Université de Montréal

Inferring API Usage Patterns and Constraints: a Holistic Approach

par
Mohamed Aymen Saied

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Août, 2016

Résumé

Les systèmes logiciels dépendent de plus en plus des librairies et des frameworks logiciels. Les programmeurs réutilisent les fonctionnalités offertes par ces librairies à travers une interface de programmation (API). Par conséquent, ils doivent faire face à la complexité des APIs nécessaires pour accomplir leurs tâches, tout en surmontant l'absence de directive sur l'utilisation de ces API dans leur documentation. Dans cette thèse, nous proposons une approche holistique qui cible le problème de réutilisation des librairies, à trois niveaux. En premier lieu, nous nous sommes intéressés à la réutilisation d'une seule méthode d'une API. À ce niveau, nous proposons d'identifier les contraintes d'utilisation liées aux paramètres de la méthode, en analysant uniquement le code source de la librairie. Nous avons appliqué plusieurs analyses de programme pour détecter quatre types de contraintes d'utilisation considérées critiques. Dans un deuxième temps, nous changeons l'échelle pour nous focaliser sur l'inférence des patrons d'utilisation d'une API. Ces patrons sont utiles pour aider les développeurs à apprendre les façons courantes d'utiliser des méthodes complémentaires de l'API. Nous proposons d'abord une technique basée sur l'analyse des programmes clients de l'API. Cette technique permet l'inférence de patrons multi-niveaux. Ces derniers présentent des relations de co-utilisation entre les méthodes de l'API à travers des scénarios d'utilisation entremêlés. Ensuite, nous proposons une technique basée uniquement sur l'analyse du code de la librairie, pour surmonter la contrainte de l'existence des programmes clients de l'API. Cette technique infère les patrons par analyse des relations structurelles et sémantiques entre les méthodes. Finalement, nous proposons une technique coopérative pour l'inférence des patrons d'utilisation. Cette technique est axée sur la combinaison des heuristiques basées respectivement sur les clients et sur le code de la librairie. Cette combinaison permet de profiter à la fois de la précision des techniques basées sur les clients et de la généralisabilité des techniques basées sur les librairies. Pour la dernière contribution de notre thèse, nous visons un plus haut niveau de réutilisation des librairies. Nous présentons une nouvelle approche, pour identifier automatiquement les patrons d'utilisation de plusieurs librairies, couramment utilisées ensemble, et généralement développées par différentes tierces parties. Ces patrons permettent de découvrir les possibilités de réutilisation de plusieurs librairies pour réaliser diverses fonctionnalités du projets.

Mots clés : Compréhension de programme, utilisabilité des API, inférence de patrons et contraintes d'utilisation, documentation des API.

Abstract

Software systems increasingly depend on external library and frameworks. Software developers need to reuse functionalities provided by these libraries through their Application Programming Interfaces (APIs). Hence, software developers have to cope with the complexity of existing APIs needed to accomplish their work, and overcome the lack of usage directive in the API documentation. In this thesis, we propose a holistic approach that deals with the library usability problem at three levels of granularity. In the first step, we focus on the method level. We propose to identify usage constraints related to method parameters, by analyzing only the library source code. We applied program analysis strategies to detect four critical usage constraint types. At the second step, we change the scale to focus on API usage pattern mining in order to help developers to better learn common ways to use the API complementary methods. We first propose a client-based technique for mining multi-level API usage patterns to exhibit the co-usage relationships between API methods across interfering usage scenarios. Then, we proposed a library-based technique to overcome the strong constraint of client programs' selection. Our technique infers API usage patterns through the analysis of structural and semantic relationships between API methods. Finally, we proposed a cooperative usage pattern mining technique that combines client-based and library-based usage pattern mining. Our technique takes advantage at the same time from the precision of the client-based technique and from the generalizability of the library-based technique. As a last contribution of this thesis, we target a higher level of library usability. We present a novel approach, to automatically identify third-party library usage patterns, of libraries that are commonly used together. This aims to help developers to discover reuse opportunities, and pick complementary libraries that may be relevant for their projects.

Keywords: Program comprehension, API usability, usage pattern mining, usage constraint inference, API documentation.

Contents

List of Tables

List of Figures

I dedicate this thesis to my parents, AbdelKader & Hajer, my wife, Wafa, and my son, Ayoub.

Acknowledgments

First and formost, I would like to express my sincere appreciation and gratitude to my supervisor Professor Houari Sahraoui for his guidance during my research. His support and inspiring suggestions have been precious for the advancement of this thesis content. I appreciate all his contributions of time and ideas to make my Ph.D. experience productive. At difficult times, I draw my strength back from his energy and good mood, and his advices go beyond academic research. One simply could not wish for a better supervisor.

I would like to thank Professor Tien N. Nguyen for accepting to be my external examiner and for furnishing efforts to review my dissertation. I thank Professor Eugène Syriani for accepting to be on my PhD committee and for reviewing my dissertation. And I would also like to thank Professor Pierre Thibault for joining my PhD committee. Many thanks for Professor Sylvie Hamel for accepting to chair my doctoral committee.

Last but not least, I would like to salute the soul of my father. He would have been proud of me if he were among us today. I would like to express my special thanks to my mother. I would have never managed any of this without her support and sacrifices. My special thanks goes as well to my sister and my brother who never stopped believing in me. I would like to express my gratitude to my wife for her kindness and compassion she helped and encouragement me to outstrip all the diffuclties and finish my doctorate.

Chapter 1

Introduction

1.1 Research context

Software reuse is a common practice in the development and maintenance of a modern software system [26]. Indeed, modern industry builds software systems more and more by assembling features offered by libraries and application frameworks. This contributes in facilitating the development of complex systems with controlled costs while maintaining the delivery schedules, and quality [24].

Libraries expose the functionalities or services they provide through an interface API (Application Programming Interface). And software developers increasingly need to reuse functionality provided by external libraries through their APIs. Thus developers have to learn how to use existing APIs to benefit from them.

Documentation is an essential resource for software comprehension in general, but is a critical resource, in particular, for APIs usability and learning. API documentation usually specifies the way in which client software can interact with the library and reuse its functionalities independently of implementation details. Therefore, APIs benefits are dependent on documentation quality and completeness. Indeed, with an incomplete or missing documentation, the client application code may be inconsistent with the library implementation, and bugs may creep into the client programs [23]. Moreover, the benefits of using APIs do not come easily; different works on API usability showed that learning how to use APIs presents challenging barriers [37, 58, 59, 70, 72].

The number and the size of APIs are continuously growing, and developers have to cope with complexity of existing APIs that are needed to accomplish their work. Novice developers are faced with the significant difficulty of learning a large number of APIs. Even experienced developers must frequently learn newer parts of familiar APIs, or newly released APIs when working on new tasks. Moreover, with the emergence of the continuous deployment, frequent release of new versions, and time at their disposal, it is not possible for developers to learn all the APIs they need in depth. In this thesis, we are interested on how to help developers to easily learn common ways to correctly use the APIs.

1.2 Problem statement

Despite recent progress in API documentation and discovery, API usability is still a challenging problem. Client application developers have to deal with the library usability challenges at different levels. First, a developer has to cope with the library usability at the method level, when he is interested in a specific method of the API. Second, he has to deal with the usability at the global level of an API, when using complementary methods. Finally, developers often use more than one library and, then, they have to tackle the usability challenges when coordinating the use of distinct libraries that may be developed by different organizations. In the remainder of this section, we highlight the different problems and challenges addressed in this thesis that are mainly related to the identification of APIs usage patterns and constraints.

Library usability at the API method scope (first level)

Once an API method is selected, developers have to use it in a consistent way with its documentation and implementation. The main challenge is then to respect precondition on the method inputs, generally known as constraints on parameter values. Unfortunately, it often happens that the API usage constraints are not explicitly described in the documentation. This may generate additional cost for debugging and correction, when the method usage constraints are violated. An interesting solution to the problem of undocumented usage constraints is to automatically recover them from the code.

Existing approaches are either interested in the redocumentation of a program in general [28, 67–69] or interested specifically in the re-documentation of usage constraints of an API. For the second category, some existing works focused on the API side by analyzing the API documentation to infer API specifications [54]. Others specify API use constraints by manually adding annotations into the API code [33]. Such suggestions may be insufficient for large API and for the non-documented constraints.

To overcome the aforementioned problem, another piece of work, try to analyze client applications, rather than the library documentation, to identify constraints [57, 86]. Regardless of the effectiveness of such approaches, the needs of client applications that cover the entire target library drastically limit their applicability.

.

Library usability across complementary API methods (second level)

The API methods are generally used by client programs jointly with other methods of the API. However, it is not obvious to deduce the co-usage relationships between API methods from their documentations. The increasing size and complexity of APIs introduce an additional difficulty. Indeed large APIs are the most challenging to learn and use.

A large API may consist of several thousands public methods defined in hundreds classes. Since API classes are typically meant for a wide variety of usage contexts, the elaborated documentation of an API class may be very detailed as it tries to specify all aspects that a client might need to know about the class of interest. Hence, software developers might spend considerable time and effort to identify the subset of the class's methods that are necessary to implement their task. Therefore, identifying usage patterns for the API can help to better learn common ways to use the API complementary methods.

Existing techniques for mining API usage patterns are valuable to facilitate API learning and usage. However, existing techniques mainly identify flat usage patterns for specific usage scenario. Such techniques are proposed to recommend usage examples relevant to one task (e.g. [87]), and/or for auto-completion in a specific context (e.g. [50]). Accordingly, inferred usage pattern cannot reflect the different interfering API usage scenarios, which is definitely required to improve the API learning resource.

From another perspective and despite the different aspects they try to cover, existing techniques are all based on client programs' code. Which is a strong constraint since client programs' code is unfortunately not available for both newly released libraries and APIs which are not widely used. Moreover, it is not possible to collect client programs that cover all the potential usage scenarios of the API of interest. Indeed, client-based identification of API usage patterns can be used only for a subset of the API of interest that is the set of the API methods which are already used, multi-times, by different selected clients of the API.

Library usability across complementary software libraries (third level)

Nowadays, open source repositories provide a wide range of reuse opportunities of functionality provided by well-tested and mature third-party libraries. However, as software libraries are docu-

mented separately but intended to be used together, developers are unlikely to fully take advantage of these reuse opportunities.

Much research effort has been dedicated to the identification of API usage patterns [79, 81, 87]. The vast majority of existing works focus on usage patterns within a single library. Indeed, these approaches assume that the set of relevant libraries is already known to the developer. However, this assumption makes the task of finding relevant libraries and understanding their usage trends a tedious and time-consuming activity.

Software developers can spend a considerable amount of time and effort to manually identify libraries that are useful for the implementation of their software. Worse yet, developers may even be unaware of the existence of these libraries. Thus, they may tend to implement most of their features from scratch instead of reusing functionalities provided by third-party libraries as pointed out by several researchers [18, 79]. Therefore, we believe that identifying patterns of libraries that are commonly used together, can help developers to discover and choose libraries that may be relevant for their projects' implementation.

All of these observations are at the origin of the work conducted in this thesis. In the next section, we give an overview of our research directions to address the above-mentioned problems.

1.3 Research objectives and main contributions

The main objective of this thesis is to propose a holistic approach that deals with the usability problem at the different levels of granularity when using external libraries. To overcome the previously identified problems, we propose the following contributions, organized into three major parts, each corresponding to a specific level of library reuse.

Part 1: the method level

Our first contribution helps developers to comply with the constraints on API method parameters. We propose to identify constraints by analyzing only the library source code. We selected four critical usage constraint types related to method parameters. This is done by static and intra-procedural analysis on control flow graphs to detect the selected constraints.

We also conduct an observational study on a large set of libraries to evaluate the presence of selected constraints in the code and the degree of their documentation as an indication of the risk of constraint violation.

Part 2: the library level

Our second contribution is related to API usage pattern mining to help developers to better learn common ways to use the API complementary methods.

We first propose a client-based technique for mining multi-level API usage patterns to exhibit the co-usage relationships between API methods across interfering usage scenarios. Our technique is based on an adaptation of a clustering algorithm, and the analysis of the frequency and consistency of co-usage relations.

Then, we proposed a library based technique to overcome the strong constraint of client programs selection. Our technique infers API usage patterns through the analysis of structural and semantic relationships between API methods. This technique can even be applied to "new" APIs, for which client programs are not available yet.

Finally, we proposed a cooperative usage pattern mining technique that combines client-based and library-based usage pattern mining. Our technique takes advantage at the same time from the precision of client-based technique and from the generalizability of library-based techniques.

Part 3: the group of libraries level

Our third contribution adds a new dimension to the library usability problem. We present a novel approach, to automatically identify third-party library usage patterns, as a collection of libraries that are commonly used together by developers. This approach is meant to help developers to discover and use libraries that may be relevant for their projects. Thus we mine the 'wisdom of the crowd' to discover usage patterns of software libraries. We evaluate the efficiency of our approach on an extremely large dataset of popular libraries and client systems.

1.4 Dissertation organization

This thesis is organized as follows. Chapter 2 discusses previous research contributions that are relevant to the main themes of this dissertation: API documentation techniques, API usage patterns

mining techniques and API usage constraints inference techniques. The first part of the thesis core consists of Chapter 3 that reports on our contribution for the detection of API usage constraints. The second part of the thesis includes three chapters. Chapter 4 describes our first technique for API usage pattern mining. We present our clustering adaptation to mine co-usage relations from API client programs. In Chapter 5, we detail our second technique for API usage pattern mining using only the library code. We investigate different library-based heuristics to infer patterns of complementary API methods. In Chapter 6, we investigate deferent strategies to combine the client-based and the library-based mining of API usage patterns. In the third part of the thesis (Chapter 7), we introduce a new dimension to the library usability, in which we consider usage patterns of groups of software libraries. Finally, Chapter 8 summarizes the contributions of the work presented in this thesis, underlines its main limitations, and describes our future research directions.

Chapter 2

Related Work

In this chapter, we provide a literature review on research work related to this thesis. First, we give an overview of empirical studies on API usability to highlight the factors that hinder the API usage and learning process (Section 2.1). Then, we present the work that targets the automation of API documentation to reduce the difficulty of learning how to use APIs (Section 2.2). Finally, we discuss the related work relevant to the main themes of this research work. This includes several approaches for extracting API usage information, and the related work according to the inferred API property (Section 2.3). We classify the inferred properties into mainly four classes: (i) unordered usage patterns, (ii) sequential usage patterns, (iii) API constraints, and (iv) API migration and mapping considerations. We conclude this chapter with a discussion on the limitations of the presented research contributions (Section 2.4).

2.1 Empirical studies on API usability

Previous studies on API usability try to identify factors that hinder the usability and learnability of APIs.

Ellis *et al.* conducted a study, with twelve participants, to evaluation with five programming tasks the usability of the Factory pattern in API design as compared to constructors for object creation [20]. The authors observed that the participants spent significantly more time to create an object from factories used in APIs than with a constructor. Additionally, the results suggest that the use of factories in APIs should be avoided in many cases where other techniques, such as constructors, can be used instead. In the same context,

Stylos *et al.* conducted a user comparative study to assess how developers use APIs with required parameters in objects' constructors as opposed to parameter-less "default" constructors [70]. Six programing tasks and thirty professional developers were involved in the study. One may presume that parameters would create more usable and self-documenting APIs by guiding developers toward the

correct use of objects and preventing errors. However, the study reported that unexpectedly, developers strongly preferred and were more efficient with APIs that did not require constructor parameters.

In another study, Stylos *et al.* evaluated the impact of method placement on the API usability. On which class or classes a method is placed is important since developers often start their exploration of an API from one "main" class. The study reports that participants were significantly faster at identifying relevant dependencies and combining objects when a class from which users generally start to explore an API had methods that reference other classes in the API. This significantly enhanced the productivity of the developers [72].

Other studies looked at the role of web resources in learning how to use APIs. Brandt *et al.* observed, in a lab study involving twenty participants and five tasks, that programmers used the Web primarily for just-in-time learning of new skills, and to clarify or remind themselves of previously acquired knowledge [9]. In a different study, Stylos *et al.* identified several challenges developers encounter when using the Web to find API elements and usage examples. For instance finding the right terminology to describe API concept, spending time looking at irrelevant search results. Even when a search did yield some relevant results, if the first few documents the developers browsed did not seem relevant, they would often give up [71].

The more recent studies were interested in understanding the difficulties encountered with unfamiliar APIs. In his study, Robillard investigated the obstacles professional developers at Microsoft faced when learning how to use APIs [58]. Around 80 developers answered the survey and a series of 12 interviews was conducted to identify what exactly does make an API hard to learn. The overarching result of this study is that the resources available to learn an API are important and that shortcomings in this area hinder the API learning progress.

In an another study, Robillard *et al.* collected the opinions and experiences of over 440 professional developers and report on the obstacles developers face when learning new APIs, with a special focus on obstacles related to API documentation [59]. The study shows that when developers learn a new API they struggle not so much in the mechanics of using the API, but in understanding how the API relates towards its problem domain. The study found that developers need help mapping desired scenarios in the problem domain to the content of the API, and in understanding what scenarios or usage patterns the API provider intends and does not intend to support. Thus showing a pattern of related calls is preferred to illustrations of individual methods. The study also found that developers

want to understand how the API's implementation consumes resources, reports errors and has side effects.

In the same context, Duala-Ekoko *et al.* conducted a study in which twenty participants completed programming tasks using real-world APIs [19]. Through a systematic analysis of the screen captured videos and the verbalizations of the participants, the study isolated twenty different types of questions the programmers asked when learning to use APIs. Among the identified question, the following two held our attention.

– What is the valid range of values for a primitive argument, such as an integer, of a given method?
– Is NULL a valid value for a non-primitive argument of a given method?

Discussion forums were also explored to understand developers' needs. Hou *et al.* conducted an exploratory study in which they manually analyzed in detail 172 programmer discussions, from a newsgroup, about specific challenges that programmers had about software APIs [32]. The objective was to identify what makes APIs hard to use, and what can be done to alleviate the problems associated with API usability and learnability. The study identified several categories of obstacles in using APIs. The most prominent observation was that developer asked for API usage solutions without actually attempted anything concretely, especially when the programing tasks require the composition of API methods calls. Another arresting obstacle was the incorrect usage of APIs. In this case, the developer generally tries the right solution but the program does not work as expected due to mistakes in performing certain steps of the solution. Sometimes, a mistake was made by supplying the wrong parameter values to some API methods. And in some cases, this was because the programmer is unaware of the special constraints that the API method imposes [32]. A similar study was conducted by Wang *et al.* [82]. The study explored API usage obstacles through analyzing API-related posts regarding iOS and Android development from a Q&A website. The Study reported some scenarios that appear to be the common cause of API usage obstacles, and presented a list of iOS and Android classes that often cause usage obstacles without being frequently used.

2.2 API documentation techniques

The impact of documentation on the usability of APIs has also been an area of active research [16, 36, 44, 48, 73, 78]. Researchers attempt to make API documentation accessible and understandable to programmers.

Recent studies were interested in determining important types of knowledge conveyed in API reference documentations. Maalej *et al.* report on a study of the nature and organization of knowledge contained in the reference documentation of hundreds of APIs [44]. They provided a global perspective on 12 types of knowledge conveyed in API documentation and their distribution throughout the reference documentation. Maalej *et al.* found that functionality knowledge is pervasive and structure is common, while other types, such as concepts and purpose, are much rarer.

Similarly, Monperrus *et al.* performed an extensive empirical study on the directives of API documentation [48]. The study proposed a taxonomy of 23 types of directives present in the documentation of Java APIs. The taxonomy was constructed by analyzing more than 4000 API documentation items. The taxonomy consists of the following high-class directives.

- Method Call Directives: related to constraints and guidelines when calling a particular library method.
- Subclassing Directives: related to requirement that has to be satisfied when subclassing a library class.
- State Directives: related to requirement on the internal state of receivers of a given method call.
- Alternative Directives: related to alternative implementations of given API element.
- Synchronisation Directives: related to concurrency on an API element.
- Miscellaneous Directives: directives related to software environment in general.

Dekel *et al.* worked on highlighting directives present in the Javadocs reference documentation [16]. Their tool, eMoose, makes programmers aware of important usage guidelines or directives from the documentation of API methods. The violation of such directives could lead to bugs. The tool provides several helpful directives, which can be identified in eMoose by means of tags in the documentation. API developers and contributors have to include such tags in the documentation, and it is difficult to identify directives in existing documentation that does not include tags.

The reference documentation is an important form of API documentation. However, studies found that developers use reference documentation only when they cannot get the needed information from other possible sources [53]. This could be due to the presentation or the content of the reference documentation. Nykaza *et al.* identified the importance of an overview section in API reference documentation [53], and Jeong *et al.* highlighted the importance of explaining the API exploration starting points to increase the quality of the documentation [34].

Kim *et al.* proposed eXoaDocs [36], a tool that integrates code snippets, mined from a source code search engine, into the Java API reference documentation. eXoaDocs queries the search engine for code examples that use a given API method, then eliminates from the code examples non relevant segments to the use of the API method, and integrates the resulting code snippet in the description section of the method in the API documentation. eXoaDocs was able to embed source code examples for more than 20,000 API elements.

Stylos *et al.* proposed Jadeite [73], a tool that takes advantage from usage statistics of the APIs classes and methods in code examples on the Web. It displays commonly used API elements more prominently in the documentation. The tool also integrates code snippet on how to create instances of API classes in the documentation. Additionally, Jadeite introduced the concept of "placeholders", a feature which enables API designers or users to annotate the API documentation with classes they expect to exist in a given package of the documentation, or methods they expected to exist on a given class, and to add forward references to the actual parts of the APIs that should be used instead of the "placeholders".

More recently Treude *et al.* proposed an approach to augmenting API documentation with insights sentences derived from Stack Overflow sentences that are related to a particular API type [78]. They proposed a machine learning based approach that uses as features the sentences themselves, their formatting, their question, their answer, and their authors as well as part-of-speech tags and the similarity of a sentence to the corresponding API documentation. The proposed approach outperformed two state-of-the-art summarization techniques as well as a pattern-based approach for insight sentence extraction. Moreover, the results show that considering the metadata available on Stack Overflow along with natural language characteristics can improve existing approaches when applied to Stack Overflow data.

2.3 API property inference

Large APIs are difficult to use, because of hidden assumptions and requirements. Developers should be aware of different API properties, to correctly use the API. This is why many approaches have been proposed to infer these API properties. Each approach comes with a new definition of API properties, new techniques for inferring these properties, and new ways to assess their correctness and usefulness. In the following subsections we classify existing techniques into four broad categories.

2.3.1 Unordered usage patterns

A basic type of property that can be expressed about an API is that of an unordered usage pattern. Patterns are typically observed from the data as opposed to being formally specified by a developer. And usage patterns describe typical or common ways to use an API. Unordered usage patterns describe references to a set of API elements that co-occur with a certain frequency within a population of usage contexts. As an example of unordered usage patterns, we may detect the pattern {open ; close}, which indicates that whenever client code calls an API method open, it also calls the method close, and vice versa. This pattern is unordered, as it does not encode any information about the sequence between open and close methods.

Michail was the first to explore the use of association rule mining between a client and its library to detect usage patterns [45, 46]. Michail's idea was to help developers understand how to reuse classes in a library by indicating relationships such as "if a class subclasses class C , it should also call methods of class D". Michail detects these relations by mining client code that uses the API of interest. This preliminary work seeded the idea of using association rule mining on software engineering artifacts. Michail targets the discovery of rules, and thus he applies his approach with very low support and confidence, observing that a filtering step is necessary for the approach to be feasible.

Unordered usage patterns can also be used to detect bugs. Li *et al.* used association rule mining, in PR-Miner, to automatically detect unordered usage patterns [40]. PR-Miner parses source code to store identifiers representing functions called, types used, and global variables accessed. The stored identifiers are then used as items in the mining algorithm. Once identified, the patterns are considered as rules and used to find violations. The assumption is that rule violations can uncover bugs. The tool

was evaluated on three C/C++ systems, with up to three million lines of code. PR-Miner extracted more than 32,000 programming rules from the evaluated systems, and detected around 82 violations to the extracted rules. However, Li *et al.* noted that a large number of the association rules are false positives, even with pruning steps.

The tool DynaMine proposed by Livshits *et al.* shares the same goal [42]. It infers usage patterns by mining the change history of an API's clients. The idea is that method calls that are frequently added to the source code simultaneously often represent a pattern. DynaMine automates the task of collecting and pre-processing revision history entries and mining for common patterns. Likely patterns are then presented to the user for review; runtime instrumentation is generated for the patterns that the user deems relevant, and patterns are checked by executing the client's source code. Results of dynamic analysis are also presented to the user. The authors find only 56 patterns in the change history of Eclipse and jEdit using the chosen confidence and support thresholds, and only 21 of which are observed to occur at runtime. Additionally, the tool detected a total of 263 pattern violations.

Another approach that focuses on bug detection is the one of Monperrus *et al.* [47]. The objective of this approach is to detect missing API method calls. They collect statistics about type-usages, a type-usage being simply the list of methods called on a variable of a given type in a given client method. Then, they use this information to detect other client methods that may need to call the missing method. Their idea is implemented in a tool called DMMC (Detector of Missing Method Calls). Monperrus *et al.* do not use any standard data mining algorithm as part of their approach. Rather, for a given variable x of type T, they generate the entire collection of usages of type T in a given code corpus. From this collection, the authors compute various metrics of similarity and dissimilarity between a type usage and the rest of the collection. The missing method calls are detected through the characterization of deviant code on top of similarity and dissimilarity metrics. The tool produces warnings for type-usages whose degree of deviance reaches a certain threshold.

Other techniques that detect unordered usage patterns have been proposed to recommend API elements useful for programming tasks. Bruch *et al.* proposed FrUiT, a tool to help developers learn how to use a framework by providing them with context-sensitive framework usage patterns, mined from existing code examples [10]. Based on these patterns, suggestions about other relevant parts of the framework or the API are presented to novice users. The tool combines data mining techniques with a context-dependent recommendation. Currently, FrUiT's suggestions use the whole class in the

active editor as a context for recommendation. A more recent technique proposed by zhang *et al.* was interested in API method parameters recommendation [86]. The tool, called Precise, mines existing code base, uses an abstract usage pattern representation for each API usage scenario, and then builds a parameter usage database. Upon request, Precise queries the database for abstract usage patterns in similar contexts and generates parameter candidates by concretizing the patterns adaptively.

2.3.2 Sequential usage patterns

Sequential usage patterns differ from unordered patterns through considering the order in which API operations are invoked. Sequential pattern mining would be able, for instance, to alert the programmer that open should precede close for the pattern {open; close}. The most common goals for mining sequential API usage patterns are API documentation and bug detection. Techniques developed for API documentation infer some high-level patterns from a program, and assume that these patterns will be valuable for API documentation. On the other hand, techniques developed for bug detection typically go one step further. They detect usage patterns, and use these patterns for anomaly detection.

Sequential patterns can be derived from a wide variety of inputs. Inference techniques can be distinguished by the input they require. The main difference is naturally between dynamic and static approaches. Dynamic approaches typically read a single execution trace as input. Whereas for static approaches, the most popular strategy is to analyze API client programs source code. Gabel *et al.* proposed a runtime tool for inferring and checking simple temporal patterns using a sliding-window technique that considers a limited sequence of events and mine as usage patterns, regular expressions, with exactly two method calls involved in each regular expression [25]. The static technique by Whaley *et al.* uses interprocedural analysis and constant propagation to find call sequences of methods that may establish conditions of predicates that guard throw statements [85]. These sequences are considered illegal. Whaley *et al.* assume that programmers of languages with explicit exception handling make use of defensive programming: A component's state is encoded in state variables, and state predicates are used to guard calls to operations and cause exceptions to be thrown if satisfied.

We further distinguish sequential mining approaches by the kind of mined patterns. A significant number of approaches mine instances of a single sequential pattern. Such sequential patterns can consist simply of an ordered pair of API elements, indicating that the usage of one element should

occur before the other element in a program's execution. An example of this category is the previously mentioned approach proposed by Gabel *et al.* [25].

A further class of approaches mine instances of several patterns at once. Such patterns are generally instances of special temporal patterns such as Initialization, Finalization, Push-Pop, or Strict Alternation [41]. Other approaches are also based on mining instances of certain patterns but describe these patterns using temporal formulas, boolean formulas or temporal logic [43].

Many techniques mine API usage patterns by encoding temporal order as finite-state automata [5], or using special representation models such as Groums. Groums are Graph-based object usage models a special-purpose property representation, used by Nguyen *et al.* in GrouMiner [52]. Groums associate events in a directed acyclic graph (DAG). In contrast to finite-state automata, this graph can hold special nodes to represent control structures, such as loops and conditionals. Furthermore, edges not only represent sequencing constraints, but also data dependencies.

Some approaches use frequent item set mining, and include temporal information in the definition of the elements in the item sets [84]. Other approaches directly mine sequential patterns by using closed frequent sequential pattern mining. This mining technique exhibits a higher computational cost, but has the advantage of retaining useful information like the frequency of elements. Most of these approaches use the BIDE algorithm [80]. For instance, Zhong *et al.* proposed MAPO, a tool that clusters frequent API method call sequences extracted from client programs, then use the BIDE algorithm to mine closed sequential patterns from the preprocessed method call sequences [87]. Thus, MAPO capture groups of API's method that are frequently used together. Wang *et al.* build on MAPO and propose an approach that add pre and post clustering to reduce the number of redundant patterns and detects more succinct ones [81].

2.3.3 API constraint

A number of approaches have been developed to describe the valid and invalid behavior of the API when certain API properties are either met or more typically not met. One form of behavioral description is through constraint, such as pre-conditions, post-conditions, and invariants defined over an abstract data type or a class. A typical example of a pre-condition is that a value passed as an argument to a function should not be a null reference.

Existing techniques infer the API constraint through various analyses on the source code either of the API or its client program. To this end, two general strategies are used to generate behavioral specifications of APIs: the conjecture/refute strategy and the symbolic execution strategy.

Henkel *et al.* proposed an approach that follows the conjecture/refute strategy [29]. It systematically explores the state space of the class for which specifications are generated. They first create an instantiation transition for the class, and then systematically construct a sequence of invocations of increasing complexity on this instance. Then they focus on the detection of invalid sequences. To invalidate the conjectured invariants the tool executes code corresponding to the synthetically generated transitions. When executions result in an exception, the corresponding sequence is flagged as invalid. Buse *et al.* proposed an exception documentation reverse-engineering approach [13]. The approach follows the symbolic execution strategy. The use of symbolic execution in this approach only identifies paths that result in an exception. Thus the tool produces for each (method, exception-type) pair, a path predicate that describes constraints on the values of the method's variables that will result in a control-flow path ending in an exception of that type to be raised. Specifically, Buse *et al.* approach infers, for a given API method, the possible exception types that might be thrown, the predicates over paths that might cause these exceptions to be thrown, and human-readable statements that describe these paths.

2.3.4 API migration and mapping

APIs evolve continuously and in a very fast way, client program developers may need support to update clients of an API when the API evolves with backward-incompatible changes. Alternatively, and due to license constraints, client application may need to switch between different, but equivalent, APIs. In this context, several techniques have been proposed to infer migration mappings of elements declared in one API to the corresponding elements in a different API, or in a different version of the same API.

The most basic mappings are those that correspond to simple API refactorings, such as renaming API elements or moving them to a different module. For instance, RefactoringCrawler by Dig *et al.* infers mappings between refactored versions of an API [17]. It applies text similarity metrics to the signatures of API elements. After identifying the most similar API element pairs by performing a syntactic comparison between all API element pairs across the two API versions, RefactoringCrawler

validates each refactoring candidate by checking whether references to and from the old element are similar to the references to and from its candidate replacement element.

Other techniques go beyond standard refactorings by discovering more general mappings between API elements. SemDiff by Dagenais *et al.* is an example of such techniques [14]. SemDiff does not relate API elements by the syntactical similarity of their signatures. It rather identifies possible candidate replacements for a method by analyzing the change history of a client of the API that has already been migrated. SemDiff makes the hypothesis that, generally, calls to deleted methods will be replaced in the same change set by one or more calls to methods that provide a similar functionality. Having identified such change sets, SemDiff then ranks all detected replacements with respect to a popularity-based metric, as well as name similarity, and presents these as a ranked list to the user.

Other techniques also present some kind of migration guidelines that illustrate how references to the current element can be replaced by references to its target element(s). Nguyen *et al.* tool LibSync falls in this category by discovering these series of steps required to update a client [51]. LibSync represents API usages as directed acyclic graphs called GROUMs that basically capture reference and inheritance based usage of API methods and types, as well as various control structures and dataflow dependencies between them. Given some mapping between an initial set of elements in the current API version and their replacements set in a different version, LibSync first identifies GROUMs describing usages of the elements in the initial set in the old versions of client code, and then computes edit scripts to describe how those usages differ from the usages of the elements in the replacements set in the new versions of its client code. GROUM-based edit scripts are provided to frequent item set mining to generalize common edit operations.

2.4 Summary

Through this chapter, we presented a review of the existing work related to our contributions. We first presented different studies that motivate the importance of facilitating usability and learnability of APIs. We share with all the authors of these studies the idea that when developers learn a new API, they struggle not so much on the mechanics of using the API, but in understanding how the API elements are related to scenarios or usage patterns the API provider intends to support. Thus, showing a pattern of related calls is preferred to illustrations of individual detailed method usage.

Several approaches and tools have been proposed to deal with the learnability and usability problems of APIs. We presented various techniques that range from intuitive solutions to very complex approaches. The majority of presented techniques either infer simple intuitive patterns or require high computational cost. However, efficient tools should be scalable to work with large code bases with millions lines of code and thousands of public API methods. To be inline with this requirement, we need a non-expensive technique useful during the early step of the API learning process. Thus, we opt for the mining of non-sequential usage patterns, and we avoid complicated data representations such as partial order graphs.

The majority of the used techniques rely on the existence of large collections of API client programs to be effective. Consequently, newer APIs or non-popular parts of existing APIs may not have sufficient usage examples to infer API patterns. We try to overcome this strong constraint and propose technique based on relationships inside the library itself. Our technique is thus applicable for non-popular API new releases and even new APIs.

As compared to approaches that infer unordered usage patterns, the majority of these approaches use a form of frequent itemset mining. Thus, the mined pattern tends to be many and redundant, posing significant barriers to their practical adoption, and introducing an additional effort to filter, classify and learn the patterns. Moreover, inferred patterns are flat and valuable only for recommendation or auto-completion in a specific context and for a specific usage scenario. To overcome these constraints, our technique should reflect interfering usage scenarios.

Through the presented overview of the usage patterns inference techniques, we can distinguish different targeted goals. Several techniques were interested in documentation of usage patterns, and others were interested in detection of violations to usage patterns or recommendation of API elements. In our case, we try to ease the understanding and learnability of APIs for the early step of API learning and usage process. The inferred constraints and patterns can then be easily integrated in an API exploration process or to enrich documentation.

In the next chapters, we describe our contributions for API usage constraints, and patterns detection, and we show how to circumvent the above-mentioned problems.

Part I

Library usability at the API method scope

Chapter 3

API Usage Constraints Inference

Nowadays, APIs represent the most common reuse form when developing software. However, the reuse benefits depend greatly on the ability of client application developers to use correctly the APIs. In this chapter, we present an observational study on the API usage constraints and their documentation. To conduct the study on a large number of APIs, we implemented and validated strategies to automatically detect four types of usage constraints in existing APIs. We observed that some of the constraint types are frequent and that for three types, they are not documented in general. Surprisingly, the absence of documentation is, in general, specific to the constraints and not due to the non-documenting habits of developers. These findings justify the importance of supporting library usability at the API method level. This contribution was published at the *IEEE International Conference on Software Analysis, Evolution, and Reengineering* [64].

3.1 Introduction

As mentioned earlier, much research effort has been dedicated to the redocumentation of APIs [12, 13, 30, 60], and proposed to recover various types of information such as usage constraints and usage examples. Nevertheless, most of these contributions were devoted to a specific type of constraints (method-call sequences, exceptions, etc.) and tried to redocument valid and invalid behavior of the API. None of these contributions could address all possible types of constraints. Consequently, some constraint types, especially those related to the parameters of API methods, may not be considered in redocumentation tasks.

This can be understandable if we conjecture that these constraints are not frequent, and if they are usually documented by the API developers. However, even if some studies were interested in building taxonomies of constraints [48], to our best knowledge, there is no empirical evidence about the frequency and documentation level of such constraint types.

In this chapter, we present an observational study that targets some usage constraints and their documentation in existing APIs. To conduct the study, we selected four critical usage constraint types

that deal with method parameters, namely, *Nullness not allowed*, *Nullness allowed*, *Range limitation*, and *Type restriction*. These were among the usage constraints identified in [48]. We implemented automated strategies for finding instances of the selected usage constraints and validated them, with subjects, on 13 APIs of JDK7.

Our study was conducted on a sample of 11 real-world APIs excluding the 13 APIs used to validate the detection algorithms, except for one of the research questions. The results of our study show that some of the constraint types are used extensively and that for three types, these constraints are poorly documented in general. Moreover, the absence of documentation is, in many cases, specific to the constraints and not due to the non-documenting habits of developers.

This chapter is structured as follows. Section 3.2 discusses with examples the importance of identifying usage constraints. Section 3.3 describes the usage constraints selected for our approach and show their detection strategies. Section 3.4 gives the setting and the results of our observational study, including the validation of the detection algorithms. The threats to validity and a conclusion are provided respectively in sections 3.5 and 3.6.

3.2 Motivating examples

After deciding which method to call, providing the right values for the parameters is among the most important decisions when using an API. This is why client developers usually ask several questions in relation to method's parameters when they reuse an API [19]. Since the signature of a method is rarely enough expressive about most of the parameters' usage constraints, it is necessary to be aware of such constraints. In this section, we provide examples showing that relying only on method signatures can induce the developer in error. That is why the explicit documentation of usage constraints is needed. The following examples are all extracted from the JDK7 APIs.

– **Example 1.** Consider the method `public Object parseObject(String source, ParsePosition pos)` from the Java class `DateFormat` which is a class for date and time formatting. This method parses a string to produce a `Date`. It attempts to parse text starting at the position given by `pos`.

Just from the method signature, a developer can possibly conjecture that if the `pos` parameter is not given (null), then the method is going to parse the whole text. However, the null value has no particular semantic here, and will result in an exception.

– **Example 2.** This example shows exactly the opposite situation, where the null value has a specific semantics important for the usage of the API method. Consider the following method `public boolean hasListeners(String propertyName)` from the utility class `VetoableChangeSupport`. The method in which we are interested checks if there are listeners for a specific property.

  Based on the signature, a developer can legitimately understand that passing a null value is not allowed. Actually, when no property name is given, the method checks for listeners registered on all properties. As in the previous example, a documentation is required to explain how the null value is handled.

– **Example 3.** Another interesting example is one from the class `MulticastSocket`. This class is useful for sending and receiving IP multicast packets, with additional capabilities for joining `groups` of other multicast hosts on the Internet. The signature of the `setTimeToLive` method is `public void setTimeToLive(int ttl)`. This method set the default time-to-live for multicast packets sent out on the considered `MulticastSocket`.

  When looking at the declared type of the `ttl` parameter, the developer assumes that any integer value can be passed to specify the default time-to-live for multicast packets. Nevertheless, a restriction on the possible values of the parameter is imposed and only a value in the ranges $[0, 255]$ is accepted, otherwise the parameter is considered as illegal.

– **Example 4.** The last example considered to show the importance of inferring usage constraints is found in class `Proxy`. The following method in this class uses a parameter having as type `Object`.

  `public static InvocationHandler getInvocationHandler(Object proxy)`

  Although the signature uses a generic type, only instances of class `java.lang.reflect.Proxy` can be passed as arguments of the method, which returns the invocation handler for the specified instance. Otherwise, an `IllegalArgumentException` is going to be thrown.

These examples illustrate well the need to report such usage constraints, to API users. Still, it is important to study if these kinds of constraints are actually documented or not in the existing APIs.

3.3 Approach

This section presents our approach to infer API usage constraint from the library source code. For this study, we focus on usage constraint related to API method's parameters. We first present the selected constraint types, for which we will inspect the presence in library source code and the degree of documentation. Then we describe the strategies defined to detect the selected constraints.

3.3.1 Constraint type selection

A clear understanding of what developers usually document will help us targeting a subset of usage constraints to include in our study. As mentioned earlier, Monperrus et al. in [48] conducted a study on API documentation to understand the kind of included directives. In their work, a directive is defined as a natural-language statement that makes developers aware of constraints and guidelines related to the API usage. As a result of the study, they extracted a taxonomy containing 23 directive types grouped into six categories. Almost the half of these directive types (11) belong to the method call category, which also includes the largest portion of directive occurrences in the studied APIs (43.7%). Among the 11 directive types of this category, the most frequent one refers to the fact that a parameter cannot be null (13%). Conversely, they also found that many directives refer to the possibility of passing a null value to a parameter and explain its semantic. This directive type is less frequent in general than the first one, but more frequent in some of the considered APIs such as in JDK.

In addition to the previous contribution, we also considered the exploratory study conducted by Duala-Ekoko and Robillard on the questions asked by client developers when using unfamiliar APIs [19]. Among the frequent questions, many are related to values that can be passed to the methods and especially the valid types and value ranges.

Consequently, we retained the following constraint types:

Nullness not allowed: A situation in which a null parameter, passed as an argument to a method in the library, causes failures during execution.

Nullness allowed: A situation in which the argument passed to a method can be null. This value has a specific semantics for this parameter.

Type restriction: A situation in which the type declared by the method parameter is not enough to be aware of various restrictions on the parameter type. Only a subset of type is allowed to avoid the execution failures.
Range limitation: A situation in which the restriction on the values of a numeric parameter goes beyond possible restrictions through the declared types.

3.3.2 Detection strategies

Almost all the occurrences of the selected usage constraints can be detected using a static intra-procedural analysis, which is applied to the control flow graph (CFG). The analysis for some constraint types is flow-sensitive whereas, the one of the others is path-sensitive [49].

3.3.2.1 Nullness not allowed analysis

The objective of this analysis is to identify situations that prohibit a method parameter from being null. To this end, we defined and combined two forward branched flow analyzes. More precisely, the CFG is traversed from the entry node and, for each node, we determine if a given parameter is, before this statement, definitely not null (NON_NULL tag), definitely null (NULL tag), both values are possible (TOP tag) or we just don't know (BOTTOM tag). The first analysis method, named *NullnessAnalysis* starts from the basic idea that a variable `x` (the parameter in our case) is considered not-null after instructions of the form '`x = new()`', '`x = this`' or any other assignment of something not derived from x itself, we can also know that the parameter is not null if tests such as '`x instanceof T`' succeed. In addition, we can deduce that the variable `x` is null on the true branch after conditional expressions that test the nullness of the variable. This node tagging is then used in a second analysis that locates statements containing references to arrays, field references, method invocations and monitor statements. These types of statements may generate unchecked exceptions when the manipulated variables are null. Thus, the objective of our analysis is to detect the use of the parameters in one of the statements mentioned above. If the analysis couldn't determine that the considered parameter is always not null before such statements, a *Nullness not allowed* constraint is then detected.

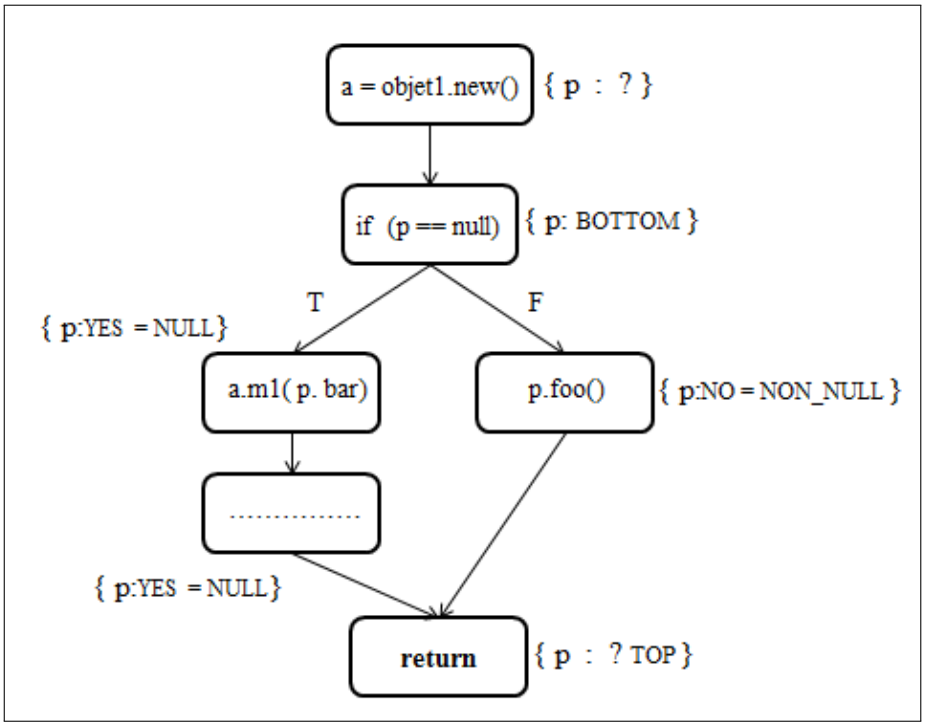


Figure 3.1: NullnessAnalysis example.

In the example of Figure 3.1, the *NullnessAnalysis* tags the nodes of the CFG for a parameter `p`. Then, the second analysis traverses the CFG and locates the statement '`a.m1(p.bar)`' in which the field `bar` is accessed while the node is tagged NULL for `p`. Consequently, a *Nullness not allowed* constraint is detected for the analyzed method.

3.3.2.2 Nullness allowed analysis

One can conjecture that if the previous analysis does not detect a *Nullness not allowed* constraint, then, null is allowed for the method parameters. However, the fact that it is allowed to pass a null value as an argument to a method, does not mean that the null value has a particular semantics that should be known by the client developer. For this reason, we are only interested in methods where a null value is not prohibited for a parameter and where the null value has a semantic which is reflected by a particular behavior of the method. The intuition behind our analysis is to consider each parameter as a potential candidate for the *Nullness allowed* constraint, from the moment we can be sure using the *NullnessAnalysis*, that the parameter in question is definitely null, at a given node, and that this node

dominates a block of other nodes, representing a potential behavior. A node n_1 dominates another one n_2 if all the paths that pass by n_2 pass by n_1 before. If such a case occurs, then, a *Nullness allowed* constraint is detected.

3.3.2.3 Range limitation analysis

In this analysis, we identify the cases where only a specific range of values is allowed for a numerical parameter, and the declared type is not specific enough to describe this restriction. This analysis gives, when necessary, the legal ranges for the parameter in question. To achieve this goal, we combine two forward-Analyzes, the first is path-sensitive and the second is flow-sensitive. The goal of the first analysis (call it *RangeLimitationAnalysis*) is to determine for each numerical parameter at each location of CFG (before each node), the range of values that the parameter can have. The basic idea is to initially consider that the value range of each numerical parameter is bounded by the smallest and largest value defined by its declared type. Then, based on the comparison operators involving the parameter and used in conditional statements, we reduce the initial range for the different branches of the CFG. This information is then reused in a second analysis which checks whether reduced parameter ranges are present in all legal exit points of the method. In this case, a constraint is detected with the reduced ranges for the given parameter.

If we reconsider the example mentioned in Section 3.2 of the `MulticastSocket` class (Example 3), a simplified control flow graph of the method `setTimeToLive(int ttl)` is shown in Figure 3.2. In this example, the *RangeLimitationAnalysis* starts by setting the range of parameter `ttl` in the entry point to the initial range of an integer $]-\infty,+\infty[$ (actually, $\left[-2^{31},2^{31}-1\right]$). Then, this range is reduced as the graph is traversed to reach $[0,255]$ for the legal exit point. Any other value results in an exception. As the range at the legal exit point is reduced compared to the initial one, a constraint is detected.

3.3.2.4 Type restriction analysis

The objective of this analysis is to detect type restrictions that are not expressed by the type declared for the method parameter. Indeed, several methods declare parameter types by conformity with inheritance and method overloading. However, it is possible that the generic types do not work

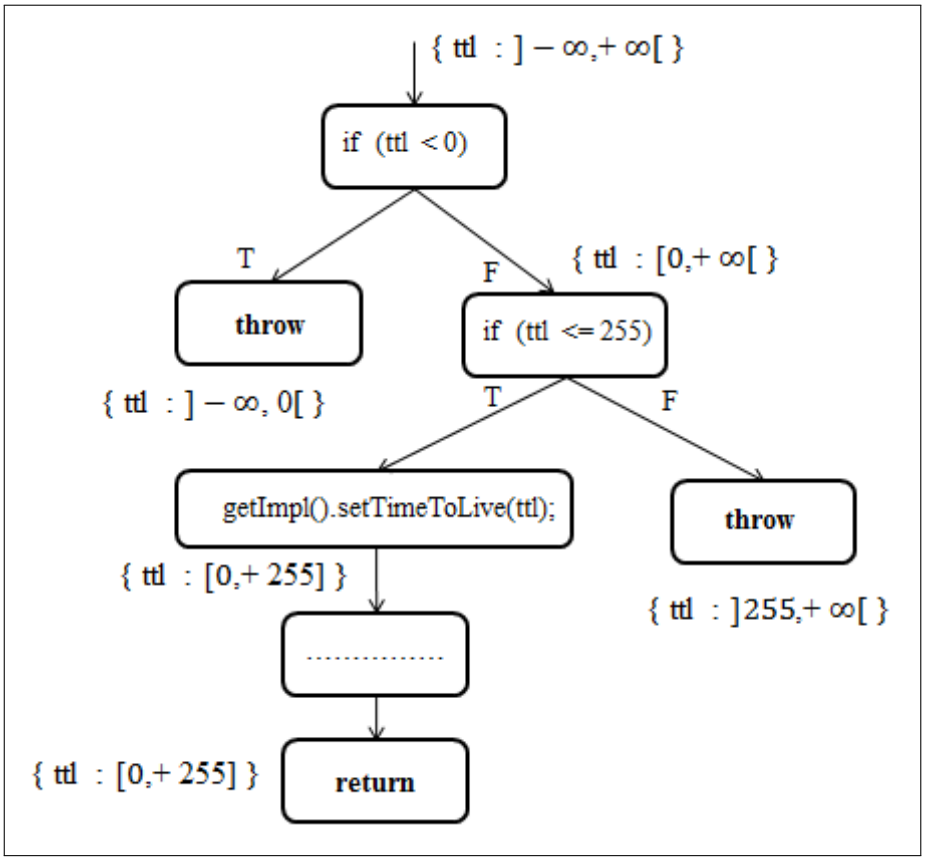


Figure 3.2: RangeLimitationAnalysis example.

for a given method redefinition. To detect type restriction cases, we combined a forward path-sensitive analysis with a flow-sensitive one.

The first analysis (call it *TypeRestrictionAnalysis*) is similar to *NullnessAnalysis* and to *RangeLimitationAnalysis*. The difference is that the propagated information here is related to the parameter type and not to its values. This analysis tags each location, for each parameter, as NOT RESTRICTED, RESTRICTED (with the restricted type), TOP, or BOTTOM. To this end, we mainly use the instructions that contain casting statement `d=(T)a`, and conditional expressions that use the *instanceof* operator to find out to which type a given instance belongs, e.g., *if (o instanceof X)*. The type information is then used in a second analysis to ensure that the restriction is valid on all the legal exit points of the method. If this is the case, a *Type restriction* constraint is detected.

3.4 Evaluation

To evaluate our approach, we first evaluated the correctness and validity of our constraints detection strategies. Then, we studied the usefulness of detecting the selected constraint types, looking at the amount of their presence in real-world APIs, and the degree of their coverage in standard documentation. In this section, we describe our experimental setup and present the obtained results.

3.4.1 Detection validity evaluation

The objective of this part of our study is to evaluate the correctness of our constraint detectors. We address the following research question.

– **RQ1:** To what extent can the proposed strategies detect the constraint of interest?

3.4.1.1 Setting

The process we followed to address our research question, starts by selecting a set of APIs known for the quality of their documentation. To this end, we selected 13 APIs of JDK7, enumerated in Table 3.I. To check the correctness, we ran our detection algorithms on the public methods of the 13 APIs and randomly selected 300 occurrences to manually check for the detection precision.

We selected 15 independent subjects (practitioners and graduate students) to manually assess the correctness of the 300 constraints of our sample. Our subjects include one senior developer, four junior developers, two M.Sc. students, and eight Ph.D. students. All the participants have at least an experience of three years in java programming and are familiar with API usage and Javadoc documentation. The task of each participant involved reading the automatically generated description for the detected constraints and checking whether the Javadoc for the corresponding method mention this constraint.

To minimize the dependency on subjects' judgement, each constraint was evaluated by three subjects. As a consequent, we assigned 60 detected constraints to each subject. We ensured, when preparing the material, that each subject evaluates occurrences of the four constraint types and that the same constraint appears in random positions for the three concerned subjects. The evaluation resulted in 900 opinions ($15 * 60$), with three opinions per constraint. If at least two of the three subjects decide that the Javadoc refers to the constraints, the detection is considered correct. Otherwise, two

API Name	Nb of evaluated classes	Nb of evaluated methods
java.lang	167	2394
java.awt	267	4844
java.math	7	300
java.beans	41	547
java.io	74	1012
java.rmi	50	235
java.net	69	1040
java.applet	1	27
java.nio	186	1980
java.security	128	968
java.sql	27	222
java.text	42	661
java.util	205	4060

Table 3.I: Selected APIs for the detection evaluation

cases are possible: either the detected constraint is a false positive or the Javadoc does not report this constraint. To distinguish between these two cases, we manually checked the suspected false positives in the API source code to make a final decision.

To train the participants on the evaluation tasks, we gave them written instructions, including the evaluation process and the definitions of constraint types. Then, we organized a training session, in which the participants were asked to evaluate some constraints. After completing the tasks, we discussed with them the answers and clarified any mistake or misunderstanding. To avoid fatigue and boring effects, we developed a web application that allows the subjects to complete the evaluation at their convenience. The web application offers the possibility to save the current state of the validation session and resume it later.

We calculate the precision as the proportion of detected constraints that are actual constraints.

$$Precision = \frac{Nb\_correct\_constraints}{Nb\_detected\_constraints} \tag{3.1}$$

To calculate the recall, we need to know, before hand, the list of actual constraints in the code. Here again, we used a sample of constraint. To define the sample, we searched automatically in the javadoc of the selected APIs, keywords that describe the four constraints types. Then, we manually

check the query results to determine if the documentation indicates the presence of a constraint. This process led to a sample of 123 occurrences.

The recall calculation was accordingly defined as the proportion of the manually sampled constraints detected by our algorithms.

$$recall = \frac{Nb\_correct\_constraints}{Nb\_actual\_sample\_constraints} \tag{3.2}$$

3.4.1.2 Results

We are interested, in this section, in the precision and recall of our detection algorithms. The complete results of the detection on the 13 API of JDK7 are presented in the sections 3.4.2. For the precision, the results are almost perfect for the four constraint types (96% to 100%) as shown in Table 3.II. The error margins are low enough to trust our automatic constraints detection strategies.

	Nb Detected constraint	**Nb correct constraint**	**precision**
Range limitation	75	75	100%
Nullness allowed	78	75	96%
Nullness not allowed	111	108	97%
Type restriction	36	36	100%

Table 3.II: Constraints detection precision

The recall results are reported in Table 3.III. A high majority of the actual constraints of our sample was detected (between 78% and 94%). The constraints that were not found require, essentially, inter-procedural analyzes. This is the case when, for example, the argument is used to set a class attribute. Then, this attribute throws an exception in another method. Another case is when the argument is directly used in another method call. As we are dealing with constraints that affect directly APIs methods called from the clients, we consider that the recall of our detection algorithms is sufficient for our study.

	Nb Detected constraint via queries	Nb Detected constraint via analysis	recall
Range limitation	26	22	85%
Nullness allowed	18	14	78%
Nullness not allowed	61	51	84%
Type restriction	18	17	94%

Table 3.III: Constraints detection recall

3.4.2 Detection usefulness evaluation

The main objectives of this part of our study is to evaluate the presence of usage constraints in Java APIs' source code and to observe the degree of their documentation, as an indication of the usefulness of detecting the selected constraints. To this end, we address the following research questions.

- **RQ2:** To what extent, usage constraints are present in real-world APIs?
- **RQ3:** When usage constraints exist in an API source code, are they documented in the Javadoc?
- **RQ4:** When usage constraints are not documented, is it specific for the constraint or because of the non-documentation of the method as a whole?

3.4.2.1 Setting

In order to address our research questions, we selected 11 APIs. To achieve a good level of representativeness, we balanced the following criteria: application domain, size, popularity in terms of the number of downloads, and intended audience (development vs research). The evaluated APIs are enumerated in Table 3.IV.

Figure 3.3 summarizes the followed process to address our research questions. This process is organized in four steps. The first step consists in applying the constraint detectors on all the selected APIs. The result of this step allows us to know, for each constraint type, the number of occurrences

1. Number of downloads made before the 30th April 2014

API Name	Description	Nb of evaluated classes	Nb of evaluated methods	Nb of downloads [1]	Intended audience
Super CSV	An open-source library for reading/writing CSV files.	64	336	105648	Development
Java MythTV	A library to query and control digital video recorder back-end and database.	168	1605	3038	Development
GeneticLibrary	A Genetic Algorithms Library.	3	39	604	Research
OpenJava Weather	A library for providing access to weather information from different sources.	27	187	36	Research
PetriNetExec	A library for embedding Petri Nets into Java applications.	36	218	466	Research
IPtablesJava library	A library for firewall logs, connection tracking and rule management.	27	290	1171	Development
Simple2D	A library to simplify menial and advanced graphics tasks.	14	149	5	Research
Java Marine	A library for enabling easy access to data provided by electronic marine devices.	57	500	4172	Research
tcpchannel	A processes communication library with TCP sockets.	7	48	142	Development
laverca	An open-source library for requesting signatures using mobile signature services.	29	226	514	Development
xtarget	A Library for automated editing of small XML Databases.	73	613	75	Development

Table 3.IV: Selected APIs

found in each API. By analyzing these results in terms of frequency and distribution over the APIs, we are able to answer our research question **(RQ2)**.

The second step is to extract the Javadoc documentation of the methods concerned with the detected usage constraints. Then, in a third step, the javadoc of a method is aligned with a simple, yet legible, description that we automatically generate to document a detected usage constraint. An

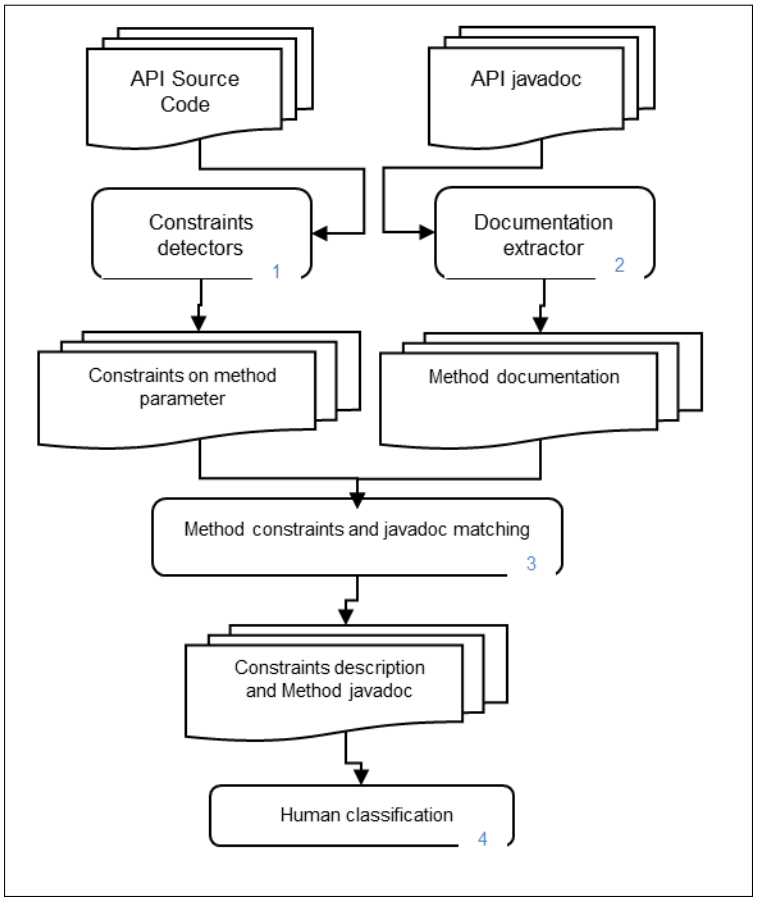


Figure 3.3: Overview of the evaluation process.

example of such an alignment is shown in Figure 3.4. The constraint description (1) is on the left part of the figure, and the method javadoc (2) is on the right part.

In the fourth step, we manually check if a detected usage constraint is documented, implicitly or explicitly, in the javadoc of the corresponding method, as sketched in the taxonomy presented in Figure 3.5. The documentation is considered as implicit when this latter mention an exception for illegal argument without giving explicitly the cause. We consider the documentation as explicit when the cause is stated. The data derived in this step, is used to answer the research question **(RQ3)**.

When a considered usage constraint is not documented, two cases are possible. The first one is that the developer did not comment the method at all. The other alternative is that the javadoc exists for the method, but the constraint is not mentioned in it. The distinction between the two possibilities is important. In the second case, it is difficult for the client application developer to suspect the

net.sf.marineapi.nmea.util.SatelliteInfo.setAzimuth(int) Constraint detected: the Range of parameter azimuth is limited to: [0, 360]	Set satellite azimuth, in degrees from true north (0..359°). @param azimuth the azimuth to set @throws IllegalArgumentException If value is out of bounds 0..360 deg.
(1)	(2)

Figure 3.4: side-by-side constraints description and javadoc.

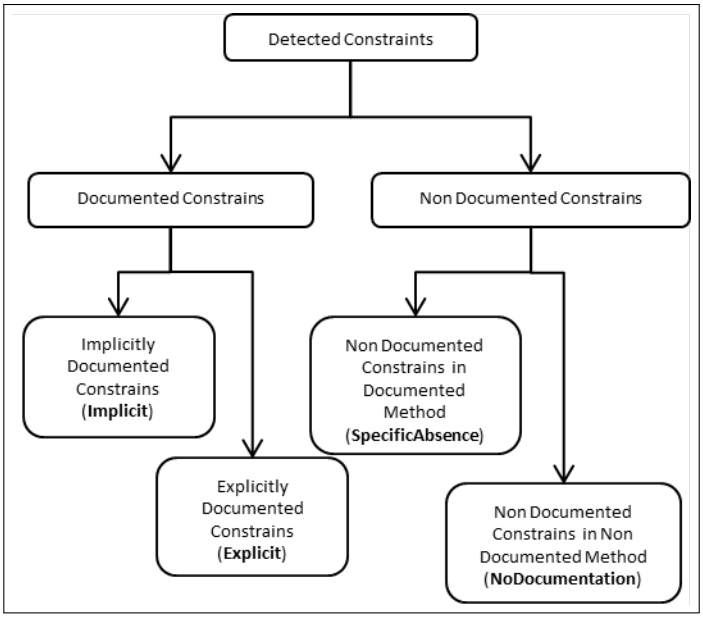


Figure 3.5: Constraint classification taxonomy.

existence of constraints not mentioned in the documentation. Consequently, the second case is more serious. This last classification allows us to answer the research question **(RQ4)**.

3.4.2.2 Results

In this section, we discuss the results for each research question.

Usage constraints existence (RQ2):

Table 3.V gives the detection results for the 11 APIs of Table 3.IV. As we are not dealing with the documentation issue at this stage, we also present, in Table 3.VI, the detection results for JDK APIs of Table 3.I.

API Name	Nullness not allowed	Nullness allowed	Type restriction	Range limitation
Super CSV	87	6	12	0
Java MythTV	183	132	2	3
GeneticLibrary	8	0	0	0
Open Java Weather	36	18	0	0
PetriNetExec	45	0	0	0
IPtables Java library	27	7	0	3
Simple2D	28	0	0	4
Java Marine	57	1	0	15
tcpchannel	15	1	0	0
laverca	28	24	0	0
xtarget	87	21	2	3

Table 3.V: Number of Detected Constraints

Globally, the four constraint types occur in the considered libraries with different degrees. As expected, the *Nullness not allowed* constraint is the most frequent one for the 24 APIs. The *Nullness allowed* constraints are also frequently present, especially for the JDK APIs. The third constraint type in importance is the *Range limitation*. We found occurrences in 16 APIs. The fourth constraint type is the less frequent. Occurrences were found for almost all the JDK APIs, but in only three APIs of our sample.

To give some insight on the found constraints, we tried to characterize the detected cases. Looking at the APIs' source code, we noticed that the *Range limitation* constraints concern numerical parameter with a fixed semantic. For example, we found constraints that limit the months to (1..12) , the day to (1..31), and hours to (0..23). Other constraints deal with angles, positions, ports, etc.

Nullness allowed constraints are mainly related to situations were null signify that a default value has to be used, as in the case of the method `valueOf(ProtocolVersion protoVersion, ...)` in the `Schedule` class of the MythTV API:

```
public static Schedule valueOf(ProtocolVersion protoVersion, int dbVersion,
                  IRecorderNextProgramInfo nextProgramInfo,
                  IProgramRecordingType.Type recordingType) {
                              ...
            if (recordingType == null) {
                  recordingType = IProgramRecordingType.Type.SINGLE_RECORD;
            }
                              ...
      }
```

Null allowed constraints are also related to situations where the parameter is not required in some method execution, for example, in one of the `java.beans.PropertyDescriptor` constructors, the javadoc mentions that the `readMethodName` parameter may be null if the property is write-only and the `writeMethodName` parameter may be null if the property is read-only.

For the other two types of constraints, we have not found particular characterizations. The only observation is that *Type restriction* constraints are present in overloaded methods.

Constraints documentation (RQ3):

To answer the third research question, we do not use the APIs of JDK7 as these were selected for the detection validation (Section 3.4.1). In fact, since they are well known for the quality of their documentation. They are not representative of the existing APIs in terms of documentation.

Figure 3.6 present the results found for the 11 APIs that we have selected for the study. The first conclusion we can draw is that, for three constraint types, the non-documented usage constraints are by far more frequent than the documented one (more or less 80%). The only exception is the case of *Type restriction* constraints where the two thirds of the occurrences are documented. *Nullness allowed* constraints are the one with the lowest documentation rate (only 12%). This is really surprising since

API Name	Nullness not allowed	Nullness allowed	Type restriction	Range limitation
java.awt	788	243	6	135
java.beans	100	30	1	1
java.io	98	25	2	37
java.lang	369	43	15	40
java.math	83	0	2	9
java.net	84	20	18	23
java.nio	147	7	10	25
jaba.rmi	22	4	0	0
java.security	105	42	3	10
java.applet	1	0	0	0
java.sql	7	7	1	2
java.text	73	19	10	7
java.util	535	128	32	84

Table 3.VI: Number of Detected Constraints in JDK 7

the null value has, in general, specific semantics that must be understood by the client developers. For example, if you look at the class `DatabaseVersionRange` in MythTV API, the constructor is intended to construct a version-range object from two version objects given as parameters. Here, when the parameters are null, the range is set with default version values. This is not obvious to guess when it is not explicitly stated in the documentation.

```
public DatabaseVersionRange(DatabaseVersion from, DatabaseVersion to) {
            super(
                  from==null?DB_VERSION_1029:from,
                  to==null?DB_VERSION_LATEST:to
            );
      }
```

In the case where the constraints are documented, it is worth looking at the nature of the documentation, i.e., implicit vs explicit. We conjecture that explicit documentation is more efficient as it does not require from the client developers a cognitive effort to fully understand the constraint. Conversely, implicit documentation could lead to constraint misinterpretations.

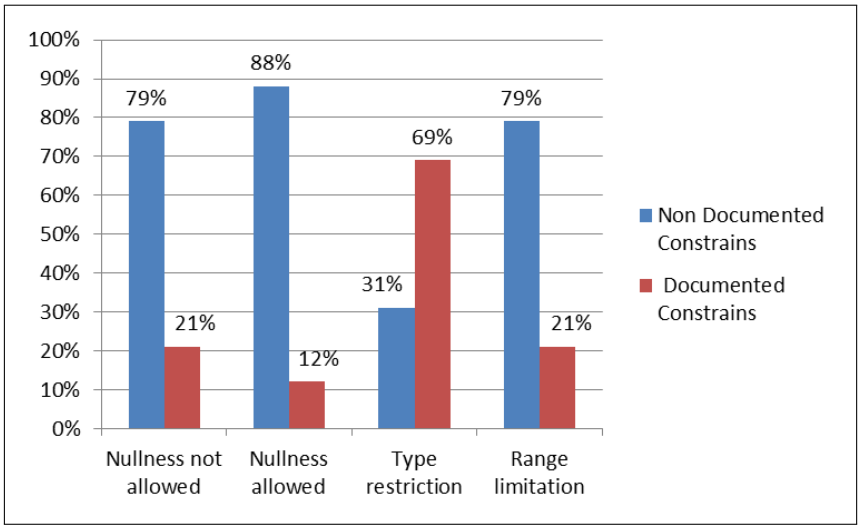


Figure 3.6: Documented vs non-documented constrains.

As shown in Figure 3.7, explicit documentation is clearly more frequent than implicit ones, especially for the *Type restriction* and *Range limitation* constraints (100%). For example, in the class `SatelliteInfo` of the Marine API, the javadoc explicitly mentions that if the parameter value is out of bounds 0..99 dB an `IllegalArgumentException` is thrown for method `setNoise(int)`, which sets the satellite signal noise ratio. The *Nullness allowed* constraint type is the one with the most implicit documentation (35%). For example, in the class `Pixmap` of MythTV API the method, `valueOf(..., Date lastModifiedDate)` creates a new pixmap object to read the preview image of a recording. We have detected that the parameter `lastModifiedDate` is null-allowed, and the javadoc implicitly mention it, saying that `lastModifiedDate` is the last modified date of the pixmap, if any. When we investigated this case, we understood that if the `lastModifiedDate` parameter is null, the current preview image should be downloaded, whereas when it is not null, the Pixmap object will be used to download a previously generated preview image. Something more explicit than '`if any`' should have been mentioned. *Nullness not allowed* constraint type has some implicit constraint documentation (9%). For instance, for the constructor *SentenceParser(String nmea)* in Marine API, the javadoc mentions that the parameter `nmea` is NMEA 0183 sentence and that an `IllegalArgumentException` is thrown if the specified sentence is invalid. The developer has to understand that NMEA 0183 is a combined electrical and data specification for communication between marine electronics that have its specific format and criteria

so the `nmea` parameter will have to meet those criteria and format which wouldn't be the case if the parameter is null.

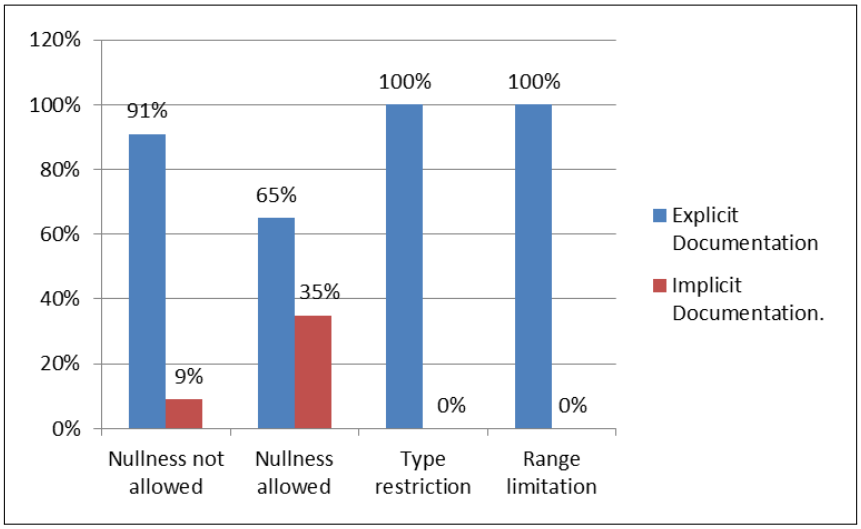


Figure 3.7: Explicit vs Implicit documentation.

Non-documentation scope (RQ4):

To better estimate the severity of the constraint documentation absence, we looked at the presence/absence of the global documentation of methods. Knowing if the constraint was not specifically documented in spite of the method javadoc presence, gives an indication about the risk of constraint violation, since with a method globally undocumented, the developer of a client application is more vigilant while in the other case, the confidence in the documentation increases the developer's vulnerability and therefore, increases the risk of constraint violation.

The results, shown in Figure 3.8 , indicate that the rate of specific absence is variable depending on the constraint type. The case of *Type restriction* constraints seems particularly worrying as for 100% of the non-documented constraints, the method javadoc exists. However, the sample is too small (only 5 non-documented constraints), to generalize the finding.

The case of *Nullness not allowed* constraints is more serious as half of the numerous constraints are not specifically documented. Let us consider the following example, in the class `QueryHint` of the Weather API. The example shows a portion of the source code and the Javadoc for the method `query(GeoLocation location, ...)`. Looking at the code, it is clear that the `location` parameter cannot be null. However, the developer of a client application might think that a null value

for this parameter corresponds to the use of a default location or activates a mechanism for geolocation based on IP address, for example. Although this case could be confusing, nothing is mentioned in the javadoc in relation to the *Nullness not allowed* constraints given below:

```
/**
 * Queries the data source about the weather in alocation from a specific date onward.
 * @param location the location
 * @return the weather in the given location in the given period of time
*/

public List<WeatherReport> query(GeoLocation location, Date beginTime,
                  QueryHint hint) {
            if (location == null) {
                  throw new IllegalArgumentException("Invalid coordinate: null.");
            }
                                                ...
      }
```

Another critical example is shown below. In the class `DefaultDetachedNode` of the Xtarget API, the javadoc of the method `addChild( XTDetachedNode child, int index )` does not report any constraint on the `child` parameter.

```
/**
       * Inserts the given child node at the given index.
       * @param child The node to insert
       * @param index The new index of the inserted node
                                    ...
       */
      public void addChild(XTDetachedNode child, int index) {
                                   ...
            testRecursion(child);
            children.add(index, child);
            child.setParent(this);
      }
```

While looking at the code, we found that `child` is used to invoke the method `setParent(...)`. This invocation will lead to a `NullPointerException` if the parameter is null. Two cases are possible, either the API developer is not aware of the constraint, or he assumes that the client developer will take care of testing his parameter before using the API method. In both cases, the constraint should have been mentioned in the javadoc.

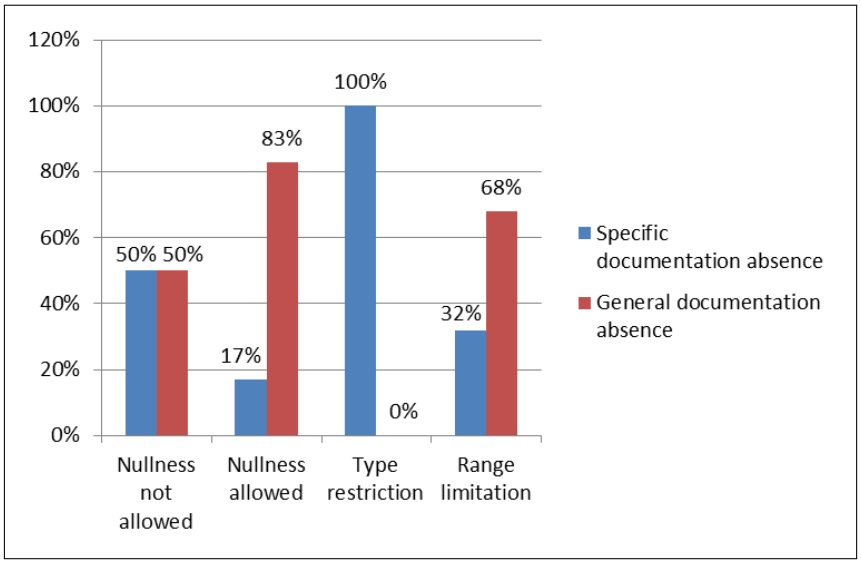


Figure 3.8: Specific vs General documentation absence.

3.5 Threats to validity

Our study is an observational one. Thus, we do not apply specific statistics as we are not comparing groups or identifying relationships between data. Consequently, we cannot generalize our results. Still, we used criteria-based sampling that increases our confidence about the results, which give an interesting portrait about the presence of usage constraints and their documentation. Moreover, our study raises questions that can be addressed by randomized studies, e.g., *are well-documented APIs more used than poorly documented ones?* or *is there a relationship between the quality of API documentation and the number of errors attributed to its usage?*

A possible threat to validity of our study concerns the recall. We automatically looked in the javadoc for keywords that describe the four constraint types, and we examined whether these constraints were detected by our algorithms. So our procedure will estimate recall based on a sample that contains only methods for which the constraint is mentioned in the Javadoc. To alleviate the impact of this threat, we applied this process only on the JDK APIs which are widely used and known for the quality of their documentation.

Another possible threat is the documentation inspection by the authors to answer questions **RQ3** and **RQ4**. We do not view this as an issue since we do not have particular result expectancies. The

only expectancies that we could have are related to the validity of our constraint detection strategies. In this case, we used independent subjects.

3.6 Conclusion

This chapter presents a set of analysis procedures to automatically detect certain types of API usage constrains based only on the library source code. The detection technique is based on intra-procedural analysis to determine when a particular type of constraint applies to a method. Our detection has a precision which reaches 100% for certain constraint and a recall rate between 78% and 94%. We also report the results of an observational study on the occurrence and the documentation of a set of API usage constraint types. Our study is based on both, looking at usage constraints inside the API source code and examining their documentation. The results of our study show that almost all the considered usage constraints are frequently present in API source code. We also discovered that many important usage constraints are not documented. Our finding is compelling evidence that the research effort dedicated to the automatic detection of usage constraint is justified.

Part II

Library usability across complementary API methods

Chapter 4

Mining Multi-level API Usage Patterns

Software developers need to cope with complexity of Application Programming Interfaces (APIs) of external libraries or frameworks. However, typical APIs provide several thousands of methods to their client programs, and such large APIs are difficult to learn and use. An API method is generally used within client programs along with other methods of the API of interest. Despite this, co-usage relationships between API methods are often not documented. In this chapter we present our approach for mining *Multi-Level API Usage Patterns* (MLUP) to exhibit the co-usage relationships between methods of the API of interest across interfering usage scenarios. We evaluated our technique through the usage of four APIs having up to 22 client programs per API. For all the studied APIs, our technique was able to detect usage patterns that are, almost all, highly consistent and highly cohesive across a considerable variability of client programs. This contribution was published at the *IEEE International Conference on Software Analysis, Evolution, and Reengineering* [63].

4.1 Introduction

Software developers might spend considerable effort to identify the subset of the class's methods that are necessary to implement their task at hand. To learn how to use API's methods, software developers usually search for code snippets by means of source code search engines, such as the Ohloh or OpenHUB [1] Code search engine. Yet, existing source code search engines return a huge number of code snippets that use the API's methods of interest. Therefore, we believe that identifying *usage patterns* for the API can be helpful for a better learning of common ways to use the API.

Recently, automated techniques to detect API usage properties have gained a considerable attention [11, 31, 79, 87]. These existing techniques are valuable to facilitate API understanding and usage. However, since the interference between different API usage scenarios could negatively impact the usage pattern mining process, existing techniques mainly focus on identifying API patterns in a specific usage scenario. Such techniques are proposed for recommending usage examples of the API that

1. Ohloh Code search engine renamed to OpenHUB: https://www.openhub.net/

are relevant to one task at hand (e.g. [87]), and/or for auto completion of source code in a specific context (e.g. [50]). In consequence, they fail short in identifying API usage patterns that are, at the same time, *context independent* and *reflect different interfering API usage scenarios*.

In this chapter , we present our technique for mining *multi-level API usage patterns*. We define a multi-level API usage pattern as a distribution on different usage cohesion levels of some API methods which are most frequently used together in client programs, in a consistent way, and regardless of the variability of client programs. Our technique is based on the analysis of the frequency and consistency of co-usage relations between the API methods within a variety of client programs of the API of interest. The rational behind this multi-level distribution of methods in a usage pattern is to identify the pattern's core, which represents the pattern's methods that are 'always' used together, and to reflect interfering usage scenarios of the pattern's core with the rest of the API methods. Hence, multi-level usage patterns add a new dimension which can be used to enhance the API documentation with co-usage relationships between methods of the API of interest.

This chapter is structured as follows. Section 4.2 motivates the usefulness of this work with two actual examples from HttpClient and Swing APIs. We explain our approach in Section 4.3 and present the case study used for evaluating it in Section 4.4. Section 4.5 presents and analyzes the results of our study. In Section 4.6 we discuss further our approach and the evaluation results, before concluding in Section 4.7.

4.2 Motivation examples

In this section we present two motivation examples to illustrate the following aspects of the addressed problem:

- The same usage pattern can be used in different contexts (we have to be context independent)
- Interfering API usage scenarios could be presented using multi-level usage patterns.

4.2.1 HttpClient Authentication

We consider the HttpClient API [2], which provides a feature-rich package implementing the client side of HTTP standards and recommendations. For this API, each time the `setProxy(String,int)` method of `HostConfiguration` class is invoked, our technique will inform the developer that he

will need to think about the method `setProxyCredentials(AuthScope,Credentials)` of the class `HttpState`. This hypothesis on the co-usage relationship between `setProxy` and `setProxyCredentials` methods is due to our analysis of various client programs of the Http-Client API, which found that both methods are always called together.

For instance, Figure 4.1 shows two code snippets of `HostConfiguration.setProxy` method [2]. In these code snippets, the `HttpState.setProxyCredentials` method is also invoked, after `setProxy`. These code snippets are from two different projects *MailBridege* [3] and *jUploader* [4]. In the first code snippet, the methods of interest are used in the constructor of `NetTools` class, while in the second code snippet the methods of interest are used in a factory of `HttpClient` objects (in `HttpClientFactory.getHttpClient()`). The purposes of these client classes/methods are completely different as their names indicate. It is worth noting that in these client methods, as in others, the methods of interest (`setProxy` and `setProxyCredentials`) are used with other methods of HttpClient, such as `HttpClient.getParams` and `HttpClient.setParameter`. However, deeply analyzing a variety of client methods to `setProxy` method, we observed that its co-usage relationship with `setProxyCredentials` method remains consistent, but its co-usage relationships with other methods of HttpClient API are not.

Actually, for a given HTTP client, once proxy settings are set (using `setProxy` method), client program should consider updating the HTTP client's state, which contains all HTTP attributes that may persist from request to request (such as authentication credentials). Therefore, the client program should set the proxy credentials for the given authentication realm using `setProxyCredentials` method.

Despite the very well elaborated documentation of the HttpClient API, we had to spend a noticeable time to figure-out the above-mentioned information in this documentation. Indeed, reading the Java doc of the aforementioned classes and methods, and reviewing the HttpClient's code sample [5], we did not find any documentation regarding the co-usage relation between the two meth-

2. With the query *"setProxy & language = Java"*, the Ohloh code search engine returned more than 5 million results (the query was on 15th June 2014). Verifying all the code snippets found is practically impossible. However, most of the results that we were able to verify were not relevant to the method of interest (`HostConfiguration.setProxy`), but for other methods having the same name (*"setProxy"*).

3. MailBridege is a HTTP tunnel for POP and SMTP access via a proxy.

4. jUploader is a cross platform and cross-site photo uploader.

5. svn.apache.org/viewvc/httpcomponents/oac.hc3x/trunk/src/examples/

ods `HostConfiguration.setProxy` and `HttpState.setProxyCredentials`. Finally, reading the Authentication [6] section in the HttpClient's user guide, we found that the first paragraph "Server Authentication" and the fourth paragraph "Proxy Authentication" provide an *implicit* documentation of that co-usage relationship.

```
public class NetTools {
   private HttpClient client = null;
   ...
    public NetTools() {
      ...
         client.getHostConfiguration().setProxy( proxyHost, proxyPort );
         ...
         client.getState().setProxyCredentials( AuthScope.ANY, defaultcreds );
         ...
```

```
public class HttpClientFactory {
   public static HttpClient getHttpClient() {
      HttpClient client = new HttpClient(new MultiThreadedHttpConnectionManager());
      ...
         HostConfiguration hc = client.getHostConfiguration();
         hc.setProxy(proxyHost, proxyPort);
            ...
         client.getState().setProxyCredentials(scope, creds);
      ...
```

Figure 4.1: Code snippets of "`setProxy`" found using Ohloh code search engine.

4.2.2 The Swing GroupLayout's interface

To better illustrate the benefit of identifying consistent API usage patterns, we consider the class `GroupLayout` [7] in the Swing API [4]. `GroupLayout` is a LayoutManager that hierarchically groups components in order to position them in a Container. It is typically used by every client programs of the Swing API for managing layouts in JPanels.

The (public) `GroupLayout`'s interface consists of 30 methods, and the class defines one constructor `GroupLayout(Container host)`. Reading the `GroupLayout`'s Java doc, the associated examples and the Java Swing tutorial, *"How to Use GroupLayout"* [8], we found that it is not trivial at all to identify the smallest subset of `GroupLayout`'s methods that are actually required for managing Swing GUI layouts. Indeed, the example provided with the `GroupLayout` documentation

6. hc.apache.org/httpclient-3.x/authentication.html
7. http://docs.oracle.com/javase/7/docs/api/javax/swing/GroupLayout.html
8. http://docs.oracle.com/javase/tutorial/uiswing/layout/group.html

uses, in addition to the `GroupLayout`'s constructor, 6 methods of the `GroupLayout`'s interface. In the Java Swing tutorial, the provided example on `GroupLayout` [9] uses additional method (7 methods in total).

Analyzing a wide variety of client methods to the `GroupLayout` class, we found that a relatively small subset of the `GroupLayout`'s methods are always (consistently) used together to build layouts of Swing GUIs. Those methods are: the `GroupLayout`'s constructor, then `setHorizontalGroup` and `setVerticalGroup` methods. Hence, these methods represent a usage pattern of Swing API, that we refer to as *core usage pattern* of `GroupLayout`. Moreover, our analysis revealed that building a layout (using GroupLayout) cannot be complete without using either `createParallelGroup`, `createSequentialGroup`, or both methods, for specifying the type of layout's horizontal and vertical groups. In other words, the set {`createParallelGroup`, `createSequentialGroup`} is, *partially* or *totally*, used with the core usage pattern of GroupLayout. We call these methods *peripheral usage pattern* of `GroupLayout`.

In summary, our technique for mining API usage patterns can inform Swing users that layouts can be built using 5 methods: the core usage pattern (3 methods) and peripheral usage pattern (2 methods) of `GroupLayout`. Hence, developers can focus only on 5 methods of the `GroupLayout`'s interface (instead of 30 methods) for building the layouts of their Swing GUIs. Then, as needed, developers can then modify properties of their GUI layouts using other methods in `GroupLayout`.

4.3 Approach

In this section, we detail our approach for detecting multi-level API usage patterns. Before detailing the used algorithm, we provide a deeper definition of multi-level usage patterns and their representation in our approach.

4.3.1 Multi-level API usage patterns

As outlined earlier, we define an API usage pattern (UP) as a group (i.e. cluster) of methods of the API of interest that are co-used together by the API client programs. An UP includes only methods

9. http://docs.oracle.com/javase/tutorial/uiswing/layout/groupExample.html

which are accessible from client programs (i.e. public methods of the API), and each UP represents an exclusive subset of the API's public methods.

Ideally, the co-usage relations between the UP's methods remain the same across *all possible* client methods of the UP. However, APIs are open applications, and it is unfeasible to analyze all their possible usage scenarios. Hence, we need a technique that can identify API usage patterns independently of the variability of features provided by the API's client programs and of the API's usage scenarios. Therefore, our technique for identifying API usage patterns should (1) capture out interference in co-usage relationships between the API's methods, and (2) isolate noises with respect to the degree of co-usage relationships in detected patterns.

To illustrate the representation of multi-level usage patterns, we use our example on `GroupLayout` class (Section 4.2.2). In this example, we showed that the core usage pattern of `Grouplayout` consists of 3 methods, which are used together by all analyzed client methods of `Grouplayout`. We also outlined that the `Grouplayout` core usage pattern was always used with other methods of Swing. The issue here is to associate this usage pattern to other methods that are closely related to it and can enhance its informativeness, and to isolate it from other methods that can degrade its co-usage relationships and consistency. To address this issue, we incrementally cluster this core usage pattern with closely related methods, from the closest to the least close ones, so that the resulted multi-level usage pattern of `Grouplayout` will include the core and peripheral usage patterns of `Grouplayout` as shown in Figure 4.2. This incremental clustering provides valuable information: all client methods of the `Grouplayout` class, which invoke the `Grouplayout(Container)` constructor, utilize methods in cluster *L0* for `Grouplayout` initialization; most of these client meth-

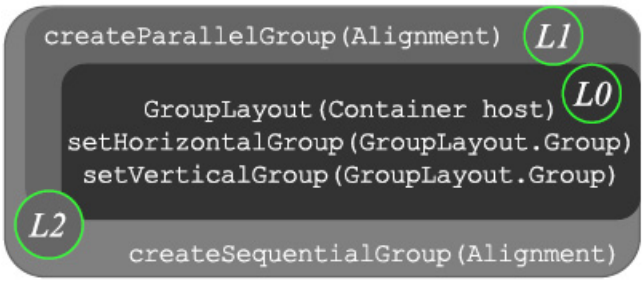


Figure 4.2: The cluster *L2* which represents the MLUP of class `GroupLayout`: *L0* represents the `GroupLayout`'s *core* usage pattern, then the cluster *L1*/*L2* includes *partially*/*totally* the `GroupLayout`'s *peripheral* usage pattern.

ods create parallel groups using the `createParallelGroup` method; and the other client methods create sequential groups using the `createSequentialGroup` method, or create both parallel and sequential groups.

4.3.2 Approach overview

Our approach takes as input the source code of the API to study and multiple client programs making use of this API. The output of our approach is a set of usage patterns as described in Section 4.3.1.

The API usage patterns detection approach proceeds as follows.

- First, the API's and client programs source code is statically analyzed to extract the references between the methods of the client programs and the public methods of the API. The static analysis is performed using the Eclipse Java Development Tools (JDT).
- Second, we compute usage vectors for the API public methods. Each public method in the API is characterized by a usage vector which encodes information about its client methods.
- Finally, we use cluster analysis to group the API methods that are most frequently co-used together by client methods.

4.3.3 Information encoding of API methods

In our approach, each API public method is represented by a usage vector that has constant length l, that is the number of all client methods which use the API methods. Figure 4.3 shows that the API of interest is used by 7 client methods. And, these client methods use 5 methods of the API. Note that the client methods could belong to different client programs of the considered API. For an API method, m, an entry of 1 (or 0) in the i^{th} position of its usage vector, denotes that m is used (or not used) by the i^{th} client method corresponding to this position. Hence, summing the entries in the method's usage vector represents the number of its client methods. For instance, in Figure 4.3, the usage vector of *API.m1* shows that this API method is used by 4 client methods, which are *C1.m1*, *C1.m2*, *C2.m3* and *C3.m1*. We can see that these client methods use also the API methods *API.m2, API.m3* and *API.m4*, but do not use *API.m5*.

	C1.m1	C1.m2	C2.m1	C2.m2	C2.m3	C3.m1	C3.m2
API.m1	1	1	0	0	1	1	0
API.m2	1	1	1	0	1	1	0
API.m3	0	1	1	0	1	0	0
API.m4	1	1	1	0	1	1	0
API.m5	0	0	0	1	0	0	1

Figure 4.3: The usage vector representation of five API methods with seven client methods.

4.3.4 Clustering algorithm

Our clustering is based on the algorithm DBSCAN [22]. DBSCAN is a density based algorithm, i.e. the clusters are formed by recognizing dense regions of points in the search space. The pseudocode of the DBSCAN algorithm is shown in Algorithm 1. The main idea behind DBSCAN is that each point to be clustered must have at least a minimum number of points in its neighborhood. This property of DBSCAN permits the clustering algorithm to leave out (not cluster) any point that is not located in a dense region of points in the search space. In other words, the algorithm clusters only relevant points and leaves out noisy points. This explains our choice of DBSCAN to detect API usage patterns. Indeed, not all public methods of the API are to be clustered because some are simply not co-used together nor with specific subsets of the API methods.

DBSCAN constructs clusters of API methods by grouping those that are close to each other (i.e. similar methods) and form a dense region. For this purpose, we define the Usage Similarity, $USim$ in Equation (4.1), between two API methods m_i and m_j, using the Jaccard similarity coefficient with regards to the client methods, Cl_mtd, of m_i and m_j. The rationale behind this is that two API methods are close to each other if the corresponding methods share a large subset of common client methods.

$$USim(m_i, m_j) = \frac{|Cl\_mtd(m_i) \cap Cl\_mtd(m_j)|}{|Cl\_mtd(m_i) \cup Cl\_mtd(m_j)|} \tag{4.1}$$

Where $Cl\_mtd(m)$ is the set of client methods of the API method m. For example, the $USim$ between the API methods *API.m1* and *API.m2* in Figure 4.3 is $\frac{4}{5}$,since these API methods have in total 5 client methods, and 4 of them are common for *API.m1* and *API.m2*. The distance between methods

m_i and m_j is then computed as defined in Equation (4.2) and the function `getNeighbors(P, epsilon)` in the pseudocode of the DBSCAN, uses the distance defined in Equation (4.2) to decide if a point belongs to the neighborhood of a given point.

$$Dist = 1 - USim(m_i, m_j). \tag{4.2}$$

DBSCAN depends upon two parameters to perform the clustering. The first parameter is the minimum number of methods in a cluster. In our context, we set it at two, so that a usage pattern must include at least two methods of the studied API. The second parameter, epsilon, is the maximum distance that within which two methods can be considered as neighbors to each other. In other words, epsilon value controls the minimal density that a clustered region can have. The shorter is the distance between the methods within a cluster the more dense is the cluster.

In practice, the choice of the epsilon value is not straightforward, since we do not know before hand which density and which threshold of similarity between methods will lead to good-quality usage patterns. Therefore, as it will be shown in the next section, we adapt DBSCAN algorithm to use different epsilon values for identifying multi-level usage patterns.

4.3.5 Incremental clustering

In DBSCAN, the value of the epsilon parameter influences greatly the resulting clusters. Indeed, in our approach, a value of 0 for epsilon, means that each cluster must contain only API methods that are completely similar (i.e. distance among methods belonging to the same cluster must be 0). Relaxing the epsilon parameter will relax the constraint on the requested density within clusters.

On the one hand, if we set epsilon at fixed small value, such as epsilon $= 0$, this will produce usage patterns that are very dense. Yet, resulted usage patterns will not capture out interference in co-usage relationships between the API's methods: a usage pattern will include only methods that are all always co-used together. On the other hand, fixing epsilon to relatively large value, such as epsilon $= 0.8$, DBSCAN will produce usage patters that include some noises. Thus, for a given usage pattern, it will not be easy to distinguish between dense subsets that capture out interference in co-usage relationships from subsets that include noises.

In our approach, we decided to build the clusters incrementally by relaxing the epsilon parameter, step by step to tolerate approximation of co-usage. Algorithm 2 shows the pseudo-code of our incre-

Algorithm 1 DBSCAN algorithm

```
DBSCAN(DataSet, epsilon, MinNbPts){
clusters <- {} ; noisyPoints <- {} //output of the algorithm
for each unvisited point P in DataSet do
    mark P as visited
    Neighborhood_P = getNeighbors(P, epsilon)
    if (Neighborhood_P.size) < MinNbPts then
        noisyPoints <- noisyPoints + {P}
    else
        currentCluster <- new cluster
        constructCluster(P, Neighborhood_P, currentCluster, epsilon, MinNbPts)
        clusters <- clusters + {currentCluster}
    end if
end for
}
constructCluster(P, Neighborhood_P, currentCluster, epsilon, MinNbPts){
currentCluster <- currentCluster + {P}
for each point Q in Neighborhood_P do
    if Q is not visited then
        mark Q as visited
        Neighborhood_Q <- getNeighbors(Q, epsilon)
        if Neighborhood_Q.size >= MinNbPts then
            Neighborhood_P <- Union(Neighborhood_P,Neighborhood_Q)
        end if
    end if
    if Q is not yet member of any cluster then
        currentCluster <- currentCluster + {Q}
    end if
end for
}
getNeighbors(P, epsilon){
for each point Q in DataSet do
    if DIST(P,Q) < epsilon
    then Q is neighbor of P.
end for
}
```

mental clustering. First, we construct a dataset containing all the API methods and cluster them using DBSCAN algorithm with epsilon value of 0. This results in clusters of API methods that are always used together, and multiple noisy API methods left out. At the end of this run, for each produced cluster we aggregate the usage vectors of its methods using the logical disjunction in one usage vector. Then, a new dataset is formed including the aggregated usage vectors and the usage vectors of noisy methods from the first run. This dataset is fed back to the DBSCAN algorithm for clustering, but with a slightly higher value of epsilon, that is epsilon = $0 + \delta$. This procedure is repeated in each step corresponding to an epsilon value of α. And, the clustering process is stopped when epsilon reaches a maximum value, β, given as parameter.

Algorithm 2 H-DBSCAN: Hierarchical DBSCAN algorithm

```
1: H-DBSCAN(DataSet, maxEpsilon, MinNbPts, epsilonStep){
2: epsinon <– 0
3: while epsilon < maxEpsilon do
4:    DBSCAN(DataSet, maxEpsilon, MinNbPts, epsilonStep)
5:    clusters <– DBSCAN.clusters
6:    noisyPoints <– DBSCAN.noisyPoints
7:    compositePoints <– constructPoints(clusters)
8:    Dataset <– noisyPoints + compositePoints
9:    epsilon <– epsilon + epsilonStep
10: end while
11: }
12: constructPoints(clusters){
13: for each C in clusters do
14:    compositePoints <– OR(all points of C)
15: end for
16: }
```

For example, Figure 4.4 shows that result of our incremental clustering for the API methods in Figure 4.3. In this example, the initial dataset contains 5 methods, *API.m1, ..., API.m5*, the epsilon parameter is incremented in each step by $\delta = 0.2$, and the epsilon maximum value was set to $\beta = 0.5$. Figure 4.4 shows that the algorithm will produce one multi-level usage pattern that contains in total 4 methods (*API.m1, ..., API.m4*), and *API.m5* is left out as a noisy method. The produced multi-level usage pattern involves 3 levels of density. The first level, which is the most dense one, is clustered at epsilon $= 0$, and it includes only *API.m2* and *API.m4*. These two methods represent the core

of the identified usage pattern since they are always co-used together (i.e. they have a perfect usage similarity). The second level, is clustered at epsilon = 0.2, and includes *API.m1* in addition to *API.m2* and *API.m4*. This method, *API.m1*, is included in this level since its distance from both *API.m2* and *API.m4* is smaller than 0.2. Finally, the third level, which is the less dense one, is clustered at epsilon = 0.4. This level includes, in addition to the methods of the first and second level, the method *API.m3*. At the end, the method *API.m5* is left out as a noisy method since its distance from any clustered method is larger than epsilon maximum value, which is 0.5. This multi-level usage pattern can be interpreted as that the pattern's core, which includes *API.m2* and *API.m4*, can have 2 interference usage scenarios: (1) the core methods can be co-used, most frequently, with *API.m1*, which shares 4/5 common client methods; (2) the core methods can be co-used with *API.m3*, which shares 3/5 common client methods.

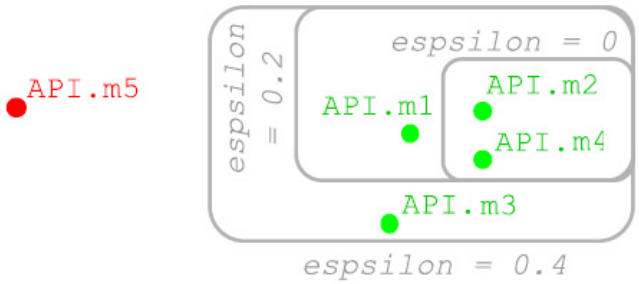


Figure 4.4: Resulting clusters of applying the incremental algorithm to API methods of Figure 4.3.

4.4 Evaluation

The objective of our study is to evaluate whether our technique can detect API usage patterns which are cohesive enough to provide valuable information that can help in learning and using APIs, and which are generalizable independently of the API usage contexts. We formulated the research questions of our study as follows:

- **RQ1:** to which extent the detected usage pattern are cohesive?
- **RQ2:** to which extent the detected usage pattern could be generalized to other "new" client programs, that are not considered in the mining process?

4.4.1 Systems studied

We evaluate our technique through the usage of four widely used APIs: HttpClient, Java Security, Swing and AWT (Table 4.I). To perform our study, we selected 22 client programs for the Swing and AWT APIs, and 12 client programs for the HttpClient and Java Security APIs (see Table 4.II and Table 4.III). We selected these four API and thier client programs for our validation because they are well studied in the related work.

API	Description
Java Security	Provides features to improve security of Java applications
HttpClient	Implements standards for accessing resources via HTTP
Swing	An API providing a graphical user interface (GUI) for Java programs
AWT	An API for providing a platform dependent graphical user interface (GUI) for a Java program

Table 4.I: Selected APIs for the case study

APIs	Client programs	Description
HttpClient & Java Security	Jakarta Cactus	A simple test framework for unit testing server-side java code
	Apache JMeter	A project that can be used as a load testing and measure performance tool
	Heritrix	A web crawler
	HtmlUnit	A GUI-Less browser for Java programs
	OpenLaszlo	An open source platform for the development and delivery of rich Internet applications
	Mule	A lightweight enterprise service bus (ESB) and integration framework
	RSSOwl	An aggregator for RSS and Atom News feeds.
	Apache Jackrabbit	Is an open source content repository for the Java platform.
	Axis2	A core engine for Web services.
	RESTEasy	A JBoss project that provides various frameworks to build RESTful Web Services
	WildFly	An application server
	WSO2 Carbon	An SOA middleware platform

Table 4.II: Client programs used in our case-study for HttpClient & Java Security

APIs	Client programs	Description
Swing & AWT	G4P (Processing GUI)	A library that provides a rich collection of 2D GUI controls
	Valkyrie RCP	A Spring port of the current Spring Rich Client codebase.
	GLIPS Graffiti editor	A cross-platform SVG graphics editor
	Mogwai Java Tools	Java 2D and 3D visual entity relationship design and modeling (ERD,SQL)
	Violet	An UML editor
	Mobile Atlas Creator	This application creates off-line raster maps
	Metawidget	A smart widget Building User Interfaces for domain objects
	Art-of-Illusion	A 3D modelling and rendering studio
	VASSAL	An engine for building and playing human-vs-human games
	Neuroph	A lightweight Java Neural Network Framework
	WoPeD	A Java-based graphical workflow process editor, simulator and analyzer
	jEdit	A mature programmer's text editor
	Spring-RCP	Provide a way to build highly-configurable, GUI-standards
	GanttProject core	An application for project management and scheduling
	Pert	The PERT plugin for GanttProject
	Htmlpdf	The html and pdf export plugin for GanttProject
	Msproject	MS-Project import/export plugin for GanttProject
	swingx	Contains extensions to the Swing GUI toolkit
	JHotDraw	A Java GUI framework for technical and structured Graphics
	RapidMiner	An integrated environment for machine learning and data mining
	Sweet Home 3D	An interior design application
	LaTeXDraw	Is a graphical drawing editor for LaTeX

Table 4.III: Client programs used in our case-study for Swing & AWT

4.4.2 Comparative evaluation

To address our research questions we opted for a comparative evaluation, we compared our technique for mining multi-level API usage patterns (MLUP) to the most similar approche MAPO as configured in [87]. To mine API usage patterns, MAPO clusters frequent API method-call sequences extracted from client programs, then use the BIDE [80] algorithm to mine closed sequential patterns from the preprocessed method-call sequences: i.e. to capture groups of API's method that are fre-

quently used together. This comparison allow us to better position our approach and characterize the obtained results. For MLUP we fixed the `maxEpsilon` value to 0.35. Obtained results are not computed for the core and the peripheral patterns separately, but rather for the usage pattern as a whole.

4.4.3 Metrics and experimental setup

The following describes the process of our experiments for a given API of the case study. The setup settings described in this section were applied for the two compared approaches MLUP and MAPO.

Pattern Co-Usage Relationships

To address our first research question **(RQ1)**, we need to evaluate whether detected usage patterns are cohesive enough to exhibit informative co-usage relationships between the API methods. To measure the usage cohesion of the detected patterns, we use the Pattern Usage Cohesion Metric (PUC), PUC was originally proposed and used in [55, 56] for assessing the usage cohesion of service interfaces. It evaluates the co-usage uniformity of an ensemble of entities, in our context, a group of API methods which forms a pattern.

PUC takes its value in the range [0..1]. The larger the value of PUC is, the better the usage cohesion is. PUC states that a usage pattern has an ideal usage cohesion (PUC = 1) if all the pattern's methods are actually always used together. The PUC for a given usage pattern p is defined as follows:

$$PUC(p) = \frac{\sum_{cm} ratio\_used\_mtds(p, cm)}{|CM(p)|} \tag{4.3}$$

Where *cm* denotes a client method of the pattern p; *ratio_used_mtds(p, c)* is the ratio of methods which belong to the usage pattern p and are used by the client method *cm*; while $CM(p)$ is the set of all client methods of the methods in p.

To answer our first research question **(RQ1)**, we apply our technique and MAPO for all the selected APIs of the case study, using the APIs client programs described in Table 4.II and Table 4.III. Then we compare the cohesion of the detected usage patterns through the two techniques, using the PUC metric.

Pattern Generalization

The detection of usage patterns for an API depends on the used set of API's client programs (training client programs). Hence, to address our second research question **RQ2**, we need to evaluate whether the detected API's usage patterns will have similar usage cohesion in the context of new client programs of the API (validation client programs). Our hypothesis is that: *detected usage patterns for an API are said "generalizable" if they remain characterized by a high usage cohesion degree in the contexts of various API client programs.* This is regardless of the natures and features of those client programs, and of whether those client programs were used or not for detecting the API's usage patterns. Such generalizable usage patterns can contribute to learn common ways of using the API of interest.

To evaluate the generalizability of detected patterns, we perform leave-one-out cross-validations for all the selected APIs of the case study, using the API's client programs described in Table 4.II and Table 4.III. Let N represents the number of used client programs for the considered API (e.g. N = 22 for Swing), we perform N runs of the two compared techniques (MLUP and MAPO) on the API. Each run uses N-1 client programs as training client programs for detecting usage patterns, and leaves away one of the API's client programs as validation client programs. The results are sorted in N runs, where each run has its associated usage patterns, and its corresponding training and validation client programs.

Then, we address our second question **(RQ2)** in two steps, as follows. In the first step, we evaluate the cohesion of the detected usage patterns (as measured by PUC) in the contexts of validation sets. However, in a given run, it is possible that some detected usage patterns involve only methods that are not used at all in the validation client programs. Therefore, to evaluate the generalizability of detected patterns in a run, we consider only patterns which contain at least one method that is actually used by the run's validation client programs. We call such patterns as the eligible patterns for the validation client programs. For an eligible pattern, if only a small subset of its methods are used by the validation client programs, while the other methods are not used, the pattern will have a low usage cohesion. As a consequence, it will be evaluated as "not generalizable". At the end of this step, we compare between the cohesion results obtained with MLUP and MAPO.

In the second step, for each run we evaluate the consistency of the detected usage patterns between the training client programs and the validation client programs. A pattern is said consistent if the

co-usage relationships between the pattern's methods in the context of the training client programs remain the same (or very similar) in the context of the validation client programs. We define the consistency of a usage pattern as:

$$Consistency(p) = 1 - |PUC_T(p) - PUC_V(p)| \tag{4.4}$$

Where PUC_T and PUC_V are the usage cohesion values of the pattern p in the training client programs context and validation client programs context, respectively. This metric takes its value in the range [0..1]. A value close to 1 indicates that the co-usage relationships between the pattern's methods remain the same between training client programs and the *new* validation client program, while a value close to 0 indicates a dissimilar behavior of the pattern between the two sets of clients. This metric allows us to see whether between changing contexts good patterns remain good ones and bad patterns remain bad ones.

4.5 Results analysis

Before addressing our research questions, we analyzed the dependencies between the client programs in Table 4.II 4.III and the selected APIs. We also applied the two compared techniques for detecting usage patterns of selected APIs. Table 4.IV summarizes the results of this phase, which shows that, for all studied APIs, our technique (MLUP) has been able to detect more usage patterns than MAPO. And, MLUP usage patterns are, overall, larger than MAPO ones. In the following, we investigate the results of our experiments explained in Section 4.4 to address our research questions **RQ1** and **RQ2**.

4.5.1 Patterns cohesion (RQ1)

To answer our research question RQ1, we analyze the average and the distribution of usage cohesion values for all detected usage patterns per studied API, using the two compared approaches. Table 4.V clearly show that, in average, MLUP outperforms MAPO for detecting cohesive usage patterns. Moreover the Wilcoxon rank sum test statically confirms these statements. The usage cohesion values obtained for MLUP reflect very strong co-usage relationships between the pattern's methods

		MLUP				MAPO			
			UP's size				UP's size		
API	**cov. mtd**	**UPs**	**Avg**	**Min**	**Max**	**UPs**	**Avg**	**Min**	**Max**
Java Security	125	12	2.8	2	4	4	2.0	2	2
HttpClient	343	20	2.7	2	4	12	2.5	2	4
Swing	1618	102	3.9	2	7	69	2.0	2	2
AWT	1019	75	4.3	2	9	50	3.1	2	6

Table 4.IV: Overview on the number of covered/analyzed methods and the number of detected usage patterns per API

for all studied APIs. Indeed, the average usage cohesion values for MLUP are between 90% and 96%, whereas MAPO's average values are between 55% and 71% .

API	**MLUP**	**MAPO**
Java Security	0.90	0.71
HttpClient	0.96	0.55
Swing	0.94	0.61
AWT	0.94	0.60

Table 4.V: Average Cohesion of identified API usage patterns, for MLUP and MAPO.

The distribution of usage cohesion values for all detected usage patterns in Figure 4.5 confirms the above-mentioned finding. Indeed, in the worst case for MLUP and the best case for MAPO, the case of Java Security API, the median usage cohesion with MLUP and MAPO are respectively around 90% and 71%. For the other studied APIs, the medians and lower quartiles remain larger than 90% for MLUP, whereas with MAPO the medians and upper quartiles persist under 70%.

In summary, the co-usage relationships between the methods of every usage pattern detected with MLUP are comparatively strong, where at least 70%, and upto 100%, of the pattern's methods are co-used together.

4.5.2 Patterns generalization (RQ2)

The cross-validation allow us to observe the generalizability of the detected patterns on two levels. First we inspect the co-usage relationships of detected patterns in the context of potential new

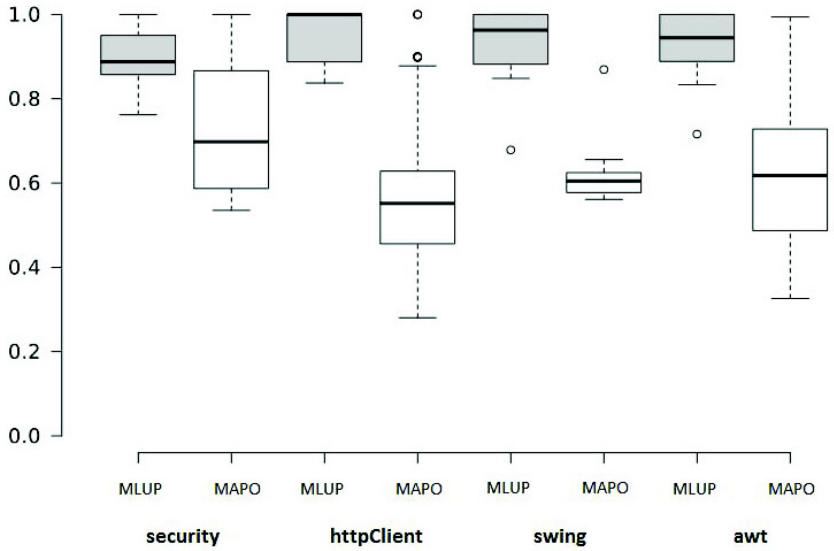


Figure 4.5: Cohesion values of identified API usage patterns, for MLUP (gray boxes) and MAPO (white boxes).

client programs. Then we characterize the usage cohesion deterioration between the training and validation client programs. In both levels, we first analyze the average value of corresponding measurements collected from all cross-validation runs (Table 4.VI and Table 4.VII), and then we analyze the distribution of collected values using median boxplots (Figure 4.6 and Figure 4.7).

Patterns Validation Cohesion

Table 4.VI summarizes, for both MLUP and MAPO, the usage cohesion of detected usage patterns in the contexts of validation client programs. For MLUP we notice that the average values remain high (around 85%), but with a slight degradation in the case of HttpClient API where the average usage cohesion is 79%. We also notice that the standard deviation values are very low. This reflects that, overall, the detected patterns using MLUP had always very good usage cohesion in the context of validation client programs. As for MAPO, the average usage cohesion values are significantly lower.

In Figure 4.6, although patterns cohesion values were degraded for both MLUP and MAPO, as compared to the patterns cohesion values in Figure 4.5, we observe that the median values for MLUP remain high. We also notice that the degradation of cohesion values is much more visible for MAPO. Precisely, for MLUP, in the worst case (HttpClient API) the median usage cohesion for detected

	MLUP			MAPO		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Java Security	0.85	0.12	1.00	0.67	0.12	1.00
HttpClient	0.79	0.16	1.00	0.45	0.08	0.58
Swing	0.85	0.09	1.00	0.58	0.03	0.64
AWT	0.84	0.06	0.95	0.39	0.09	0.67

Table 4.VI: Statistics on the Cohesion of identified API usage patterns for MLUP and MAPO , in validation clients

patterns is around 75%. For the other studied APIs, the medians and lower quartiles remain larger than 80%.

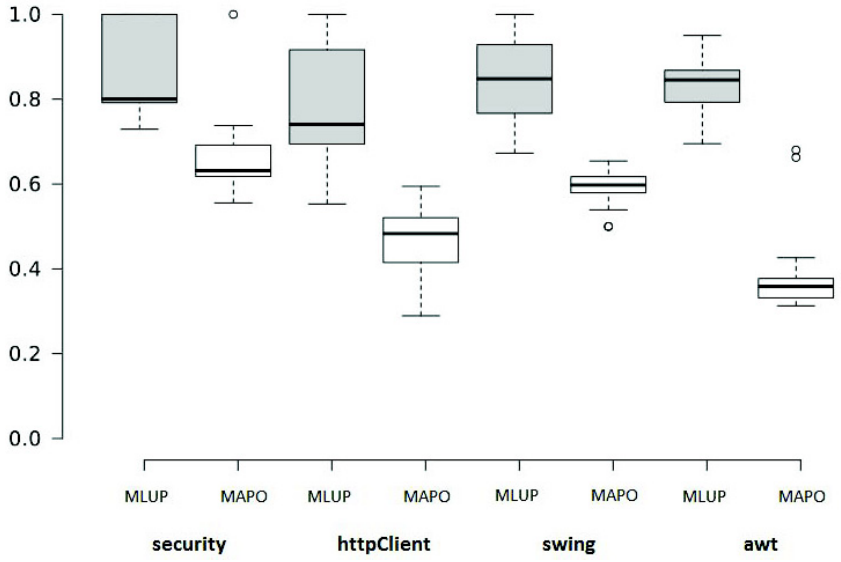


Figure 4.6: Cohesion values of identified API usage patterns, for MLUP and MAPO in the contexts of validation clients

In summary, compared to usage patterns that are detected by MAPO, usage patterns detected with our technique remain highly cohesive across various client programs of the API of interest. This shows that MLUP detected usage patterns retain their informative criteria independently of the API usage scenarios. Examples of such API usage patterns are the patterns that we discussed in Sections 4.2.1 and 4.2.2. These results show that API usage patterns detected with our technique can be used

to enhance the API documentation with co-usage relationships with high confidence without a need to consider all possible usage contexts (client programs) of the API of interest.

Patterns Consistency

As it can be seen in Table 4.VII, the results show that, using both MLUP and MAPO, the identified usage patterns for the four studied APIs are overall characterized with high consistency. Indeed, for both MLUP and MAPO, the average consistency values of detected patterns across multiple validation client programs are around 80%. The table also shows that maximum consistency values for all studied APIs are very close to the ideal value, which is 1, and the standard deviation values are very small. We consider as example the case of AWT API, where 22 leave-one-out cross validations have been made, using 22 client programs. In this case the results show that, for any AWT's detected pattern, on average 84% (for MLUP) and 77% (for MAPO) of co-usage relationships between pattern methods remain similar across 22 client programs of AWT.

	MLUP			MAPO		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Java Security	0.86	0.09	0.98	0.87	0.06	0.96
HttpClient	0.77	0.09	0.89	0.86	0.03	0.90
Swing	0.83	0.05	0.92	0.91	0.02	0.95
AWT	0.84	0.04	0.94	0.77	0.03	0.88

Table 4.VII: Statistics on the Consistency of identified API usage patterns for MLUP and MAPO

The boxplots in Figure 4.7 give more information on the consistency of detected usage patterns in each run of the compared techniques. Overall, the boxplots in Figure 4.7 show that, for detected patterns with MLUP, the median and lower quartile values are almost equal for 3 APIs (Java Security, Swing, and AWT), and they are around 85% and 80%, respectively. Hence, for these 3 APIs, we observe that almost all detected usage patterns (precisely 75% of detected usage patterns, according to the box lower quartile) are characterized with a high consistency, where the pattern's consistency value is greater than 80%. In comparison with MAPO results for these 3 APIs, we observe that the consistency of usage patterns detected with MLUP is comparable to that of MAPO, with a very small delta in favor of MAPO in some cases. However, recalling that the cohesion scores for MLUP were much higher than those for MAPO, in the context of both validation sets (Figure 4.6) and training

sets (Figure 4.5), we believe that this degradation of patterns consistency for our technique MLUP is actually acceptable.

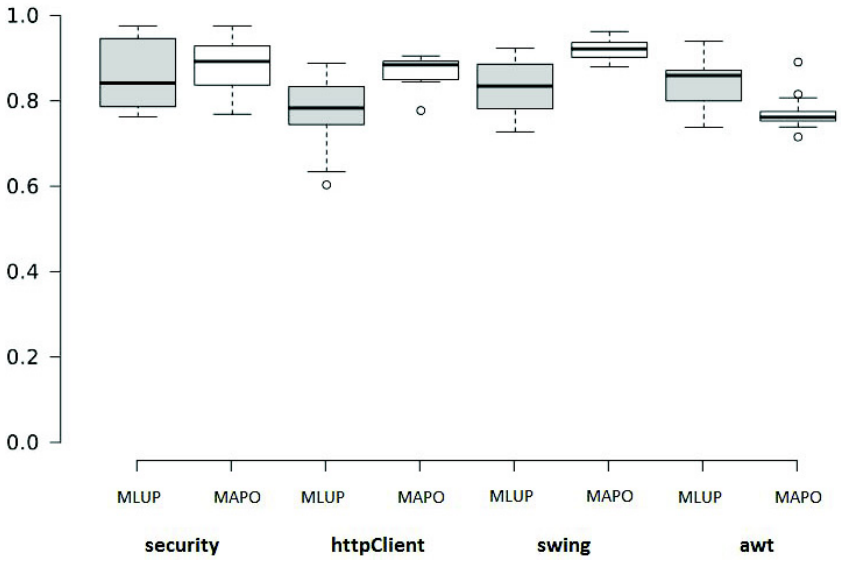


Figure 4.7: Consistency values of identified API usage patterns for MLUP and MAPO, across multiple validation clients.

In summary, the consistency metric reflects only the behavioral similarity of detected patterns with reference to the training and validation client programs, regardless of the cohesion quality. Both techniques are equivalent in terms of consistency, while MLUP remain much better in terms of cohesion quality of detected patterns.

4.6 Discussion

We applied our approach to four APIs and detected usage patterns that are informative and which could help to enhance the API's documentation. The evaluation of our approach took into account the generalization to other client programs of the same API and showed that the usage pattern of an API remain informative for other clients. Although, we selected four APIs of different domains, the approach may not be generalized to all APIs because they may be of different natures.

The API usage pattern detected by our technique have the valuable property of generalizability. An API pattern detected using a certain set of client programs of the API, as input to our approach, is very likely to be evenly detected using a different set of client programs of the API of interest. This aspect

of our approach also ensures that the detected patterns are independent of specific usage scenarios. The generalizability of the patterns suggests that they are used by different client programs involving various features. From this perspective, our approach is meant for enriching the API documentation and facilitating the development tasks when using new APIs.

One of the key contributions of our work is the adaptation of DBSCAN algorithm for mining API usage patterns. We have opted for this technique rather than for classic hierarchical clustering techniques since DBSCAN has a notion of noise, and is robust to outliers. Indeed, when mining API usage patterns, it is very probable to find many API methods that are not used jointly with specific set of methods. Such methods are utility methods that can be used with distinct subsets of the API methods. These methods are considered as noise and should not be clustered.

Finally, the application of our technique to detect multi-level API usage patterns requires the setting of thresholds that may impact its output. For instance, the epsilon parameter in the clustering algorithm controls the *co-usage strength* of the detected patterns. A small value leads to highly cohesive clusters which means that the detected patterns are very informative. Hence, decreasing the value of this parameter would result in an improvement in cohesion of detected patterns. However, in this case the number and the generalization of detected patterns could decrease. The highly cohesive detected patterns may not be shared by a large number of clients because they may not reflect different interfering API usage scenarios. Therefore, there is a tradeoff between the co-usage relationship of our detected patterns and their generalization to a large number of clients. Based on our initial experience we set the `maxEpsilon` value to 0.35 which reflects a similarity of 0.65 but this value still need more investigation.

4.7 Conclusion

This chapter presented a novel technique that identifies multi-level API usage patterns. We detect groups of API methods that are highly cohesive in terms of usage by client programs. We analyzed four APIs along with up to 22 client programs per API. Our approach detected API usage patterns that are generalizable for a wide variety of API client programs. The detected patterns are constructed in a hierarchical manner and are useful to enrich the API's documentation.

The efficacy of our approach relies on the coverage of the API methods, provided by the used client programs for mining the API usage patterns. This coverage has also to be redundant, i.e. an API method should be covered (used by) several client methods from different client programs. This limitation is shared by all existing work around making use of multiple API clients. This observation lead us to deal with non client based techniques, to infer API usage patterns. In Chapter 5 , we present our technique for mining API usage patterns only using the library source code.

Chapter 5

Mining API Usage Patterns only using the Library Source Code

As mentioned earlier learning to use existing or new software libraries is a difficult task for software developers, which would impede their productivity. Our approach presented in chapter 4, and much existing work has provided different techniques to mine API usage patterns from client programs in order to help developers on understanding and using existing libraries. However, considering only client programs to identify API usage patterns is a strong constraint. In fact, the client programs source code is not always available or the clients themselves do not exist yet for newly released APIs. In this chapter, we propose a technique for mining *Non Client-based Usage Patterns* (NCBUPminer). We detect unordered API usage patterns as distinct groups of API methods that are structurally and semantically related and thus may contribute together to the implementation of a particular functionality for potential client programs. We evaluated our technique through four APIs. The obtained results are comparable to those of client-based approaches in terms of usage-patterns' cohesion. This contribution was published at the *IEEE International Conference on Program Comprehension* [62].

5.1 Introduction

Identifying usage patterns for the API can help to better learn common ways to use the API, even if there exists several different ways to combine API elements (e.g. methods). In recent years,[60] much research effort has been dedicated to the identification of API usage patterns [11, 42, 46, 79, 81, 87]. These existing techniques are valuable to facilitate API understanding and usage. Despite the different aspects they try to cover, these techniques are all based on client programs' code. However, client programs' code is unfortunately not available for both newly released API libraries and APIs which are not widely used. Even if client programs are available, all the usage scenarios of the API of interest may not be covered by those clients. Indeed, from the coverage perspective, client-based identification of API usage patterns can be used only for a subset of the API of interest, that is the set of the API methods which are already used, multi-times, by different clients of the API. Hence,

such techniques need to access and analyze the code of different clients of the API, and guarantee that those clients cover the possible variety of the API usage contexts.

In this chapter, we propose a technique which does not rely on client programs' code to determine API usage patterns, namely Non Client Based Usage Patterns miner (NCBUPminer). Usage patterns describe references to a set of API methods that could co-occur with a certain frequency within a population of usage scenarios [60].

Our approach is based on the idea that API methods can be grouped together based on their mutual relationships. Our intuition is that related methods of the API may contribute together to the implementation of a domain functionality in client programs and thus may form an API usage pattern. We are interested, in particular, in two types of relationships, namely, structural and semantic (conceptual) dependencies [8, 65].

NCBUPminer is premised on the analysis of structural and semantic dependencies of API methods within the API code itself. More precisely, we start from two assumptions: (1) the API methods that change the state and manipulate the same object could be complementary in their contribution to a functionality, i.e. structural relationship assumption; and (2) the API methods can be related to the same domain functionality if they share similar vocabulary, i.e. semantic relationship assumption.

The proposed approach is not an alternative to client-based ones. It is rather a solution when client programs are not available, i.e. for newly released API libraries and non-widely used ones. Therefore, we do not expect that it performs better for usage pattern identification. Still, our goal is to obtain results that are close to those of client-based approaches. In this context, to assess how much confidence we could have on the non-client based patterns, we performed a comparative evaluation of our technique NCBUPminer against a client based one MLUP presented in Chapter 4.

This chapter is organized as follows. Section 5.2 motivates the usefulness of this work with two actual examples from AWT and JavaSecurity APIs. We explain our approach in Section 5.3 and present the setting and API used for evaluating it in Section 5.4. Section 5.5 presents and analyzes the results of our study. In Section 5.6 we discuss further our approach and the evaluation results. Finally,Section 5.7 concludes the chapter and describes our future research work.

5.2 Motivation examples

In this section, we present two motivation examples to illustrate how the structural and semantic dependencies could be interpreted as an indicator of co-usage relationships.

5.2.1 Java Security example

Java Security API[3] provides features to improve security of Java applications. The `KeyStore` class in the `java.security` API represents a storage facility for cryptographic keys and certificates. A `KeyStore` object manages different types of entries used to authenticate other parties such as `PrivateKeyEntry, SecretKeyEntry` and `TrustedCertificateEntry`. Before a `Keystore` object can be accessed, it must be loaded, then, it would be possible to read/write entries from/into it.

NCBUPminer detected a usage pattern reflecting this functionality through the following 3 API methods:

1. `load(LoadStoreParameter)` loads `Keystore` objects using the given parameter.
2. `getEntry(String, ProtectionParameter)` gets the keystore entry for the specified alias with the specified protection parameter.
3. `setEntry(Str,Entry,ProtectionParameter)` saves the keystore entry under a specified alias and with respect to the protection parameter.

These methods have strong structural dependencies and semantic similarity. Precisely, both the `getEntry ()` and the `setEntry()` methods need to start by looking up for the initialization state of the keystore through the *`initialized`* field in the `Keystore` class, which is set via the `load()` method. The aforementioned capabilities of the three methods are achieved via the *`KeyStoreSpi`* field, that defines the *Service Provider Interface* (SPI) for the `KeyStore` class. Moreover, `LoadStoreParameter` object, which must be passed to the `load()` method, is used to set the `ProtectionParameter` object, which is used to protect the keystore data. That object is then used as a parameter in both methods `getEntry()` and `setEntry()`.

5.2.2 AWT Example

AWT API [1] provides a reach toolkit for creating user interfaces and for painting graphics and images. We consider the class `Raster` in the AWT API. `Raster` is used to represent image data through a rectangular array of pixels. A `Raster` encapsulates a `DataBuffer` that stores the sample values of image bands datum and a `SampleModel` that describes the layout of the image data and how to locate a given sample value in a `DataBuffer`. Whenever a programmer need to manipulate image low-level information, he can directly manipulate samples and pixels in the `DataBuffer` of the `Raster` class. For that, a conventional way consists on starting with the creation of a compatible sample model which describes the data of the manipulated image throw the method `createCompatibleSampleModel(int, int)` in the class `SampleModel`. In the second step, the programmer need to use the factory method `createWritableRaster(SampleModel, Point)` in the `Raster` class. This provides pixel writing capabilities throw the created object `WritableRaster`, and the programmer could manipulate the image low-level information. The next step would be the use of `createChild` method of the `Raster` class, to copy either all bands or only a subset of the image bounding rectangle. Finally the `setDataElements` method of the `WritableRaster` class is used to set the data for a rectangle of pixels from the manipulated image. Our technique detected these 4 methods as a usage pattern, since they are manipulating the same objects. For instance, returned objects by some methods of the pattern are used as parameters for the other methods. We can also notice the presence of some similarity between the vocabularies of these methods. Figure 5.1 shows a code snippet from the `SimpleRenderedImage` class in `GanttProject`, where the patterns' methods are used to copy a rectangular region.

5.3 Approch

This section presents our approach for detecting non-client based usage patterns. We define a non-client based usage pattern for an API (an API usage pattern, UP, for short) as a subset of the API's methods that are structurally and semantically related. A usage pattern includes only public API methods that can be accessed from client programs, and each pattern represents an exclusive subset of the API's methods.

```
public WritableRaster copyData(WritableRaster dest) {
      Rectangle bounds;Raster tile;
      if (dest == null) {
            bounds = getBounds();
            Point p = new Point(minX, minY);
            /* A SampleModel to hold the entire image. */
            SampleModel sm = sampleModel.createCompatibleSampleModel(width,height);
            dest = Raster.createWritableRaster(sm, p);
                                    ...
      for (int j = startY; j <= endY; j++) {
            for (int i = startX; i <= endX; i++) {
                  tile = getTile(i, j);
                  Rectangle intersectRect = bounds.intersection(tile.getBounds());
                  Raster liveRaster = tile.createChild(intersectRect.x,
                              intersectRect.y, intersectRect.width,
                              intersectRect.height, intersectRect.x, intersectRect.y,
                              null);
                  dest.setDataElements(0, 0, liveRaster);
            }
      }
      return dest;
}
```

Figure 5.1: Code snippets of `Raster` from `GanttProject`

The rationale behind fetching the co-usage relationships in the API code itself is that public methods that change the state of or manipulate the same set of objects cooperate to accomplish certain domain functionality. However, even if some API methods are cooperating by manipulating the same object states, this cooperation could be for different domain purposes. Hence, those methods may not be co-used together, for one particular domain purpose, in client programs. As a matter of fact, the domain knowledge is encapsulated in the methods vocabulary [6, 27]. Therefore, our technique for identifying API usage patterns should isolate noises with respect to the degree of structural and semantics relationships in detected patterns.

Our approach takes as input the source code of the API to study and the output is a set of usage patterns as described earlier. The detection approach proceeds as follows.

– *Extracting API methods, references and terms.* First, the API source code is statically analyzed, and its public methods are retrieved. We collect, for each public API method, all the fields that are referenced either directly inside it or through the methods that it uses. We also collect terms composing the public method name and those composing its parameters and the local variable identifiers.

– *Encoding methods information.* Then, we compute states and terms vectors for the API public methods. Each public method in the API is characterized by: (1) a vector of states which encodes information about objects and states manipulated by the method, (2) a vector of the method's terms, that will be used by LSI technique [15, 66] to construct a semantic space representation for the API of interest. The static analysis is performed using the Eclipse Java Development Tools (JDT).
– *Clustering.* Finally, we use cluster analysis to group the API methods that are most structurally and semantically related.

5.3.1 Information encoding of API methods

In our approach, an API public method represents a point in the search space. As mentioned above, each point is represented by two vectors.

The first vector is the states' vector; it has constant length that is the number of all the manipulated classes and fields through the API public methods. Figure 5.2 shows an example where eight fields of four different classes are manipulated by the public methods of the API of interest. On that basis, the API methods will have a states' vector of length 12. For a given API method, an entry of 1 (or 0) in the i^{th} position of the states' vector, denotes that the i^{th} field is referenced (or not referenced) through the API method. If at least one field of a class is referenced, then the position of the corresponding class is also set to 1. This is done to, when computing the state similarity between two methods, give more importance to situations where both methods access fields from the same class than fields from different classes.

	C1	C2	C3	C4	C1.f1	C1.f2	C2.f1	C2.f2	C3.f1	C3.f2	C4.f1	C4.f2
API.m1	1	0	1	0	1	0	0	0	1	0	0	0
API.m2	1	0	1	0	0	1	0	0	0	1	0	0
API.m3	0	1	0	1	0	0	1	1	0	0	1	0
API.m4	0	1	0	1	0	0	1	0	0	0	1	1

Figure 5.2: The state vector representation of 4 API methods. In this API, 8 fields (`C1.f1 ... C4.f2`) of 4 different classes (`C1 ... C4`) are manipulated by the API methods.

The second vector is the terms' vector; it is computed from the lemmatized collected terms composing the public method name and its parameter and local variable identifiers. Similarly to states' vectors, the terms' vectors have constant length that is the number of all lemmatized collected terms composing the public methods in the API of interest. For a given API method, an entry of 1 (or 0) in the i^{th} position of the terms' vector, denotes that the i^{th} term appears (or does not appear) in the method vocabulary.

The terms' vectors of the API are used in a Latent Semantic Indexing process [66] to create a term-document matrix C. Each of the rows of C represents a term, and each of its columns represents a document (i.e. an API public method). C is an $M \times N$ matrix where M is the number of all terms collected from the API and N is the number of public methods of the API. To better infer semantic similarity relations based on terms co-occurrence, a Singular Value Decomposition (SVD) [66] is applied. From the term-document matrix C, the SVD constructs three matrices: U_k is the SVD term matrix; Σ_k is the singular values' matrix; and V_k^T is the SVD document matrix, where in our case $k = min(M,N)$. The SVD document matrix, V_k^T, yields a new representation for each document (API public method), that enables us to compute document-document similarity scores in the semantic space representation as the cosine between the term vectors of the API methods.

5.3.2 Similarity and distance metrics

As mentioned earlier, our approach constructs clusters of API methods by grouping those that are close to each other (i.e. similar methods). For this purpose, we define two similarity metrics, State Manipulation Similarity *StateSim* and Semantic Similarity *SemanticSim*.

The rationale behind the first metric, *StateSim*, as defined in Equation (5.1), is that two API methods m_i and m_j, are close to each other (i.e. similar) if they share a large subset of the classes and fields they are manipulating.

$$StateSim(m_i, m_j) = \frac{|accessed(m_i) \cap accessed(m_j)|}{|accessed(m_i) \cup accessed(m_j)|} \tag{5.1}$$

As for the semantic similarity metric, *SemanticSim*, we use the SVD document matrix, V_k^T, as mentioned above, to compute the cosine similarity between the API methods, as defined in Equa-

tion (5.2). This measure is used to determine how much relevant semantic information is shared among two API methods.

$$SemanticSim(m_i, m_j) = \frac{\overrightarrow{V_i} \times \overrightarrow{V_j}}{||\overrightarrow{V_i}|| \times ||\overrightarrow{V_j}||} \tag{5.2}$$

Where $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ are the j^{th} and i^{th} column corresponding to m_i and m_j in the document matrix V_k^T. The semantic similarity is then normalized between 0 and 1.

Using the similarity metrics, we compute the distance between two points p_i and p_j representing respectively two API methods m_i and m_j as opposite to the average similarity between p_i and p_j:

$$Dist(P_i, P_j) = 1 - \frac{StateSim(m_i, m_j) + SemanticSim(m_i, m_j)}{2} \tag{5.3}$$

5.3.3 Incremental clustering

For the non-client based usage pattern mining, we also used the DBSCAN based incremental clustering. The description of the algorithme and the pseudo-code are given in Section4.3.4 and Section 4.3.5.

We adapt DBSCAN algorithm for identifying usage patterns that may have variant densities with regard to the similarity between the pattern methods. That is to avoid limiting our patterns by one, unjustified, threshold of similarity, which may lead to less-good solutions.

The incremental clustering first construct a dataset containing all analyzed methods of the API of interest and cluster them using DBSCAN with an epsilon value of 0. The distance defined in Equation (5.3) is used to decide if a point belongs to the neighborhood of a given point. This results in clusters of the most similar API methods, and multiple noisy points left out, i.e. points that could not be clustered because there is no other point exactly similar in terms of state and semantic similarity. Then, we construct for each produced cluster of this run a representative point (new state and term vectors), by aggregating the vectors of its composing methods using the logical disjunction. The new term vector $\overrightarrow{T}$ is then mapped into its representation in the LSI semantic space by the following transformation:

$$\overrightarrow{T_k} = \Sigma_k^{-1} \times U_k^T \times \overrightarrow{T} \tag{5.4}$$

A new dataset is formed including the new vectors and the noisy points from the first run. This dataset is fed back to the DBSCAN algorithm for clustering, but with a slightly higher value of epsilon. In the second run, some clusters form the first run are identical, other clusters are incremented with other points, and new clusters could be formed. The incremental clustering process is repeated until epsilon reaches a maximum value given as parameter.

5.4 Evaluation

The objective of our study is to evaluate whether our technique can detect API usage patterns of good quality, that are comparable to those detected using several clients to the API of interest. We formulate the research questions of our study as follows:

- **RQ1:** *what is the quality of inferred usage patterns from the perspective of client programs?* and *what is the impact/contribution of the two used assumptions (structural and semantic)?*
- **RQ2:** *to which extent the inferred patterns are comparable to those detected by the client-based technique, MLUP?*

5.4.1 Comparative evaluation

To fairly evaluate the quality of our detected API usage patterns from the perspective of the API clients, and compare them to client-based detected usage patterns, we evaluate our technique using the data set presented in Chapter 4. To perform our study we used the four APIs presented in Table 4.I and their client programs Table 4.II, and Table 4.III.

To address our first research question, **RQ1**, we apply our technique on the four selected APIs and analyze the quality of detected patterns in the contexts of selected client programs, using the parameters and metrics that we detail in Section 5.4.2. To analyze the impact of different used assumptions (*state similarity* vs. *semantic similarity*) on our detected usage patterns, we use each heuristic alone for detecting patterns in the selected APIs, and compare the quality of the detected usage patterns. We also evaluate the quality of the patterns produced by considering the combination of the two heuristics.

To address our second research question, **RQ2**, we compare our technique for mining Non Client-based Usage Patterns (NCBUPminer) to our client-based approach (MLUP) Chapter 4. We compare

with MLUP since it is a client-based approach, and both approaches, MLUP and NCBUPminer, use the same clustering algorithm. For a fair comparison, we need to assess how the newly proposed heuristics in this work (structural and semantic similarity) will perform on the same set of API methods that can be clustered using MLUP, i.e. only the subset of methods used by the considered clients.

5.4.2 Experimental setup

This section describes the used metrics in our study, as well as the setting and process of the performed experiments in our study.

To assess the quality of the detected API usage patterns from the perspective of the API client programs, we need to evaluate whether these patterns are cohesive enough to exhibit informative co-usage relationships between the API methods. To measure the usage cohesion of the detected patterns, we use the Pattern Usage Cohesion Metric (PUC) Equation (4.3), that was previously adopted in Chapter 4 to evaluate the quality of API usage patterns detected with MLUP.

To assess the quality of inferred API usage patterns using NCBUPminer, and analyze the impact of different used heuristics (**RQ1**), we run NCBUPminer three times on each studied API. Each run uses all the API's public methods as the data set to be clustered. In the first run, we consider both heuristics, structural and semantic similarity between methods, whilst in the second and third runs, we dissociate the two heuristics and consider just structural similarity and semantic similarity, respectively. For each studied API, we collect the inferred API usage patterns for the three runs and analyze their quality w.r.t. their usage cohesion (i.e. PUC values) in the context of the API client programs in Table 4.II and Table 4.III. Note that some of the inferred patterns by our technique, NCBUPminer, may not be covered by the selected client programs –although we use a large variety of client programs for each studied API. Therefore, we collect and consider the usage cohesion only for eligible patterns. In our study, an API usage pattern is said eligible if at least one of its methods is used/covered by one of the analyzed API client programs. In addition to the usage cohesion property, we also compare the number and average size of inferred patterns in each run. Based on the comparison results, we decide which heuristic/s is/are the best for inferring API usage patterns using NCBUPminer.

To answer our second research question **(RQ2)**, for all the selected APIs we apply NCBUPminer (using both heuristics) and MLUP (using the APIs client programs described in Table 4.II and Table 4.III). For a given API, only the API's methods which are covered by the analyzed client pro-

grams of the API can be clustered by MLUP. Hence, for a fair comparison between NCBUPminer and MLUP, first we identify the set of methods that can be clustered by MLUP, then we use only this set of methods as an input for NCBUPminer and MLUP. Then, we compare the detected usage patterns through the two techniques, w.r.t. their usage cohesion in the context of the API's client programs, that MLUP used for identifying its usage patterns. We collect the PUC values of identified patterns by NCBUPminer and MLUP, and use the Wilcoxon rank sum test with a 95% confidence level to test whether a significant difference exists between the measurements for the two techniques. Moreover, we opt for Cliff's delta tests for estimating the effect-size delta between the cohesion scores of usage patterns identified by the two techniques.

As both techniques use the DBSCAN incremental clustering algorithm, we set the maximum epsilon value for both techniques, in all runs, to 0.35. For NCBUPminer, this value can be interpreted as follows: the value of maximal (structural/semantic) distance between two methods within an inferred pattern should be smaller than 0.35 (where 0 and 1 are, respectively, the smallest and largest values of distance). As for MLUP, this value means: for all detected patterns, the maximal non-uniformity in the co-usage of the pattern's methods should be smaller than 0.35.

5.5 Results analysis

In the following paragraphs, we report the results of our experiments.

5.5.1 Impact of used heuristics (RQ1)

To answer our research question RQ1, we analyze the quality (usage cohesion) of inferred API usage patterns, as well as their number and size, and we inspect the impact/contribution of different used heuristics.

Usage Cohesion

As illustrated in Figure 5.3, for all the studied APIs, when only the semantic similarity between API methods is used, the inferred usage patterns reflect less-good co-usage relationships between the pattern's methods. Indeed, the average usage cohesion values for this case are between 44% for HttpClient and 52% for Java Security. Nevertheless, when the structural similarity between the API methods is considered, the usage cohesion values of inferred usage patterns are improved. Using

this heuristic alone, on average 61% and up to 69% of the pattern's methods are uniformly co-used together. The obtained cohesion values using this heuristic alone are overall acceptable. However, they are still lower than the average usage cohesion value obtained for the Java Security API while combining both heuristics, which is 73%. In this last case, we notice that the average usage cohesion is high, but with a slight degradation in the case of Swing API, where the average usage cohesion is 64%. This is mainly due to the large size of Swing. Indeed, Swing declares 7226 public methods, as compared to Java Security API which declares 901 public methods. Despite the large number of declared public methods in Swing, and with regard to 22 different clients of this API, the results show that for an inferred usage pattern by NCBUPminer on average 64% of the pattern's methods are uniformly co-used together.

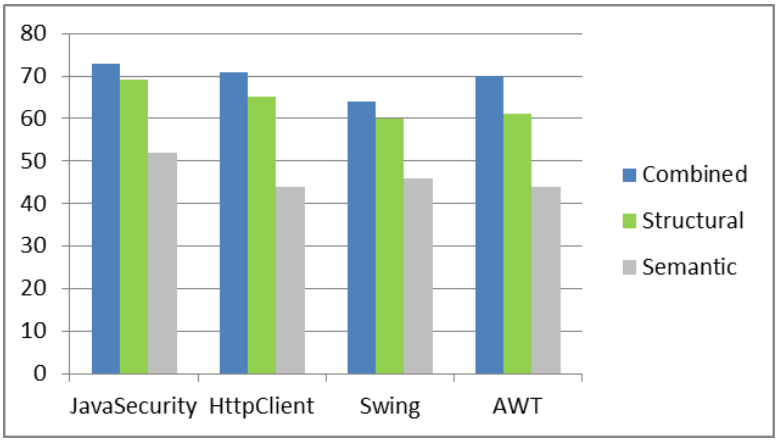


Figure 5.3: Average *cohesion* of inferred patterns using different heuristics

Number of Inferred Patterns

Figure 5.4 shows that an order relation can be observed between the numbers of patterns inferred through the three heuristics. For all studied APIs, the lowest number of inferred usage patterns was obtained while only the semantic heuristic is used. In second place, while combining the two heuristics the number of inferred patterns for each studied API was much greater than the previous case. Here, the number of inferred patterns for the Swing API reached a peak of 275 patterns.

Size of Inferred Patterns

The results in Figure 5.5 show that either when only the structural heuristic is used or both the structural and semantic heuristics are combined, the obtained sizes are almost equivalent with more or

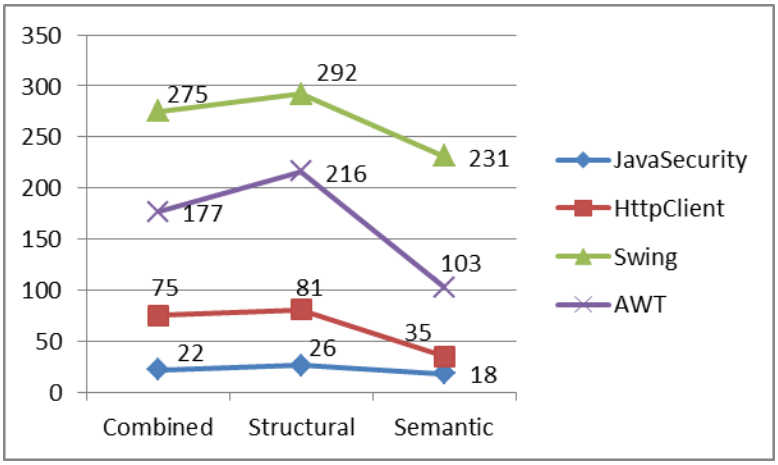


Figure 5.4: Average *number* of inferred patterns using different heuristics

less three methods per usage pattern. The largest usage patterns were inferred while only the semantic heuristic is used. Here, we observe that the average sizes of inferred clusters for HttpClient, Swing and AWT are around 15, 17 and 22, respectively. This can explain the small number of inferred patterns using this heuristic, as well as the low usage cohesion of the inferred patterns. When analyzing the inferred patterns using this heuristic, we found that they group methods having similar vocabulary but in most cases contribute to different functions in the domain. For some inferred clusters, we found that clustered methods belong in general to different classes implementing the same interface. Moreover, we were able to identify sub-clusters of methods that have strong structural similarity and contribute to one specific function in that vocabulary domain.

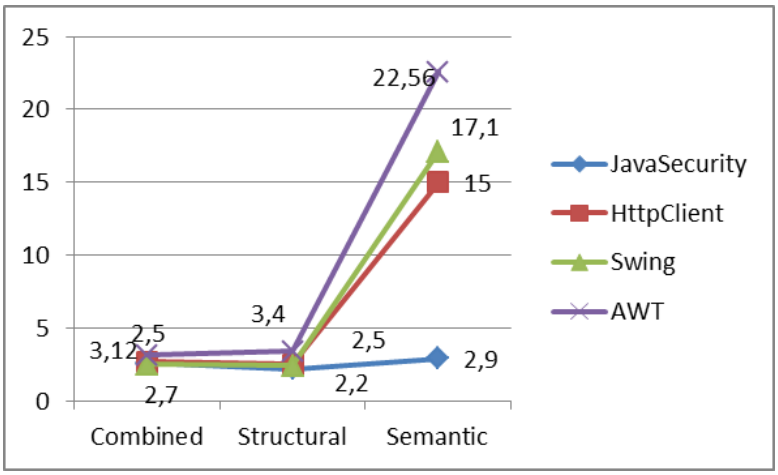


Figure 5.5: Average *size* of inferred patterns using different heuristics

As a summary, the results show that NCBUPminer can infer co-usage relationships between the API's methods with high precision. And, the structural similarity between the API methods has better contribution than the semantic similarity for inferring good API usage patterns. Still that, combining both heuristics NCBUPminer performs the best for inferring the co-usage relationships. In this case, with regard to all studied APIs and their analyzed client programs, on average 64% and upto 73% of methods in an inferred usage pattern are always uniformly co-used together, that is in the context of a large variety of API client programs. More specifically, using the structural similarity leads to cluster the API's methods which tightly collaborate together, but might contribute to different domain functions. The semantic heuristic enables our technique to identify clusters of methods that belong to the same vocabulary domain, and the structural heuristic enables it to identify sub-groups in those clusters, in which methods contribute together to one specific functionality in that domain.

Such good usage patterns detected by NCBUPminer, while combining both heuristics, are the examples that we outlined in Section 5.2. In addition to those patterns, NCBUPminer was able to infer other informative API usage patterns that were not covered/used by the API client programs considered in our study –although we used a large variety of client programs for each studied API. For instance, NCBUPminer inferred a usage pattern of the HttpClient API for validating certificate chain. The inferred pattern consists of the following four methods defined in the classes `PKIXParameters` (methods 1 and 2) and `CertPathValidator` (methods 3 and 4):

1. `PKIXParameters(java.security.KeyStore)`,
2. `setRevocationEnabled(boolean)`,
3. `getInstance(String)`, and
4. `validate(CertPath, CertPathParameters)`

Although these methods are not covered by the 12 client programs of HttpClient API that we used in our study, our analysis revealed that they form an informative usage pattern of HttpClient API. Indeed, using code search engines, we found that these methods are uniformly used together in several client programs of HttpClient API, as in the `Waterken` [1] project, for the purpose of validating certificate chains. For instance, the code snippet in Figure 5.6 shows how this pattern is used in a method of the URL `Handler` class in `Waterken`, for checking the trusted server. The description of this

1. Waterken is a platform for secure interoperation using a capability messaging protocol.

usage pattern is as follows. First the method `PKIXParameters()` is used to create an instance of `PKIXParameters`, that the `CertPathValidator` will use to validate certification chains. Then, the second method `setRevocationEnabled()` is used to enable or disabled the default revocation checking mechanism of the underlying PKIX (Public-Key Infrastructure X.509) service provider. In the last step, the method `getInstance()` is invoked to create a `CertPathValidator` object which implements the specified algorithm that can be used to validate certification paths by calling the fourth method `validate()`.

5.5.2 Comparative evaluation (RQ2)

To address our research question **RQ2**, we applied our technique (with the combined assumptions) and MLUP for detecting usage patterns of selected APIs.Then, we compared the results.

We, first, analyze the average usage cohesion values for all detected usage patterns per studied API obtained after using the two techniques as shown in Table 5.I. The results reveal that using both NCBUPminer and MLUP, the identified usage patterns for the four studied APIs are overall characterized with high usage cohesion values, which reflect very strong co-usage relationships between the methods of identified patterns. Indeed, the average usage cohesion values of identified patterns across multiple validation client programs are around 80% for NCBUPminer and 90% for MLUP. Although the results of NCBUPminer are, as expected, slightly lower than the one of MLUP, they are close and higher enough, considering that the client programs are not seen in the derivation process. Actually, using the Wilcoxon rank sum test, we found that the difference between the two compared approaches, with regard to the usage cohesion of detected patterns, is not statistically significant at $\alpha = 0.05$. To estimate the effect-size between the usage cohesion scores of NCBUPminer patterns and that of MLUP patterns, we performed Cliff's delta test at 95% confidence interval. The result is that the estimated delta value, which is $d = 0.10$ in favor of MLUP patterns, is not significant. More precisely, there is a probability of only 42.8% that a pattern randomly chosen from MLUP results will have a higher usage cohesion score than a randomly chosen pattern from NCBUPminer patterns, as compared to 32.3% probability in favor of NCBUPminer, and to 25% probability for the equality of scores. Hence, we state that the performance of NCBUPminer for inferring API usage patterns is comparable to that of MLUP.

```
public void checkServerTrusted(X509Certificate[] chain, String authType) throws
    CertificateException{
                // Validate the certificate chain.
                                        ...
                    PKIXParameter params = new PKIXParameters(Collections.singleton(ta));
                    params.setRevocationEnabled(false);
                    CertPathValidator.getInstance("PKIX").validate(path, params);
                                        ...
     }
```

Figure 5.6: Code snippet for validating certificate Chain, in method `checkServerTrusted` from class `Handler` in `Waterken`.

API	**NCBUPminer**	**MLUP**
Java Security	0.84	0.90
HttpClient	0.83	0.96
Swing	0.78	0.94
AWT	0.81	0.94

Table 5.I: Average Cohesion of identified API usage patterns, for NCBUPminer and MLUP

Table 5.II summarizes the number and size of API usage patterns identified by NCBUPminer and MLUP in this phase. The table shows that, for all studied APIs, MLUP has been able to identify more usage patterns than NCBUPminer. Moreover, MLUP usage patterns are, overall, slightly larger than NCBUPminer ones. We believe that this is mainly due to the capabilities of MLUP in resolving the interference in co-usage relationships between API methods. Indeed, thanks to the valuable information about the usage of API methods in the API client programs, MLUP can identify API's methods that do not belong to the same vocabulary domain, but their usage frequently interferes with each others. Still, except for Java Security, the number of patterns inferred by NCBUPminer is comparable to ones of MLUP. In HttpClient, Swing and AWT, the percentages are, respectively, 95% ($^{19}/_{20}$), 91% ($^{93}/_{102}$), and 77% ($^{58}/_{75}$).

In conclusion, the detected usage patterns with NCBUPminer retain their informative criteria independently of the API usage scenarios, and our technique can be used to enhance the API documentation with co-usage relationships with high confidence, when the client programs are not available or are not numerous enough to cover the multiple usage scenarios. Indeed, the majority of usage patterns detected by the client-based technique, MLUP, were inferred by our non-client based technique,

		NCBUPminer				MLUP			
			UP's size				UP's size		
API	**cov. mtd**	**UPs**	**Avg**	**Min**	**Max**	**UPs**	**Avg**	**Min**	**Max**
Java Security	125	4	2.5	2	3	12	2.8	2	4
HttpClient	343	19	2.4	2	4	20	2.7	2	4
Swing	1618	93	2.8	2	8	102	3.9	2	7
AWT	1019	58	2.5	2	6	75	4.3	2	9

Table 5.II: Overview on the number of covered/analyzed methods and the number of detected usage patterns per API

NCBUPminer. However in some cases the inferred usage patterns were slightly different from the exact real usage in client programmes.

5.6 Discussion

Software clustering techniques, which are based on structural or semantic coupling between software entities, have been widely used to support program comprehension, software remodularization, concept location or feature identification [8, 65]. However, this is the first time it is shown that combining both heuristics, structural and semantic similarity between methods, can lead to identifying new dimensions of dependencies between API methods, that are co-usage relationships within API client programs. We evaluated the impact of the two aforementioned heuristics on the quality of inferred API usage patterns. The results show that combining both heuristics performs the best for inferring the co-usage relationships between the API's methods within client programs.

The valuable contribution of NCBUPminer over existing techniques around identifying API usage patterns, is that it can be applied on "new" APIs, where client programs are not available. In fact, our technique can be used even before the release of the API for assisting API developers in comprehension tasks and in enriching the API documentation.

To evaluate the performance of our technique in inferring API usage patterns, we compared it to the our client-based approach MLUP. We compared with MLUP since both techniques use the same clustering algorithm. As both techniques use the DBSCAN incremental clustering , we used for both the same configuration used for MLUP , where the value of the parameter *maxEpsilon* is set to

0.35. However, as the comparative techniques use different heuristics, the parameter *maxEpsilon* has a completely different interpretation in each technique.

We applied our approach to four APIs and detected usage patterns that are informative and, which could help to enhance the API's documentation. We state that the performance of our technique for inferring API usage patterns is comparable to that of MLUP. However, as a threat to validity, this finding is related to the used client programs in our study. Still, our study used a fair set of validation client programs to evaluate the quality of inferred usage patterns by our technique.

5.7 Conclusion

In this chapter, we presente a technique that infers API usage patterns using only the API source code, independently of the availability of API client programs or unit tests to cover the API functionality. Our technique uses only structural and semantic similarity between the API methods to infer their co-usage relationships. We applied our technique on four APIs that differ in size, utility and usage domains. To evaluate the performance of our technique, we analyzed the quality of inferred API patterns in terms of usage cohesion using a large variety of API client programs. We found that our technique can infer API usage patterns with a precision that is not fare from the most-recent client-based technique for inferring API usage patterns. Furthermore, we evaluated the contribution of each used heuristic for inferring good usage patterns. We found that by combining both heuristics, structural and semantic similarity, our technique performs the best.

Despite these encouraging results, there is still room for improvement, . Indeed the library based technique infers patterns that could apply to any client program, However, some of the inferred patterns do not reflect real usage scenario. Conversely client based technique infers precise usage patterns with actual instances in the used client programs, yet inferred patterns are limited to the usage scenarios in the selected client. In chapter 6 we explore the combination of client-based and library-based usage pattern mining.

Chapter 6

A Cooperative Approach for Combining Client-based and Library-based API Usage Pattern Mining

As mentioned earlier much existing work has provided different techniques to mine API usage patterns based on client programs in order to help developers understanding and using existing libraries, such as our technique presented in Chapter 4. Other techniques propose to overcome the strong constraint of clients' dependency and infer API usage patterns only using the library source code, such as the technique presented in Chapter 5 . In this chapter, we propose a cooperative usage pattern mining technique (COUPminer) that combines client-based and library-based usage pattern mining. We evaluated our technique through four APIs and the obtained results show that the cooperative approach allows taking advantage at the same time from the precision of client-based technique and from the generalizability of library-based techniques. This contribution was published at the *IEEE International Conference on Program Comprehension* [61].

6.1 Introduction

The client-based techniques are known for their accuracy. However, client programs' code is unfortunately not available for both newly released libraries and APIs that are not widely used. And even if client programs are available, in an exhaustive scenario, those clients may not cover all the possible usage contexts of the API of interest. To address these issues, the solution is to overcome the strong constraint of client dependency by only considering the library code.

Deriving the usage patterns from library code is interesting as the inferred patterns could apply to any client program (generality property). However, this kind of derivation could be unsafe since it could lead to inferring usage patterns that do not reflect a real-world behavior. Conversely, client-based derivation infers precise usage patterns with actual instances in the used client programs (accuracy property). However, these patterns are specific to the considered clients. In this chapter, our objective is to take advantage of the properties of generality and accuracy by combining the client-based and library-based usage pattern mining. Our idea is similar to that of hybrid static-dynamic

analysis, which aims at finding a good compromise between the static-analysis soundness and the dynamic-analysis accuracy [21].

We specifically study which form of combination is better suited to achieve the best tradeoff between the generality and accuracy properties. As both techniques follow an iterative mining process to refine the patterns, the obvious combination is the sequential one, by applying a first technique (client or non client-based mining) to derive a set of patterns and then apply the second technique to refine these patterns. A more sophisticated combination is to interleave the different iterations of the two techniques (starting by one or the other technique) in a parallel and cooperative manner to solve a common goal. As both techniques use parameters, the values of the parameters can be varied to improve the accuracy and to explore the search space more efficiently.

We evaluated our hybrid technique using the four APIs of the dataset presented in Chapter 4: The results indicate that the cooperative approach is better than the sequential one.

This chapter is organized as follows. Section 6.2 discuss the motivation behind this contribution. We explain our approach and detail the cooperative mining in Section 6.3. The approach evaluation setting and the used APIs are described in Section 6.4. Section 6.5 presents and discuss the results of our study. Finally,Section 6.6 concludes the chapter.

6.2 Motivation examples

In this section, we illustrate through a motivating example how the combinations of client-based and library-based techniques can lead to a better approximation of real-world behavior.

Digital signatures are used for authentication and integrity assurance of digital data. `Signature` class of the Java Security API[3] is used to provide this functionality. A `Signature` object can be used either for signing data or for verifying digital signatures. For that, a client program usually starts by getting the `Signature` object that implements a specified standard signature algorithm through the method `getInstance(String algorithm)` in the `Signature` class. Then, one needs to initialize the `Signature` object either with a private key for signing (`initSign(PrivateKey)`), or with a public key for verification (`initVerify(PublicKey)`). In the next step, the method `update(byte[])` is used to update the data to be signed or verified. Finally, depending on the initialization type, either the method`sign()` is used for signing the updated data or the method

`verify(byte[]signature)` is used to verify the passed-in signature. Let us, now, show how such a pattern has been mined using the client-based and library-based techniques.

The client-based technique presented in Chapter 4 detected a usage pattern reflecting the integrity assurance functionality through the following 4 API methods `getInstance(String algorithm)`, `initSign(PrivateKey), sign()` and `update(byte[])`. The client-based technique missed the methods related to verifying digital signatures, since these methods were not sufficiently used by the considered client programs.

The library-based technique Chapter 5 grouped 12 API methods in a pattern reflecting the integrity assurance functionality. The pattern includes the following methods: `initVerify(PublicKey)`, `initSign(PrivateKey)`, `sign()`, and `verify(byte[] signature)` in `Signature` class; `generatePublic(..), generatePrivate(..)` and `getInstance(..)` in the class `KeyFactory`; `getInstance(..)`, `initialize(..)` and `generateKeyPair()` in `KeyPairGenerator` class; `getPrivate()` and `getPublic()` in `KeyPair` class. Although this pattern includes the digital signature verification functionality, some of the associated methods, e.g., generation of pairs of public and private keys, are not always relevant.

We will show later in this chapter that combining both client and non-client based techniques help us restricting the mined pattern to only the 8 necessary API methods 6 of them are in the `Signature` class : `getInstance(String algorithm), initSign(PrivateKey), sign(), initVerify(PublicKey)`,`verify(byte[]signature), update(byte[] )` and the other are in the `KeyFactory` class : `generatePublic(..)`,`generatePrivate(..)`.

6.3 Approch

This section introduces our cooperative approach for detecting API usage patterns. Before presenting the cooperative patterns mining, we provide a brief overview of our approach.

6.3.1 Overview

We define a usage pattern for an API as a subset of the API's methods that are either co-used together by the API client programs, or structurally and semantically related and thus may jointly contribute to the implementation of a domain functionality in client programs. A usage pattern includes

only public API methods that can be accessed from client programs, and each pattern represents an exclusive subset of the APIs methods.

APIs are open applications, and it is unfeasible to analyze all their possible usage scenarios (clients). For this reason, client-based mining emphasizes patterns' accuracy over their generality. Conversely, for the library-based mining, public methods that change the state of or manipulate the same set of objects while sharing the same semantic context, could cooperate to accomplish certain domain functionality. This allows to find general patterns independently from a given subset of clients. However, those methods, although similar, may not represent an actual usage pattern in practice. Hence, the library-based mining emphasizes patterns' generality over their accuracy. To benefit from both generality and accuracy properties, our approach combines both client-based and library-based techniques.

Our approach takes as input the source code of the API to study and multiple client programs making use of this API. The output is a set of usage patterns as described earlier. The detection approach follows three steps:

- *Extracting API methods, references, terms and usage.* First, the API source code is statically analyzed, and its public methods are extracted. Exactly as for the client-based and library-based techniques, We collect, for each public API method, all the fields that are referenced either directly inside it or through the methods that it uses. We also collect terms composing the public method name and its parameters as well as the local variable identifiers. In addition to fetching information inside the library, all the provided client programs are statically analyzed to extract the occurrences of API methods that are used in the clients' methods.
- *Encoding methods information.* Again as for the client-based and library-based techniques, after the information extraction, we derive states, terms and usage vectors for the API public methods. Each public method in the API is characterized by: (1) a state vector which encodes information about objects and states manipulated by the method, (2) a vector of the method's terms, that will be used by LSI technique [15, 66] to construct a semantic space representation for the API of interest, and (3) a vector of the method's usage which encodes information about its client methods.

– *Clustering.* Finally, we use cluster analysis to group the API methods that have the best trade-off between structural, semantics and co-usage relationships. We are also based on the adaptation of DBSCAN algorithm.

6.3.2 Cooperative patterns mining

This section details the combination process of the cooperative API usage-pattern mining. As mentioned before, the API public methods represent points in the search space. These points will be cooperatively clustered from two different perspectives. (i.e. client-based and library-based), which allows examining the search space more efficiently. The challenge is then, how to combine these two perspectives to achieve their respective benefits. For this question, we have two possibilities: (1) the two techniques are ran one after the other (sequential combination), or (2) they are intertwined (parallel combination).

6.3.2.1 Cooperative sequential combination

As mentioned earlier, the client-based and the library-based mining could be performed one after the other. The rationale behind the sequential combination is that the obtained usage patterns with one mining technique can be completed/enriched by the other technique. For the sequential combination, we will test the possibilities of starting with one or the other technique.

1) Start with the client-based mining

In this case, as shown in Figure 6.1, we start the combined process with the client-based mining, and the clustering is performed using the client based distance Equation (4.2). At the beginning, we construct a dataset containing all the API methods and cluster them using DBSCAN algorithm with an epsilon value close to 0. This results in clusters of API methods that are always used together. The other methods are considered as noisy points. After this run, for each produced cluster, we aggregate the usage vectors of its methods using the logical disjunction into one vector to form a new dataset. The new dataset includes the aggregated usage vectors and the usage vectors of noisy methods from the first run. This dataset is fed back to the clustering algorithm, but with a slightly higher value of epsilon. This procedure is repeated until epsilon reaches a maximum threshold value.

At this point, we switch to the library-based mining. For that, all the resulting clusters and noisy points are used to construct the input dataset. The clustering is then performed using the library-based distance Equation (5.3). Here again, we start by an epsilon value of 0. Then, we repeat the process with the new technique: (1) clustering, (2) dataset updating, and (3) epsilon slight increase, until we reach a second epsilon threshold. To update the dataset, the state and term vectors of the already-clustered methods are respectively aggregated using the logical disjunction. The new term vector $\overrightarrow{T}$ is then mapped into its representation in the LSI semantic space by the following transformation:

$$\overrightarrow{T_k} = \Sigma_k^{-1} \times U_k^T \times \overrightarrow{T} \tag{6.1}$$

2) Start with the library-based mining

For this second sequential combination possibility, we start with the library-based mining, and then the resulting clusters and noisy points are used to construct the input dataset for the client-based mining, which is repeated until epsilon reaches a maximum value given as a parameter.

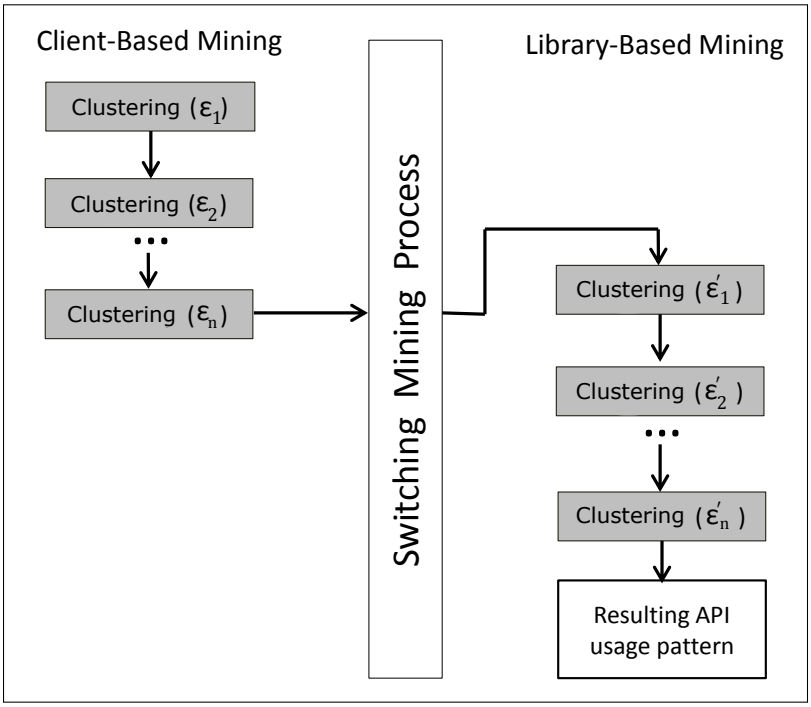


Figure 6.1: Sequential combination start with the client-based mining

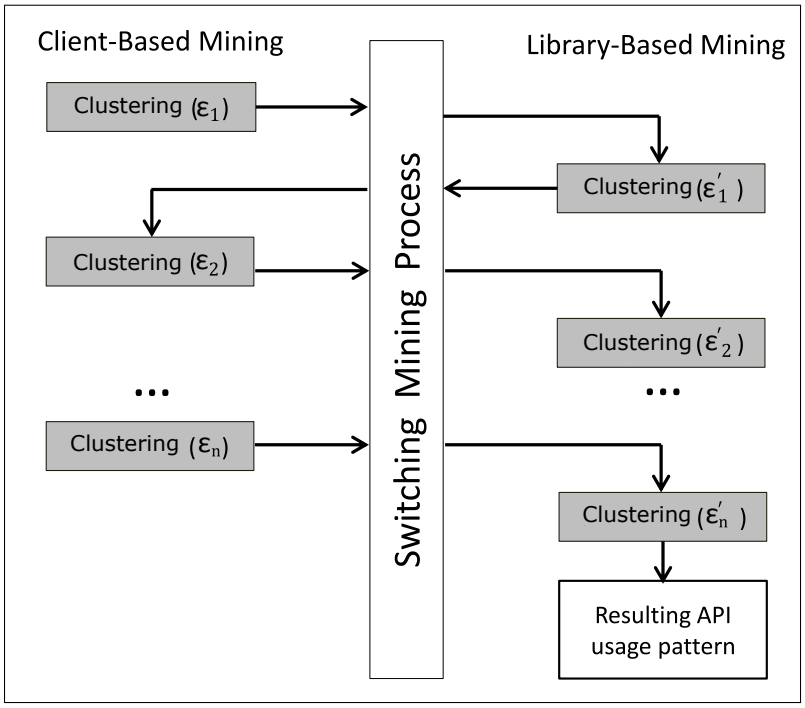


Figure 6.2: Parallel combination start with the client-based mining

6.3.2.2 Cooperative parallel combination

For the parallel combination, the client-based and the library-based mining are intertwined. The two mining process will, step by step, incrementally evolve in parallel exchanging the clustered data. In each step, either an iteration of the client-based mining or an iteration of the library-based mining is performed. Two epsilon values are maintained in parallel one for each technique: epsilonC and epsilonL. In the parallel combination, we can choose to start with the client-based mining or the library-based mining. The rationale behind the parallel combination is to go along the same lines of incremental clustering allowing the data to be mutually influenced from the beginning of the pattern construction.

1) Start with the client-based mining

In this case, as shown in Figure 6.2, we start the parallel combination with the client-based mining, and the clustering is performed using the client based distance Equation (4.2). At the beginning, all the API methods are clustered using DBSCAN algorithm with epsilonC value close to 0. At the end of this run, the mining process is switched after updating the dataset according to the resulting clusters

and the noisy methods. As for the next iteration we use the library-based clustering, the dataset is updated by considering the state and term vectors, and the clustering is performed using the library-based distance Equation (5.3) with an epsilonL value of 0. At the end of this iteration, we switch back to the client-based clustering. The dataset is updated by the aggregation of the usage vectors, and epsilonC value is slightly increased. This cooperative process is repeated until epsilonC or epsilonL reach a value given as a parameter.

2) Start with the library-based mining

In this case, we reproduce the previous mining processus with the difference that we start the parallel combination with the library-based mining.

6.4 Evaluation

The objective of the evaluation study is to assess whether a cooperative approach can lead to a better inference of API usage patterns compared to individual client-based and library-based techniques, in terms of the tradeoff between the generality and accuracy properties of the mined patterns. The evaluation is performed in two steps. We first determine which of the four combination options leads to the best tradeoff. Then, we compare the cooperative approach, i.e. the best combination option, with individual techniques. We formulate the research questions of our study as follows:

- **RQ1:** *what is the best strategy to combine the client-based and the library-based mining, and what is the impact of the different strategies on the quality of the inferred API usage patterns from the perspective of client programs?*
- **RQ2:** *to what extent does the cooperative parallel approach performs better than the individual techniques?*

6.4.1 Comparative evaluation

In this chapter, we study the benefits of the cooperative approach (COUPminer) with respect to individual techniques. To be consistent with the previous comparison studies, the evaluation in this chapter is performed using the data set presented in Chapter 4. on the same four well-known APIs presented in Table 4.I and their client programs Table 4.II, and Table 4.III.

To address the first research question, **RQ1**, we apply each combination strategy for detecting patterns in the considered APIs. Then, we compare the quality of detected usage patterns in the contexts of the selected client programs, using the parameters and metrics that we detail in Section 6.4.2. We use the Wilcoxon rank sum test with a 95% confidence level to assess whether a significant difference exists between the measurements of the four combination strategies.

To address our second research question, **RQ2**, we compare the best cooperative strategy (COUPminer) with the client-based (MLUPminer) and library-based (NCBUPminer) techniques.

6.4.2 Experimental setup

This section describes the used metrics in our study, as well as the setting and process of the performed experiments.

To assess the quality of the detected API usage patterns from the perspective of the API client programs, we need to evaluate whether these patterns are cohesive enough to exhibit informative co-usage relationships between the API methods. To measure the usage cohesion of the detected patterns, we use the Pattern Usage Cohesion Metric (PUC)Equation (4.3), that was previously adopted in Chapter 4 and Chapter 5 to evaluate the quality of API usage patterns detected with MLUPminer and NCBUPminer.

6.4.2.1 Impact of used combination strategies (RQ1)

To assess the quality of inferred patterns using COUPminer, and analyze the impact of different combination heuristics, we run the cooperative technique four times on each studied API. Each run uses all the API's public methods as the dataset to be clustered. In the first and second run, we consider the sequential combination heuristic, whilst in the third and fourth runs, we consider the intertwined parallel combination heuristic. Each heuristic was run twice, once starting with the client-based mining, and once starting with the library-based mining. For each studied API, we collect the inferred API usage patterns for the four runs and analyze their quality w.r.t. their usage cohesion (i.e. PUC values) in the context of the API client programs selected for the study. Note that some of the patterns inferred by our technique, COUPminer, may not be covered by the selected client programs. Therefore, we collect and consider the usage cohesion only for eligible patterns. In our study, an API

usage pattern is said eligible if at least one of its methods is used/covered by one of the analyzed API client programs. Based on the comparison results, we decide which strategy is the best for inferring API usage patterns.

6.4.2.2 Comparative evaluation (RQ2)

We address our second research question in two steps, as follows. In the first step, for all the selected APIs of the case study, we apply NCBUPminer, MLUPminer and the best cooperative strategy COUPminer. Then, we compare the number and average size of inferred patterns for each technique. In the second step, we perform leave-one-out cross-validations to assess whether COUPminer achieves a good tradeoff between library-based and client-based techniques.

The cross-validations was performed using the API's client programs selected for the study. Let N represents the number of used client programs for the considered API (e.g. N = 22 for Swing), we perform N runs of the three compared techniques (NCBUPminer, MLUPminer and COUPminer) on the API. Each run uses N-1 client programs as training client programs for detecting usage patterns, and leaves away one different API's client program for validation. Obviously for NCBUPminer both the training clients and the validation client only served to assess the patterns' quality, since NCBUPminer is a non client-based technique.

To ensure that experimental observations are due to the proposed heuristic and not to the parameter choice, we reused the same configuration for the three techniques, e.g. the maximum value of the epsilon parameter is set to 0.35.

Patterns Validation Cohesion We used the results of the cross-validations to evaluate the cohesion of the detected usage patterns (as measured by PUC) in the contexts of validation sets. However, in a given run, it is possible that some detected usage patterns involve only methods that are not used at all in the validation client programs. Therefore, we consider only patterns that contain at least one method that is actually used by the run's validation client programs. For such a pattern, if only few of its methods are used by the validation client programs, whereas the other methods are not, the pattern will have a low usage cohesion. At the end of this step, we compare the cohesion results obtained with NCBUPminer, MLUPminer and COUPminer.

Patterns Consistency For each run of the cross-validations, we evaluate the consistency of the detected usage patterns between the training client programs and the validation client program. A pattern is said consistent if the co-usage relationships between the pattern's methods in the context of the training client programs remain the same (or very similar) in the context of the validation client programs. We define the consistency of a usage pattern as:

$$Consistency(p) = 1 - |PUC_T(p) - PUC_V(p)| \tag{6.2}$$

Where PUC_T and PUC_V are the usage cohesion values of the pattern p in the training client programs context and validation client programs context, respectively. This metric takes its value in the range [0..1]. A value close to 1 indicates that the co-usage relationships between the pattern's methods remain the same between training client programs and the *new* validation client program, while a value close to 0 indicates a dissimilar behavior of the pattern between the two sets of clients. This metric allows us to see whether, between changing contexts, good patterns remain good ones and bad patterns remain bad ones.

6.5 Results and discussion

In the following sections, we discuss the results of our experiments.

6.5.1 Impact of used combination strategies (RQ1)

To answer RQ1, we inspect the impact/contribution of both the sequential combination heuristic and the intertwined parallel combination heuristic. For each of these two heuristics, we first need to compare the possible strategies, i.e. starting with the client-based mining and starting with the library-based mining.

As illustrated in Table 6.I, for the sequential combination strategy, when the client based mining is considered first, the inferred usage patterns exhibit less-good co-usage relationships between the pattern's methods. The average usage cohesion values for this case are between 73% for Swing and 81% for Java Security. Nevertheless, when the library-based mining is considered first, the usage cohesion values of inferred patterns are improved for three out of four libraries (between 75% for

Swing and 85% for Java Security). The only slight degradation was observed for HttpClient from 78% to 77%.

For the intertwined parallel combinations, we observed the same trend. When the process starts with the client-based mining, all the average usage cohesion values were higher than 80%, except for Swing with 77%. This low value can be attributable to the large size of Swing. Indeed, Swing declares 7226 public methods, as compared to Java Security API, which declares only 901 public methods. The values for Swing were also the lowest for the sequential combination strategies. The best cohesion values are obtained for the cooperative strategy with library-based mining is used as the initial technique. In this case, the usage cohesion values improve to reach, in the worst case, 82% for HttpClient and in the best case 88% for Java Security.

For both the sequential and the intertwined parallel combinations, starting library-based mining was the best option. The mined patterns reveal that with this option, it is possible to retrieve the majority of potential patterns, then, the client-based mining allows refining these patterns. However, the refinement is not performed at the same scale for the two combination heuristics. It is more efficient at a low scale, when the refinement is performed at the early stage of the pattern inference, which is the case for the intertwined mining.

Indeed, Table 6.I clearly show that, in average, the intertwined parallel combination outperforms the sequential combination for detecting cohesive usage patterns. These results are statistically significant according to the Wilcoxon rank sum test. The usage cohesion values obtained for intertwined parallel combination reflect strong co-usage relationships between the pattern's methods for all the studied APIs.

	SequentialCombination		**ParallelCombination**	
API	**Start with clientBased**	**Start with libraryBased**	**Start with clientBased**	**Start with libraryBased**
Security	0.81	0.85	0.85	0.88
HttpClient	0.78	0.77	0.80	0.82
Swing	0.73	0.75	0.77	0.83
AWT	0.77	0.79	0.82	0.85

Table 6.I: Average Cohesion of identified API usage patterns, for NCBUPminer, MLUPminer and COUPminer.

6.5.2 Comparative evaluation (RQ2)

To answer our research question RQ2, we applied NCBUPminer, MLUPminer and COUPminer, then we analyze the number and size of inferred API usage patterns, as well as their consistency and validation cohesion.

6.5.2.1 Number and size of inferred patterns

Figure 6.3 shows that a trend can be observed between the numbers of patterns inferred through the three mining techniques. For all studied APIs, the lowest number of inferred patterns was obtained while only the client-based heuristic (MLUPminer) is used. Whereas the highest number of patterns was inferred when only the library-based heuristic (NCBUPminer) is used. We notice that a compromise is achieved while combining the library-based and client-based heuristics (COUPminer). For instance, in the case of AWT the number of pattern was adjusted to 143 while it was 75 with MLUPminer and 177 with NCBUPminer. The results in Figure 6.4 show that a similar trend can be observed for the patterns' size. The largest usage patterns were inferred when NCBUPminer is used, and the smallest patterns were inferred when MLUPminer is used. Once again, we notice that COUPminer finds a tradeoff in terms of patterns' size.

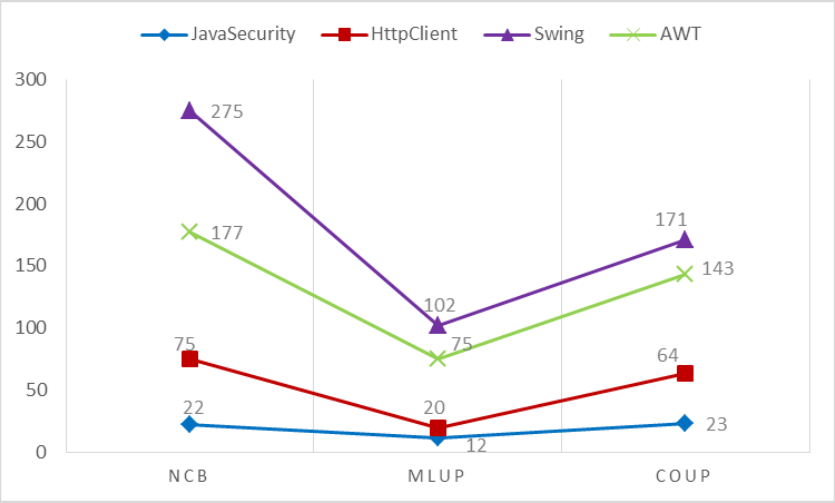


Figure 6.3: *Number* of inferred patterns using different techniques

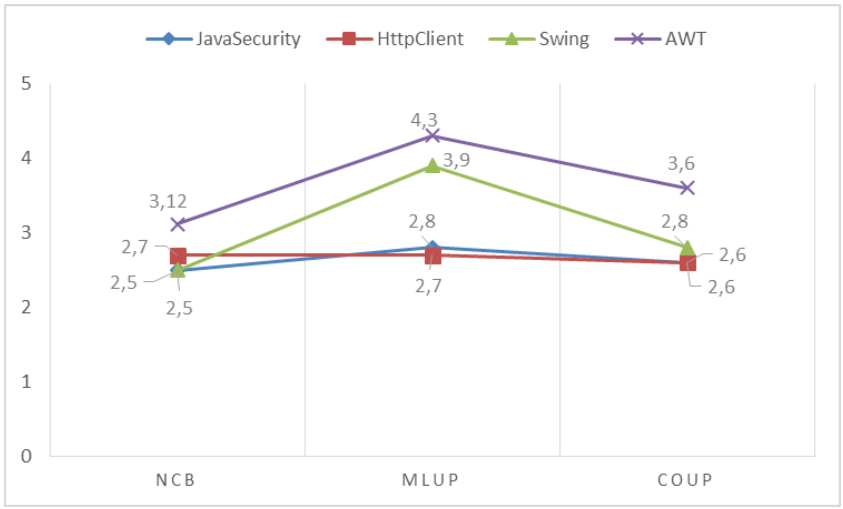


Figure 6.4: Average *size* of inferred patterns using different techniques

6.5.2.2 Cross-validation

The cross-validation allows us to observe on two levels the effect of COUPminer on inferred patterns. First, we inspect the co-usage relationships of the detected patterns in the context of potential new client programs. Then, we characterize the usage cohesion deterioration between the training and validation context.

Patterns Validation Cohesion As it can be seen in Table 6.II, with MLUPminer, we obtained the highest average validation cohesion with around 83%. This indicates very strong co-usage relationships within the inferred patterns. In the case of NCBUPminer, we notice a visible decline in the validation cohesion value with an average validation cohesion around 67%. In the case of COUPminer the decline was very limited with average values around 81%. We also notice that the standard deviation values are very low. This shows that, overall, the detected patterns using COUPminer have always a good usage cohesion in the context of the validation client programs. Looking at the cohesion distribution in Figure 6.5, we observe that, for all studied APIs, the lower quartile in the case of COUPminer is higher than the upper quartile of NCBUPminer. This means that at least 75% of the cooperatively mined patterns are better than 75% of the patterns mined with NCBUPminer.

	NCBUP			MLUP			COUP		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Security	0.70	0.04	0.76	0.85	0.12	1.00	0.86	0.12	1.00
HttpClient	0.69	0.07	0.78	0.79	0.16	1.00	0.79	0.08	1.00
Swing	0.60	0.04	0.68	0.85	0.09	1.00	0.80	0.06	1.00
AWT	0.68	0.06	0.77	0.84	0.06	0.95	0.81	0.05	0.88

Table 6.II: Statistics on the Cohesion of identified API usage patterns for NCBUPminer, MLUPminer and COUPminer , in the contexts of validation clients

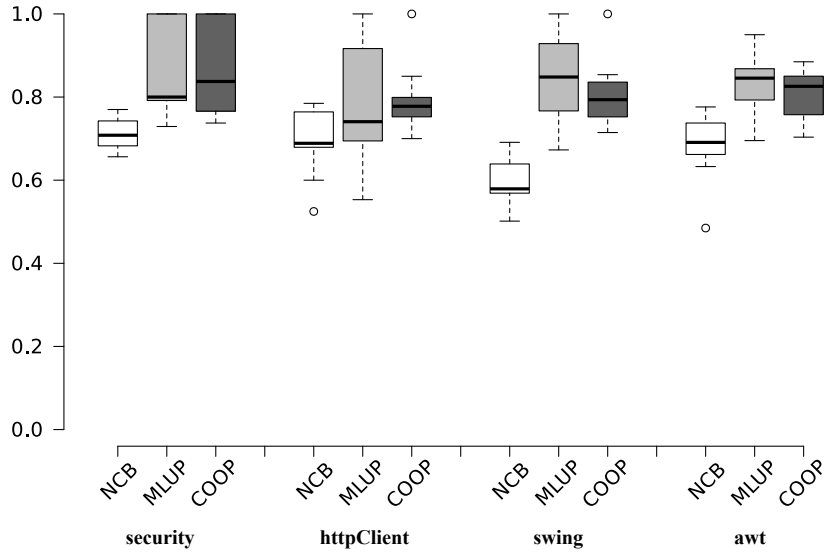


Figure 6.5: Cohesion values of identified API usage patterns, for NCBUPminer, MLUPminer and COUPminer in the contexts of validation clients

Patterns Consistency As it can be seen in Table 6.III, the results reveal that the patterns identified with NCBUPminer are overall characterized with very high consistency. Indeed, the average value across multiple validation client programs is around 94%. A slight deterioration is observed for the consistency of patterns mined with MLUPminer average values are around 81%. The table also shows that for all studied APIs, in the case of COUPminer, consistency values are close to those from NCBUPminer. We consider as an example the case of Swing API, where 22 leave-one-out cross validations have been made, using 22 client programs. In this case, the results show that, for any detected pattern in Swing, on average 87% (for COUPminer) and 94% (for MLUPminer) of co-usage relationships between pattern methods remain similar across 22 client programs of AWT. When

comparing median boxplots in Figure 6.6 we note that, for all studied APIs, at least half of the patterns detected with COUPminer are more consistent than half of the patterns detected with MLUPminer.

	NCBUP			MLUP			COUP		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Security	0.94	0.03	0.98	0.86	0.09	0.98	0.88	0.07	0.98
HttpClient	0.96	0.03	0.97	0.77	0.09	0.89	0.83	0.08	0.94
Swing	0.94	0.05	0.98	0.83	0.05	0.92	0.87	0.05	0.95
AWT	0.92	0.03	0.89	0.84	0.04	0.94	0.86	0.07	0.95

Table 6.III: Statistics on the Consistency of identified API usage patterns for NCBUPminer, MLUPminer and COUPminer , across multiple validation clients.

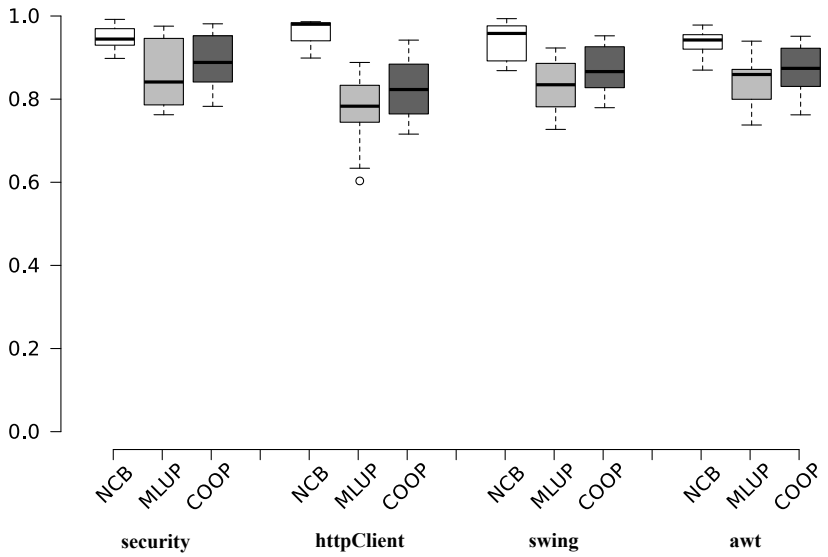


Figure 6.6: Consistency values of identified API usage patterns for NCBUPminer, MLUPminer and COUPminer across multiple validation clients.

6.5.3 Discussion and threats to validity

Overall, our technique COUPminer has been able to infer usage patterns that are more cohesive than those of NCBUPminer and more consistent than those of MLUPminer. In addition, and above all, as expected the cooperative cohesion was slightly lower than the one of MLUPminer, and the cooperative consistency was slightly lower than the one of NCBUPminer. However, the results are

close and high enough to reflect accurate co-usage relationships and generalizable patterns. In conclusion, the cooperative approach allows achieving a very good tradeoff between the accuracy of the client-based mining and the generalizability of the library-based mining.

Detected usage patterns could help to enhance the APIs documentation and the evaluation of our approach took into account the generalization to other client programs of the same API and showed that the usage patterns of an API remain informative for new clients. As a threat to the validity of our study, this finding can be attributable to the client programs used in our study. Still, we used a fair number of validation client programs to evaluate the quality of the inferred usage patterns by our technique.

Another possible threat to validity, is the evaluation of our technique on four APIs only. Although, we selected four APIs of different domains, the approach may not be generalized to a larger set of APIs because these may be of different natures and qualities. Indeed, we are combining, on the one hand, a library based heuristic that depends on the code quality and programming style, with, on the other hand, a client based heuristic that depends on the API size and the amount of functionalities provided to client programs. To better assess our cooperative approach, we are planning to investigate the effect of these properties on the effectiveness of our technique, in a larger experimental setting.

6.6 Conclusion

In this chapter, we present a technique that infers API usage patterns using both library-based and client-based heuristics. Our technique uses structural, semantic and co-usage similarity between the API methods to infer usage patterns. We applied our technique on four APIs that differ in size, utility and usage domain. To evaluate the performance of our technique, we analyzed the quality of inferred API patterns in terms of consistency and usage cohesion through a large variety of API client programs. We found that our technique allows taking advantage at the same time from the accuracy of the client-based techniques and the generalizability of the library-based techniques.

In the next chapter we close the circle of our holistic approach through another level of library usability across complementary software libraries.

Part III

Library usability across complementary software libraries

Chapter 7

Automated Inference of Software Library Usage Patterns

As we mentioned previously, today's open-source repositories provide a wide range of libraries, and it is widely recognized that using mature and well-tested third-party libraries can improve developers' productivity, reduce time-to-market, and produce more reliable software. However, as software libraries are documented separately but intended to be used together, developers are unlikely to fully take advantage of these reuse opportunities. In this chapter, we present our approach to automatically identify third-party library usage patterns, i.e., collections of libraries that are commonly used together by developers. Our approach employs a clustering technique to group together software libraries based on external client usage. To evaluate our approach, we mined a large set of 6,000 popular libraries from Maven Central Repository and investigated their usage in 38,000 client systems from the Github repository. Our experiments show that our technique is able to detect the majority (77%) of highly consistent and cohesive patterns.

7.1 Introduction

Today's code repositories provide an increasingly large number of reusable software libraries with a variety of functionalities. Automatically analyzing how software projects utilize these libraries, and understanding the extent and nature of software library reuse in practice is a challenging task for developers. Indeed, software developers can spend a considerable amount of time and effort to manually identify libraries that are useful and relevant for the implementation of their software. Worse yet, developers may even be unaware of the existence of these libraries. Developers tend to reinvent the wheel and implement most of their features from scratch instead of reusing functionalities provided by third-party libraries as pointed out by several researchers [18, 79]. Therefore, we believe that identifying patterns of libraries commonly used together, can help developers to discover and choose libraries that may be relevant for their projects' implementation.

In this chapter, we propose an approach for mining Library Co-Usage Patterns, namely LibCUP. We define a usage pattern of libraries as a collection, with different usage cohesion levels, of libraries

that are most frequently jointly used in client systems. Our approach uses the adaptation of DBSCAN clustering algorithm, to detect candidate library usage patterns based on the analysis of their frequency and consistency of usage within a variety of client systems. Different client systems may use utility libraries (e.g., `JUnit`, `log4j`, etc.) as well as domain-specific libraries (e.g., `httpclient`, `groovy`, `spring-context`, etc.). Thus, the rationale behind the distribution on different usage cohesion levels of libraries in a pattern, is to distinguish between the most specific libraries and the less specific ones. Moreover, our approach is intended to be used first to identify patterns of particular libraries that interest a developer. These libraries could then be fed to the previously proposed approaches (in chapters 4, 5, and 6) to recommend particular methods to be used in different contexts. Moreover, LibCUP provides a user-friendly visualization tool to assist developers in exploring the different library usage patterns.

We evaluate our approach on a large dataset of over 6,000 popular libraries, collected from Maven Central repository [1] and investigated their usage from a wide range of over 38,000 client systems from Github repository [2], from different application domains. Furthermore, we evaluated the scalability of LibCUP as compared to LibRec [77], a state-of-the-art library recommendation technique based on association rule mining and collaborative filtering. We also performed a ten-fold cross validation to evaluate the generalizability of the identified usage patterns to potential new client systems. Our results show that across a considerable variability of client systems, the identified usage patterns by LibCUP remain more cohesive than those identified by LibRec.

The remainder of this chapter is organized as follows. Section 7.2 motivates the usefulness of LibCUP with two real-world examples. We detail our approach in Section 7.3. We present our experimental study to evaluate the proposed approach in Section 7.4, while providing discussions in Section 7.5. Finally, Section 7.6 concludes the chapter.

7.2 Motivation and challenges

In this section, we present two real-world scenarios to motivate the usefulness of library co-usage patterns. In the first example, the goal is to find a set of libraries that allow to meet the requirements of a given software system. In the second example, the goal is to decide between two libraries with

1. http://mvnrepository.com
2. www.github.com

similar functionalities to be used in a software system. In this context, we assume that a library with more potential to be used with other related libraries is preferred. The related libraries are assumed to extend the features of the software system.

7.2.1 Learning-environment example

Let us consider a software-development team responsible of the task of maintaining a Web portal for a growing private university with around 4,000 undergraduate and graduate students. The university is planing to move from a simple Web portal to an advanced course management system to provide adequate service to their students and faculty members. As a first step, the development team decided to go through an exploratory phase, during which they developed a situational application to assess the turnout rate in the new learning environment. This application allows students and faculty to schedule activities related to courses and maintain deadlines related to projects. It should also allow real-time conversations between course or project participants.

Based on these requirements, developers found that their application requires some basic functionalities including a *scheduling* and an *emailing* services that have to be integrated. In this situation, developers can either implement the different features from scratch, or reuse features provided by existing libraries. In both cases, they may spend a considerable time and effort for either implementing the features or finding compatible and useful libraries to be integrated in the application.

The development team later find out that they are required to use the `quartz` [3] library to implement the scheduler. With this new constraint, the developers have to solve the following challenges:

- *What is the recommended emailing library that best complements the* `quartz` *library?* The selection should take into account assumed compatibility with the `quartz` library as well as the effort needed to integrate the library into the system.
- *More generally, what related libraries can be used to implement the remaining features of their software system?* The developers might be interested in related libraries that are commonly used by similar systems with the `quartz` library.

Addressing these two challenges could be a complex task for developers if done manually. Indeed, developers should check in open-source code repositories to find similar projects, and investigate their

3. http://mvnrepository.com/artifact/org.quartz-scheduler/quartz

library usage. Manually finding libraries that are commonly used together in a particular scenario and understanding the current usage practice for a particular library is unlikely to be effective.

7.2.2 Web application frontend example

We now consider another scenario with Aaron, a freelance programmer, who seeks to implement an inventory-management web application. Aaron decided to develop his web applications in an industrial setting, where the back-end is implemented in Java, and the front-end is implemented in a Java/XML based framework. For the user interfaces, several libraries can be used; the most popular ones are `primefaces`[4] the UI component library for Java Server Faces, and `gwt-user`[5] of the Google Web Toolkit.

Aaron has to decide which library to use: `primefaces` or `gwt-user`. In other words:

– *Which library is the best option in terms of future extension of the software system's functionalities?* Aaron prefers libraries that are usually used with many other libraries, which offers a large variety of functionalities. This provides a high potential of extensions of his software system.

In both examples, we consider that mining patterns of libraries used jointly by many client systems may provide insights to make the best decisions.

7.2.3 Challenges: mining library usage

In this work, we mine the 'wisdom of the crowd' to discover usage patterns of software libraries. Studying the current library usage within similar systems may provide hints on compatibility and relevance between existing libraries. We assume that libraries that are commonly used together are unlikely to have compatibility and integration issues.

The goal is to discover which sets of libraries are commonly used together by similar systems. To this end, our approach is designed to find multiple layers, i.e., levels of relevant libraries according to their usage frequency. For effective reuse, developers can go through the different levels inside the usage patterns to discover relationships, with different strengths, between the collection of related libraries.

4. http://mvnrepository.com/artifact/org.primefaces/primefaces
5. http://mvnrepository.com/artifact/com.google.gwt/gwt-user

For the first motivating example, we use the usage patterns to discover that `commons-email` library [6], which is a popular emailing library, complements the `quartz` library. Furthermore, by using the multi-layers structure of our patterns, developers can then find related libraries that would complement, at different degrees, both `quartz` and `commons-email`.

For the second motivating example, we found that `gwt-user` library is part of a usage pattern with many other related libraries including `gwt-dev` [7], `gwt-servlet` [8], `gwt-incubator` [9], and `gin` [10]. This collection of libraries covers different functionalities such as browser support, widgets, optimization, data binding, and remote communication. All these features are opportunities for future extensions, and we are confident that they can be integrated together as demonstrated by the client systems that already used them. Conversely, Aaron found that, although `primefaces` library might be useful for his system, it is not widely used with other libraries and, then, does not offer a sufficient guarantee of future integration with other libraries.

These two examples show that the task of identifying library usage patterns becomes more and more complex, especially with the exponentially growing number of libraries available in the Internet. This motivates our proposal of automatically identify library usage patterns to assist developers in reusing and integrating libraries and, then, increase their productivity.

7.3 Approach

In this section, we present our approach, LibCUP, for mining library usage patterns. Before detailing the used algorithm, we provide a brief overview of our approach, we finally describe our visualization technique to explore the identified library usage patterns.

7.3.1 Approach overview

Our approach takes as input a set of popular libraries, and a wide variety of their client systems extracted from existing open-source repositories. The output is a set library usage patterns organized within different layers according their co-usage frequency.

6. http://mvnrepository.com/artifact/org.apache.commons/commons-email
7. http://mvnrepository.com/artifact/com.google.gwt/gwt-dev
8. http://mvnrepository.com/artifact/com.google.gwt/gwt-servlet
9. http://mvnrepository.com/artifact/com.google.gwt/gwt-incubator
10. http://mvnrepository.com/artifact/com.google.gwt.inject/gin

We define a *library co-usage pattern* as a collection of libraries that are commonly used together. It represents an exclusive subset of libraries, distributed on different usage cohesion layers. A usage cohesion layer reflects the co-usage frequency between a set of libraries.

Indeed, similar client systems may share some domain-specific libraries, but they may at the same time share some utility libraries, which are more commonly used by a large number of systems. For this reason, we seek a technique that can capture co-usage relationships between libraries at different levels.

Our approach proceeds as follows. First, the input dataset is analyzed to identify the different client systems depending on each library. Then, the dependency information is encoded using dependency vectors. Indeed, each library in the dataset is characterized with a dependency vector which encodes information about (1) their client systems and (2) the rest of other systems in the dataset that are not using it. Finally, we use a clustering technique to group the libraries that are most frequently co-used together by clients. All libraries that have no consistent usage through the client systems are isolated and considered as noisy data.

7.3.2 Multi-layer library co-usage pattern mining

For the library co-usage pattern mining, we also use the DBSCAN-based incremental clustering. The description of the algorithm and the pseudo-code are given in Chapter 4, sections 4.3.4 and 4.3.5. DBSCAN allows the clustering algorithm to filter out all points that are not located in a dense region of points in the search space. This specific property explains our choice of DBSCAN to detect usage patterns of libraries. Indeed, not all libraries of the dependency dataset are to be clustered because some are simply not co-used with specific subsets of the libraries, while others are co-used with almost all the subsets of libraries.

In our approach, each library is represented as a dependency vector that has constant length l. The vector length is the number of all client programs which use the libraries in the dataset. Figure 7.1 shows that the considered dataset represents 8 client systems depending on 8 third-party libraries. For an external library, Lib_x, an entry of 1 (or 0) in the i^{th} position of its dependency vector, denotes that the client system corresponding to this position depends (or does not depend) on the considered library. Hence, summing the entries in the library's vector represents the number of its client program in the dataset. For instance, in Figure 7.1, the dependency vector of *Lib1* shows that the four client

systems *C1*, *C2*, *C3* and *C6* depend on this library. We can also see that these systems depend on other libraries including *Lib2, Lib3* but none of them depends on *Lib5*.

DBSCAN constructs clusters of libraries by grouping libraries that are close to each other, thus forming a dense region (i.e. similar libraries) in terms of their co-usage frequency. For this purpose, we define the Dependency Similarity, *DSim* in Equation (7.1), between two libraries Lib_i and Lib_j, using the Jaccard similarity coefficient with regards to the client programs, *Cl_sys*, of Lib_i and Lib_j.

The rationale behind this is that two libraries are close to each other (short distance) if they share a large subset of common dependent client systems.

$$DSim(Lib_i, Lib_j) = \frac{|Cl\_sys(Lib_i) \cap Cl\_sys(Lib_j)|}{|Cl\_sys(Lib_i) \cup Cl\_sys(Lib_j)|} \tag{7.1}$$

where $Cl\_sys(Lib)$ is the set of client programs depending on the library *Lib*. For example, the *DSim* between the libraries *Lib1* and *Lib6* in Figure 7.1 is $\frac{2}{4}$ since these libraries have in total 4 client programs, and 2 of them are common for *Lib1* and *Lib6*. The distance between the points in the search space corresponding to two libraries Lib_i and Lib_j is then computed as $Dist = 1 - DSim(Lib_i, Lib_j)$.

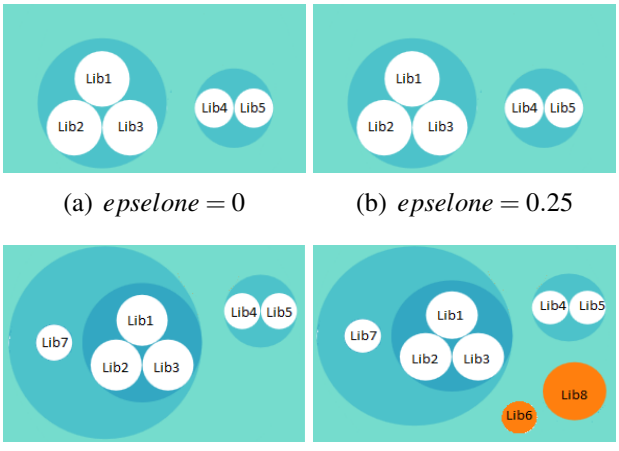
	C1	C2	C3	C4	C5	C6	C7	C8
Lib1	1	1	1	0	0	1	0	0
Lib2	1	1	1	0	0	1	0	0
Lib3	1	1	1	0	0	1	0	0
Lib4	0	0	0	1	1	0	0	0
Lib5	0	0	0	1	1	0	0	0
Lib6	0	0	0	0	0	0	1	0
Lib7	0	0	1	0	0	1	0	0
Lib8	1	1	0	1	1	0	1	1

Figure 7.1: The dependency vector representing the dependency between eight client programs and eight libraries.

We build the clusters incrementally by relaxing the epsilon parameter, step by step. We first, takes as input a dataset containing all the libraries and their client systems within a specific format, then

we cluster them using an epsilon value of 0. This step results in clusters of libraries that are always used together, as well as multiple noisy points left out. For each produced cluster, we aggregate the dependency vectors of its libraries using the logical disjunction in one dependency vector. Then, a new dataset is formed, which includes the aggregated dependency vectors and the dependency vectors of noisy libraries from the previous run. This dataset is then fed back to the DBSCAN algorithm for clustering, but with a slightly higher value of epsilon. This procedure is repeated in each step until reaching a maximum value for epsilon, given as a parameter.

For example, Figure 7.2 shows the incremental clustering of the libraries. In this example, the initial dataset contains 8 libraries, *Lib1, ..., Lib8*, the $epsilon$ parameter is incremented in each clustering step by 0.25 with the epsilon maximum value set to 0.55. As shown in Figure 7.2(a), the first step produces two clusters at $epsilon = 0$. The two clusters include respectively (*Lib1, Lib2, Lib3*) and *Lib4, Lib5*. These libraries are clustered at the most dense level since, in each cluster, these libraries were frequently co-used together. The second step is performed with $epsilon = 0.25$ as illustrated in Figure 7.2(b). For this step, there is no change in the dataset since the distances are larger than the current $epsilon$ value. Finally at $epsilon = 0.5$, as illustrated in Figure 7.2(c) a new cluster involving 2 density level is generated. This cluster includes *Lib7* in addition to (*Lib1, Lib2, Lib3*) since they share 2 out of the 4 common client systems.



(a) $epselone = 0$ (b) $epselone = 0.25$

(c) $epselone = 0.5$ (d) $epselone = 0.75$

Figure 7.2: Resulting clusters of applying the incremental algorithm using ε-DBSCAN to the library dataset presented in Figure 7.1.

We can notice that *Lib6* is a rarely used library and *Lib8* is a utility library used with almost all the considered client systems, showing no particular usage trend. Thus at the last iteration of the incremental clustering as illustrated in Figure 7.2(d), the libraries *Lib6* and *Lib8* are left out as noisy points since their distance from the clustered libraries is larger than the maximum epsilon value, which is 0.55.

7.3.3 Pattern visual exploration

With the exponentially growing number of software libraries in the Internet, the size of possible usage patterns tends to be very large. To simplify the exploration of the inferred patterns, we opted for an interactive and user-friendly visual exploration technique. The distribution of libraries on different usage cohesion levels follows a hierarchical structure by inclusion that can be visualized through a treemap layout [35]. However, to better reveal the hierarchy, we opted for a circle packing visualization [83] even if circle packing is not as space-efficient as a treemap.

In our visualization, the libraries are represented as white dots and the clusters as circular region with deferent shades of blue. Figure 7.3 shows an example of the circle packing visualization. Elements belonging to the same cluster (co-used elements) are placed at the inner periphery of the circular region representing the cluster. Thus the nesting between circular regions indicates the different clustering level. Our visualization enables us to map different information to the graphical component of the visualization scene. For instance, library popularity can be mapped to the dot size, and the cluster cohesion can be mapped to the region color.

The developer can use our visualization to explore the inferred patterns. The exploration is based only on pattern cohesion and the circle packing nesting. Our visualization environment provides also an interactive navigation where the user can click on any usage pattern, i.e., cluster, to zoom in or zoom out. The visualization allows also to place the mouse over a library to get a tooltip with some more information. For example, when clicking on a library the user is redirected automatically to its artefact home page on the Maven Central repository [11] in order to download the library jar file. Our visualization allows also the developer to provide as input the set of libraries he is already using in his system and for which he is trying to find related and commonly used libraries. Thereafter, the

11. http://mvnrepository.com

entered libraries are highlighted in a different color, and the developer can start his exploration from the highlighted points.

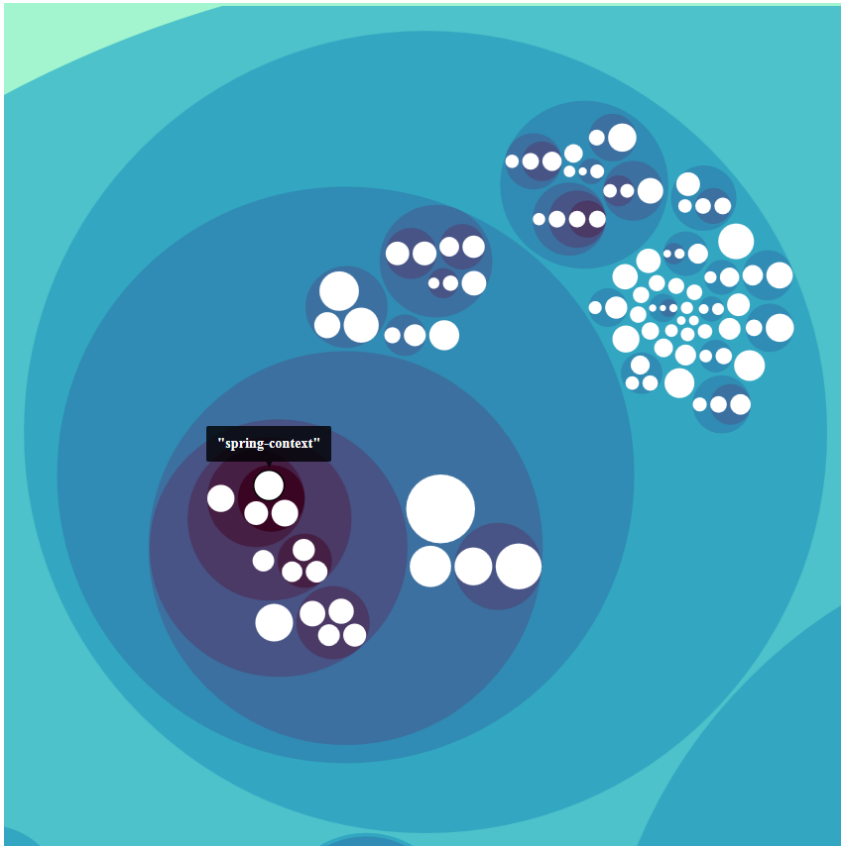


Figure 7.3: circle packing visualization

7.4 Empirical evaluation

In this section, we present the results of our evaluation of the proposed approach, LibCUP. Our study aims at assessing whether LibCUP can detect *usage patterns of libraries* that are (*i*) *cohesive* enough to provide valuable information to discover relevant libraries, and (*ii*) *generalizable* for new client systems. We also compare the results of our technique LibCUP to the available state-of-the-art approach, LibRec [77]. LibRec combines association rule mining and collaborative filtering to recommend libraries based on their client usage. For each experiment in this section, we present the research questions to answer, the research method to address them, followed by the obtained results.

7.4.1 Data collection

To evaluate the feasibility of our approach on real-world scenarios, we carried out our empirical study on a large dataset of Open Source Software (OSS) projects. As we described earlier, our study is based on widely used libraries collected from the popular library repository Maven and a large set of client systems collected from Github repository. Since Github is the host of varying projects, to ensure validity of quality Github projects, we performed the following filtering on the dataset:

- *Commit size*. We only included java projects that had more than 1,000 commits.
- *Forks*. We only include projects that are unique and not forks of other projects.
- *Maven dependent project*. We only included projects that employ the maven build process (use `pom.xml` configuration file).

Each Github repository may contain multiple projects, each having potentially several systems. Each of these systems are dependent on a set of maven libraries, that are defined in a `pom.xml` file within the project.

	Dataset
Snapshot Date	15th January 2015
# of github systems	38,000
# of unique dependent libraries	6,638

Table 7.I: Dataset used in the experiment

Note that for all data, we first downloaded an offline copy of the original software projects (the source code) from Github and the libraries (the jar files) from Maven before extraction. Thereafter, for each library, we selected the latest release. In the beginning, we started with 40,936 dependent libraries. However, to remove noise, we filtered out libraries having less than 50 identifiers based on methods, attributes and classes. This process removed libraries that we assume very small or partial copies of their original libraries and thus are not relevant. Our dataset resulted in 6,638 different Maven libraries extracted from unique 38,000 client systems from Github.

The dataset is a snapshot of the projects procured as of 15th January 2015. Our dataset is very diversified as it includes a multitude of libraries and software systems from different application domains and different sizes. Overall, the average number of used libraries per system is 10.56, while the median is 6.

7.4.2 Sensitivity analysis

As a first experiment, we evaluated the sensitivity of the patterns' quality, identified by LibCUP, with respect to different settings, including the dataset size and *maxEpsilon* values. We aim at addressing the following research question.

RQ1. *What is the impact of various experimental settings on the patterns' quality?*

7.4.2.1 Analysis method

To address **RQ1**, we need to evaluate whether the detected patterns are cohesive enough to exhibit informative co-usage relationships between specific libraries. Hence, we use the Pattern Usage Cohesion metric (PUC), as defined in Equation (7.2) to capture the cohesion of the identified patterns.

$$PUC(p) = \frac{\sum_{cp} ratio\_used\_Libs(p, cp)}{|C(p)|} \in [0, 1] \tag{7.2}$$

where *cp* denotes a client system of the pattern *p*, *ratio_used_Libs(p, cp)* is the ratio of libraries that belong to the pattern *p* and that are used by the client system *cp*, and $C(p)$ is the set of all client systems of all libraries in *p*.

To answer our first research question **(RQ1)**, we perform two studies.

- **Study 1.A.** We apply LibCUP to our collected dataset described in Section 7.4.1. Then, we investigate the impact of different *maxEpsilon* values on the PUC results of the detected patterns.
- **Study 1.B.** We investigate the scalability of our technique. We fix the *maxEpsilon* value and we run LibCUP several times while varying the dataset size to observe the patterns cohesion and the time efficiency.

7.4.2.2 Results for RQ1

Study 1.A: Sensitivity to *maxEpsilon* parameter. Figures 7.4, 7.5, and 7.6 report the effect of different *maxEpsilon* values. Our experiments show that the *maxEpsilon* parameter influences different characteristic of the inferred patterns, including the pattern usage cohesion, the number of inferred patterns, and the patterns size. Figure 7.4 shows that the average PUC ranges from 1 to 0.5, while varying the *maxEpsilon* in the range [0,0.95]. We notice that even when the *maxEpsilon*

reaches high values, the inferred patterns maintain acceptable cohesion values. This is due to the incremental construction of the patterns that generates multiple layers of libraries, each reflected at a different À density level. Thus, layers inferred at the early steps influence the overall cohesion of the final pattern.

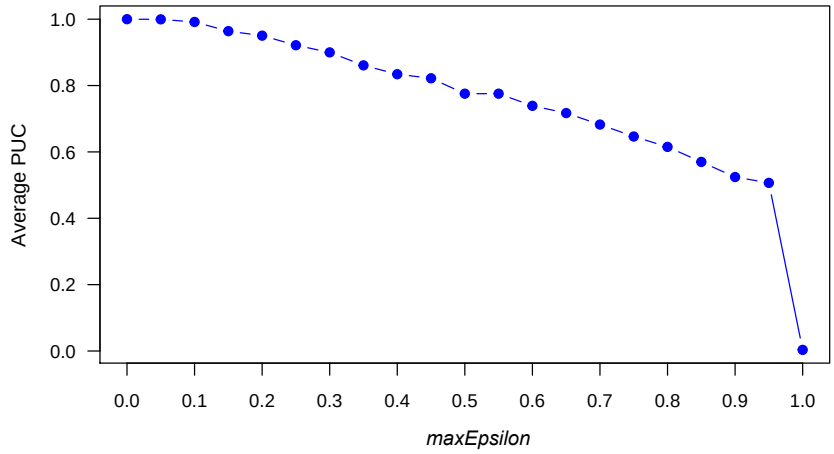


Figure 7.4: Effect of varying *maxEpsilon* parameter on the average cohesion of the identified patterns.

When *maxEpsilon* is set to 1 the patterns' cohesion drop down to 0. This is because in the last step all the libraries are clustered into one usage pattern as depicted in Figure 7.5. Moreover, we can clearly see from this figure that the number of inferred patterns increases to reach a peak of 1,061 when *maxEpsilon* is set to 0.80. In more details, we observe that:

- Before the peak, some of the existing patterns are enriched with new libraries to add new external layers to the original usage pattern. Moreover, we also noticed that some new patterns are identified with libraries that were considered as noisy. This is since we are tolerating less density within clusters when *maxEpsilon* increases. We observe that this has an effect to increase the global number of inferred patterns.
- After the peak, some of the existing patterns are merged without losing their internal structure. This result, in turn has an effect to reduce the overall number of inferred patterns.

To get a more qualitative sense of the obtained results, we noticed that for the low values of *maxEpsilon* and up to intermediate values (i.e., 0.5, 0.6), the inferred patterns tend to mainly cover

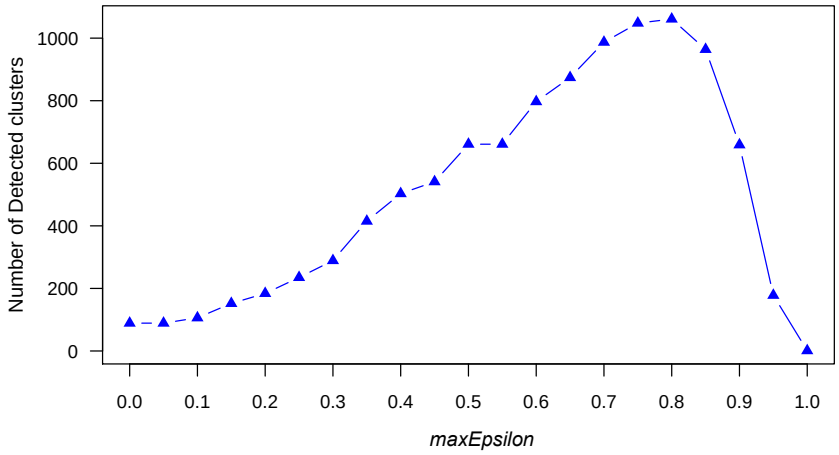


Figure 7.5: Effect of varying *maxEpsilon* parameter on the number of identified patterns.

domain specific libraries (e.g., program analyzers `jdepend`[12], graphics manipulation `batik`[13], etc.). Those patterns are characterized with an average number of client systems that do not exceed 50 clients per pattern. The more the *maxEpsilon* parameter is relaxed, the more the patterns are enriched with other libraries. Starting with specific libraries, the patterns reach a step in which they become enriched with utility or more generic libraries such as `JUnit`[14] and `log4j`[15]. For sake of simplicity, we do not present in Figure 7.6 the last step where all the libraries are clustered into one single cluster with a larger number of client systems. Based on these results, LibCUP uses a *maxEpsilon* threshold of 0.5 as a defaults parameter.

Study 1.B: Sensitivity to the dataset size. To carry out this experiment, we set the *maxEpsilon* value to 0.5. This is a proactive choice to ensure that libraries appearing in the same pattern are used more frequently together than separately. Thereafter, we run LibCUP with different dataset sizes. In each run, we augmented the previously used dataset with 1000 libraries, and we observed the average cohesion of patterns as well as the execution time taken to infer them. All experiments were carried out on a computer with an Intel core i7-4770 CPU 3.40 GHz, with 32 GB RAM.

Figure 7.7 depicts the obtained results for this experiment. We noticed from the figure, that the shape of the graph is consistent for the different dataset size. The PUC score slightly increases from

12. http://mvnrepository.com/artifact/jdepend/jdepend
13. http://mvnrepository.com/artifact/batik/batik
14. http://mvnrepository.com/artifact/junit/junit
15. http://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api

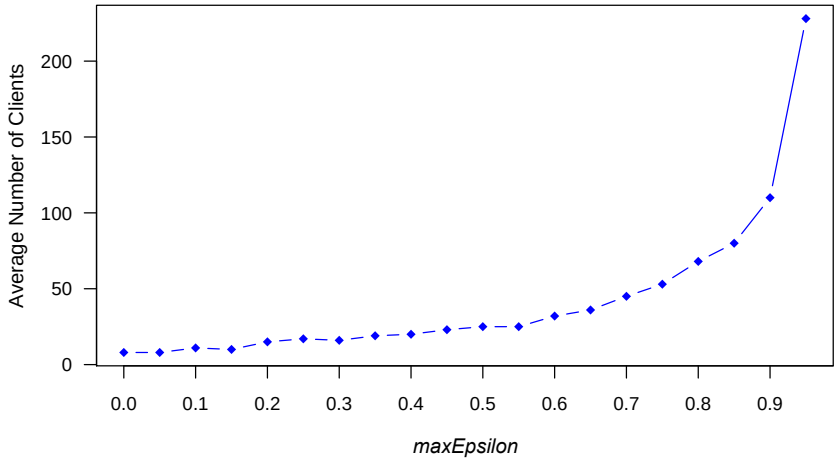


Figure 7.6: Effect of varying $maxEpsilon$ parameter on the average number of clients per pattern.

0.79 to stabilize at 0.82 for the last three runs. In more details, we found that there is an increase in terms of the number of inferred patterns from 62 in the first run with an average size of 3 libraries per pattern, to reach the bar of 500 patterns at the last run with an average size of 5.5 libraries per pattern. These results confirm that when considering more libraries, LibCUP is able to enrich the inferred patterns with new libraries while detecting new patterns.

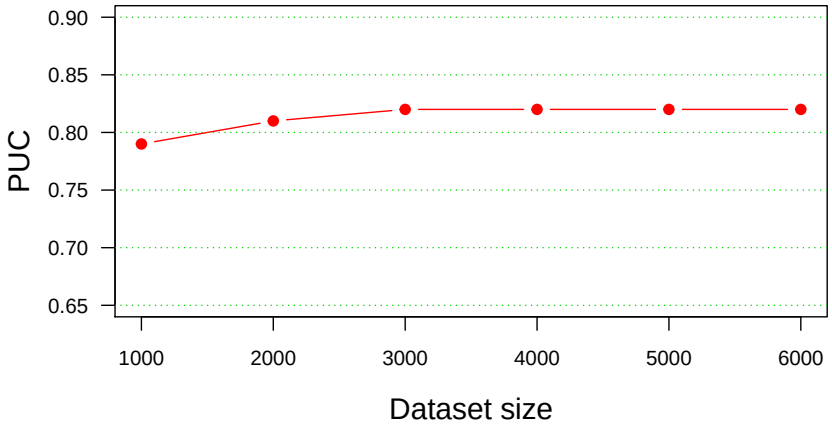


Figure 7.7: Effect of varying the dataset size on the average pattern cohesion $maxEpsilon = 0.5$.

In terms of time efficiency, Figure 7.8 depicts the influence of the dataset size on the execution time. As it can be seen on the figure, the execution time of LibCUP is sensitive to the dataset size,

as expected. At the first run, LibCUP took less than 7 minutes to mine a set of 1,000 libraries, while reaching 159 minutes of execution time to mine the large set of 6,000 libraries with their 38,000 client systems. However, it is worth saying that even with 159 minutes of execution time, LibCUP can be considered time efficient, since the inference process is done off-line once, then the identified patterns can be easily explored using our interactive visualization tool.

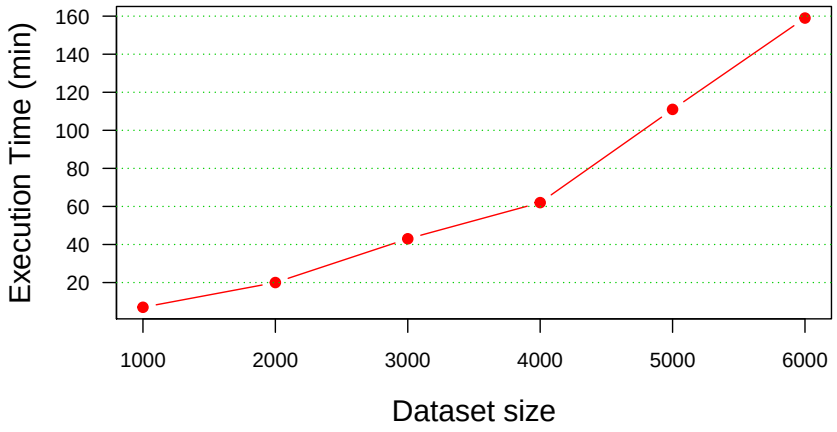


Figure 7.8: Effect of varying the dataset size on the time efficiency with $maxEpsilon = 0.5$.

In summary, the obtained PUC results of the identified usage patterns provide evidence that LibCUP exhibits consistent cohesion using our adopted of DBSCAN algorithm. Using a default $maxEpsilon = 0.5$, we found that at least 50%, and up to 100%, of the usage patterns are co-used together with high PUC. Moreover, our technique is stable and time efficient when varying the size of the mined library set.

7.4.3 Evaluation of patterns cohesion

As a second experiment, we conduct a comparative study to evaluate the cohesiveness of the identified library usage patterns against the state-of-the-art approach LibRec [77]. To the best of our knowledge, LibRec [77] is the only existing approach that has addressed this problem. We aim at addressing the following research question.

RQ2. *To which extent are the identified library usage patterns cohesive as compared to those inferred with LibRec?*

7.4.3.1 Analysis method

To address our second research questions **(RQ2)** we conducted a comparative evaluation of our approach with LibRec in order to better position our approach and characterize the obtained results.

To infer usage patterns, LibRec is based on mining association rules obtained from closed itemsets and generators using the Zart algorithm [74]. We applied both LibCUP and LibRec to all the selected libraries of our dataset (cf. Section 7.4.1). Then, we compare the identified usage patterns of both approaches in terms of PUC. More specifically, we compare the average PUC values for all detected patterns of each approach. For LibCUP, we fixed the $maxEpsilon$ value to 0.5 as explained earlier. For LibRec we fixed the $Minconf$ to 0.8 , the $Minsup$ to 0.002 and the $Number\ of Nearest Neighbors$ to 25 [77].

7.4.3.2 Results for RQ2

Table 7.II reports the obtained results for **RQ2**. On average, LibCUP achieves an average PUC score of 0.82, which outperforms LibRec that was only able to achieve 0.72 of PUC. The achieved PUC values by LibCUP reflect high co-usage relationships between the pattern's libraries making them more cohesive.

In terms of the number of inferred patterns, we observe that our multi-layer clustering technique allows detecting a reasonable number of patterns of 531, with a medium size of libraries distributed on the different layers (i.e., 5.5). On the other hand, LibRec inferred an abundant number of patterns up to 3,952, even though it relies on closed itemsets and generators to construct a compact set of association rules [77]. Indeed, the set of patterns obtained from the closed itemsets and generators is supposed to be much smaller than the complete set of rules. However, in practice the inferred patterns with LibRec tend to be many but with smaller size (on average, it generates 2 libraries per pattern). We believe that this large number of small size library patterns will in turn limit the practical adoption and usefulness of the LibRec approach.

Furthermore, we studied the number of clients per pattern. We noticed from the results of Table 7.II, that the patterns inferred by LibCUP are used on average with 30 client systems. Indeed, by manually investigating these client systems, we found that they generally share common domain-specific features. For LibRec, the inferred patterns are used within too many client systems that share

pairs or triplets of libraries, which are, in most of the cases, utility libraries such as `JUnit`, `log4j`, `slf4j-api`, and `commons-lang`. These libraries are likely to be used by several unrelated client systems.

Number of patterns	LibCUP	LibRec
Avg PUC	0.82	0.72
Nb Patterns	531	3,952
Avg pattern size	5.5	2.0
Nb Clients per Pattern	30	2,269

Table 7.II: Average cohesion and overview of the inferred usage patterns for LibCUP and LibRec.

7.4.4 Evaluation of patterns generalization

In this study, we aim at evaluating whether the identified library usage patterns with LibCUP can be generalizable in comparison with those of LibRec. We aim at addressing the following research question.

RQ3. *To which extent are the detected usage patterns generalizable to other "new" client systems, that are not considered in the training dataset?*

7.4.4.1 Analysis method

To answer **RQ3**, we investigate whether the detected patterns will have similar PUC values in the context of new client systems. Our hypothesis is that:

> *Detected patterns are said "generalizable" if they remain characterized by a high usage cohesion degree in the contexts of various client systems.*

To evaluate the generalizability of the detected patterns, we perform a ten-fold cross-validation on all the client systems in the dataset. We randomly distribute the dataset into ten equal-sized parts. Then, we perform ten independent runs of both approaches, LibCUP and LibRec. Each run uses nine parts as training client systems to detect possible patterns, and leaves away the remaining part as a validation dataset.

The results are sorted in ten runs, where each run has its associated patterns, and its corresponding training and validation client systems. Then, we address **RQ3** through two experimental studies as follows.

- **Study 3.A.** We evaluate the cohesion of the detected patterns (as measured by PUC) in the context of the validation datasets. In a given run, it is possible that some detected patterns contain only libraries that are never used in the validation client systems. Consequently, to evaluate the generalizability of the detected patterns in each run, we consider only the patterns that contain at least one library that is actually used by the run's validation client systems.
  We call such patterns the '*eligible patterns*' for the validation client systems. An eligible pattern will have a low PUC if only a small subset of its libraries is used by the validation client systems, while the other libraries have not been used. As a consequence, it will be considered as "non-generalizable". This study aims at comparing the PUC results obtained for the training client systems context and validation client systems context for both LibCUP and LibRec.
- **Study 3.B.** In this study, we push further the comparison, as LibRec is specifically designed for library recommendation. We attempt to evaluate whether our approach is also useful in a recommendation context. To this end, we define for the library patterns inferred by LibCUP an ad-hoc ranking score based on the pattern cohesion and the library usage similarity.
  For each fold, we identify a recommendation set of useful libraries for the validation client systems. For each system, we drop half of its libraries and use them as the ground truth. The remaining half is used as input to the recommendation process. This methodology was also used in [77] and mimics the scenario where a developer knows some of the useful libraries but needs assistance to find other relevant libraries.
  For each system that should receive library recommendation, we first identified potentially useful patterns containing at least on library from the ground truth set. Thereafter, we rank the libraries of these patterns according to their recommendation score as defined below:

$$RecScore(L) = \max_{i}\{DSim(L, Lib_i)/Lib_i \in GT\} \tag{7.3}$$

  where *DSim* is the Dependency Similarity in Equation (7.1), and *GT* is the set of libraries conserved as ground truth of the client system that should receive library recommendations.

We evaluate the ranking for both LibCUP and LibRec using two metrics commonly used in recommendation systems for software engineering [7, 75, 76]: (*i*) the recall rate@k, and (*ii*) the Mean reciprocal rank (MRR) as follows. To measure the recall@k, we consider N target systems S that should receive library recommendations. For each system $S_i \in S$, if any of the dropped libraries is found in the top-k list of recommended libraries, we count it as a hit. The recall rate@k is measured by the ratio of the number of hits over the total number N of considered systems. Inspired by the previous studies [7, 75, 76], we choose the k value to be 1, 3, 5, 7, and 10. Formally, the recall rate@k is defined as follows:

$$Recall\ rate@k(S) = \frac{\sum_{\forall S_i \in S} isCorrect(S_i, \text{Top-}k)}{N} \tag{7.4}$$

where the function $isCorrect(S_i, \text{Top-}k)$ returns a value of 1 if at least one of the top-k recommended libraries is in the ground truth set, or 0 otherwise.

The MRR is a statistic measure that is commonly used to evaluate recommendation systems. Let N be the number of systems that should receive recommendations, and $rank_i$ is the rank of the first relevant recommendation for the i^{th} system, then the MRR score is calculated as follows:

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i} \tag{7.5}$$

7.4.4.2 Results for RQ3

Study 3.A: Patterns generalizability. To assess the PUC score variation between the training and validation client systems, we first analyze the average value of their corresponding scores collected from all cross-validation runs. Then, we analyze the distribution of the collected values using the median. The results of this study are summarized in Table 7.III and Figure 7.9.

Table 7.III summarizes the PUC results of the detected patterns in the contexts of training and validation client systems. In the training context, we notice that the average values are high for both LibCUP and LibRec with respectively 77% and 72% of the patterns libraries that are co-used together. For LibCUP, a slight degradation of the PUC value is observed in the context of validation

PUC	LibCUP		LibRec	
	Training context	**Validation context**	**Training context**	**Validation context**
Avg	0.77	0.69	0.72	0.54
Max	0.78	0.78	0.88	0.89
StdDev	0.01	0.05	0.06	0.27

Table 7.III: Average Training and Validation Cohesion of identified usage patterns for LibCUP and LibRec.

client systems. We also notice that the standard deviation values are very low (0.01 and 0.05). These results reflect that, overall, the detected patterns had good PUC in the context of both validation and training client systems. However, for LibRec the achieved average PUC values are significantly lower in the validation context comparing to the training context.

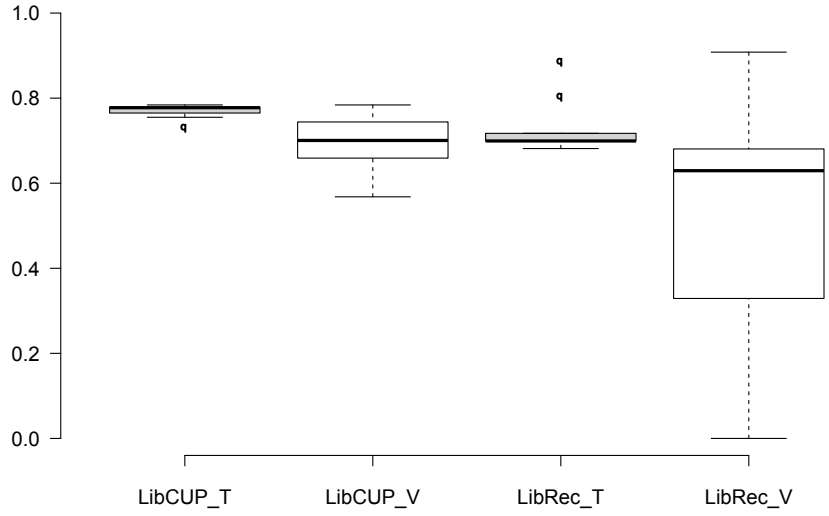


Figure 7.9: PUC results of the identified library usage patterns in the contexts of training (T) and validation (V) clients achieved by each of LibCUP and LibRec.

In more details, the distribution of PUC values for all detected usage patterns in Figure 7.9 confirms the above-mentioned finding. Indeed, the medians and lower quartiles in the context of validation clients remain larger than 66%. Figure 7.9 also provides evidence that the degradation of cohesion values for each inferred pattern is much more visible in the case of LibRec.

In summary, we can say that almost all detected usage patterns achieved by LibCUP retain their informative criteria. Precisely, 75% of detected usage patterns, according to the boxplot's lower quartile, are characterized with a high usage cohesion in both training and validation contexts.

Study 3.B: Library recommendation. Table 7.IV reports the recall rate@k for both LibCUP and LibRec while varying the value of $k \in \{1,3,5,7,10\}$. We notice that, as expected, larger k values achieve higher recall rates for both approaches. More specifically, we can see that when comparing the recall rate, LibCUP performs clearly better in terms of recall@1 and recall@3. However, the starting from $k = 5$, LibRec tends to achieve better results. This indicates that LibCUP is more efficient in recommending correct libraries in the top ranks within the recommendation list when using LibCUP. The MRR results support this observation with a score of 0.15 for LibCUP.

Conversely, good recommendations are achieved for more targeted client systems when using LibRec. This is mainly due to the fact that LibRec's patterns are mainly composed of utility libraries, unlike the LibCUP's patterns which are mainly composed of domain-specific libraries. However, one can notice that recommending utility libraries that are commonly used is less useful in practice. It is also worth mentioning that for each fold, due to the large number of systems (3,800 validation client systems) that require library recommendations, the recall values achieved by both LibCUP and LibRec are still low (below 0.34) as reported in Table 7.IV.

	LibCUP	**LibRec**
Recall@1	0.12	0.01
Recall@3	0.14	0.11
Recall@5	0.15	0.19
Recall@7	0.17	0.27
Recall@10	0.22	0.34
MRR	0.15	0.09

Table 7.IV: Recommendation recall rate and MRR results achieved by both LibCUP and LibRec.

7.5 Discussion

We applied our approach to over 6,000 popular third-party libraries and 38,000 client systems in order to detected possible library usage patterns. The detected patterns should be informative to

help developers in automatically discovering existing library sets and therefore, relieve the developers from the burden of doing so manually. More interestingly, our adaptation of DBSCAN shows high scalability and performance with our large dataset.

The evaluation of our approach took into account the potential generalization of the identified patterns to other client systems and showed that these usage patterns remain informative for other clients.

The application of our technique to detect library usage patterns requires the setting of thresholds that may impact its output. For instance, the max epsilon value in the clustering algorithm controls the cohesion (PUC) strength of the detected patterns. A small value leads to highly cohesive clusters, which means that the detected patterns are more informative. Hence, decreasing the value of this parameter would result in an improvement in cohesion of the detected patterns. However, in this case, the number and the consistency (generality) of the detected patterns could decrease because the highly cohesive detected patterns may not be shared by a large number of clients. To ovoid bothering potential users of our approach with tuning the max epsilon value, we set it to a default value of 0.5 which ensures that the libraries within patterns are at least used more frequently together than separately.

To get a more qualitative sense, we describe one of the inferred patterns identified by our technique that can fulfil the requirements of the case scenario discussed in Section 7.2.1 and that provide useful libraries for potential extensions of the system. The developer would use `scheduler-api` and `mailsender-api` rather than the `quartz` and `commons-email`. This pattern has different cohesion layers, and provides at the first layer, the libraries `sakai -calendar-api` [16] and `sakai-presence-api` [17]. In the second layer, we find three libraries that are added to the pattern, namely `portal-chat` [18], `messageforums-tool` [19] and `mailsender -api` [20]. At the external usage cohesion layer, the `scheduler-api` is added to the pattern. Indeed, these libraries have been frequently co-used in a set of 18 client systems at least in our dataset.

16. mvnrepository.com/artifact/org.sakaiproject.calendar/sakai-calendar-api/
17. mvnrepository.com/artifact/org.sakaiproject.presence/sakai-presence-api/
18. mvnrepository.com/artifact/org.sakaiproject.portal/portal-chat/
19. mvnrepository.com/artifact/org.sakaiproject.msgcntr/messageforums-tool/
20. mvnrepository.com/artifact/org.sakaiproject.mailsender/mailsender-api/

It is worth noticing that we found a trade-off between the usage cohesion of the detected patterns and their generalization. Indeed, another example of more generalizable patterns that was inferred when max epsilon reached a relatively high value is the one formed in his core layer with some libraries of the Spring framework such as `spring-beans`[21], `spring-context`[22] and `spring-orm`[23]. Then, in the second layer, we find some libraries of the Hibernate framework such as the `hibernate-entitymanager`[24] and `hibernate-annotations`[25]. Finally, in the third layer, we find some json libraries such us the `jackson- databind`[26]. The pattern continue growing until including some utility libraries of logging and testing. These libraries have been co-used in a set of hundreds of client systems.

To better assist the developers in exploring potential library usage patterns, LibCUP provides an interactive and friendly-user visualization tool. The visualization takes as input the inferred patterns and distributes them through their different layers.

7.6 Conclusion

Third-party library reuse has become vital in modern software development. The number of libraries provided on the Internet is exponentially growing, which would provide several reuse opportunities. In this chapter, we introduced our automated approach to detect multi-level library usage patterns, as a collection of libraries that are commonly used together by client systems, distributed through multiple levels of cohesion. We evaluated our approach on a large dataset of 6,638 popular libraries from Maven repository, and a large population of 38,000 client systems from Github, and we compared its results to those of a state-of-the-art approach. The results indicate that our approach gives a comprehensive overview on third-party library usage patterns. The obtained usage patterns exhibit high usage cohesion with an average of 77% , and could be generalizable to other systems. Automatically detecting library usage patterns would support developers in enhancing the library space discovery, and attract their attention the missed reuse opportunities.

21. mvnrepository.com/artifact/org.springframework/spring-beans
22. mvnrepository.com/artifact/org.springframework/spring-context
23. mvnrepository.com/artifact/org.springframework/spring-orm
24. mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager
25. mvnrepository.com/artifact/org.hibernate/hibernate-annotations
26. mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind

As future work, we are planning to unify our library-level usage pattern detection with method-level usage pattern detection techniques in order to provide a comprehensive package for developers supporting them in understanding and reusing third-party libraries.

Chapter 8

Conclusion

In this chapter, we summarize the results and conclusions of this thesis. We also discuss the limitations and future research directions.

8.1 Contributions

The main objective of this thesis was to develop a holistic approach to help software developers during the early step of library usage and learning process. To this end, we proposed different techniques to deal with the library usability challenges at different levels of reuse. Our three main contributions consider library usability respectively at the method level, across complementary API methods, and within the combination of different complementary libraries.

The first contribution of our thesis, described in Chapter 3, helps developers to avoid the additional cost for debugging and correction, due to the non-respect of method usage constraints. We focused on four types of constraints that have been reported on several studies on API learning and usage obstacles. The considered constraints are related to API method parameters, and our detection algorithms succeeded in identifying the majority of the considered usage constraints. Typically, researchers and practitioners try to identify usage constraint through the analysis of library documentation or multiple client applications that cover the entire target library, which drastically limit the applicability of existing techniques. To support developers and relieve them from the burden of doing so, our technique is based only on the analysis of the library source code, and achieved, in general, 100% of precision and 94% of recall.

API methods are generally used within client programs jointly with other methods of the API. The second contribution of our thesis is related to this fact. We proposed three techniques to identify API usage patterns from different information sources. The identified patterns help developers to better learn common ways to use the API complementary methods.

As described in Chapter 4 for the first technique, we applied multi-level clustering to mine usage patterns that reflect interfering usage scenarios of the API. The technique is based on the analysis of

the frequency and consistency of co-usage relations between the API methods within client programs. Our technique was able to detect usage patterns that are highly consistent and highly cohesive across a considerable variability of client programs. The efficiency of this technique relies on the redundant coverage from different client programs. Indeed, an API method should be used by several client methods to infer usage patterns. This limitation is shared by all existing work around making use of multiple API clients programs. This leads us to consider non client-based techniques as a potential alternative.

Considering only client programs to identify API usage patterns is a strong constraint as the client programs are not always available. In Chapter 5, we proposed a technique that supports pattern mining using only the library source code. Our technique is based on the analysis of structural and semantic dependencies of API methods within the library code itself. We assume that API methods that manipulate the same object could be complementary in their contribution to a functionality. We also assume that API methods can be related to the same domain functionality if they share similar vocabulary. The proposed technique is not an alternative to client-based ones. It is rather a solution when client programs are not available. It can be applied for non popular libraries and even for "new" APIs. The proposed technique can infer usage patterns with a precision comparable to those of client-based approaches.

Despite these encouraging results, there is still room for improvement. In Chapter 6, we proposed our third technique for mining API usage pattern. The propose technique combines library-based and client-based heuristics. Indeed, the library-based technique infers patterns that could apply to any client program, but some of the inferred patterns do not reflect a real usage scenario. Conversely, client-based technique infers usage patterns with actual instances in the used client programs, but inferred patterns are limited to the usage scenarios in the selected client. We study which form of combination is better to achieve the best tradeoff between library-based and client-based heuristics. We evaluated the three techniques through the usage of four APIs having up to 22 client programs per API. The obtained results show that the cooperative approach allows taking advantage at the same time from the precision of client-based technique and from the generalizability of library-based techniques.

From another perspective, developers spend considerable time and effort to identify libraries that are useful and relevant for the implementation of their software. Worse yet, developers may tend to

implement most of their features from scratch if they are unaware of the existence of libraries reuse opportunities. In Chapter 7, our third contribution takes the form of a technique to automatically identify third-party library usage patterns, as a collection of libraries that are commonly used together when analyzing open source repositories. We evaluated our technique on a large dataset of around 6,000 popular libraries from Maven Central Repository and around 38,000 of their client systems from the Github repository. Our technique is able to detect the majority of highly consistent and cohesive patterns.

Finally, we believe that the inferred constraints and patterns can easily be integrated in an API exploration process or to enhance API documentation, or even to recommend API elements.

8.2 Future Perspective

In this section, we discuss some open research directions and broader applications related to our proposal for future work.

As we previously said, the inferred patterns by our approach are valuable for early steps of library learning and usage process. However, when a developer gains experience with a library and takes control of it, he may need more complex details on how to use the API. We are working to extend our initial approach to support complex API usage patterns represented as linear temporal logic formulas (LTL). The LTL specification can express different types of complex API usage properties, such as safety properties that state the API calls that never happens together, or sequential properties that state the temporal order between API calls. This kind of specification can be inferred from the execution trace, collected for other software systems in similar contexts.

An important future direction consists of adapting our approach to detect API misuse. Over the last few years, researchers proposed a multitude of techniques for bug-detection in the context of library reuse. These techniques commonly mine API usage patterns from different API client programs and find violations of those patterns, assuming that they may correspond to bugs. Such pattern violations are called API misuses. As part of our future work, we intend to introduce a completely deferent technique for API misuse detection. The technique will be based on the image similarity of API usage glyphs. For the human-computer interaction community, glyphs [39] are very popular and generally used to automatically generated visual icons, employed as object identification

technique. Unfortunately, glyphs are rarely used to address software engineering problems. We plan to generate API usage glyph dataset from trusted API client systems, and apply content-based image retrieval techniques [38] to detect API misuse based on the glyphs' image shapes and textures similarity.

With tasks intended for experienced developers, such as API misuse related bug detection or complex specification inference, we may need to use more complicated data representations. As part of our future research on complex API usage tasks, we plan to study the usage of graph-based object usage models (Groums) [52]. Groums can hold special nodes to represent control structures, such as loops and conditionals. Furthermore, edges not only represent sequencing constraints, but also data dependencies. Our intention will be to compare our multi level pattern to groums when used as input for API misuse or complex specification inference technique.

From another perspective, our clustering-based technique for pattern mining can be adapted to mine software components from object-oriented APIs, or to improve the service interface modularization for service-oriented application.

Bibliography

[1] The java.awt package and all its subpackages (12 public packages in total). URL http://docs.oracle.com/javase/7/docs/api/.

[2] The jakarta commons httpclient component. URL http://hc.apache.org/httpclient-3.x/.

[3] The java.security package and all its subpackages (5 public packages in total). URL http://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html.

[4] The swing api (18 public packages in total). URL http://docs.oracle.com/javase/7/docs/technotes/guides/swing/.

[5] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[6] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gael Gueheneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[7] Iman Avazpour, Teerat Pitakrat, Lars Grunske, and John Grundy. Dimensions and Metrics for Evaluating Recommendation Systems. In Martin P Robillard, Walid Maalej, Robert J Walker, and Thomas Zimmermann, editors, *Recommendation Systems in Software Engineering*, chapter 10, pages 245–273. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-45134-8. doi: 10.1007/978-3-642-45135-5_10. URL http://dx.doi.org/10.1007/978-3-642-45135-5{_}10.

[8] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):4, 2014.

[9] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In

Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1589–1598. ACM, 2009.

[10] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. Fruit: Ide support for framework understanding. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 55–59. ACM, 2006.

[11] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

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

[13] Raymond PL Buse and Westley R Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 273–282. ACM, 2008.

[14] Barthélémy Dagenais and Martin P Robillard. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):19, 2011.

[15] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

[16] Uri Dekel and James D Herbsleb. Improving api documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 320–330. IEEE Computer Society, 2009.

[17] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[18] Ekwa Duala-Ekoko and Martin P Robillard. Using structure-based recommendations to facilitate discoverability in apis. In *European Conference on Object-oriented Programming*, pages 79–104. Springer, 2011.

[19] 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*, pages 266–276. IEEE Press, 2012.

[20] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.

[21] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[23] Martin Feilkas and Daniel Ratiu. Ensuring well-behaved usage of apis through syntactic constraints. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 248–253. IEEE, 2008.

[24] William Frakes, Kyo Kang, et al. Software reuse research: Status and future. 2005.

[25] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 15–24. ACM, 2010.

[26] John E Gaffney Jr and RD Cruickshank. A general economics model of software reuse. In *Proceedings of the 14th international conference on Software engineering*, pages 327–337. ACM, 1992.

[27] Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An experimental investigation on the effects of context on source code identifiers splitting and expansion. *Empirical Software Engineering*, 19(6):1706–1753, 2014.

[28] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44. IEEE, 2010.

[29] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from java classes. In *European Conference on Object-Oriented Programming*, pages 431–456. Springer, 2003.

[30] William C Hill and James D Hollan. History-enriched digital objects: Prototypes and policy issues. *The Information Society*, 10(2):139–145, 1994.

[31] Reid Holmes, Robert J Walker, and Gail C Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32 (12):952–970, 2006.

[32] Daqing Hou and Lin Li. Obstacles in using frameworks and apis: An exploratory study of programmers' newsgroup discussions. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 91–100. IEEE, 2011.

[33] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *European Conference on Object-Oriented Programming*, pages 27–51. Springer, 2009.

[34] Sae Young Jeong, Yingyu Xie, Jack Beaton, Brad A Myers, Jeff Stylos, Ralf Ehret, Jan Karstens, Arkin Efeoglu, and Daniela K Busse. Improving documentation for esoa apis through user studies. In *International Symposium on End User Development*, pages 86–105. Springer, 2009.

[35] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*, pages 284–291. IEEE, 1991.

[36] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Adding examples into java documents. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 540–544. IEEE, 2009.

[37] Andrew J Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[38] Michael S Lew, Nicu Sebe, Chabane Djeraba, and Ramesh Jain. Content-based multimedia information retrieval: State of the art and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 2(1):1–19, 2006.

[39] John P Lewis, Ruth Rosenholtz, Nickson Fong, and Ulrich Neumann. Visualids: automatic distinctive icons for desktop interfaces. *ACM Transactions on Graphics (TOG)*, 23(3):416–423, 2004.

[40] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[41] Chang Liu, En Ye, and Debra J Richardson. Ltrules: an automated software library usage rule extraction tool. In *Proceedings of the 28th international conference on Software engineering*, pages 823–826. ACM, 2006.

[42] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 296–305. ACM, 2005.

[43] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[44] Walid Maalej and Martin P Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[45] Amir Michail. Data mining library reuse patterns in user-selected applications. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 24–33. IEEE, 1999.

[46] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

[47] Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*, pages 2–25. Springer, 2010.

[48] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[49] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[50] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[51] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. A graph-based approach to api usage adaptation. In *ACM Sigplan Notices*, volume 45, pages 302–321. ACM, 2010.

[52] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392. ACM, 2009.

[53] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 133–141. ACM, 2002.

[54] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 815–825. IEEE Press, 2012.

[55] Mikhail Perepletchikov, Caspar Ryan, and Keith Frampton. Cohesion metrics for predicting maintainability of service-oriented software. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 328–335. IEEE, 2007.

[56] Mikhail Perepletchikov, Caspar Ryan, and Zahir Tari. The impact of service cohesion on the analyzability of service-oriented software. *IEEE Transactions on Services Computing*, 3(2): 89–103, 2010.

[57] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 925–935. IEEE Press, 2012.

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

[59] Martin P Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[60] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5): 613–637, 2013.

[61] Mohamed Aymen Saied and Houari Sahraoui. A cooperative approach for combining client-based and library-based api usage pattern mining. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[62] Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari Sahraoui. Could we infer unordered api usage patterns only using the library source code? In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 71–81. IEEE Press, 2015.

[63] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. Mining multi-level api usage patterns. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 23–32. IEEE, 2015.

[64] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. An observational study on api usage constraints and their documentation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 33–42. IEEE, 2015.

[65] Giuseppe Scanniello and Andrian Marcus. Clustering support for static concept location in source code. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 1–10. IEEE, 2011.

[66] Hinrich Schütze. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, 2008.

[67] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[68] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 101–110. IEEE, 2011.

[69] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 71–80. IEEE, 2011.

[70] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, pages 529–539. IEEE Computer Society, 2007.

[71] Jeffrey Stylos and Brad A Myers. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 195–202. IEEE, 2006.

[72] Jeffrey Stylos and Brad A Myers. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008.

[73] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A Myers. Improving api documentation using api usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–126. IEEE, 2009.

[74] Laszlo Szathmary, Amedeo Napoli, and Sergei O Kuznetsov. Zart: A multifunctional itemset mining algorithm. 2006.

[75] Chakkrit Tantithamthavorn, Akinori Ihara, and Ken-ichi Matsumoto. Using co-change histories to improve bug localization performance. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, pages 543–548. IEEE, 2013.

[76] Chakkrit Tantithamthavorn, Rattamont Teekavanich, Akinori Ihara, and Ken-ichi Matsumoto. Mining a change history to quickly identify bug locations: A case study of the eclipse project. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 108–113. IEEE, 2013.

[77] Ferdian Thung, LO David, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE 2013): Proceedings: Koblenz, Germany, 14-17 October 2013*, pages 182–191, 2013.

[78] Christoph Treude and Martin P Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, pages 392–403. ACM, 2016.

[79] Gias Uddin, Barthélémy Dagenais, and Martin P Robillard. Temporal analysis of api usage concepts. In *Proceedings of the 34th International Conference on Software Engineering*, pages 804–814. IEEE Press, 2012.

[80] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 79–90. IEEE, 2004.

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

[82] Wei Wang and Michael W Godfrey. Detecting api usage obstacles: A study of ios and android developer questions. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 61–64. IEEE, 2013.

[83] Weixin Wang, Hui Wang, Guozhong Dai, and Hongan Wang. Visualization of large hierarchical data by circle packing. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 517–520. ACM, 2006.

[84] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[85] John Whaley, Michael C Martin, and Monica S Lam. Automatic extraction of object-oriented component interfaces. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 218–228. ACM, 2002.

[86] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836. IEEE Press, 2012.

[87] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.