently, obtain the most popular token. A similar measure is calculated also for the document and after a formula showed in the paper, authors retrieve weights and put them in a vector; in this way, each bag of words is associated to a weight that measure its relevance for the recommendation.

The proposed approach consists of three main parts: history based recommender, descriptor based recommender, and an integrator component that puts all together. For the history recommender, the algorithm compares the indicator on the Jira platform using a similarity distance matrix, starting from Jira fields and in particular by considering summary and description as key values of comparison and store them in its knowledge base called Historical Feature Request Database. Once the similarity scores are obtained, the algorithm performs the aggregation of these scores and perform the final comparison between the historical scores (calculated in this step) and the new feature request that is coming. To do this, they create a top-k request by looking at the history and choose the recommendation with the highest value among them. Description based component, instead, compares the new feature request with the Javadoc of the method, to have a more detailed recommendation. Subsequently, a preprocessing phase is performed to extract from the API documentation method calls by taking into account the @param and @return annotations. As similarity measure, in this phase they use cosine similarity between the current features and the preprocessed APIs. The last component is the integration, that merges the historical part with the description part, applies Gibbs sampling, and tries to calculate the best results at each iteration. In particular, they pick first the no-zero historical recommendations and then compare them with the results of descriptor module; from these results, the algorithm creates a top ranked recommendation related to the developer's features.

3.9 CodeBroker – API recommendation with reuse conducive development environment

In [33] authors propose the CodeBroker tool, which makes use of information retrieval techniques to make recommendations, which are based on Javadoc generated from Java source files. The tool is based on two communication channels with the developer that is interested in API recommendation: one is implicit since the system autonomously retrieves methods information and details from a given query. In this case it shows information organized in three layers, namely task relevance, signature details, and full JavaDoc. Regarding the second channel, it is explicit because the developer can refine the query based on its current needs and the system can adapt itself to this new situation and this technique is called retrieval by reformulation. Furthermore, CodeBroker creates a discourse model to represent the projects and the user. It uses LSA as similarity techniques to do the comparison and Java core libraries as dataset to test the tool. About the implicit communication, this term describes a set of information that can be inferred by the system without taking in account the user's hints; for the explicit channel communication, instead, the tool considers the user's need by looking its model plus the discourse one previously mentioned.

As first step, CodeBroker arranges the query by taking into account the context of the developer, called constrain part, the program that is the concept of functionality, and the code that is the embodiment of it. So, for the similarity analysis, the authors consider the context, the conceptual similarity, and the constrain similarity, that involved between two different signatures of the methods. About this last concept, they reuse it to apply the Latent similarity analysis (LSA) as main technique to perform comparison between two different API methods. Going in deep, the tool performs the so called signature matching that outlines the similarity of two components based on their signature structure. Although this comparison should be not

representative enough, the authors claim that is suitable for their purpose: so, the value of the comparison is in the range [0.0,1.0], that represents the match between two different methods. However, this first analysis is enhanced by taking into account the explicit communication channel to allow the developer to refine the initial query: the typical use case is that a user wants to improve the initial query with other components and so she changes the query in order to retrieve more information or very different components with respect to the initial one.

A previously mentioned, the approach introduces the notion of discourse model of the developer that represents the sequence of tasks necessary to implement the wanted features and it is used to improve the final components recommendation. At the beginning, this model is empty as the developer is starting to develop and she doesn't know at the beginning what are the components that are useful for the task. During the query phase, this model is filled with respect to the developer's choices. There is another model that CodeBroker takes into account during its analysis: the user model, that represent the developer's knowledge in abstract form. It is very different with respect to the discourse model and it is partially filled based on the developer skills. This model is used to remove possible components that the user already knows and so to avoid the redundancy problem. The authors define the knowledge as the number of implemented class by the developer and, in general, it is different from one user to another. The final recommendation is performed in terms of three layers: the first one shows the components related to the query, the second is linked to mouse movement (it displays signature information about the retrieved components) and, finally, a completed description in a HTML external page, included the JavaDoc. For the evaluation task, the authors used Java 1.1.8 core libraries and JGL library, with 663 classes and about 7,000 methods to analyze.

3.10 Usetec – Mining API usage example from the test code

In [36] authors propose an Eclipse plugin to cover the most recent and new APIs that typically lack documentation. To this end, the approach considers as baseline testing code developed in JUnit. The core aspect of the approach is based on heuristic slicing on the test units that are split in different test scenarios and, from them, Usetec extracts the code examples that represent the recommendations. The main issue of using testing code is to separate the different test scenarios into different parts, in order to have recommendation for the right context. In the literature there are no approaches that perform this kind of separation, so the authors are forced to implement a new approach, based on code patterns that represent different kinds of methods in the JUnit code. They represent five categories of code patterns: the first one contains only assertion methods, which are used into result verification phase, and represented by regular expression to summarize the content. The second code pattern type is composed of assertion and non assertion methods, such as data declaration and re-initialization of variables. In the third scenario, we have at least a method invocation of the API that is under testing. In this case, it is not enough to simply slice the sequence because the authors want to extract the data-relevant statement for the recommendations. So, they used the proposed heuristic slicing algorithm to achieve their aim. These extracted statements, combined with the relevant statements of the second scenario, form a complete scenario. Finally, in the last scenario, Usetec analyzes the unit test that includes sub sequences of method invocations. To do this, the authors use the already mentioned LCS techniques in order to complete the scenario and give a more accurate recommendations.

The Usetec supporting tool retrieves the testing method invocations using name convention techniques, that is based on similarities among testing classes of JUnit. Then, by using the code patterns described above (in the following order: pattern 1, pattern 2, pattern 3 and 4), the tool is able to extract the code examples from the test methods. The last step is the clusterization of the results, in order to have the most representative code examples. For this purpose, Usetec uses the LCS similarities and the harmonic mean to identify a cluster. The final recommendations consist of automatically retrieved JavaDoc together with the usage examples shown in a GUI in Eclipse.

The evaluation of the approach involved about 200 extracted methods for each project and they selected manually a golden set to do the comparison. Moreover, they involved human evaluators that had experience with Java. Usetec is compared with eXoaDoc tool on the minimum edit distance.

3.11 PAM – Parameter-free probabilistic API mining

In [8] authors presents the PAM approach, which is able to extract patterns directly from the source code of a given project. It follows the same technique shown in MAPO by visiting the AST but it does not consider conditional statements such as if else structures. At the end of this process, the approach retrieves the list of API calls in the form of method invocations by considering their qualified name. Notice that PAM represents an improvement of the original MAPO approach because it is able to dynamically infer the call sequences. When this extraction phase is finished, the probabilistic model if considered to retrieve the most probable and useful API calls in the form of patterns. The model is based on a generative algorithm that takes as input the API call patterns and generates the interesting patterns for each of them. To this end a greedy algorithm is also used to maximize at each step the probability to choose more interesting patterns starting from the original sequence. All these concepts are used in the main algorithm used by PAM, called structured EM algorithm. This algorithm requires some form of training data, represented by client methods and the associated probability to determinates the more interesting related API patterns. All the steps are independent so the EM algorithm calls the previous algorithms in a parallel way. At the end, the authors have as results the list of the most API significant patterns with the associated probability.

The dataset used by PAM is the same of CLAMS. PAM is implemented as a Maven project available on Github. As input, it takes a file in ARFF format (the same format used also in CLAMS) that represents the client API methods. We can parametrize the execution with different ARFF files, by setting the maximum number of iterations or structured steps. Here below there is an example output of PAM and it consists of a ranked list of invocations depending on their probability to occur.

```
prob: 0,02059
[java.lang.String)]

prob: 0,01866
[java.lang.String]

prob: 0,01138
[int)]

prob: 0,01127
[int]
```

```
prob: 0,00922
[com.google.protobuf.ExtensionRegistryLite)]

prob: 0,00916
[java/io/PrintStream/println(java.lang.String)]

prob: 0,00763
[java/lang/String/equals(java.lang.Object)]

prob: 0,00762
[java/util/ArrayList/ArrayList()]

prob: 0,00725
[java/util/List/size()]

prob: 0,00702
[com/google/protobuf/GeneratedMessage/Builder/onChanged()]

prob: 0,00675
[java/util/Map/get(java.lang.Object)]

prob: 0,00637
[java.util.Map)]

prob: 0,00559
[java/lang/StringBuilder/append(java.lang.String)]

prob: 0,00551
[com/google/protobuf/GeneratedMessage/getUnknownFields()]

prob: 0,00544
[java/util/Map/put(K]
```

3.12 Summary

Table 3.1 shows an overview of the approach previously summarized. They are classified according to the following features:

- *Used parser*: it specifies what are the techniques adopted for parsing the source files or AST;
- *Similarity*: this parameter indicates the degree of similarities and what are the algorithms involved in this phase;
- *Clustering*: it refers to the employed clustering techniques;
- *Supported language/platform*: this feature specifies the language and/or the platform that has been used to validate the considered approach;
- *Produced recommendations*: it refers to the kind of recommendations the considered approach is able to produce.

Approach Name	**Used parser**	**Similarity**	**Clustering**	**Supported language/-platform**	**Provided recommendations**
MAPO	JDT	API call sequence	Data-driven	Java	API usage patterns
UP-Miner	Roselyn AST parser	SeqSim technique	BIDE algorithm	C#	Probabilistic graph of APIs
CLAMS	JDT parser	Distance matrix	LCS,HDBSCAN	Java	API usage patterns
APIRec	GumTree AST parser	Association-based model	inference mode	Java	Most frequent API calls
Buse-Weimer algo	Symbolic execution for path enumeration	Distance matrix	K-medoids algorithm	Java	Human readable documentation
APIMiner	Slicing algorithm	Structural similarity of APIs	FP-growth with WEKA tool	Android	Documentation enhancement
API patterns with object usages	Extract object usage	Co-existence relation	Modularity index and Gamma index	Android	API usage patterns
JIRA platform	JDT	Cosine similarity	Integrator component	Java	Top ranked methods
CodeBroker	Back-end search engine	LCS	Discourse and user model	Java	Relevant tasks, signature and JavaDoc
Usetec	JUnit	heuristic slicing algorithm	LCS technique	Java	API method invocations
PAM	JDT	Structured EM algorithm	Probabilistic model	Java	Ranked list of method invocations

TABLE 3.1: Summary of the considered API mining techniques

Chapter 4

The proposed approach to recommend API usage patterns

4.1 Overview

In this chapter, we present our proposed approach to API function call recommendation. This conceived architecture is depicted in Figure 4.1. We exploit CLAMS as the tool for extracting usage patterns. Furthermore, Simian is used to detect code cloning. At the beginning of the process and after a preprocessing phase, we have patterns files and the developer's file, represented by a single string. This helps keep trace on the context in which the user is developing. Then, all these files are used to extract the recommendations in form of patterns by using Simian integrated in Eclipse platform following the options specified in the next sections.

At the end of this phase, Simian retrieves the cloned clone between the developer's file and the CLAMS patterns related to the library that the developer has included. By using these files, the tool recommends by removing the cloned part and suggests the user the new lines of code that represent the missing patterns. We are going to describe the entire system and as well as the integration in the next sections. In particular, Section 4.2 gives an introduction to code cloning taxonomy. Some code clone techniques are going to be presented in Section 4.3. In Section 4.4, we are going to introduce Simian.

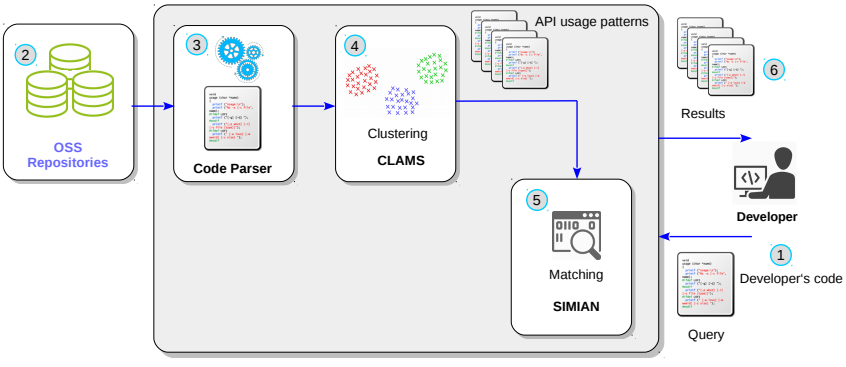

FIGURE 4.1: API usage patterns recommendation using CLAMS and Simian

4.2 Code cloning taxonomy

To support recommendations, we choose an approach that involves code cloning analysis. When we analyze a software complex system, it is quite common to encounter duplicated lines of code, especially in big projects. This is done by the copy and paste technique, since it offers an easy solution to the current problem that the developer is facing, thus helping save time. Although this works in theory, it is not the best solution because the cloned code might bring some unexpected side effects on the other parts of the software.

The purpose of clone cloners is to analyze software complex system in order to find the common parts among them. During the analysis, it is also important to clarify how the code has been compared and what the word cloned really means. Two fragments of code could be declared clones also if they are not exact duplicate but even if they share most of the structure (such as variable name, statement, and method calls). So, a code cloning tool must analyze also the structure, the AST and the token composition, plus the textual plain code. There are several techniques in the code cloning field, and the work by *Chanchal et. al* [3] provides a clear overview on this topic. In general, a clone detector tries to find the similarities between two fragments of code. Such an analysis depends first of all on the level of details that the tool wants to reach. For example, each code cloner could specify a different similarity function in order to set the level of cloning. They differ also in terms of the comparison of two snippets of code, such as AST, textual comparison, etc. There are a lot of concepts and techniques and Table 4.1 depicts a taxonomy to classify the activity of code cloning.

Code cloner type	**Level of similarity**
Type-1	The code fragments differs only from the white spaces, comments ad layout
Type-2	Two code fragments are syntactically equal except for the same conditions of Type-1 plus identifiers, literals and name variables
Type-3	This kind of clone detector looks for variation (add, delete or change) in statement that appears in the fragments, plus the previous conditions
Type-4	We have this kind of cloner when the computation that the fragments perform are equal without considering the syntactic implementation

TABLE 4.1: Code cloner tools taxonomy

The most common code clone detecting approach is matching textual content, as the transformation and normalization phases are often very slight. The fixed lines used in the comparison are called window and they are encoded with a hash function. To obtain fragments with different lengths, the tool apply simply a slicing on the window. The lexical approach, instead, works on the tokens obtained from the source code through the compiler-style lexical analysis. This technique is more robust because it avoid the whitespaces and other dirty code that we want to exclude from the comparison. The big issue of this approach is that it not consider the syntax; so, the founded clones may overlap different syntax units but preprocessing or postprocessing can avoid this situation, like pretty-printing techniques to format the code in a better way.

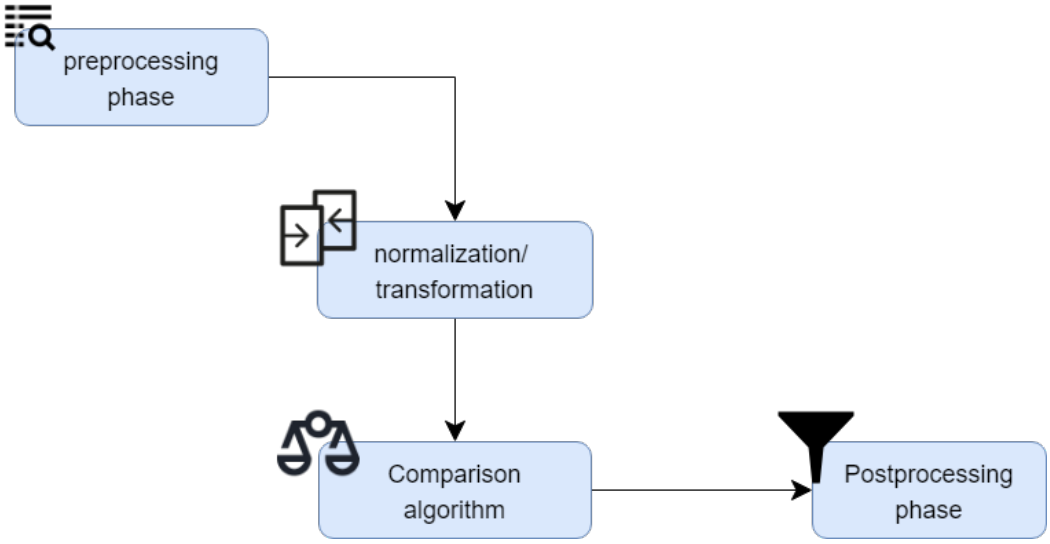

FIGURE 4.2: Typical code cloning phase

4.3 Code cloners: techniques and features

In this section, we introduce some code cloners that use first track the cloned code and then return the results back to the developer. Among the techniques, we focus on the string matching one, as it is closer to the code snippets used in our approach. By this technique, there are many important issues that need to be taken into account as suggested in [26]. First of all, it is necessary to avoid false positive, i.e., the part of code that is marked as duplicated but is actually not.

Moreover, scalability plays an important role since a code cloning tool must analyze complex software system which grows exponentially in recent years. An highly related issue is also the analysis of multiple languages: a good code cloner, should be able to recognize the cloned code beyond different syntaxes and lexical differences among the various programming languages.

Finally, the apply some filters on the results, as the single line is not enough to be significant for the cloning analysis. Therefore, they set two metrics, one related to minimum length sequence of the duplicated part, and the second related to the maximum gap size, measured between two compared sequences. To improve the recall of the results, they also set up a second normalization based on regular expression and they test the overall approach on the same dataset of Bellon's study, composed by the Cook and Weltab application.

4.4 Simian: A code clone detector

As already mentioned in Chapter 1, it is possible to perform recommendation at different levels of abstraction, e.g., pattern, documentation, code snippets in order to give complete and useful suggestions to developers. In our approach, the overall idea is to perform API recommendation at the level of code snippets that represent the patterns related to the developer's file. To do this, we exploit the code cloning analysis presented in the previous section. We choose Simian, a project developed in Java that performs code analysis for many languages such as Java, C, C#, Ruby, JavaScript, to name a few. Following the taxonomy in Table 4.1, Simian is a Type-2

code cloner with flexible options on variables, literals, modifiers. All possible options are described in Table 3, although we discard some options that are related to languages different from Java, such as C.

To test the main functionalities of the tool, we run a jar file available on the website [11] by specifying options and the input file. As output, we get on the console the textual representation of source coordinates that describe the number of duplicated lines and the original source files. It is possible to change the type of output using the formatter option (in Table 3). As mentioned in the tool classification, Simian has the following features:

- It supports object oriented and Web languages;
- There are no constraints regarding any additional tools or dependencies;
- It is language and platform independent;
- It has free granularity and it analyzes line by line for each source file;
- It exploit fingerprint techniques for code representation;
- It applies transformation on variable, types and literals using options.

Among the main drawbacks, Simian does not include IDE supports and one has to do a manual integration which is going to be discussed in the next section. Moreover, it doesn't include any pre-processing or post-processing phases as well as an heuristic algorithms for the threshold or an aggregation phase at the end of the process. Also there is no specific evaluation regarding the algorithm complexity.

Table 4 describes a very simple scenario in which we pick four pairs of Java project with the description of their main features and how lines of code are in common.

From the scenario, we can see that similar projects share more common lines of code, like the first two pairs that are both Eclipse plugin projects. This happened because these projects are built with the same wizard procedure and share the initialization phase of the plugin, such as the activate method. The second pair of projects, instead, doesn't share any lines because the CyberGea project is another plugin projects that uses Mqtt paho client and JDBC libraries mainly while the Neo EMF project is related to construct a metamodel with the aim of creating a Neo4j database. Then, we compare Cybergea with another project developed in this university, the Scuna project that involves Swing framework and the MySql library for Java. It is worth noting that in this case, the two projects share only the latter part and there are few lines in commons.

The last example is related to Java Servlet in the Web context and Simian analyzes two kinds of servlet, one without and the other with the handling of session. The tool is able to detect the lines in common, as the servlet shares the initialization part in common like the doGet and doPost methods. All the projects shown in this simple comparison are developers in the context of university projects and their aim is only to show an example of the code cloning activity of Simian. As an additional remark, for this comparison we use the default options for Simian and launch it from the console.

Before going in deep in the explanation of our approach, we recall some existing approaches that deal with the problem of API recommendations.

Option name	Default value	Description
-threshold	6	This option fix an lower bound on the number of duplicated lines of code (if present)
-formatter	none , possible values: plain, xml, emacs, vs (visual studio), yaml, null	This option is used to obtain results in a specified format
-reportDuplicateText	disable , type + to add	With this option, the duplicated lines of code present in all projects are printed on the console
-language	disable , type + to add	This option specifies the language of the input files to compare
-defaultLanguage	disable , type + to add	If the file type is not specified, Simian inferred the type and set it as default
-failOnDuplication	able , type - to remove	If this option is enabled, it causes an exception when the checker finds duplicate code
-reportDuplicateText	disable , type + to add	With this option, the duplicated lines of code present in all projects are printed on the console
-ignoreRegions	disable , type + to add	It ignores block in region structures (only for C# programming language)
-ignoreBlocks	disable , type + to add	It excludes specified blocks from the comparison (start/end line must be specified
-ignoreCurlyBraces	disable , type + to add	The curly braces are ignored so it should be match as duplicate line
-ignoreIndentifier	disable , type + to add	With this option, the variable with different identifiers match as equal
-ignoreIdentifierCase	able, type - to disable	This option doesn't consider the case of identifiers present in the code: so Name and name are considered equal
-ignoreStrings	disable , type + to able	This option considers all strings in the comparison and doesn't care about the form
-ignoreStringCase	able, type - to disable	Same as the previous option but considers the upper and lower case the same
-ignoreNumbers	disable, type + to add	This option considers different numbers as equal
-ignoreCharacter	disable, type + to add	With this option, all character type are marked as equal
-ignoreCharacterCase	able, type - to disable	Same as ignoreStringCase but considers char by char
-ignoreLiterals	disable, type + to add	All literals should be seen as equal for Simian
-ignoreVariableNames	disable, type + to add	This option allows Simian to see different variable names as equal
-ignoreModifiers	able, type - to disable	This option doesn't consider modifiers of methods (public, private, protected as element of diversity in the code)

TABLE 4.2: Simian options used in the experiment

Projects name	Main features	Similarity level (duplicated LOC)
ADTPlugin, ModiscoPlugin	Plugin projects created with same wizard	39 lines of code in common
CyberGea, NeoEMFExample	Very different projects that realize different features	No lines in common
CyberGea, Scuna project	The projects share only database part	12 lines on common
Simple Servlet, ServletSession	Web projects with servelts	35 lines of code in common

TABLE 4.3: Projects considered in the comparison

4.5 Simian within Eclipse platform

Once the input has been defined, there is another step in order to use Simian for API recommendations: it is necessary to integrate with the Eclipse platform to have a more flexible and usable version of the tool, as Simian doesn't support IDE integration. As already mentioned in the Related Work section, the basic version of Simian is a jar file launched from the terminal console with different options (see the Table 3). Although it is easy to use, this version is not very suitable for our purposes. To keep the integration with a Maven project, we create a repository that contains an update version of this jar and put it as reference in the *pom.xml* file of the project. The following main classes need to be set:

Simian class	Description
Auditstener	This class is needed to initialize Simian and collects all notifications from events that occur
Block	This class represents the duplicated block of code as an object and we can interact using method utilities
FileLoader	It is used to load all files for the comparison, with the method load
Checker	This class is used to perform the real comparison by calling the method check() on preloaded files
StreamLoader	Once we load files and create the Checker, this class loads them into the Checker
Options	A data structure that encapsulates all options enabled for the comparison
Option	This class represents a single option and we can specify it by accessing to a static field
Language	This class contains static fields to set all supported languages as type of input files
CheckSummary	It contains all statistical data such as cloned code, number of total files, requested time and duplicated files

TABLE 4.4: Overview of the classes of Simian

The project has the following structure, divided into different sub-packages:

- business: it contains all the interfaces that expose the utility functions;
- business.impl: It contains the classes that implement the interfaces and represents the business logic of the entire application;

- model: It contains the representation of the SimianPattern object which is useful to keep all information for the cloning phase;
- evaluation: It contains all the functions necessary for the evaluation framework, specified in the proper section.

There are three main classes as follows. SimianDataExchangeImpl collects all data needed for the analysis; SimianFileUtilitiesImpl contains all the operations related to files; and ApiCallRecommenderImpl takes all information provided by these two classes and performs recommendations. In particular, SimianDataExchangeImpl implements the original Simian class shown in the table above and it initializes the tool in order to perform the code cloning activities. Among the implemented methods, we use the function *block()* to retrieve all necessary information for a duplicated block of code and the file that contains it. The *endCheck()* function is called at the end of the process and manipulated the class CheckSummary mentioned before. In this way, we can obtain all information the total number of analyzed files, the duplicated ones, the time required for the comparison and the total number of blocks. This class implements also the ExchangeData interface, that is used as a bridge for ApiCallRecommenderImpl class, as the Simian interface provides only void methods without the possibility to return the necessary information.

We describe the APICallRecommenderImpl class, that collects the data coming from the previous class and analyzes them in order to produce the recommendation. The main function is *findPattern()* that loads the necessary files and launches Simian analysis by exploiting the Aulistener interface. The files are loaded in pairs, in which we have the snippet of code coming from the developer's file and the other component is the list of CLAMS pattern. During the analysis phase, it is necessary to check the files and in particular, we have to discard from the analysis the files that contain duplicated blocks within themselves. It is possible because Simian retrieves for each pair the files that contains a duplicated blocks of code and, in this way, we can reduce some bias. This leads to the fact that in the developer's snippet we can have some duplicated lines of code that are not useful for the recommendation. Once we have the block, it is possible to create the object SimianPattern that represents the discovered pattern for the snippet among the CLAMS results.

The class for this object belongs to the model sub-package, that represents the extracted pattern. Following the POJO structure, we have the *getter* and *setter* functions for each property of the object, gradually filled during the analysis. We have duplicated lines to store the cloned code, the filename of the pattern and the elapsed time for each pattern that Simian has found in the analysis. In this way, we can easily write to a file to show the final results in a more comprehensive way. The original Simian output, in facts, shows only the duplicated lines and CLAMS puts the patterns in a ranked list but without the context, represented in this case by the import at the beginning of the file. Thanks to this structure, we can also rank the patterns from the one that have more lines in commons, using the proper field in the wrapper class.

The last main component is SimianFileUtilities, in which we open all necessary files, write the recommendations and create temporary files for the code cloning activity. All these classes are integrated in test class that calls in the proper sequence all the methods to perform the final recommendation. In particular, the method *scan()* takes all files that contain patterns extracted by CLAMS while the function *createTemporaryFile()* creates the temporary files to perform the comparison. To extract the ground truth part used in the evaluation part, we use *parseAST()*. This function uses the JavaParser library to traverse the AST and take the body of the method that we

are interested in. In this way, Simian inputs are only partial fragments of code that represent the typical developing scenario.

The project contains also classes with the evaluation task, such as the function to build the Rascal project structure in order to analyze the corresponding method invocations and to apply the metrics on them. More details are going to be provided in the evaluation framework section. The overall architecture is depicted in Figure 4.3.

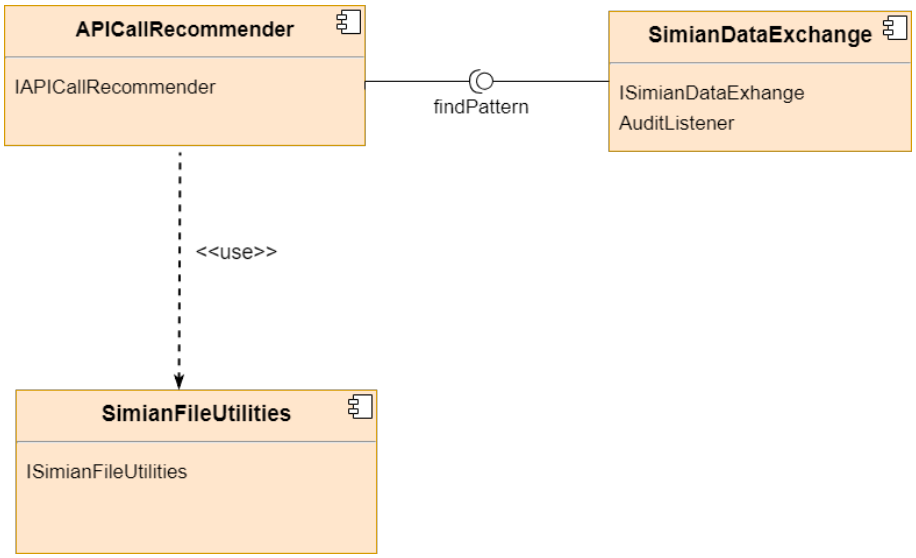

FIGURE 4.3: Component diagram

4.6 Preprocessing

The input files that are necessary to initialize the tool, we have from one side the developer's file with the real snippet of code that she is implementing and needs support for this. From the original file, we extract a portion that represents the context of the recommendation that can be a list of method invocations or simply a list of variable declarations. This portion is called ground truth and it is the part used as the context of the recommendation. On the other hand, there are patterns mined by CLAMS, in the form of ranked Java files sorted by rational specified in CLAMS paper. These files contain patterns, defined as a sequence of API method calls that define, instantiate class belonging to the APIs contained in the developer' string. The number of the files and also their dimensions in term lines of code depends on the considered libraries. As Simian is a tool based on file comparison, we use temporary files of Java to do the comparison, after the process, the files are discarded to save memory.

This preprocessing phase is required as mentioned in the Related Work section, the tool doesn't include any built-in preprocessing. To extract the snippet of code from the developer's file, we use Java Parser, an open source project that allows one to analyze, modify and generate Java code, to traverse the AST of the input files and take the body of a method randomly selected that composes our ground truth. We consider only compilable files, otherwise Java Parser is not able to build the corresponding AST for the analysis. We take into consideration only the ground truth files because we are in the typical scenario in which the developer is starting to implement some features and she has written only some code fragments. In general a

recommendation relies on the context in on which the developer is working. In our case, the context is the developer's code snippet with imports and variable declarations.

For CLAMS, we create a golden set of 5 libraries, chosen among the set of 15 libraries available at the authors' website. For each of them, CLAMS retrieves a list of patterns represented by Java files and their number depending on the library that we consider. The precision and the lines of code of this list depend on the clients and examples files. We can also decide which methods and classes that CLAMS analyzes through the namespaces file in the proper folder. This phase is necessary because Simian doesn't have a well-defined preprocessing phase and needs to defined in a proper way. We can define the entire procedure that just describes a human preprocessing, since we don't use any automatic procedure or heuristic algorithm to select the input files. We select relevant client files, and avoid test classes. For CLAMS pattern, we don't put any limitation and we consider all possible patterns retrieved for a specific library.

4.7 CLAMS adaptation

Regarding the output by CLAMS, we describe the original structure as well as the necessary modification to integrate it into our tool. The complete source code has been made available by the authors. CLAMS is written in Python and uses srcXML and Astyle to produce XML files and to formatter in a human-readable way the code respectively. As claimed by the authors, there are no specific constrains about the technologies to use in case of a new implementation. As input, CLAMS takes two kinds of files: client files that represent the real project on Github related to the dataset that authors use for evaluation phase while example files are used as training set.

All these files are collected in a folder and CLAMS loads by getting the path. Moreover, there is a namespace file that identifies the name of classes used in the clients and example files by using their complete namespaces such as org.codehaus.jackson. The last input used by the main.py file, that is used to initialize the platform, is the list of methods that are represented by an ARFF (Attribute-Relation File Format) file [10]. It is an ASCII text file that describes a list of instances sharing a set of attributes, specified in the header section. In our case, the attributes are the method declaration (the caller) and the method invocations (the calls).

There is a phase of preprocessing in which CLAMS extracts API calls and their AST using JDT utilities and represents them in XML using srcXML. The core of the project is the snippet generator module (represented by summarise.py file) that takes as input a source code file (Java in this case) and using srcXML they first replace literals with XML types and delete comment. Then, they separate the API code from the code that doesn't contain API calls and highlights the variable in local scope of API. Finally, the code without API call is removed and CLAMS adds some comments near the API statement and needed variables. The approach considers also the classical statement like if-else structure as a part of API statement.

For clustering, both HDBSCAN and k-Medoids algorithms are used that differ only in the precision of the returned snipped (HDBSCAN is more accurate but k-Medoids covers more methods). The rank is based on the example files that contains a sequence of API call; if the sequence within the file is a super-sequence of the sequence of snippet that we considered, so this snippet is supported, and its rank is increased. In the result folder, CLAMS puts the library that we want to analyze,

the methods, the source file (both in .java and xml format), some JSON files that represent all information about a method (class, package, rank, id) and the ARFF file related to the library.

For the integration step in our platform, it is necessary to slightly modify the original approach to have better results. In particular, if we use the pattern of CLAMS as they are, there are some bias since through srcML, CLAMS substitutes the literals with its own type and Simian is not able to detect them as cloned code, even using all available options regarding the code. To avoid this, we modify the function that substitutes literals, putting some default values instead of srcML types. This modification doesn't affect the validity and accuracy of extracted path as it is just a matter of modifying literals with another and allow Simian to avoid bias in the code cloning analysis.

4.8 API recommendations

At the end of the preparatory phases, we describe now the core of this project, the API recommendations. Once Simian is launched, it performs the detection of code cloning activity on the CLAMS patterns files and the developer's code snippet. The typical scenario is depicted by the use case diagram in Figure 4.4, in which the developer asks for recommendations.

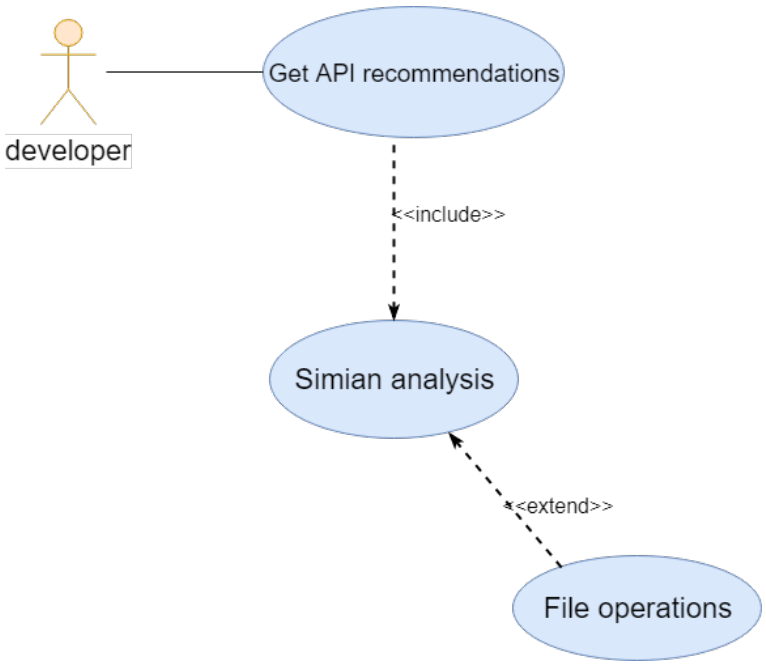

FIGURE 4.4: Use case scenario

The notion of cloned code depends on the options that we have selected and turn on: the mandatory option to enable is the threshold, that sets the minimum lines of code in commons, reportDuplicateText, otherwise we couldn't show and manipulate the result and language that is Java because we analyze projects related to it. Without this options, Simian gives us only the fingerprints that represent the source coordinates of the files, which are not significant for our purpose. Thus, to have a better representation, we put the results in a wrapper class that represent the Pattern object in which we have all attributes to describe the recommendation in the right manner.

Other options, such as strings, identifiers or modifiers that should be introduced in the comparison, can be enabled with respect to the level of cloning that we want to reach. To find useful results, it is necessary to set at least ignoreIdentifiers, ignoreIdentifierCase, ignoreLiterals, ignoreVariableName, ignoreNumbers and ignoreModifiers since Simian goes beyond the developer personal implementations and looks only for the structure of the code, in order to use the concept of pattern in a more effective way. Based on these options, Simian applies the proper transformations on the original textual code in order to perform the comparison.

Furthermore, Simian compares the pair developer's snippet - pattern because some CLAMS patterns include some duplicated lines of code and this can bring some bias. Once we load the files, the check is performed and the results that include lines of code, name of pattern file and time to perform the comparison and put all in the wrapper class mentioned before.

At the end of this step, we have the patterns (a complete one or only partial) that the developer can start to implement and we can discard it from the comparison, as the developer is not interested to see what he has done so far. The last step is to remove the duplicated lines of code from the suggested patterns and show to the developer only the new part, that integrates his code or proposes new patterns for that feature, related of course to the library that he is implementing. About the ranking, we order the pattern by considering the number of cloned lines, so the first pattern contains more duplicated lines than second one and so on. The ranking is simply performed on the SimianPattern object that we produce as output. All the steps are summarized in 4.5.

The example below is related to the *MQTT paho* library, we have extracted the method *publish()* from the developer's file with Java Parser and run Simian on it. The last columns show two top rank CLAMS patterns.

Developer's original file

```
package org.eclipse.paho.sample.mqttv3app;
import java.io.IOException;
import java.sql.Timestamp;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MqttDefaultFilePersistence;

public class Sample implements MqttCallback {
public Sample(String brokerUrl, String clientId, boolean cleanSession,
    boolean quietMode, String userName, String password) throws
    MqttException {
try {
conOpt = new MqttConnectOptions();
conOpt.setCleanSession(clean);
if(password != null ) {
conOpt.setPassword(this.password.toCharArray());
}
if(userName != null) {
conOpt.setUserName(this.userName);
}
client = new MqttClient(this.brokerUrl,clientId, dataStore);
```

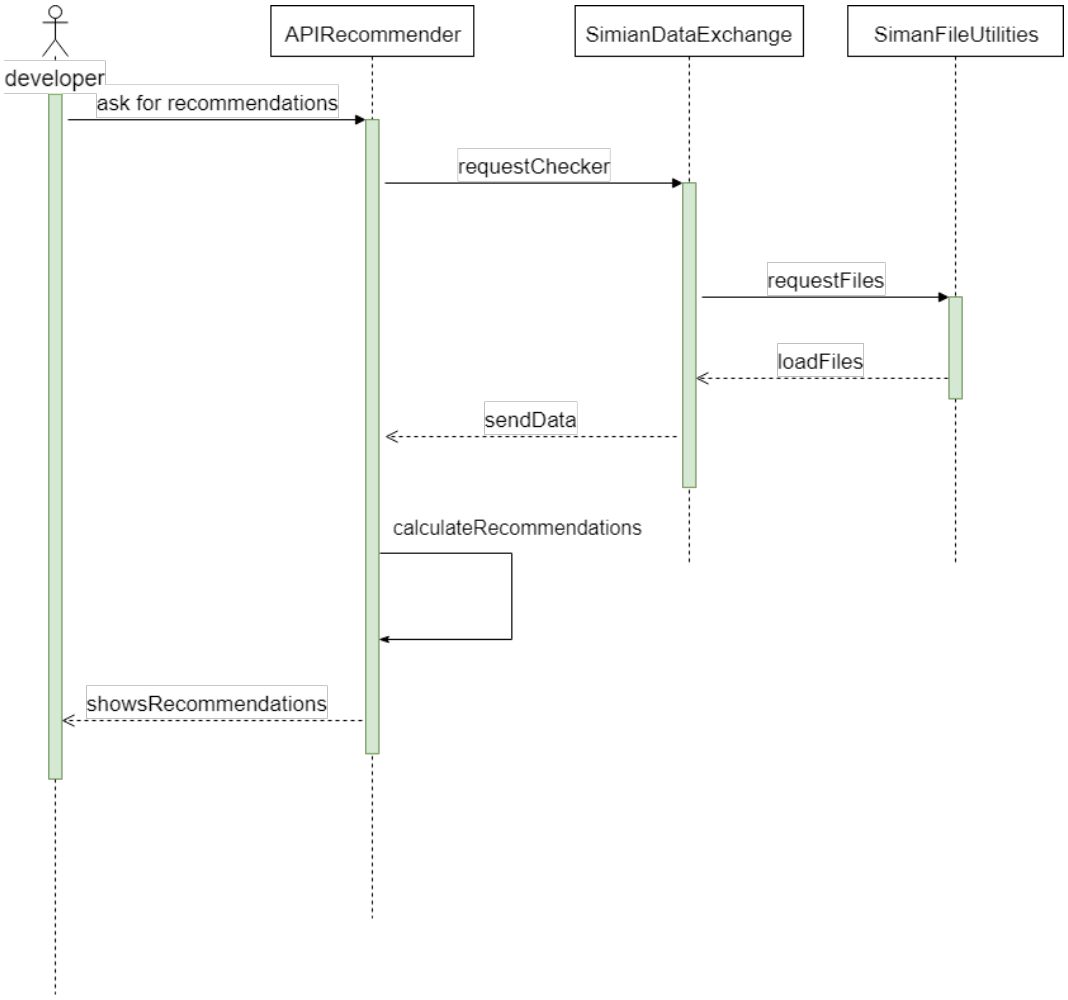

FIGURE 4.5: Sequence diagram

```
client.setCallback(this);

} catch (MqttException e) {
e.printStackTrace();
log("Unable to set up client: "+e.toString());
System.exit(1);
}
}

public void publish(String topicName, int qos, byte[] payload) throws
    MqttException {

// Connect to the MQTT server
log("Connecting to "+brokerUrl + " with client ID "+client.getClientId());
client.connect(conOpt);
log("Connected");
String time = new Timestamp(System.currentTimeMillis()).toString();
log("Publishing at: "+time+ " to topic \""+topicName+"\" qos "+qos);
MqttMessage message = new MqttMessage(payload);
message.setQos(qos);
client.publish(topicName, message);
// Disconnect the client
client.disconnect();
```

```
log("Disconnected");
}


public void subscribe(String topicName, int qos) throws MqttException {
client.connect(conOpt);
log("Connected to "+brokerUrl+" with client ID "+client.getClientId());
log("Subscribing to topic \""+topicName+"\" qos "+qos);
client.subscribe(topicName, qos);
// Continue waiting for messages until the Enter is pressed
log("Press <Enter> to exit");
try {
System.in.read();
} catch (IOException e) {
//If we can't read we'll just exit
}
// Disconnect the client from the server
client.disconnect();
log("Disconnected");
}
}
```

Extracted method:

```
// Connect to the MQTT server
log("Connecting to " + brokerUrl + " with client ID " +
    client.getClientId());
client.connect(conOpt);
log("Connected");
String time = new Timestamp(System.currentTimeMillis()).toString();
log("Publishing at: " + time + " to topic \"" + topicName + "\" qos " +
    qos);
// Create and configure a message
MqttMessage message = new MqttMessage(payload);
message.setQos(qos);
// Send the message to the server, control is not returned until
// it has been delivered to the server meeting the specified
// quality of service.
client.publish(topicName, message);
// Disconnect the client
client.disconnect();
log("Disconnected");
```

```
CLAMS pattern #1
{
String pubTopic;
MqttClient pubClinet;
String payload;
int qos;
pubClinet = new
    MqttClient(url,
    clientId);
pubClinet.setCallback(this);
pubClinet.connect();
MqttMessage message = new
    MqttMessage(payload.getBytes());
pubClinet.publish(pubTopic,
    message);
pubClinet.disconnect();
}
```

```
CLAMS pattern #2
{
String topicName;
MqttAsyncClient client;
MqttConnectOptions conOpt;
String brokerUrl;
byte[] payload;
int qos;
log("a string"+brokerUrl
+ "a string"+client.getClientId());
IMqttToken conToken =
    client.connect(conOpt,null,null);
log("a string"
    +System.currentTimeMillis()+
    "a string"+topicName+"a
    string"+qos);
IMqttDeliveryToken pubToken =
    client.publish(topicName,
    message, null, null);
pubToken.waitForCompletion();
IMqttToken discToken =
    client.disconnect(null, null);
discToken.waitForCompletion();
}
```

In particular, the two extracted recommendations show two possible uses of the object MqttMessage that the developer declared in the publish method. The first recommendation adds an MqttClient while the second uses the interface IMqttDeliveryToken as a different way to send the *mqtt* message. The number of recommendation is strongly related to the length of the method and how many lines Simian can detect in its textual analysis. This example is used just to show the expected output after running our approach. In general, given a project that uses different libraries, with this approach we are able to recommend patterns of whatever libraries that the user is interested in. As mentioned before, recommendations are at level of code snippet and provide a concrete and immediate hint. As we choose CLAMS for the second element of the clone pair, we are not able to provide tutorial or complete

application as recommendation, because only patterns are available for this purpose.

We can support almost the dataset of CLAMS and different libraries like MQTT or Json. The quality of recommendations depends first of all on the blocks in the developer's file that Simian is able to detect in the patterns provided by CLAMS. Providing more patterns means more support for the library and this brings more API recommendations at the end of the process. Another aspect to be taken into account is that Simian analyses the duplicated blocks of code and maybe some methods invocations that are not in the correct sequence are discarded automatically from the final recommendation.

In the next chapter, we are going to present an evaluation framework by using part of the dataset of CLAMS (as we have already the patterns used by Simian for the comparison). We do this as a double check validation because Simian doesn't have a post-processing phase and we need a comparison that goes beyond the lexical one in order to apply the metrics. Moreover, we will be comparing the results with those generated by PAM.

Chapter 5

Evaluation

At this point, we proceed with the evaluation of the results shown in the previous Chapter 4. The concept of recommendation changes from a context to context and it depends also on the developer's skills. We look for an evaluation framework that evaluates the results without bias related to this aspect. To do this, we choose Rascal tool to parse the AST of the developer's file and the API recommendations obtained combining Simian and CLAMS results. From Rascal, we obtain a list of method declarations and the related invocations for each file, and compare them in order to calculate four metrics: precision, recall, success rate and F-measure. These metrics are very useful because they allow us to analyze the results at a more abstract level, going beyond the code cloning activity. Simian doesn't perform this kind of comparison because it is a textual code cloner and looks only for duplicated lines of code. The main structure of the validation framework is depicted in Figure 5.1. Moreover, we compare the proposed approach with PAM, the probabilistic tool already analyzed in term of results, computational time and also by applying the same metrics on the method invocations. Next sections explains in details each component of this system, especially how to run Rascal and how the metrics have been calculated and how covert the PAM format in order to perform the comparison in the right manner.

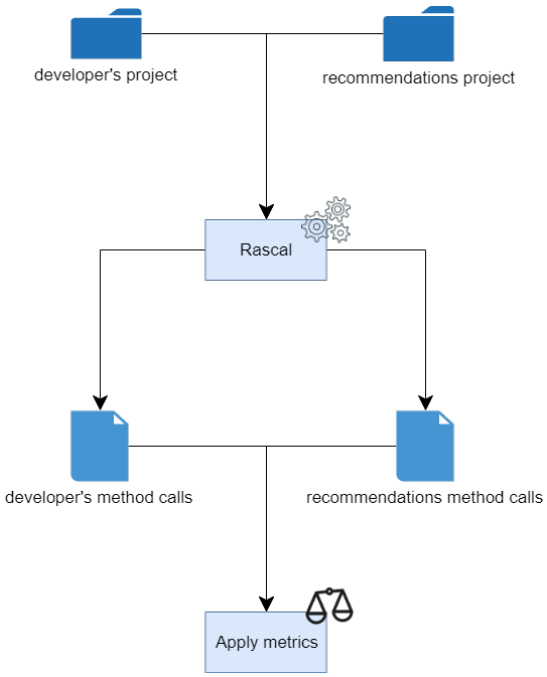

FIGURE 5.1: Validation framework

5.1 Research questions

The research questions we are going to address are as follows:
RQ$_1$: *Is the approach able to provide consistent recommendations?* This research question measures the quality of the recommendations provided by comparing the context, represented by the ground-truth part, and the retrieved patterns by the proposed approach. We analyze the possible situations in which there are some false positives by computing the metrics mentioned before and report the results in order to make an evaluation at the level of method invocations.
RQ$_2$: *Are the final results comparable to those recommended by PAM?* In order to validate the proposed approach, we perform a comparison with PAM. PAM produces a list of method invocations, so to do in a better way the comparison we use Rascal to transform the recommendations provided by our approach into method invocations.
RQ$_3$: *What are the timing performances of the proposed approach?* With this research question, we measure the time computation for a single recommendation. We take into account also the time needed to write the output file with the recommended code snippet and compare it with PAM.

5.2 Study methodologies

To validate and apply the metrics, we simulate the typical scenario in which the developer is working on a project. To do this, we select randomly a fragment of code coming from the client file using Java Parser as described before. We obtain the context, that is only a small part of the original file and perform our approach on it. To apply metrics, however, we need a transformation from the snippet of code to method invocations, that is a more comparable format. To do this, we use Rascal [29], a language for meta programming and it is able to create programs that read, analyse, transform, generate and/or visualize other programs. The range of programs to which meta-programming can be applied is wide: from programs in standard languages like C and Java to domain-specific languages for describing high-level system models or applications in specialized area. In some cases, even test results or performance data are used as input for meta-programs.

We look also to meta programs that can be analyzed by Rascal like reverse engineer and statically analyse of a big software system before visualizing the results. The main aim of Rascal is to provide a reusable set of primitives to build and manipulate program representations. The point is not to provide a unified representation of programs to let generic algorithms operate on. We can consider Rascal as an engineering tool for programmers that need to construct meta programs because it allows one to run, inspect, debug, etc. just as normal programs do. The main advantages are as follows:

- The syntax is very easy to learn and is used even for model and represent sophisticated concepts;
- Complex built-in data types provide standard solutions for many meta-programming problems;
- Safety is achieved by finding most of the errors before the program is executed, so the debug phase can be reduced;
- Local type inference makes local variable declarations redundant;

- Pattern matching can be used to analyze all complex data structures;
- Syntax definitions make it possible to define new and existing languages for specific purposes;
- Traversing the data structures is doing in an effective way and it is possible to extract information from them or to synthesize results;
- Templates enable easy code generation;
- The integration in Eclipse simplifies the usage and the iteration with all Rascal features.

Furthermore, Rascal implements the so called EASY (Extract-Analyze-SYnthesize) paradigm, used in the meta programming domain. Any meta-programming problems follow a fixed pattern. Starting with some input systems (a black box that we usually call system-of-interest), first relevant information is extracted from it and stored in an internal representation. This internal representation is then analyzed and used to synthesize results. If the synthesis indicates this, these steps can be repeated over and over again. Figure 5.2 represents the EASY paradigm that is quite common in the meta programming domain.

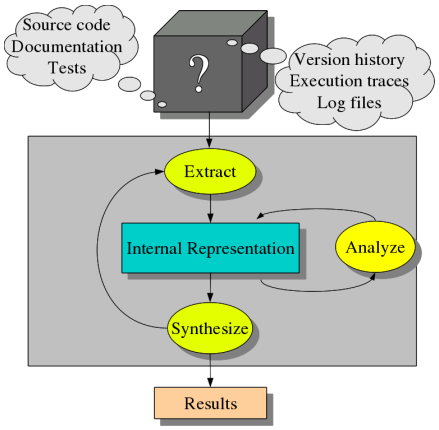

FIGURE 5.2: The EASY paradigm used by Rascal

For our purposes, we use Rascal to parse the AST of an entire project and retrieves all necessary information from it, such as class names, packages, methods and variables. To do this, however, we need a runnable project and the classpath file in which all dependencies are specified. In our scenario, the only required dependency is the jar file for the library. As the evaluation takes places among method invocations, we are forced to create this structure for the developer's file and the patterns that represents our recommendations. The structure is represented in Figure 5.3, in which we have:

- src folder: it contains Java file with the source code;
- lib folder: it contains the jar files that represent the libraries used in the project;
- classpath file: it contains the dependencies for the project.

Where the files in the **src** folder represent the developer's file in one project and the recommendations in the second one. These two projects will be the input of Rascal. The structure is built in the evaluation sub-package as mentioned before. In

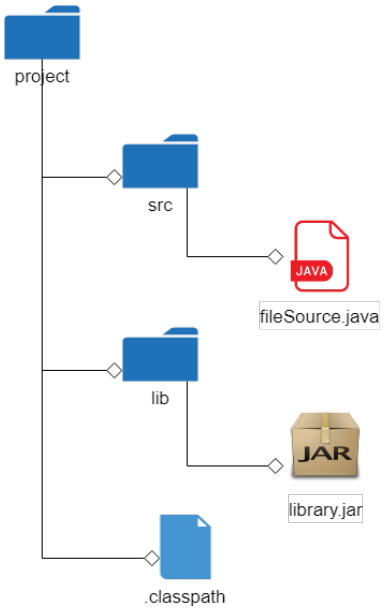

FIGURE 5.3: Folder structure for Rascal

particular, we use functions that take as input the library name and give the correct classpath file to have the structure mentioned before. Moreover, we have a function that builds the class in the correct way putting together the CLAMS patterns in a single class (for each of them, the function creates a method) where at the beginning the same import of the ground truth is inserted. For the patterns, we take as import the same contained in the namespaces file included in the original CLAMS dataset, in order to have all possible method invocations.

Also for the ground-truth data we create a similar structure, by putting the considered method in a Java class in the Rascal structure. We do this task for the ten client projects for each considered library twice, since we want to analyze also the pattern in form of code snippets extracted by the proposed approach. For the ground-truth code fragments, we put all imports used in the original client file; for the mined patterns, we encode in the built files all the imports that belong to the CLAMS namespace file. This task is necessary because we don't know at the beginning what the possible recommendations are. This approach doesn't bring any bias because Rascal retrieves only the method invocations with the corresponding import. An example file created for Rascal is given below:

```
import twitter4j.Query;
import twitter4j.Tweet;
import twitter4j.Twitter;
import twitter4j.TwitterFactory;
public class Twitter4jSample{
public void ground() {
    List<MyTweet> tweets = new ArrayList<MyTweet>();
    try {
        // get some tweets about java
        Twitter twitter4j = new TwitterFactory().getInstance();
        for (int i = 0; i < 3; i++) {
            Query q = new Query("java");
            q.setRpp(100);
            for (Tweet tw : twitter4j.search(q).getTweets()) {
                MyTweet myTw = new MyTweet(tw.getId(), tw.getFromUser());
```

```
                myTw.setText(tw.getText());
                myTw.setCreatedAt(tw.getCreatedAt());
                myTw.setFromUserId(tw.getFromUserId());
                tweets.add(myTw);
            }
            Thread.sleep(1000);
        }
    } catch (Exception ex) {
        logger.error("Error while grabbing tweets from twitter!", ex);
    }
    return tweets;
}
}
```

In particular, we added the code fragment from the developer's client file and put it a method called ground, plus the required imports for the snippet. Once we setup this structure, Rascal is able to parse the AST and retrieve the list of method invocations used to validate the approach. Rascal is launched using Eclipse RCP-RAP as platform and for this reason, we must use it as standalone to produce the list of method invocations starting from the output file of Simian and CLAMS. The main component of Rascal is a module in which we define the function used for our purpose. In particular, the proper function is called on the project's folder and gives as output a file that contains the method invocations. It is possible by creating the AST and takeing from it method declaration and invocations. An example of a set of method invocations is given below:

Method invocations of the ground truth

```
MqttSample/ground()#org/eclipse/paho/client/mqttv3/MqttMessage/MqttMessage(byte[])
MqttSample/ground()#org/eclipse/paho/client/mqttv3/MqttMessage/setQos(int)
MqttSample/ground()#java/lang/System/currentTimeMillis()
```

Method invocation of the mined patterns

```
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttMessage/MqttMessage(byte[])
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttClient/MqttClient
(java.lang.String,java.lang.String)
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttClient/disconnect()
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttMessage/setQos(int)
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttClient/publish
(java.lang.String,org.eclipse.paho.client.mqttv3.MqttMessage)
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttClient/connect()
MqttSample/pattern56()#java/lang/String/getBytes()
MqttSample/pattern56()#org/eclipse/paho/client/mqttv3/MqttClient/setCallback
(org.eclipse.paho.client.mqttv3.MqttCallback)
MqttSample/pattern59()#org/eclipse/paho/client/mqttv3/MqttMessage/MqttMessage(byte[])
MqttSample/pattern59()#org/eclipse/paho/client/mqttv3/MqttMessage/setQos(int)
MqttSample/pattern59()#org/eclipse/paho/client/mqttv3/IMqttToken/waitForCompletion()
MqttSample/pattern59()#java/lang/System/currentTimeMillis()
MqttSample/pattern27()#org/eclipse/paho/client/mqttv3/MqttMessage/MqttMessage(byte[])
MqttSample/pattern27()#org/eclipse/paho/client/mqttv3/MqttMessage/setQos(int)
MqttSample/pattern27()#org/eclipse/paho/client/mqttv3/MqttMessage/setRetained(boolean)
```

These files are not so suitable for an immediate analysis. However, we can apply metrics used in the statistics and information retrieval domains in order to evaluate

the Simian approach. We also select the code fragments that compose our ground truth by trying to cover the most important objects and methods for each library. To do this, we compose different scenarios, one in which Simian analyzes the first lines of a certain method, another in which we pick the last method and so on. The aim is to cover all the possible features that a certain library offers. For example, for the Twitter4j library, we select code snippets have both methods for user authentication procedure and methods and procedure to publish a tweet.

5.3 Dataset

First of all, we show the list of libraries that the proposed approach provides. They are the same at those by CLAMS, as they are already tested and most popular in the Java project context. We select 5 libraries among the CLAMS original dataset and they are showed in Table 5.1. In this table, we depict also the number of recommended patterns provided by CLAMS. For each library, we select 10 files in order to keep the same context and to perform average values for the metrics. From this, we build our dataset that contains 50 files. Each file represents the developer's file that we have seen in the proposed approach figure and we extract from them the ground truth part. For each of them, we run the proposed approach and evaluate the results through the metrics.

Library	**No. of patterns**
twitter4j	107
drools	309
camel	152
wicket	717
restlet-framework	182

TABLE 5.1: Libraries supported by Simian and CLAMS

A brief description of the golden set is as follows:

- twitter4j: it is used for integrate the Twitter services in Java environment;
- drools: it is a Business Rules Management System (BRMS) that helps to define the internal rule for the language;
- camel: it defines mediation rules and routing for specific domain languages;
- wicket: it is a component based web framework;
- restlet framework: it helps to build web APIs following the REST architecture.

5.4 Metrics

Once we have the two lists of method invocations, we can define metrics to do the evaluation, we consider *precision*, *recall*, *F-measure* and *success rate* and they are explained below:
$Precision = \frac{corr}{all_{rec}}$

$Recall = \frac{corr}{all_{gt}}$

$$F\ Measure = \frac{2*precision*recall}{recall+precision}$$

$$\text{Success rate} = \begin{cases} 1 & \text{if at least one method of the ground truth} \\ & \text{belongs to the recommended pattern invocations} \\ 0 & \text{otherwise} \end{cases} \quad (5.1)$$

Where *corr* is the number of correct API method invocations by the approach related to the context, all_{rec} is the number of method invocations in the recommendations and all_{gt} is all method invocations of the ground truth part, extracted from the initial files. With the first metric, we want to measure the precision of the recommendations related to the context considering all the patterns and all the method invocations related to them. Recall points out the rate related to the ground truth part, that is more restricted than the original file and so. In this case, Simian gives as out put less method invocations but more focused on the context in which we are. F-measure considers both precision and recall in order to give an average value on the accuracy of the approach. The classical index considers the harmonic mean between precision and recall. Finally, success rate is a binary value that is equal to 1 if at least one method in the ground truth is found in the recommendation, 0 otherwise. In the table, we measure this metric for all ten clients and so we give a rate that represents an average success rate.

All the rates range from 0 to 1 and represent the accuracy of the approach. The number is affected also by the number of patterns extracted by CLAMS. The metrics are computed by using Eclipse, and in particular, once we have the files coming from the Rascal computation, described in the previous section, the function applyMetrics calculates all the metrics.

5.5 Results

The metrics work on method declarations and invocations, so for each mined pattern by the approach we can have a taste of the results. In particular, we go further with respect to the code cloning analysis, that looks for only the lexical differences among the lines of code. Reversely, we are able to add some post-processing phases on the results and we can do more accurate comparison with this evaluation framework.

We also make a comparison between the proposed approach and an existing API recommendations, PAM. The metrics are useful to evaluate the accuracy of Simian applied to CLAMS but we decide to improve the validation framework adding an existing and validated approach. PAM uses probabilistic techniques to retrieve the most probable method invocation starting from file in an ARFF format which contains method invocations and declarations. We can compare the result of PAM with that of the proposed approach considering the top rank method invocations retrieved by CLAMS. As the format of our recommendation differs from that of PAM, we have to use Rascal to extract method invocations that represent the API recommendations of the ground truth data. What different is the ranking method to order the list of invocations.

For CLAMS, we keep trace of the context with a ARFF file that contains the method caller and calls for the library and the client file that represents the real implementation in Java. On the other hand, PAM takes as input only ARFF file without any client files for the context. It is necessary to represent the context in we are in order to reduce bias in the analysis. We can do this by converting the clients files (that are Java source code) in the ARFF format and combine the original files with this

new one. We can reuse Rascal in the same for the metrics evaluation, by extracting method invocations from client files. After this phase, we transform the invocations into the ARFF format by using a Python script. In this way, PAM is able to keep trace of the context and produce as final output the list of API function calls. After we run PAM, it produces the results in the format, in form of method invocations and the corresponding probability. However, as the ground-truth data is usually only a fragment of the original developer's file, the invocations have small probabilistic impact because the original ARFF file for each libraries has more than 2,000 lines. So, in order to apply the evaluation metrics, we have to split the PAM result files and select the section where we can find proper invocations. Figure 5.4 explains the process applied to obtain the same format used in the comparison.

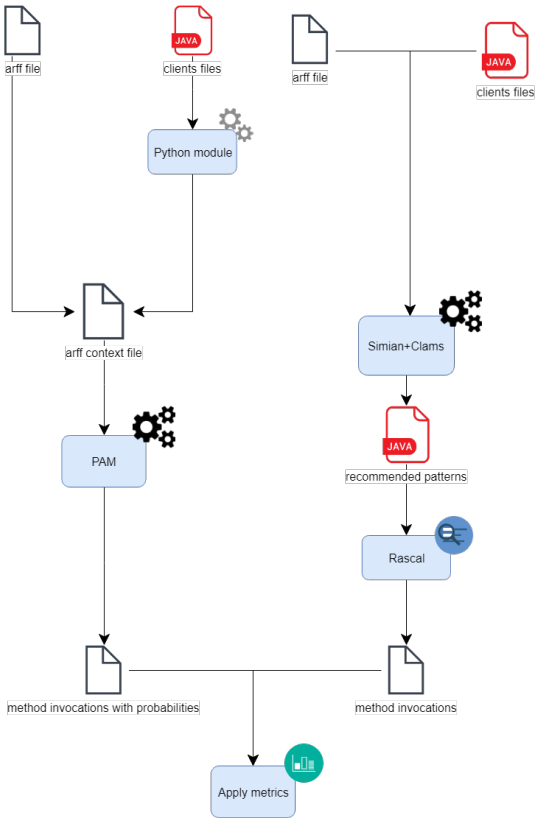

FIGURE 5.4: Process for PAM comparison

At the end of the process, we have the list of invocations ranked by probabilities from PAM and the list of invocations extracted by Rascal. Since they have the same format, we can apply the metrics in the same way described before. Tables 5.2 and 5.2 depict a comparison between our proposed approach and PAM:

Library	**precision**	**recall**	**success rate**	**F-measure**
twitter4j	0.506235119	0.74285538	0.9	0.545789625
drools	0.22455129	0.344444445	0.7	0.215914793
camel	0.430785693	0.598096273	1	0.448844577
wicket	0.104704642	0.23564139	0.6	0.152584271
restlet-framework	0.292169883	0.539874547	0.7	0.218620598

TABLE 5.2: Average values for the proposed approach

Library	precision	recall	success rate	F-measure
twitter4j	0.471779376	0.601220656	0.9	0.443219823
drools	0.224551356	0.467142966	0.7	0.261774121
camel	0.243655335	0.562697394	0.7	0.238838121
wicket	0.080743729	0.358903284	0.6	0.119715519
restlet-framework	0.246895116	0.377042229	0.7	0.235095648

TABLE 5.3: Average values for PAM

The results are used to answer research questions $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. The best case is obtained by the *Twitter4j* library, where precision and recall are larger than 50%, while the worst case is seen by the library *wicket*. This happens since the code snippet that represents the ground truth doesn't contain any method invocations.

As can be seen, PAM has a worse precision compared to the results obtained by applying our proposed approach. This attributes to the fact that PAM discards some invocations that belong to the ground truth as they are not relevant from the probabilistic point of view. Recall is affected by the splitting procedure as already described before. The number of the ground truth invocations are different from the original ones because PAM puts at the end of the files the method invocations related to the considered library and so we can consider only this part of the file. As a result, the F-measure is lower than the original values while success rate remains almost unchanged. Figures 5.5 and 5.6 depict the precision and recall scores for the proposed approach and PAM.

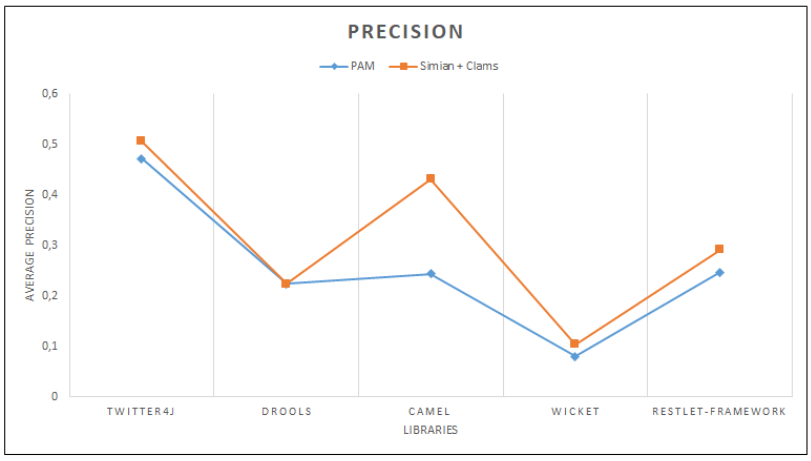

FIGURE 5.5: Precision comparison

Moreover, we measure the time needed for the computation by both approaches. For PAM, we get the time needed to produce the list of method invocations starting from a single ARFF file that summarizes the context, while for our approach, we considered the time needed to produce the recommendations in form of code snippet, considering as context the developer's ground truth (in this case the ARFF is used by CLAMS to produce the patterns chosen as baseline). The results shows that the times are almost equal, except in the case of the *wicket* library. This is due to the fact that PAM relies only on ARFF and in this case the file contains a huge amount of data.

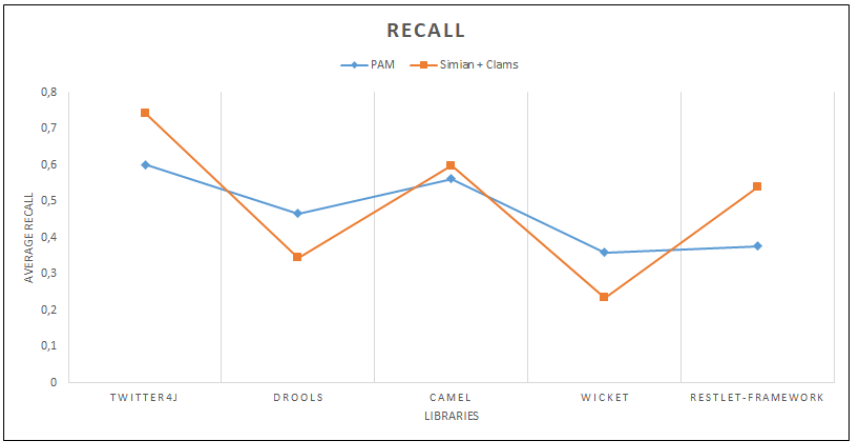

FIGURE 5.6: Recall comparison

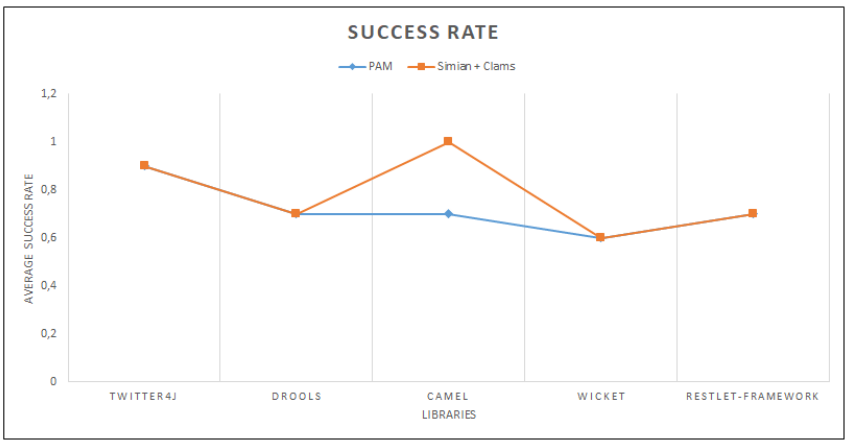

FIGURE 5.7: Success rate comparison

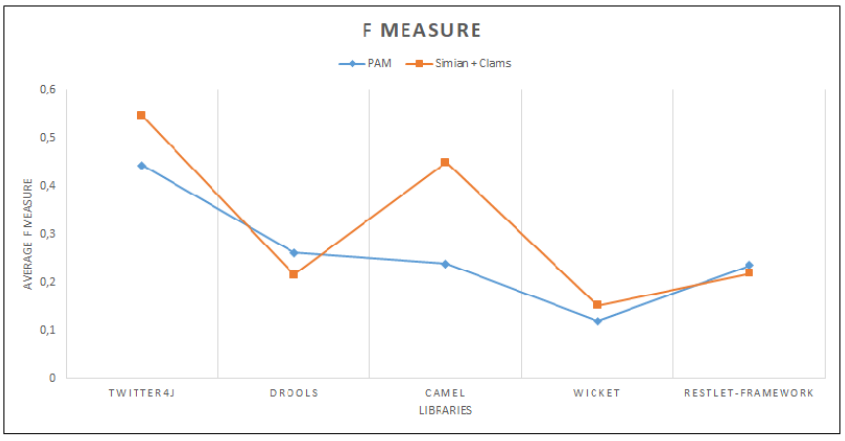

FIGURE 5.8: F measure comparison

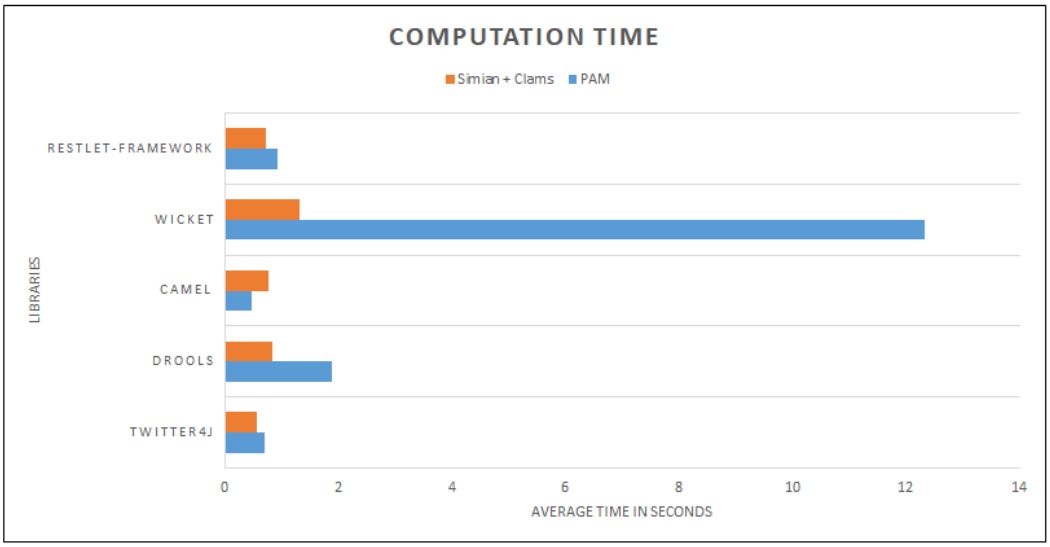

FIGURE 5.9: Time comparison

Chapter 6

Conclusion

Aiming to assist developers in their development activities by mining open source software repositories, we propose a framework for providing API function calls. This topic has attracted attention from the research community. Our work is built based on some existing tools and it can be integrated into Eclipse. On the one side, we have Simian, a code cloner that is able to retrieve cloned code among files without needing the pre-processing or post-processing phases. On the other hand, we have CLAMS, a tool that gives as output a ranked list of patterns. The final goal is to provide API function call recommendations in the context of software complex system. This means that we have to analyze Java user's file put within a more bigger structure (as Eclipse project for example). So, a code cloner tool is not enough to generate proper recommendations at the level of code snippet and we integrate the concept of patterns thanks to CLAMS approach. From this, we reuse output files that contain useful patterns for our purposes. In this way, we provide recommendations at level of code snippet, which is closer to code and helps developers complete.

From the evaluation framework, it can be seen that the approach is strongly related to the developer's code snippet that Simian takes as input. From the comparison with PAM, we notice that the proposed approach can produce accurate recommendations. Only in the case of some values of recall and f-measure are inferior to those of than PAM. Another issue is that CLAMS patterns have a structure composed first by a list of variable used by the method calls and then the chain of method declaration to realize a specific feature. This structure may introduce some bias because Simian can find similarities only among the list of variables and not in the method calls. Moreover, Simian divides source code to analyze into blocks, limited by the threshold parameter. In some cases this can lead to false positives since Simian cannot identify a certain pattern in the code if it is not smaller than the threshold value. For example, in source code there are two methods that belong to a CLAMS pattern which includes another method invocations, if in the original source code the third invocation differs from that, Simian is not able to retrieve the pattern. Some threats could arise from the evaluation framework in which we used Rascal. In order to validate the approach, we had set up an Eclipse structure as shown in the related section. Although the structure is correct, it is manually built so it may affect the metrics, in particular precision and recall.

We opt for a code cloner, however there are a lot of alternative approaches that can be exploited, like another code cloner or change completely the approach. We can beautify the output results that are presented in a simple file by setting up an Eclipse plugin, with all necessary components. Regarding the evaluation, we can provide a human survey by involving developers with different skills and knowledge to asses the provided results with a different perspective. Considering other approaches, many studies use probabilistic techniques, for example, define models or probability distribution. The central point of our work is the use of a code

cloner to detect similarities between the pattern retrieved by CLAMS and the actual code snippet. Although Simian doesn't have any pre- or post-processing phases, we overcome this limitations by manually selecting source files and ranking the results, considering the duplicated lines of code. Moreover, we added also the AST concept in the evaluation framework. In this way, we are able to equip Simian with a new functionality that considers only the textual similarity.

Bibliography

[1] Hudson S. Borges and Marco Tulio Valente. "Mining usage patterns for the Android API". In: *PeerJ Computer Science* 1 (2015), e12. DOI: 10.7717/peerj-cs.12. URL: https://doi.org/10.7717/peerj-cs.12.

[2] Raymond P. L. Buse and Westley Weimer. "Synthesizing API usage examples". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 2012, pp. 782–792. DOI: 10.1109/ICSE.2012.6227140. URL: https://doi.org/10.1109/ICSE.2012.6227140.

[3] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". In: (2009), pp. 470–495. URL: www.elsevier.com/locate/scico.

[4] Christopher Scaffidi. "Why are APIs Difficult to Learn and Use?" In: (2006).

[5] Barthélémy Dagenais et al. "Moving into a New Software Project Landscape". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 275–284. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806842. URL: http://doi.acm.org/10.1145/1806799.1806842.

[6] Ekwa Duala-Ekoko and Martin P. Robillard. "Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 266–276. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337255.

[7] Jaroslav Fowkes and Charles Sutton. "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. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950319.

[8] Jaroslav M. Fowkes and Charles A. Sutton. "Parameter-free probabilistic API mining across GitHub". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 254–265. DOI: 10.1145/2950290.2950319. URL: http://doi.acm.org/10.1145/2950290.2950319.

[9] https://www.crossminer.org/. "Last accessed on 30/09/2018". In: ().

[10] https://www.cs.waikato.ac.nz. "Last accessed 30/09/2018". In: ().

[11] https://www.harukizaemon.com/simian/. "Last accessed on 25/09/2018". In: ().

[12] Patricia Jablonski and Daqing Hou. "CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE". In: *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*. eclipse '07. New York, NY, USA: ACM, 2007, pp. 16–20. ISBN: 978-1-60558-015-9. DOI: 10.1145/1328279.1328283. URL: http://doi.acm.org/10.1145/1328279.1328283.

[13] Nikolaos Katirtzis, Themistoklis Diamantopoulos, and Charles A. Sutton. "Summarizing Software API Usage Examples Using Clustering Techniques". In: *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 2018, pp. 189–206. DOI: 10.1007/978-3-319-89363-1_11. URL: https://doi.org/10.1007/978-3-319-89363-1_11.

[14] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. "An Empirical Study of API Usability". In: (2013).

[15] Martin P. Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: (2009).

[16] João Eduardo Montandon et al. "Documenting APIs with examples: Lessons learned with the APIMiner platform". In: *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. 2013, pp. 401–408. DOI: 10.1109/WCRE.2013.6671315. URL: https://doi.org/10.1109/WCRE.2013.6671315.

[17] Seyed Mehdi Nasehi et al. "What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow". In: *28th IEEE International Conference on Software Maintenance*. IEEE. Piscataway, 2012, pp. 25–34. DOI: 10.1109/ICSM.2012.6405249.

[18] Anh Tuan Nguyen et al. "API code recommendation using statistical learning from fine-grained changes". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 511–522. DOI: 10.1145/2950290.2950333. URL: http://doi.acm.org/10.1145/2950290.2950333.

[19] Nils Gode. "Incremental Clone Detection". PhD thesis. University of Bremen, 2008.

[20] Haoran Niu, Iman Keivanloo, and Ying Zou. "API Usage Pattern Recommendation for Software Development". In: *Journal of Systems and Software* 129.C (2017), pp. 127–139. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.07.026.

[21] Haoran Niu, Iman Keivanloo, and Ying Zou. "API usage pattern recommendation for software development". In: *Journal of Systems and Software* 129 (2017), pp. 127–139. DOI: 10.1016/j.jss.2016.07.026. URL: https://doi.org/10.1016/j.jss.2016.07.026.

[22] Francesco Ricci, Lior Rokach, and Bracha Shapira. "Introduction to Recommender Systems Handbook". In: *Recommender Systems Handbook*. Ed. by Francesco Ricci et al. Boston, MA: Springer US, 2011, pp. 1–35. ISBN: 978-0-387-85820-3. DOI: 10.1007/978-0-387-85820-3_1.

[23] Martin P Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: *IEEE software* 26.6 (2009), pp. 27–34.

[24] Martin P. Robillard et al. "Automated API Property Inference Techniques". In: *IEEE Transactions on Software Engineering* 39.5 (2013), pp. 613–637. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.63.

[25] Martin P. Robillard et al., eds. *Recommendation Systems in Software Engineering*. en. DOI: 10.1007/978-3-642-45135-5. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-45134-8 978-3-642-45135-5. URL: http://link.springer.com/10.1007/978-3-642-45135-5 (visited on 03/10/2017).

[26] Stephane Ducasse, Oscar Nierstrasz, and Matthias Rieger. "On the effectiveness of clone detection by string matching". In: (2005), pp. 37–58.

[27] Ferdian Thung et al. "Automatic recommendation of API methods from feature requests". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 2013, pp. 290–300. DOI: 10.1109/ASE.2013.6693088. URL: https://doi.org/10.1109/ASE.2013.6693088.

[28] Gias Uddin and Martin P Robillard. "How API Documentation Fails". In: *IEEE Software* 32.4 (2015), pp. 68–75. ISSN: 0740-7459. DOI: 10.1109/MS.2014.80.

[29] utor.rascal-mpl.org/. "Last accessed on 24/09/2018". In: ().

[30] J. Wang and J. Han. "BIDE: efficient mining of frequent closed sequences". In: *Proceedings. 20th International Conference on Data Engineering*. 2004, pp. 79–90. DOI: 10.1109/ICDE.2004.1319986.

[31] J. Wang et al. "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. DOI: 10.1109/MSR.2013.6624045.

[32] Jue Wang et al. "Mining succinct and high-coverage API usage patterns from source code". In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. 2013, pp. 319–328. DOI: 10.1109/MSR.2013.6624045. URL: https://doi.org/10.1109/MSR.2013.6624045.

[33] Yunwen Ye and Gerhard Fischer. "Reuse-Conducive Development Environments". In: *Autom. Softw. Eng.* 12.2 (2005), pp. 199–235. DOI: 10.1007/s10515-005-6206-x. URL: https://doi.org/10.1007/s10515-005-6206-x.

[34] Hao Zhong et al. "MAPO: Mining and Recommending API Usage Patterns". In: *23rd European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2009, pp. 318–343. ISBN: 978-3-642-03012-3. DOI: 10.1007/978-3-642-03013-0_15.

[35] Hao Zhong et al. "MAPO: Mining and Recommending API Usage Patterns". In: *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. 2009, pp. 318–343. DOI: 10.1007/978-3-642-03013-0_15. URL: https://doi.org/10.1007/978-3-642-03013-0_15.

[36] Zixiao Zhu et al. "Mining API Usage Examples from Test Code". In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. 2014, pp. 301–310. DOI: 10.1109/ICSME.2014.52. URL: https://doi.org/10.1109/ICSME.2014.52.