

**EXTRACTING EXAMPLES FOR API USAGE  
PATTERNS**



HUDSON SILVA BORGES

**EXTRACTING EXAMPLES FOR API USAGE  
PATTERNS**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais – Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: MARCO TÚLIO DE OLIVEIRA VALENTE

Belo Horizonte

Março de 2014



HUDSON SILVA BORGES

**EXTRACTING EXAMPLES FOR API USAGE  
PATTERNS**

Dissertation presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais – Departamento de Ciência da Computação in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

ADVISOR: MARCO TÚLIO DE OLIVEIRA VALENTE

Belo Horizonte

March 2014

© 2014, Hudson Silva Borges.  
Todos os direitos reservados.

Borges, Hudson Silva

B732e      Extracting Examples for API Usage Patterns /  
Hudson Silva Borges. — Belo Horizonte, 2014  
xxi, 102 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de  
Minas Gerais – Departamento de Ciência da  
Computação

Orientador: Marco Túlio de Oliveira Valente

1. Computação – Teses. 2. Engenharia de software –  
Teses. 3. Software – Reutilização – Teses. I.  
Orientador. II. Título.

CDU 519.6\*32(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


## FOLHA DE APROVAÇÃO

Extracting examples for API usage patterns

**HUDSON SILVA BORGES**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. EDUARDO MAGNO LAGES FIGUEIREDO  
Departamento de Ciência da Computação - UFMG

  
PROF. HEITOR AUGUSTUS XAVIER COSTA  
Departamento de Ciência da Computação - UFLA

  
PROF. HUMBERTO TORRES MARQUES NETO  
Instituto de Ciências Exatas e Informática - PUC/MG

Belo Horizonte, 20 de março de 2014.





# Agradecimentos

Agradeço a todas as pessoas que me ajudaram a chegar até aqui, desde o primário até o término do mestrado.

Agradeço especialmente aos meus pais, Diva e Hugo, que sempre estiveram ao meu lado apoiando minhas decisões e dando todo suporte necessário.

Agradeço a minha amada noiva, Elisângela, que foi a pessoa que esteve ao meu lado a todo momento durante esses dois anos. Agradeço também a minha segunda família—especialmente Elir, Luzia, Elir Jr. e Eliel—que também sempre me apoiaram.

Agradeço ao professor Marco Túlio por todo apoio durante esses dois anos de mestrado. Sempre atencioso e disponível, representou uma das importantes bases que me apoiaram durante todo curso.

Agradeço aos meus amigos do LLP por tornar mais agradável a convivência diária no departamento.

Agradeço ao Programa de Pós-graduação em Ciência da Computação (PPGCC) pelo suporte financeiro e acadêmico.

Agradeço ao CNPq pelo apoio financeiro.



# Resumo

Atualmente, sistemas computacionais reusam cada vez mais funcionalidades providas por bibliotecas e *frameworks*, que são acessados por meio de *Application Programming Interfaces* (APIs). Contudo, o reuso de tais recursos requer um esforço não trivial. Estudos empíricos recentes mostram que a falta de exemplos de uso é o maior obstáculo para o uso de APIs atualmente. Por outro lado, o crescente aumento do uso de APIs por sistemas clientes produz dados que podem ser usados para melhorar o aprendizado de tais APIs. Esta dissertação de mestrado apresenta uma extensão da plataforma APIMiner—chamada APIMiner 2.0—que instrumenta documentações tradicionais de APIs com exemplos de código fonte relativos a métodos que são frequentemente chamados em conjunto. Essa abordagem estende a plataforma APIMiner 1.0 usando algoritmos de mineração de regras de associação para extrair padrões de uso e incluir exemplos de uso mais úteis. Particularmente, os exemplos são sumarizados usando um algoritmo de *slicing* estático que considera várias estratégias propostas recentemente na literatura visando melhorar a legibilidade. Uma instância da plataforma APIMiner 2.0 foi implementada para a API de desenvolvimento do Android. Além disso, foi conduzida uma profunda análise do uso da API do Android por centenas de sistemas clientes, um estudo de campo envolvendo o uso do APIMiner 2.0 por desenvolvedores profissionais e um estudo controlado com 29 participantes. Para esses estudos, a plataforma minerou 1.952 padrões de uso em 396 sistemas clientes de código aberto. Também foram extraídos 102.442 exemplos para métodos únicos da API e 184.821 exemplos para padrões de uso em 151 sistemas clientes de código aberto. Em um período de cinco meses, a instância Android APIMiner 2.0 recebeu 32.335 visitas e 5.721 requisições de exemplos de uso para métodos únicos da API e padrões de uso. Além disso, foi conduzido um estudo controlado incluindo duas tarefas de manutenção corretivas que foram implementadas por 29 participantes. Finalmente, foi observado que a plataforma APIMiner 2.0 foi particularmente útil para participantes que não possuem conhecimentos em Android.

**Palavras-chave:** API, compreensão de programas, padrões de uso, exemplos de código fonte, regras de associação, slicing de programas.



# Abstract

In modern software development, systems increasingly reuse functionality provided by libraries and frameworks, which are accessed by means of Application Programming Interfaces (APIs). However, reusing current APIs generally requires a nontrivial effort. Recent empirical studies show that the lack of usage examples is a major obstacle for using modern APIs. On the other hand, the increasing use of APIs by client systems produces data that can be used to improve APIs learning. This master dissertation presents an extension of the APIMiner platform—called APIMiner 2.0—that supports the instrumentation of traditional API documents with source code examples on methods that are frequently called together. Our approach extends the APIMiner 1.0 platform by relying on association rules algorithms to extract usage patterns and to include more useful examples. Particularly, the provided examples are summarized using a slicing algorithm that considers several strategies recently proposed in the literature to increase readability. An instance of the APIMiner 2.0 platform was implemented for the Android API. Furthermore, we conducted an in-depth analysis on the use of the Android API by hundreds of client systems, a field study involving the use of APIMiner 2.0 by professional developers, and a controlled study with 29 subjects. In Android APIMiner 2.0, we mined 1,952 usage patterns by analyzing 396 open-source Android systems. We extracted 102,442 examples for single API methods and 184,821 examples for usage patterns from 151 open-source Android systems. Moreover, Android APIMiner 2.0 received a total of 32,335 visits and 5,721 example requests for single API methods and usage patterns in a five-month period. Finally, we conducted a controlled study including two corrective maintenance tasks that were performed by 29 subjects. We observed that APIMiner 2.0 was particularly useful for subjects who do not have knowledge on Android.

**Keywords:** API, program comprehension, usage patterns, source code examples, association rules, program slicing.



# List of Figures

1.1	APIMiner 2.0 Overview . . . . .	4
2.1	JavaDoc instrumented with an "Example Button" . . . . .	14
2.2	Database Representations . . . . .	20
3.1	APIMiner 2.0 architecture . . . . .	24
3.2	Feedback buttons . . . . .	35
3.3	JavaDoc for the SQLiteDatabase class as instrumented by APIMiner 2.0 . . . . .	38
3.4	Example window in APIMiner 2.0 . . . . .	38
3.5	Full source code dialog window . . . . .	39
3.6	API usage pattern example . . . . .	40
4.1	New main page of Android APIMiner 2.0 . . . . .	41
4.2	Ratio of methods with at least an API call, among the systems in the Mining Dataset . . . . .	44
4.3	API calls/method, considering all methods of each system and only methods with at least a single API call . . . . .	45
4.4	Distribution of the useful transactions . . . . .	47
4.5	Percentage of useful transactions per project . . . . .	48
4.6	Number of rules by varying the minimum support value . . . . .	52
4.7	API methods coverage by varying the minimum support value . . . . .	52
4.8	Number of API method calls in the examples . . . . .	54
4.9	Distribution of the examples size . . . . .	57
4.10	Number of visits per week . . . . .	60
4.11	Number of examples provided per week . . . . .	61
4.12	Relation between the number of examples requests for usage pattern and single API method . . . . .	63
4.13	Screenshots from More Aqui . . . . .	66
4.14	Subjects' expertise . . . . .	68

4.15 Professional expertise of the subjects (in years) . . . . . 68



# List of Tables

3.1	API elements and their attributes . . . . .	26
4.1	Top five projects with more useful transactions . . . . .	48
4.2	Top five most common transaction items . . . . .	49
4.3	Intra-class and Inter-class calls in useful transactions . . . . .	50
4.4	Top five API methods most called in useful transactions . . . . .	50
4.5	Top five API classes most used in useful transactions . . . . .	50
4.6	Top five methods with more rules and their average support . . . . .	53
4.7	Top five rules with higher support . . . . .	54
4.8	API methods with more examples . . . . .	55
4.9	Methods that most appeared in all examples . . . . .	55
4.10	Classes that most appeared in all examples . . . . .	56
4.11	Packages that most appeared in all examples . . . . .	56
4.12	Quality metrics (all examples) . . . . .	56
4.13	Top ten countries in visits . . . . .	61
4.14	Top ten methods with most requests for examples (#1) and their number of available examples (#2) . . . . .	62
4.15	Top ten methods with most requests for usage patterns #1, their number of single example requests #2, and usage patterns #3 . . . . .	63
4.16	Performance of the subjects in the experiment . . . . .	70
4.17	Number of subjects who completed the tasks . . . . .	71
4.18	Expertise on Android development of the subjects who concluded the <b>Task 1</b>	71
4.19	Expertise on Android development of the subjects who concluded the <b>Task 2</b>	71
4.20	Time spent in the tasks (in minutes) . . . . .	72
A.1	Mining Dataset . . . . .	85
A.2	Examples Dataset . . . . .	94



# Contents

<b>Agradecimentos</b>	<b>ix</b>
<b>Resumo</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Description . . . . .	2
1.3 Contributions . . . . .	3
1.4 Organization . . . . .	4
<b>2 Background</b>	<b>7</b>
2.1 Introduction . . . . .	7
2.2 IDE-based Recommendation Systems . . . . .	9
2.2.1 Vertical Code Completion . . . . .	9
2.2.2 Mining API Code Snippets . . . . .	10
2.3 API Documentation . . . . .	11
2.3.1 APIMiner 1.0 . . . . .	12
2.3.2 eXoaDocs . . . . .	14
2.4 Examples Quality . . . . .	16
2.4.1 A Study of Programming Q&A in StackOverflow . . . . .	16
2.4.2 Synthesizing API Usage Examples . . . . .	17
2.5 Association Rules . . . . .	18
2.6 Program Slicing . . . . .	21

2.7	Final Remarks . . . . .	22
<b>3</b>	<b>Proposed Solution</b>	<b>23</b>
3.1	Introduction . . . . .	23
3.2	Source Code Analyzer . . . . .	25
3.3	Patterns Analyzer . . . . .	26
3.4	Examples Extractor . . . . .	27
3.4.1	Summarization Algorithm . . . . .	29
3.4.2	Readability Improvements . . . . .	30
3.4.3	Removing Similar Examples . . . . .	33
3.5	Ranking Engine . . . . .	34
3.5.1	Ranking Examples for Single API Methods . . . . .	34
3.5.2	Ranking Usage Patterns . . . . .	36
3.6	JavaDoc Weaver . . . . .	37
3.6.1	Example Button . . . . .	37
3.6.2	Examples Presentation . . . . .	38
3.6.3	Usage Patterns Interface . . . . .	39
3.7	Final Remarks . . . . .	40
<b>4</b>	<b>Evaluation</b>	<b>41</b>
4.1	Overview . . . . .	41
4.2	Android API . . . . .	42
4.3	Android APIMiner 2.0 . . . . .	43
4.3.1	Dataset . . . . .	43
4.3.2	Transactions . . . . .	46
4.3.3	Association Rules . . . . .	51
4.3.4	Examples . . . . .	53
4.3.5	Usage Patterns . . . . .	57
4.4	Field Study . . . . .	60
4.5	User Study . . . . .	64
4.5.1	Study Setup . . . . .	65
4.5.2	Experiment Execution . . . . .	67
4.5.3	Experiment Results . . . . .	69
4.6	Threats to Validity . . . . .	72
4.7	Final Remarks . . . . .	73
<b>5</b>	<b>Conclusion</b>	<b>75</b>
5.1	Contributions . . . . .	75

5.2	Comparison with Related Work . . . . .	76
5.2.1	IDE-based Recommendation Systems . . . . .	76
5.2.2	API Documentation . . . . .	77
5.3	Future Work . . . . .	78
	<b>Bibliography</b>	<b>79</b>
	<b>Appendix A Android APIMiner 2.0 Datasets</b>	<b>85</b>
	<b>Appendix B Forms</b>	<b>99</b>
B.1	Subject Characterization Form . . . . .	99
B.2	Tutorial Task Form . . . . .	100



# Chapter 1

## Introduction

### 1.1 Motivation

In modern software development, systems often reuse functionality provided by libraries and frameworks, which are accessed by means of Application Programming Interfaces (API). This form of reuse brings numerous benefits such as (i) reduction of costs and time; (ii) increase the focus of developers on the system's core requirements since they do not need to re-implement the services provided by an API; and (iii) increase on software quality, assuming the API is implemented by experts.

However, reusing current APIs generally requires nontrivial effort. Robillard and DeLine [2010] conducted an exploratory survey and identified inadequate learning resources as a critical obstacle for learning new APIs. More specifically, they identified that inadequate or insufficient documentation is the most severe obstacle faced by developers. Moreover, recent empirical studies indicate that low-level documentations—which usually are automatically generated from the source code and are limited to document the signature of the API methods—represent one of the major obstacles in the API learning process [McLellan et al., 1998; Robillard, 2009; Robillard and DeLine, 2010; Hou and Li, 2011; Buse and Weimer, 2012; Duala-Ekoko and Robillard, 2012]. In common, such studies share the finding that source code examples are a central instrument to make more productive the use of APIs.

On the other hand, the increasing use of APIs by client systems produces data that can be used to improve APIs learning. For example, Uddin et al. [2011] and Uddin et al. [2012] proposed an approach to identify cohesive subsets of an API that are often introduced in client programs—called Temporal API Usage Patterns. These patterns can be used to inform developers of the next programming steps required when using an API. Silva Jr et al. [2012] presented an approach that relies on the source code of a

project to mine and suggest code sequences. Similarly, Hsu and Lin [2011] propose an approach that suggests API methods that are frequently called together (in sequence or not).

## 1.2 Problem Description

In general, the first resource accessed by developers when learning an API is its documentation [Robillard and DeLine, 2010]. However, common API documents are automatically generated from annotations in the code, like JavaDoc for Java and PyDoc for Python. Robillard and DeLine [2010] classify such documentations as low-level, because they are typically restricted to briefly describe methods and variables and do not provide a conceptual view of the API. These factors constitute the main obstacle when developers need to learn a new API. Moreover, Mar et al. [2011] argue that making a documentation that guides programmers on using an API correctly is a critical job for API developers.

A key resource frequently used for ensuring the effectiveness of API documentations is code examples [Nykaza et al., 2002; Robillard and DeLine, 2011]. For this reason, some approaches have been proposed to automatically instrument traditional documentations with source code examples, such as APIMiner [Montandon, 2013] and EXOADOCS [Kim et al., 2009]. Particularly, these approaches rely on automatic code summarization techniques for extracting code examples. As advantages, these systems have a wider reachability (because they are independent from other platforms) and greater scalability (because their results can be pre-processed). As a key disadvantage, the provided examples are limited to single API methods.

On the other hand, code examples can also be provided by IDE-based recommendation systems. Examples include VCC [Silva Jr et al., 2012] and MACs [Hsu and Lin, 2011], which take advantage of different data sources available on IDEs to produce examples. In such approaches, it is common the usage of data mining techniques to identify patterns on the analyzed data to make recommendations. However, the provided examples are not for documentation purposes due to their dependence on the IDE context.

To the best of our knowledge, there is no system centered on traditional API documents that automatically extracts source code examples for API usage patterns.



## 1.3 Contributions

The central objective of this master work is to introduce a new feature in the APIMiner platform, which is a system that provides source code examples for single API methods calls [Montandon, 2013; Borges et al., 2013; Montandon et al., 2013]. This new feature relies on data mining techniques to identify API usage patterns and to provide real source code examples that follow the mined patterns. Moreover, several strategies recently proposed in the literature are used to increase the readability of the provided examples.

For example, it is common the co-occurrence of methods like `open()` and `close()` in client systems. In this case, the usage of data mining techniques allows us to derive an usage pattern such as `open() ⇒ close()`. As another example, in Java a common usage pattern related to the SQL API is as follows:

`java.sql.DriverManager.getConnection(String)` and `java.sql.Connection.close()`

In this sense, Listing 1.1 presents an example for this usage pattern, with readability improvements—use of comments for abstract initialization (line 1) and comments in empty blocks (line 4).

Listing 1.1: Example for the usage pattern `DriverManager.getConnection(String)`  $\Rightarrow$  `Connection.close()`

---

```
1 String url; // initialized previously
2 Connection conn = DriverManager.getConnection(url);
3 try {
4     // some code
5 } finally {
6     conn.close();
7 }
```

---

Figure 1.1 illustrates the approach proposed in this master dissertation. This approach—called APIMiner 2.0—initially identifies all methods provided by an API of interest. Then, it mines API usage patterns from a set of client systems. Based on these patterns, APIMiner 2.0 extracts source code examples from the client systems with readability improvements for single API method calls and also for the identified usage patterns. Finally, the JavaDoc documentation is instrumented with the examples.

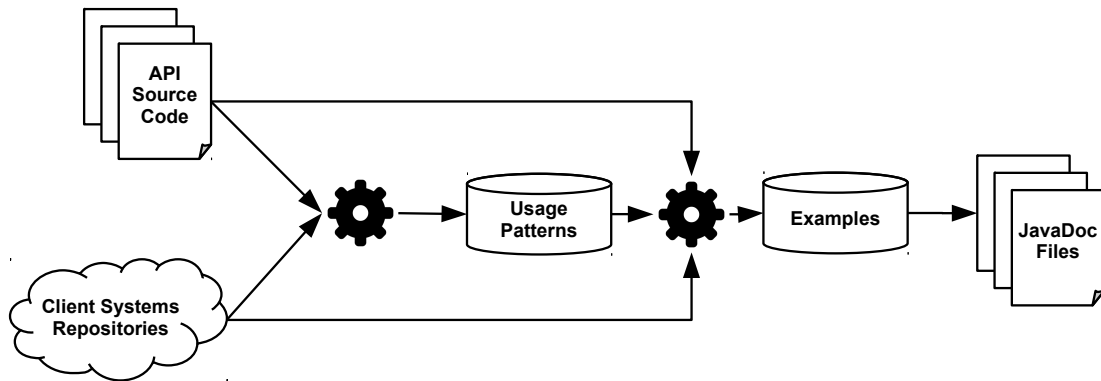


Figure 1.1: APIMiner 2.0 Overview

The following characteristics highlights the main differences between APIMiner 2.0 and existing API documentations systems:

- The mined API usage patterns contribute to the extraction of representative source code examples. Recent studies show that developers prefer examples that involve more than one method call [Robillard and DeLine, 2010; Buse and Weimer, 2012; Nasehi et al., 2012]. In this sense, APIMiner 2.0 presents usage examples including API methods frequently called together in real applications;
- Automatic summarization techniques based on static slicing in many cases may generate examples with readability problems. In order to improve the examples extracted using slicing, we propose an algorithm that includes several strategies recently proposed in the literature to increase the readability of API examples;
- A criterion for ranking the extracted examples that prioritizes examples that have no compilation errors.

To evaluate the proposed approach, we implemented a particular instance for the Android API. In this instance, we used APIMiner 2.0 to extract 287,263 source code examples and 1,672 usage patterns. Moreover, we also report a field study, with real data collected after APIMiner 2.0 was used by professional Android developers. Additionally, we conducted a controlled experiment involving 29 subjects that performed maintenance tasks in a small Android application with the help of APIMiner 2.0.

## 1.4 Organization

This master dissertation is organized in four chapters, which are described next:

- Chapter 2 describes background themes related to the central goal of this master dissertation. Basically, we cover how recommendation systems are currently used in software engineering and how such systems can be classified. Moreover, we explain in details some recent studies highlighting key quality properties of API usage examples. We also briefly present the data mining techniques and the static slicing algorithm used by APIMiner 2.0;
- Chapter 3 presents the solution proposed in this master dissertation. In this chapter, we present the central ideas behind our solution and we also highlight the main differences between the proposed solution and its first version. We also describe in details how we improve the summarization algorithm used by APIMiner 1.0 to extract examples for API usage patterns and also to extract examples with higher quality;
- Chapter 4 evaluates a particular instance of the proposed solution for the Android API, called Android APIMiner 2.0. Initially, we characterize this particular instantiation. Furthermore, we describe a field study conducted using this instance and a controlled experiment;
- Chapter 5 presents the contributions and limitations of this work. Moreover, the chapter also suggests possible future work.



# Chapter 2

## Background

### 2.1 Introduction

Nowadays, software development requires that developers deal with a huge amount of information and technologies in order to maximize their productivity and improve the quality of their products. In this context, the idea of supporting software development with recommendations systems is a promising one [Happel and Maalej, 2008].

The organizers of the 3rd ACM International Conference on Recommender Systems proposed a general definition of a recommendation system [RecSys, 2009], which was also cited by Robillard et al. [2010]:

Recommendation systems are software applications that aim to support users in their decision-making while interacting with large information spaces. They recommend items of interest to users based on preferences they have expressed, either explicitly or implicitly. The ever-expanding volume and increasing complexity of information [...] has therefore made such systems essential tools for users in a variety of information seeking [...] activities. Recommendation systems help overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance.

To help developers to navigate in large information spaces (e.g., finding a particular class from a library) many Recommendation Systems for Software Engineering (RSSEs) are emerging [Robillard et al., 2010]. Basically, RSSEs are software applications that provide information items estimated to be valuable for a software engineering task in a given context.

Robillard et al. [2010] considered three main design dimensions for RSSEs:

- **Nature of the context.** The recommendation context is the most important of the dimensions of RSSEs, because it characterizes the environment that the developer is present. The inputs to RSSEs are contextual information, which can be explicit, implicit, or a hybrid of these strategies. Explicit context is based on user-interfaces interactions, like entering text, selecting elements in the code or dragging and dropping elements on widgets. Basically, this strategy is appropriate for contexts where it is difficult to detect the user's information. On the other hand, implicit contexts do not require interactions with users, i.e., the user's information can be automatically gathered, from historical repositories. Finally, the hybrid strategy combines the previous strategies when none is sufficient to gathering the necessary information;
- **Recommendation Engine.** To make recommendations, RSSEs must analyze more than contextual data and include additional data as means to provide accurate recommendations. For example, Mining Software Repositories (MSR) is a promising approach for retrieving these additional data. In addition, RSSEs should provide a ranking mechanism that ideally puts the items most valuable to the users at the top of the presented recommendations;
- **Output mode:** The recommendations provided by RSSEs can be triggered by an user request (e.g., clicking in a button) or delivered without an explicit request (e.g., the recommendations shown while programmers are coding). Moreover, the recommendations can be presented in batch (set of recommendations about a task) or inline (annotations atop of artifacts).

RSSEs typically provide recommendations related to software development artifacts, particularly source code, but they can also provide recommendations for many other aspects of software development, such as quality measures, tools, project managers, and other software engineering tasks [Robillard et al., 2010]. For instance, Expertise Browser is a system that analyzes past changes in code and other artifacts to recommend software experts to consult [Mockus and Herbsleb, 2002]. The JMove tool relies on the set of static dependencies established by a method to recommend move method refactorings [Sales et al., 2013]. The ArchFix tool relies on architectural violations raised by static architecture conformance tools to provide repairing recommendations to guide the process of fixing such violations. [Terra et al., 2013].

Other important aspect of software development, addressed in this master dissertation, concerns API comprehension and documentation. In this context, recom-

mendation systems are still under investigation by researchers. The following sections present two main groups of RSSEs proposed for this particular context.

## 2.2 IDE-based Recommendation Systems

Integrated Development Environments (IDEs) are software platforms that provide a set of features to support the development process. In this group of platforms, stands out Eclipse [Eclipse, 2013], NetBeans [Oracle, 2013] and IntelliJ [WebStorm, 2013].

In general, IDEs are designed to support new features, usually by means of plug-ins. Such plug-ins are usually coupled to the IDE kernel and they can share functionality with existing modules. Therefore, recommendation systems developed as plug-in for IDEs can take benefit from a very important context: the work environment of the developers.

The first group of RSSEs described in this master dissertation is developed as plug-ins for IDEs and therefore they can take advantage of different data sources to trigger recommendations. The RSSE described in Section 2.2.1 uses code fragments already implemented to make recommendations of statement sequences and associations that are frequent in projects. Section 2.2.2 describes a RSSE that provides code snippets recommendations to be inserted directly into the source code.

### 2.2.1 Vertical Code Completion

Silva Jr et al. [2012] proposed a new approach for code completion, complementary to existing approaches, which takes into account the semantics of what was coded to suggest frequent code sequences. Basically, the approach relies on a sequence mining algorithm [Agrawal and Srikant, 1995] to extract sequential patterns to be compared with existing statements. In case of similarities, the remaining pattern elements are automatically suggested to the user.

In order to automate the approach, the authors built a tool called VCC (Vertical Code Completion) in the format of a plug-in for the Eclipse IDE. The implementation of the tool as a plug-in allowed the reuse of several data structures from Eclipse (e.g., the AST of the current project). Moreover, an IDE is a natural environment for presenting the suggestions of frequent sequences of statements to developers.

In the first phase, the plug-in uses the project's AST to extract information about all method calls made in the project. The extracted information is structured and processed with an algorithm for mining sequential patterns. The extracted patterns

are structured in the form of a search tree, which provides a good performance when searching for patterns in the second phase.

The second phase consists of the query and presentation of the sequential patterns recommendations to users, which is started whenever they request. When this happens, the tool receives as input a set of sequences that represent different combinations of method calls in the body of the method, maintaining the order in which they are called. For each request, the sequential patterns in the tree that match the ones passed as input sequences are ordered and suggested to the users.

An experimental evaluation in the form of a controlled experiment was conducted to evaluate how the code suggestions are useful for developers during their development tasks. The results showed that the suggestions provided by the tool achieved rates of 71.66% of approval, 20% of failure, and 8.33% were neutral about the usefulness of the suggestions.

Providing sequential patterns directly from the source code is one of the strengths of the approach, since the data used by the algorithm is specific of each project. On the other hand, the approach fails in projects at early stages, where the available mass of data is not sufficient for extracting accurate sequential patterns.

## 2.2.2 Mining API Code Snippets

Hsu and Lin [2011] proposed an approach to help APIs newcomers with recommendations of frequent API usage patterns mined from relevant source code files retrieved in real time from a code search engine. Basically, the approach takes as input one or more lines of code used for searching for relevant source code (i.e., files that contains calls to the elements in the input). The retrieved source files are mined and processed in order to obtain sequential patterns and association patterns. Next, these patterns are transformed into recommendations and later in code snippets.

To automate the approach, the authors implemented a tool called MACs (Mining API Code Snippets) as a plug-in for the Eclipse IDE. To provide recommendations MACs works in two distinct stages: (i) populating the patterns database, and (ii) making recommendations and generating code snippets.

The first stage consists in obtaining relevant source code files, extracting structural information, and mining patterns for the elements provided as input. The source code files are analyzed and the extracted information is stored in order to provide the necessary data for the mining process. This process relies on algorithms to mine sequential patterns and association patterns.



The second stage consists in the treatment and reuse of the identified patterns for generating code snippets that are inserted directly into the user's workspace. Initially, the patterns are retrieved from the database and sorted by their frequency of occurrence. When the user selects one of the patterns, the tool generates code snippets that can be edited and inserted into a desired file in the workplace.

The authors evaluated the usefulness and accuracy of the recommendations provided by the approach in eight open-source projects. In each project selected for evaluation, they randomly selected a file to be analyzed. Moreover, for each file, they randomly chosen a third-party API used in this file. At this point, the authors assumed that a developer without a prior knowledge of this API would program in the same way as in the current code. In other words, it should use the same elements of the API in a new implementation of this code.

Therefore, the evaluation was performed to measure the capability of the approach to recommend: (i) the API elements present in the class for association rules, and (ii) the API elements present in the method body for sequential patterns. The results showed that association rules for the top ten recommendations had an precision of 85% and a recall of 60%. Moreover, the first recommendation of sequential pattern achieved an precision of 85% and a recall of 82%. In this context, precision is defined as the number of relevant API elements retrieved by a search divided by the total number of retrieved API elements. On the other hand, recall is defined as the number of relevant API elements retrieved by a search divided by the total number of relevant API elements.

The authors also evaluated the performance of MACs. On average, the time spent from the request to the presentation of the results was 12 seconds. They noted that the step for accessing the Kodiers.com engine was the most important, representing 85% of the total time. These result were possible due to some optimizations such as: (i) limitation to a maximum of 20 files to be retrieved from Kodiers.com, (ii) application of some filters on the data, (iii) the association patterns are limited to two elements (i.e., one premise and one consequent), and (iv) limitation to a maximum of 50 patterns.

## 2.3 API Documentation

API documentation is often considered as a key learning resource by developers [Robillard, 2009]. However, inadequate or insufficient documentation is also often appointed as the most severe obstacle facing by developers when learning a new API [Robillard and DeLine, 2010; Hou and Li, 2011].

Traditionally published in the form of static HTML files, API documentations have very few contextual information, unlike IDEs. Therefore, documentation-based RSSEs tend to be less customized to the user's context. In other words, it may be necessary that users explicitly inform some information.

The RSSEs in this second group are developed as tools that instrument traditional documentations. In Section 2.3.1, we present a RSSE that instruments the standard Java-based API documentation format with concrete examples extracted from a private source code repository. Moreover, the examples are summarized using a static slicing algorithm. Section 2.3.2 describes a RSSE that extracts examples from source code obtained from search engines to generate a new type of documentation, called EXOADOCS.

### 2.3.1 APIMiner 1.0

To help developers using an API, Montandon [2013] proposed a platform that instruments the standard Java-based API documentation format with concrete source code examples, extracted from a private repository. The proposed approach, called APIMiner, receives as input a list of methods provided by an API of interest and a repository of client systems that use this API. After that, the platform extracts source code examples from such client systems, which are inserted in an instrumented JavaDoc documentation, called the APIMiner documentation.

APIMiner architecture is divided into two major phases: preprocessing and querying. The preprocessing phase is responsible by the computation that extracts the examples. The querying phase is responsible by the interaction between API users and the APIMiner system. More specifically, this architecture is composed by seven main components, as follows.

- **API Database:** This component stores information about the API to be analyzed and instrumented by the platform. This information includes the signature of all public methods provided by the API, and the full qualified name of each method;
- **Systems Database:** This component stores the source code of the systems used to extract code examples. These systems must use the API of interest and must be inserted in this database as a compressed file or using a version control system, like Git or Subversion;
- **Extraction:** This component locates and selects the code snippets that might represent an example of usage for a given API method. It relies on the Eclipse's

JDT to parse each source code file on the Systems Database and to analyze each method call performed by such systems. If the called method exists in the API Database (i.e., if the method belongs to the API under analysis) then the component sends the source code fragments with the method call to the Summarization Module (next component);

- **Summarization:** This component extracts the statements representing an example of API usage and stores the extracted code in the Examples Database (next component). It receives as input the statement containing the API method call and the body of the method where the API method is called. The component then relies on a slicing algorithm to extract the statements structurally related with the API call. The central goal is to provide a compact source code fragment that represents a usage example for the API method call received as input;
- **Examples Database:** This component stores the examples extracted and summarized by the previous component. Its central goal is to increase the performance of the platform during the querying phase;
- **Ranking Module:** This component orders the examples extracted for a given API method when the users queries for them. For this purpose, the authors implemented a ranking criteria based on four metrics: (a) Lines of code, to give priority to examples more concise and small; (b) Number of commits, to give priority to examples more important in their systems; (c) Number of downloads, to give priority to examples extracted from systems widely used and with active community; and (d) Users feedback, to give priority to examples more relevant according to user opinions. The final ranking is computed by applying a simple weighted average over the values obtained from the metrics. The examples with the highest scores are presented first to the end-users;
- **JavaDoc Documentation:** This component provides the interface for communication between the API user and the APIMiner platform. It instruments the original JavaDoc by adding a button with the label *Example* next to the description of each API method as illustrated Figure 2.1. When the users click on this button, a pop-up window appears to show the examples provided by the platform for the given API method. For each example, it is also presented extra information, such as project, file, and method from which the example was extracted, number of checkouts, and rating.

Public Methods		
Example <small>3 Examples</small>	abstract void	<b>cancel ()</b> Turn the vibrator off.
Example <small>0 Examples</small>	abstract boolean	<b>hasVibrator ()</b> Check whether the hardware has a vibrator.
Example <small>4 Examples</small>	abstract void	<b>vibrate (long[] pattern, int repeat)</b> Vibrate with a given pattern.
Example <small>9 Examples</small>	abstract void	<b>vibrate (long milliseconds)</b> Vibrate constantly for the specified period of time.

Figure 2.1: JavaDoc instrumented with an "Example Button"

To evaluate the platform, the authors implemented a particular instance of their solution for the Android API, called Android APIMiner [Montandon et al., 2013]. This instance provides 79,732 usage examples distributed over 2,494 methods and 349 classes of the API, which represents 18% of all methods and 19% of all classes in the Android API. The examples were extracted from 103 popular open-sources systems. Furthermore, a field study was conducted to answer some questions using the data obtained from Google Analytics service and from a private logging service implemented by the platform. The findings show that the Android APIMiner received a total of 20,038 visits, with three countries concentrating the accesses: United States (3,162 visits), India (2,086 visits), and Brazil (1,743 visits). An analysis on the queries used by the users when they reached Android APIMiner from a search engine showed that 35% had the keyword **example**, which reinforces the importance that API users give to source code examples.

### 2.3.2 eXoaDocs

Kim et al. [2009] proposed an intelligent search code engine that searches, summarizes, and automatically embeds API documents with code examples. Each API element is annotated with its popularity to help developers finding the APIs most frequently used in programming tasks. Moreover, about two to five code examples are provided for each API element. The resulting documentation generated by their approach is called EXOADOCS [Kim et al., 2009, 2010].

The proposed framework consists of four modules: (i) Summarization, that builds a repository of candidate code examples; (ii) Representation, that extracts semantic features from each summarized example for further clustering and ranking; (iii) Diversification, that clusters the summarized code into different usage types (i.e., various kinds of situations that API examples can be used); and (iv) Ranking, that ranks the

code examples in each cluster. The most representative code example from each cluster is inserted into the EXOADOCS. In the following paragraphs, we detail each of such modules.

The Summarization Module searches, collects, and stores potential code examples for each API method. To achieve this purpose, this module performs a query in a code search engine, as Koders, with the API's method name and collects the top-200 answers for each method. Next, the retrieved source code files are submitted to the summarization algorithm that transforms the collected code into a concise snippet. This algorithm identifies the API's methods and selects the semantically relevant lines related to this API element based on a slicing algorithm. More specifically, a source code line is tagged as relevant if it satisfies at least one of the following requirements: (R1) declares the input argument for the given API usage; (R2) changes the values of the input arguments; (R3) declares the class of the given API; or (R4) calls the given API element.

The Representation Module extracts information for clustering and ranking the examples. This module seeks to achieve a balance between an abstract representation (e.g., simple texts) and a specific representation (e.g., ASTs). More specifically, the module extracts element vectors from the summarized snippet's AST using a clone detection algorithm and computes the similarity between the extracted vectors.

The Diversification Module clusters the summarized code into different usage types. The final goal is to create clusters similar summarized code snippets. The process is performed by invoking the k-means clustering algorithm four times and selecting the result with the best quality according to a predefined set of quality metrics.

The Ranking Module ranks the examples in each cluster to select the most representative ones. First, the summarized code snippets are ranked based on measures such as representativeness, conciseness, and correctness. Next, the selected examples are sorted based on the number of occurrences.

To evaluate the proposed tool, the authors generated a version of EXOADOCS for the Java API. As result, the approach found examples for 20,480 methods, which represents 74% of the considered API. A user study revealed that EXOADOCS helped the subjects to complete more programming tasks and in less time than subjects who do not use the system.

## 2.4 Examples Quality

Usage examples are a key API learning resource and their absence is often cited as one of the main obstacles when learning a new API [Robillard, 2009]. Although there are many systems for extracting code examples [Kim et al., 2009; Zhong et al., 2009; Hsu and Lin, 2011; Silva Jr et al., 2012; Montandon et al., 2013], little is still known about the aspects that make a good code example.

Buse and Weimer [2012] argue that examples generated by current approaches might be useful, but they are very different from human-written examples. In Section 2.4.1, we discuss the characteristics of a good code example and the attributes distinguish them from unuseful examples. Section 2.4.2 presents a study on human-written examples, which proposes key guidelines for automatically creating examples sharing the best properties of human-written ones.

### 2.4.1 A Study of Programming Q&A in StackOverflow

Nasehi et al. [2012] conducted an exploratory study using the Stack Overflow forum of programming questions and answers. Their goal was to identify the aspects that characterize code examples, i.e., examples that effectively help developers and maintainers to solve real programming tasks. In addition, they identified the attributes that distinguish useful examples from not useful ones.

The authors collected a set of threads containing answers with high scores and manually analyzed all recognized answers. Basically, recognized answers correspond to accepted answers and unaccepted answers with score greater than or equal to 10. The authors found that only 2.4% of the responses posted in StackOverflow have such score.

The analysis of the recognized responses revealed some common attributes to the examples, which are listed below:

- **Concise Code:** The examples are small and less complex due to the elimination of implementation details and the use of place-holders;
- **Using Question Context:** The examples use information or code present in the question, resulting in more cohesive code;
- **Highlighting Important Elements:** The answer begins by highlighting the key elements of the solution;
- **Links to Extra Resources:** The answer contains hyperlinks to other sources of information, which generally have more information;

- **Multiple Solutions:** Several answers in the same thread can provide alternative solutions to the same problem, using different classes or APIs;
- **Inline Documentation:** Comments are frequently used as a way of explaining small pieces of the code.

However, the presence of these attributes does not automatically lead to a recognized answer. On the other hand, it was observed that the lack of code, the lack of explanation, and inconsistent explanations are the key attributes present in the responses not recognized. The authors also emphasize that the examples and the explanations are mutually important for building good examples. In general, good answers are customized to the needs of questioners. Moreover, the usage of familiar contexts to developers facilitates the understanding by other users, beyond the questioner himself.

### 2.4.2 Synthesizing API Usage Examples

Buse and Weimer [2012] proposed a technique for mining and synthesizing usage examples more succinct, representative, and legible to developers. Basically, the technique uses data flow analysis to model and extract usage information on API elements. This information is transformed into abstract models and clustered to generate usage examples. The synthesized examples have specific characteristics that make them more readable to developers, which were obtained by means of two main analyzes: (i) an analysis of examples hand-crafted by developers and (ii) a survey with developers.

In the first analysis, the authors explored the original documentation of the Java Software Development Kit, by collecting existing examples and analyzing them to identify key quality properties. The authors assumed that the Java SDK documentation is a good source of information because it is authoritative, generally considered of high quality and written for a general audience. The main properties of the examples were:

- **Length:** On average the examples have 11 lines of code, and the median is five lines of code;
- **Abstract initialization:** Some variables have their initialization abstracted, in order to allow users to change the initialization to fit specific values in their programming context;
- **Abstract use:** Abstract placeholders are used to indicate specific behaviors of a context, for example a comparison between two specific objects;

- **Exception Handling:** Examples need to handle possible exceptions when necessary or when commonly performed.

In the second analysis the authors conducted a survey with 150 undergraduate students questioning what factors are important in code examples. Some common themes emerged from the results:

- **Multiple uses:** Users prefer examples showing different ways to use the involved elements;
- **Readability:** Users prefer examples that are easy to read and understand;
- **Naming:** The identifiers present in the examples should preferably be clear and intuitive;
- **Variables:** Users prefer examples where all involved variables are declared in the scope of the example.

Based on the information obtained in the analysis, the authors proposed an algorithm to automatically generate usage examples with quality near to the one found in examples produced manually by developers. In order to evaluate this algorithm, the authors conducted an experiment with undergraduate students comparing examples obtained by the proposed technique with examples found in the Java SDK documentation and examples provided by the EXOADOCS tool [Kim et al., 2009].

The experiment involved 154 participants and each participant evaluated 35 pairs of API examples. Each example pair was generated from the random selection of two examples provided by two out of the three evaluated approaches. The results showed that the examples provided by their technique rank better than the Java SDK examples in 82% of the cases and in 94% of the cases, when compared to the examples provided by EXOADOCS.

## 2.5 Association Rules

In many scenarios we need to discover how often two or more objects of interest co-occur or the relationships between co-occurrences of objects. A typical example is marketing analysis, for example to identify sets of items that clients frequently buy together at a supermarket. Another example is API usage analysis, aiming to identify sets of elements frequently used together in a client system. Such sets allow us to extract co-occurrence information among items, which can be used to make statements



about how likely are two items to co-occur or to conditionally occur [Zaki and Meira Jr, 2014].

In data mining, these problems are typically known as frequent itemset mining and association rules mining, as initially introduced by Agrawal et al. [1993]. Basically, the process consists of first finding collections of items that are frequent (i.e., that appear in the database an specified number of times). After, the goal is to generate strong association rules (i.e., rules with a high probability to occur again) in the form of  $A \Rightarrow B$  [Han et al., 2006].

To define these problems, the following concepts are needed:

- **Items:** Items are the elements under analysis. For instance, in API usage analysis, the items can be the collection of all API methods;
- **Itemset:** Itemset is a subset of the set of all items. Formally, suppose that  $I = \{x_1, x_2, \dots, x_m\}$  is the set of all items. A set  $X \subset I$  is called an itemset. For example, the itemset  $I$  can represent all methods provided by an API and  $X$  can be the methods provided by a class of this API;
- **Transaction:** A transaction is a tuple of the form  $\langle t, X \rangle$ , where  $t$  is a unique transaction identifier and  $X$  is an itemset. For example, consider an API usage analysis scenario. In this case,  $t$  can represent the client's methods that call a given API method and  $X$  can represent the API methods called within this method;
- **Database:** A database is a collection of transactions and can be represented by different forms. A binary database is a binary relation on the set of transactions identifiers and items. On the other hand, a transaction database is basically a list of all transactions. For example, assume that  $I = \{m_1, m_2, m_3, m_4\}$  is the set of all methods provided by an API  $A$  and that  $T = \{\langle 1, \{m_1\} \rangle, \langle 2, \{m_2, m_3\} \rangle, \langle 3, \{m_2, m_4\} \rangle, \langle 4, \{m_1, m_2\} \rangle, \langle 5, \{m_4\} \rangle\}$  is the collection of the transactions obtained after an analysis in client systems that use  $A$ . Figure 2.2a and Figure 2.2b provide examples of the binary and transaction databases, respectively;
- **Itemset support:** Support is the number of transactions that contain a given itemset. For example, in the database presented in Figure 2.2, the support for the itemset  $\{m_1, m_2\}$  is 1, because it is present only in Transaction 4. On the other hand, the support for the itemset  $\{m_4\}$  is 2 because it is present in the transactions 3 and 5.

id	m <sub>1</sub>	m <sub>2</sub>	m <sub>3</sub>	m <sub>4</sub>
1	1	0	0	0
2	0	1	1	0
3	0	1	0	1
4	1	1	0	0
5	0	0	0	1

(a) Binary Database

id	itemset
1	m <sub>1</sub>
2	m <sub>2</sub> m <sub>3</sub>
3	m <sub>2</sub> m <sub>4</sub>
4	m <sub>2</sub> m <sub>2</sub>
5	m <sub>4</sub>

(b) Transaction Database

Figure 2.2: Database Representations

An itemset is considered frequent if it has a support greater than or equal to a predefined value. Therefore, mining frequent itemsets is not a trivial problem since there are  $2^{|I|}$  potentially frequent itemsets.

An association rule represents an implication of the form  $X \Rightarrow Y$ , where  $X$  and  $Y$  are itemsets of  $I$  and  $X \cap Y = \emptyset$ . The support of a rule is the number of times in the database in which both  $X$  and  $Y$  co-occur as subsets in the same transaction. The relative support of a rule is the fraction of transactions in the database in which both  $X$  and  $Y$  co-occur as subsets in the same transaction. The confidence of a rule is a measure of the rule's strength and it is calculated as the conditional probability that a transaction contains  $Y$  given that it contains  $X$ . In other words, the confidence of a rule is defined as the support of the rule divided by the support of its premise (the left term in the rule).

Moreover, to extract association rules statistically significant, other metrics are also often used:

- **Lift:** The lift of a rule measures its strength and it is calculated as the confidence of the rule divided by the relative support of the consequent. Values greater than 1.0 suggest that the rule is quite useful [Bayardo and Agrawal, 1999];
- **Jaccard:** The Jaccard coefficient measures the similarity between the antecedent and the consequent.

Agrawal and Srikant [1994] proposed an algorithm, called Apriori, that improves a brute-force approach for frequent itemset mining. Zaki et al. [1997] proposed an algorithm, called Eclat, that indexes the database and improves the computation of the sets frequency. Finally, Han et al. [2000] proposed an algorithm, called FP-Growth, for mining the complete set of frequent patterns by pattern fragment growth.

As previously mentioned, there are several applications that can take benefit from itemset and association rules mining. In this master dissertation, we use association rules to extract usage patterns regarding API methods calls.

## 2.6 Program Slicing

In general, real-world software systems are complex and difficult to understand. To facilitate the understanding of programs, developers frequently fragment the code into smaller pieces. For example, to debug a program which presents a problem at a given statement X, a developer may search for other statements that can affect this particular statement. In other words, the developer breaks the code into smaller fragments to identify and resolve his problem.

Weiser [1981, 1984] proposed an approach to automatically decompose a program into slices according to particular criteria, known as program slicing. Basically, a slice criterion represents a statement of the program and the slice represents a set of independent code fragments. The slicing algorithm processes each statement in the program and selects those related to the slice criterion.

According to Weiser, the power of slices comes from four facts:

1. Slices can be computed automatically, since it is possible to build algorithms to automate the whole process.
2. Slices are generally much smaller than the program from which they originate, because only the related statements are included in the slice.
3. Slices can be executed independently of one another, since all necessary statements and variables are present in a slice.
4. Slices reproduce exactly a projection of the original program's behavior, because the selected statements are independent of the unselected ones.

Considered as a refinement of program slicing, Korel and Laski [1988] introduced the concept of dynamic program slicing, which are slices that preserve the program's behavior for a specific input. Unlike static slicing, dynamic slicing is computed at runtime and collects only the statements executed for a specific value given as input [Agrawal and Horgan, 1990].

Generally, the slices produced by a static slicing algorithm are larger than the slices produced by a dynamic slicing algorithm. However, the former are not limited to a specific input [Harman and Hierons, 2001].

Moreover, static slicing can be categorized according to the program fragment affected by the slicing, as follows: (a) backward slice, and (b) forward slice. A backward slice consists of the set of statements that can affect the input statement, and a forward slice consists of the statements that would be affected by the computation of the input statement [Venkatesh, 1991].

In this master dissertation, we rely on a static slicing algorithm that accepts as input multiple statements. Our goal is to select relevant statements that can be used as examples for API usage patterns.

## 2.7 Final Remarks

This chapter presented background work related with the system proposed in this master dissertation. Initially, we presented some recommendation systems in software engineering and their potential to assist developers. We divide the recommendation systems related to our research in two main groups. The first group includes RSSEs developed as extensions of IDEs, which may provide more accurate recommendations. The second group includes RSSEs that improve API documentations with usage examples.

Next, we presented two studies that provide guidelines on improving the readability and usefulness of source code examples. These guidelines used together with a static slicing algorithm constitute the core of the approach for extracting examples for API usage patterns proposed in this master dissertation.

Finally, we presented the concept of association rules, which are used to extract the API usage patterns considered in this work.

# Chapter 3

## Proposed Solution

### 3.1 Introduction

The central idea behind our solution is to provide source code examples involving more than one API method call and representing usage patterns. We claim that real programming tasks tend to be complex, and therefore most maintenance tasks are not limited to a single method call. In other words, examples that illustrate a single API method call are limited because real tasks tend to use multiple calls. Moreover, when providing source code example it is also necessary to consider some aspects, like representativeness and readability. Representativeness relates to the capacity of the example present the correct use or an usage pattern of a given element. Readability relates to the capacity to facilitate the understanding of the content by programmers.

Our approach extends the APIMiner 1.0 platform [Montandon, 2013] by using data mining algorithms to extract usage patterns and to include more useful examples. More specifically, we rely on an association rules mining algorithm to discover usage patterns of the API methods. These patterns are used to extract and recommend examples representing a common usage of the API methods by client systems. Moreover, we propose some techniques to improve the readability of the extracted examples. We call our approach APIMiner 2.0.

Figure 3.1 presents the internal architecture of the APIMiner 2.0 platform. Basically, this architecture has six artifacts and five modules:

**API Source Code:** This artifact represents the repository of the API's source code;

**Mining Dataset:** This artifact represents a repository with API client systems—i.e., systems that use the API in their code. This repository is used to mine API usage

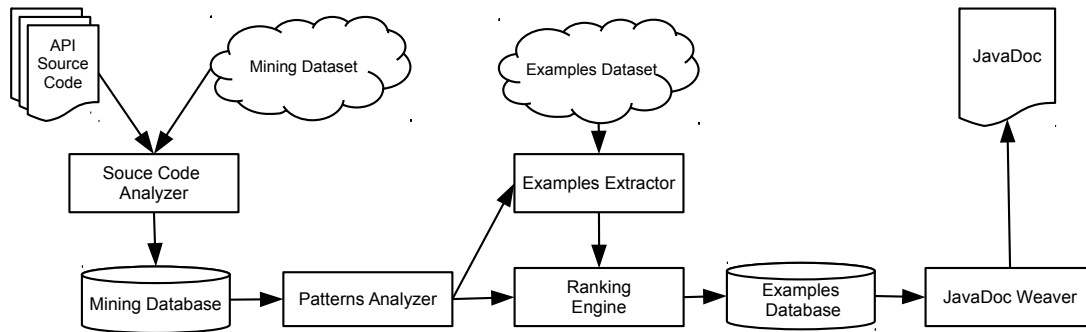


Figure 3.1: APIMiner 2.0 architecture

patterns;

**Source Code Analyzer:** This module analyzes the source code of the target API and their client systems. From the API, it extracts information on the provided classes and methods (e.g., signatures, access modifiers, etc.). From the client systems, it extracts information about the usage of the methods provided by the API;

**Mining Database:** This artifact is a database that stores the data generated by the module **Source Code Analyzer**;

**Patterns Analyzer:** This module analyzes the usage information stored in the **Mining Database** to extract usage patterns of the common API methods in the client systems. This module relies on a data mining algorithm to mine frequent association rules in the projects. We call these rules as usage patterns because they are frequently followed by the client systems;

**Examples Dataset:** This artifact represents a repository with selected API client systems to extract the examples. The systems in this artifact can be different from the systems in the **API Clients to Mining** repository;

**Examples Extractor:** This module extracts single-calls and multiple-calls examples with API methods (using the patterns mined by the **Patterns Analyzer** module). Moreover, the examples are processed by this module to improve their readability;

**Ranking Engine:** This module ranks the examples with single API method calls

and the examples for the mined usage patterns, which include multiple API methods calls. The ranked examples are stored in the **Examples Database**;

**Examples Database:** This artifact represents a database that stores examples and usage patterns produced by the **Examples Extractor** and **Ranking Engine** modules;

**JavaDoc Weaver:** This module generates the instrumented documentation with the examples and the usage patterns;

**JavaDoc:** This artifact represents the output of the proposed approach. It is a documentation in JavaDoc format instrumented with examples and usage patterns.

Regarding the APIMiner 1.0 architecture, the proposed platform introduces the **Patterns Analyzer** module and the **Mining Database** artifact. Moreover, the client systems used in this new version are divided into two groups: (a) **Mining Dataset**, which are systems for mining the usage patterns, and (b) **Examples Dataset**, which are systems for extracting the usage examples. However, the client systems are not necessarily exclusive in each dataset and may be present in both datasets. Moreover, the module that extracts the usage examples (**Example Extractor**) was improved to consider some attributes of readability suggested by the literature.

The ranking module (**Ranking Engine**) has a similar version in APIMiner 1.0. However, the criteria used to rank the examples in both versions are different and the new module also ranks usage patterns. The **Example Database** artifact and **JavaDoc Weaver** also have a similar module in APIMiner 1.0. The key difference is that the new modules were modified to store and present usage patterns.

The following sections present in details the modules in this architecture. Section 3.2 describes the **Source Code Analyzer** module. Section 3.3 shows how the patterns are mined by the **Patterns Analyzer** module. Sections 3.4 and Section 3.5 describe the extraction process and the ranking of examples and patterns by the **Examples Extractor** and **Ranking Engine** modules, respectively. Finally, Section 3.6 shows how the JavaDoc is instrumented by the **JavaDoc weaver** module.

## 3.2 Source Code Analyzer

The **Source Code Analyzer** module supports the first step that must be executed by the proposed platform. Basically, this module analyzes the source code of the target

API and their client systems to obtain information about: (i) classes and methods provided by the target API, and (ii) calls to API methods made by the client systems.

More specifically, this module implements two main steps. First, it analyzes the source code of the API to extract information about its classes and methods (as detailed in Table 3.1). The extracted data is store in the **Mining Database**, and replicated into the **Examples Database**. The decision to separate the mining data from the examples data was taken to improve the quality of the presented examples.

Table 3.1: API elements and their attributes

Element	Attribute	Description
class	access modifier	visibility of the class
	identifier	name of the class
	methods	list of methods provided by the class
method	access modifier	visibility of the method
	identifier	name of the method
	parameters	list of object types
	return type	object type or void
	throws	checked exceptions the method can throw
	class	owner class

After analyzing the API, the proposed platform analyzes the use of the API methods by client systems. Each client system is analyzed individually to collect the sets of API method calls. We relied on the method's scope to build these sets, therefore at the end of the process the number of sets is equal to the number of methods implemented by the classes of the client systems. Each set of method calls represents a transaction and are stored in the **Mining Database**.

It is worth mentioning that this module improves the procedure followed by APIMiner 1.0, where the administrator needs to provide the API method's signature and the full qualified name of the class that the method belongs to. Therefore, in APIMiner 2.0 we automated this task.

In the implementation of this module, we used the JDT component of the Eclipse IDE to create an Abstract Syntax Tree (AST) from the source code files. Moreover, the current implementation allows to automatically download the source code files from different repositories, such as Git, Subversion, Mercurial, compressed files, or local files.

### 3.3 Patterns Analyzer

The **Patterns Analyzer** module represents a key contribution of the proposed platform, regarding its first implementation. In few words, the module is responsible to mine



usage patterns of the API methods in the client systems. Basically, it retrieves the transactions stored in the **Mining Database** and then relies on an association rule mining algorithm to extract the usage patterns of API methods.

First, the module searches the **Mining Database** for all transactions of the target API containing more than one element (i.e., more than an API method call). The result of the search represents the transaction database, provided as input to the association rule mining algorithm. Moreover, it is also necessary to give as input a minimum support value and a minimum confidence value to extract the rules.

In this context, the minimum support value represents the number of times that the methods, included in a given rule, must appear in the transactions database. The minimum confidence value represents the probability that the methods in the consequent of the rule must appear in a transaction when the methods in the antecedent are also present. Currently the platform does not provide an automated procedure to select the best values of support and confidence, though it provides reports to support the choice of such values by the administrators.

As result, the module produces a set of association rules in the form  $A \Rightarrow B$ , where  $A$  and  $B$  are itemsets of API methods with a minimum support (*minSupport*) and a minimum confidence (*minConfidence*). After, these association rules are used by the **Examples Extractor** module to extract examples with more than a method call and by the **Ranking Engine** module to rank the usage patterns and their examples.

In the implementation of this module, we used the WEKA (Waikato Environment for Knowledge Analysis) library [Hall et al., 2009] to extract the association rules. Moreover, the platform allows exporting the database of transactions to a file in a WEKA specific format, which can be imported by the native user interface provided by WEKA.

## 3.4 Examples Extractor

This module is responsible for extracting examples of a given API client system. Basically, the module receives the source code of a given client to extract examples both for single API method calls and for the usage patterns mined by the **Patterns Analyzer** module. After, the duplicated examples are removed and the remaining ones are stored in the **Examples Database**.

This module has the same purpose of the **Summarization** module of the APIMiner 1.0, which aims to extract examples: (i) with few lines of code, (ii) that includes contextual information, and (iii) that highlights the computation provided by

the API. However, in the new version, we modified the previously algorithm to include some readability properties to the summarized examples.

First, the module receives as input the source code of a given API client and the API methods stored in the **Examples Database**. The process begins by analyzing each class from this API client. For each method implemented by such classes, the platform performs four steps:

1. **Identifying calls to API methods:** The platform identifies all calls to API methods in the method scope (called *calls*) and their corresponding statements (called *callsStatements*);
2. **Extracting examples for single API calls:** From each call in *calls*, the platform extracts examples using a summarization algorithm. For this purpose, the original summarization algorithm implemented by APIMiner 1.0 (presented in Section 3.4.1) is used to collect the set of statements that represents the example. However, the collected examples are submitted in this version to a readability improvement process (as detailed in Section 3.4.2);
3. **Extracting examples for usage patterns:** After extracting examples for single method calls, the platform extracts examples for the mined association rules. Initially, we search all mined association rules and group the elements of the antecedent with the consequent elements to build sets of API methods (called *sets*). This strategy allows us to extract only examples for the identified patterns.

Then, for each set in *sets* contained in *calls*, the platform retrieves the corresponding statements (called *statements*) in *callsStatements* and executes the summarization algorithm with *statements* and the method body as input. The examples retrieved by this third step are also submitted to the readability improvement process;

4. **Removing similar examples:** In a project, it is normal that the development team follows a common pattern of development, so the examples of a given API method can be very similar. In this sense, the proposed platform relies on a string comparison algorithm to detect and remove similar examples from an API client. We acknowledge that this strategy may not be the best, but at least the restriction to the examples of the same project allows improving its precision.

In the following subsections, we provide more details on each of the aforementioned steps.

### 3.4.1 Summarization Algorithm

As described earlier, we reuse the summarization algorithm implemented by APIMiner 1.0 to collect the set of statements included in an example. This algorithm relies on a static slicing algorithm (see Section 2.6) to extract the source code lines structurally related with a set of API method calls.

Basically, the algorithm has two input arguments: (a) *seed*, which are the statements that call the API method; and (b) *body*, representing all existing statements in the method where the *seed* is located. Then, for each statement in *seed*, it performs a backward slicing to identify statements that modify the variables read by the analyzed statement. It also performs a forward slicing to identify the statements that use the variable returned by the statement, if the statement returns something. The statements identified in both slicing types are stored in a list and the process is performed again using such statements. As result, the algorithm returns all the statements identified as relevant after the several iterations.

To illustrate the algorithm for multiple seeds, we will rely on the method presented in Listing 3.1. Basically, this code receives the name of a client and a boolean value indicating whether the client is authorized. If he/she is authorized, the code prints a welcome message with his/her name at the console and the corresponding hour.

Listing 3.1: Slicing example using multiple method calls

---

```
1 public void welcome() {
2     String client = "Smith, John";
3     boolean authorized = true;
4
5     Calendar cal = Calendar.getInstance();
6     int hour = cal.get(Calendar.HOUR_OF_DAY);
7
8     if (authorized) {
9         System.out.println("Welcome back Mr." + client);
10        System.out.println("Now, it is: " + hour + " hour(s)");
11    }
12 }
```

---

Consider the statements related to the calls `java.util.Calendar.getInstance()` (line 5) and `java.util.Calendar.get(int)` (line 6) as *seed* and the body of the `welcome()` method as the *body*. Initially, the algorithm starts analyzing the call to `java.util.Calendar.getInstance()`. Therefore, the statement at line 5 is marked as relevant. As this method call does not use any variable, the backward slicing is not

called. However, the return of the call is assigned to the variable *cal*. Thus, the forward slicing is called to collect the statements that use the variable *cal*.

The forward slicing for the variable *cal* identifies that only the statement at line 6 uses this variable, then it is marked as relevant. However, the *seed* already contains the statement 6, so it is not necessary to add it again.

After, the algorithm analyzes the next statement in *seed*, which is at line 6. The algorithm collects the readable variables. In this statement, there is only the variable *cal*, because *Calendar.HOUR\_OF\_DAY* is a constant. Then, the algorithm executes the backward slicing and identifies that the statement at line 5 modifies the variable *cal*, but this statement is already marked.

In the forward slicing, the algorithm searches by statements that use the variable *hour*, which in this case is the statement at line 10. However, because it is enclosed by an *if* statement (lines 8-11), the algorithm extract the variables used by the *if*'s expression and apply the backward slicing over these variables. As observed, the *authorized* variable is declared in line 3. Therefore, this statement and the statements at line 8 and 10 are marked as relevant.

Because all method calls in *seed* were analyzed, the algorithm returns the statements marked as relevant (lines 3, 5, 6, 8, 10 and 11) and finishes. Listing 3.2 shows the source code fragment generated by the presented slicing algorithm. As we can observe, this fragment represents an usage example of the methods `java.util.Calendar.getInstance()` and `java.util.Calendar.get(int)`, ignoring the lines not related with (lines 2 and 9).

Listing 3.2: Extracted example

---

```
1 boolean authorized = true;
2 Calendar cal = Calendar.getInstance();
3 int hour = cal.get(Calendar.HOUR_OF_DAY);
4 if (authorized) {
5     System.out.println("Now, it is: " + hour + " hour(s)");
6 }
```

---

### 3.4.2 Readability Improvements

Although static slicing is an interesting algorithm for extracting source code examples, in some scenarios it may generate code fragments presenting some readability problems. For example, a large number of statements can be returned if the algorithm's scope is

not limited. On the other hand, if we define a very constrained scope some variables and calls to external methods may not be extracted.

To improve the examples extracted by the APIMiner platform, we implemented an algorithm that transforms the examples in order to consider some attributes of readability presented in the literature.

In the following subsections, such improvements are discussed.

#### *A - Variables not Identified*

When we limit the execution of the slicing to a small scope, important statements may not be included. The most frequent situation is related to variables declarations. For instance, consider the use of the `java.awt` API in the class presented in Listing 3.3.

Listing 3.3: Example to illustrate problems with delimitation of scope

---

```

1 import java.awt.Graphics;
2 import javax.swing.JApplet;
3
4 public class Hello extends JApplet {
5     public static final String MESSAGE = "Hello, world!";
6
7     public void paintComponent(Graphics g) {
8         g.drawString(MESSAGE, 65, 95);
9     }
10 }
```

---

The slicing extraction using the line 8 as *seed* and the body of the `paintComponent(Graphics g)` method produces the example in Listing 3.4. We can observe that the result is an incomplete example that omits the types of *g* and *MESSAGE*.

Listing 3.4: Incomplete example due to a limited slicing scope

---

```
g.drawString(MESSAGE, 65, 95);
```

---

To transform the examples with this problem, we automatically add the declaration of variables not identified in the slicing scope. Moreover, we added the comment “initialized previously” to each declaration, abstracting its initialization and enabling the users to fill the code with a specific initialization. Listing 3.5 presents the result of this improvement applied to our example (Listing 3.4).

Listing 3.5: Example with abstract initializations

---

```
String MESSAGE; //inicialized previously
Graphics g; // inicialized previously
g.drawString(MESSAGE, 65, 95);
```

---

*B - Empty Blocks*

Another situation when the readability of the code is affected is related to decision to include control statements. Particularly, in some cases, empty blocks can be generated. For example, consider the example extracted using the original summarization algorithm presented in Listing 3.6. In this example, the `try` block is empty, because the seed is used only by the `catch` block.

Listing 3.6: Example with an empty block

---

```
Log log = Logger.getLog("ExampleLogger");
try {
} catch (Exception e) {
    log.error(e.getLocalizedMessage());
}
```

---

In addition to `try/catch` statements, the following statements may generate examples with empty blocks:

- **Loop statements:** Empty blocks are commonly generated when the seed is localized in a loop's expression that may include assignments. Listing 3.7 presents an example of this situation.

Listing 3.7: Empty blocks in loop statements

---

```
Iterator<String> strings = Arrays.asList("Foo", "Bar").iterator();
for (String string; string = strings.next(); strings.hasNext()) {
}
```

---

- **Conditional statements:** In this case, empty blocks commonly appear when the seed is placed in the conditional expression. Listing 3.8 presents an example of this situation.

In summary, although the aforementioned examples show the use of the API methods, their readability is not good. To improve this readability, we automatically

Listing 3.8: Empty blocks in conditional statements

---

```

Connection con; // initialized previously
if (!con.isClosed()) {
}

```

---

add the comment “do something”, indicating that the empty block in fact performs a specific code in the client context. Listing 3.9 presents the improved example regarding the code in Listing 3.6.

Listing 3.9: Example with a comment in an empty block

---

```

Log log = Logger.getLog("ExampleLogger");
try {
    //do something
} catch (Exception e) {
    log.error(e.getLocalizedMessage());
}

```

---

### 3.4.3 Removing Similar Examples

Users typically want different examples for the same problem. In order to remove very similar examples, we implemented a procedure that relies on an approximate matching algorithm, which measures text similarity using cosine distance [Okazaki and Tsujii, 2010].

More specifically, this procedure is performed after extracting examples. Initially, the examples are grouped by the API methods they call. Next, we executed the Algorithm 1.

This algorithm receives as input the groups of examples (*groups*) and a value between 0 and 1 (*coefficient*), which represents the minimum ratio to consider two examples as similar. First, the algorithm iterates over each group in *groups* (line 3). For each group, the algorithm selects an example, stores it in a list of examples called *examples* and compares it with all other ones in the group (lines 4-8). If the similarity coefficient between two examples is greater or equal to *coefficient*, then the algorithm removes one of them. This process is performed until only one example stays in the group. The last example of the group is also stored in *examples*. After all groups are analyzed, the algorithm returns the examples in *examples*.

---

**Algorithm 1** Removing similar examples extracted from an API client
 

---

```

1: function REMOVESIMILAREXAMPLES(groups, coefficient)
2:   examples  $\leftarrow$  { }
3:   for each group in groups do
4:     while SIZE(group) > 1 do
5:       example  $\leftarrow$  POP(group)
6:       examples  $\leftarrow$  examples + example
7:       COMPARE_AND_REMOVE(example, group, coefficient)
8:     end while
9:   end for
10:  return examples
11: end function

```

---

This procedure represents a important contribution from this new version of the APIMiner platform, since the platform in version 1.0 did not address this problem.

## 3.5 Ranking Engine

This module is responsible for ranking the examples for single API methods and for the mined usage patterns. Basically, it performs two independent tasks: (a) ranks the examples, and (b) ranks the usage patterns for an API method and the associated examples. In the next sections, we detail each one of such tasks.

### 3.5.1 Ranking Examples for Single API Methods

When the users request examples for an API method or a usage pattern, the platform should rank the examples for this request. The examples are ranked based on three metrics:

**Completeness:** This metric indicates whether an example is complete. We consider an example as complete if it compile without errors. We consider that such examples contain all information necessary for their reuse. We also consider this metric as the most important one;

**Lines of Code (LOC):** This is a complementary metric to **completeness**. Basically, we give priority to concise and small examples;

**Users Feedback:** This metric takes into consideration the feedback of the users of the APIMiner platform and it is also a complementary metric to **completeness**. The



value of this metric may vary overtime, since it depends on information provided by the platform’s users. In other words, the dynamic nature of this metric allows the community to define itself, along the time, which examples are more relevant.

This feedback is obtained from the instrumented JavaDoc by means of two buttons: (i) Like, and (ii) Dislike, as showed in Figure 3.2. A similar strategy is used by popular websites, as Youtube, Facebook, and StackOverflow.

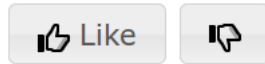


Figure 3.2: Feedback buttons

To calculate the final ranking of the examples, we propose the Algorithm 2. Basically, this algorithm ranks the examples with less compilation errors at the top. For examples with the same number of errors, the algorithm prioritizes the ones with less LOC and with a high feedback score. The feedback score is the difference between the number of positive and negative feedbacks.

---

**Algorithm 2** Prioritization algorithm

---

```

1: function COMPAREEXAMPLES(example1, example2)
2:   numErrorsE1  $\leftarrow$  COMPILATIONERRORS(example1)
3:   numErrorsE2  $\leftarrow$  COMPILATIONERRORS(example2)
4:   if numErrorsE1 = numErrorsE2 then
5:     iScoreE1  $\leftarrow$  LOC(example1) - FEEDBACKSCORE(example1)
6:     iScoreE2  $\leftarrow$  LOC(example2) - FEEDBACKSCORE(example2)
7:     if iScoreE1 < iScoreE2 then
8:       return 1
9:     else if iScoreE1 > iScoreE2 then
10:      return -1
11:    else
12:      return 0
13:    end if
14:  else
15:    if numErrorsE1 < numErrorsE2 then
16:      return 1
17:    else
18:      return -1
19:    end if
20:  end if
21: end function

```

---

Initially, the COMPAREEXAMPLES function receives as input two examples *example1* and *example2*. Then, for each example, the function calculates the number

of compilation errors and the results are stored in *numErrorsE1* and *numErrorsE2* (lines 2-3). If the number of compilation errors presented by the *example1* is less than *example2*, then the function returns the value 1, otherwise it returns -1. A positive value indicates that the first example is better than the second one.

If the number of compilation errors is the same, then the algorithm combines two metrics as a tiebreaker: **LOC** and **Users Feedback**. This new tiebreaker criterion gets the number of lines of code of the example using the function LOC and subtracts the feedback score (function FEEDBACKSCORE), smaller values are better.

### 3.5.2 Ranking Usage Patterns

Besides providing examples for single methods, a key contribution of the proposed platform is to recommend sets of API methods that are frequently called together. As usual, it is also important to recommend the most relevant examples first to users.

In Section 2.5, it was discussed that, at certain cases, only the support value for an association rule may not be the best metric to evaluate its value. Therefore, other metrics are frequently used, like Lift and Jaccard.

To rank the usage patterns we implemented an algorithm that takes into account two metrics related to association rules mining: (a) Lift, and (b) Number of Items in the Rule. Lift is a well-known statistical measure and it is particularly useful to determine whether an association rule is useful [Bayardo and Agrawal, 1999]. The Number of Items in the Rule is a basic measure that indicates the number of methods in the rule (i.e., items in the antecedent + items in the consequent).

More specifically, the algorithm sorts the usage patterns using the COMPAREUSAGEPATTERNS function presented in Algorithm 3. This function receives as input two usage patterns, *usagePattern1* and *usagePattern2*. Initially, the algorithm calculates the value of the Lift metric for both inputs using the GETLIFTVALUE function and stores the results in the *liftR1* and *liftR2* variables (lines 2-3). If these values are equal, the algorithm calculates the Number of Items in the Rule from each input using the GETNUMITEMS function and returns their difference (lines 4-7). Otherwise, the algorithm returns the difference between the Lift value of the first input and the second one (line 9).

A positive value means that the first usage pattern is more relevant than the second one. A negative value is the opposite, and a value equals to 0 means that they have the same importance.

After mining and sorting the usage patterns, the next step is to collect and sort the examples from each one. This process is similar to the one presented in Section

---

**Algorithm 3** Usage Patterns Prioritization

---

```

1: function COMPAREUSAGEPATTERNS(usagePattern1, usagePattern2)
2:   liftR1  $\leftarrow$  GETLIFTVALUE(usagePattern1)
3:   liftR2  $\leftarrow$  GETLIFTVALUE(usagePattern2)
4:   if liftR1 = liftR2 then
5:     supportR1  $\leftarrow$  GETNUMITEMS(usagePattern1)
6:     supportR2  $\leftarrow$  GETNUMITEMS(usagePattern2)
7:     return supportR1 - supportR2
8:   else
9:     return liftR1 - liftR2
10:  end if
11: end function

```

---

3.5.1 to rank examples for single API methods.

## 3.6 JavaDoc Weaver

This module supports the last step of the proposed platform. In few words, it is responsible to generate the instrumented documentation with the examples and usage patterns mined by the previous modules, in JavaDoc format. The new documentation keeps some aspects of the interface provided by the APIMiner 1.0, improving and adding some others to support the proposed approach.

This module automatically generates the JavaDoc documentation from the source code and instruments it. This instrumentation generates a new JavaDoc with tree main interfaces: (i) Example button, (ii) Single example dialog, and (iii) Associated Rules examples dialog. The following sections present these interface components.

### 3.6.1 Example Button

The Example button is reused from APIMiner 1.0. Basically, the instrumented JavaDoc includes **Example** buttons next to the list of public methods of an API target class, as presented in Figure 3.3. Also, there is a small label below the button, which indicates how many examples the platform provides for the method.

When the user clicks on an **Example** button, a dialog window is launched to show the examples (Section 3.6.2) and the usage patterns (Section 3.6.3) provided by the platform.

Public Methods		
Example 36 Examples	void	<code>beginTransaction()</code> Begins a transaction in EXCLUSIVE mode.
Example 0 Examples	void	<code>beginTransactionNonExclusive()</code> Begins a transaction in IMMEDIATE mode.
Example 1 Examples	void	<code>beginTransactionWithListener(SQLiteTransactionListener transactionListener)</code> Begins a transaction in EXCLUSIVE mode.
Example 55 Examples	<code>SQLiteStatement</code>	<code>compileStatement(String sql)</code> Compiles an SQL statement into a reusable pre-compiled statement object.

Figure 3.3: JavaDoc for the `SQLiteDatabase` class as instrumented by APIMiner 2.0

### 3.6.2 Examples Presentation

As mentioned previously, the interface that shows the examples is a dialog window. This new interface is similar to the one implemented by APIMiner 1.0, but with some presentation improvements, as presented in Figure 3.4.

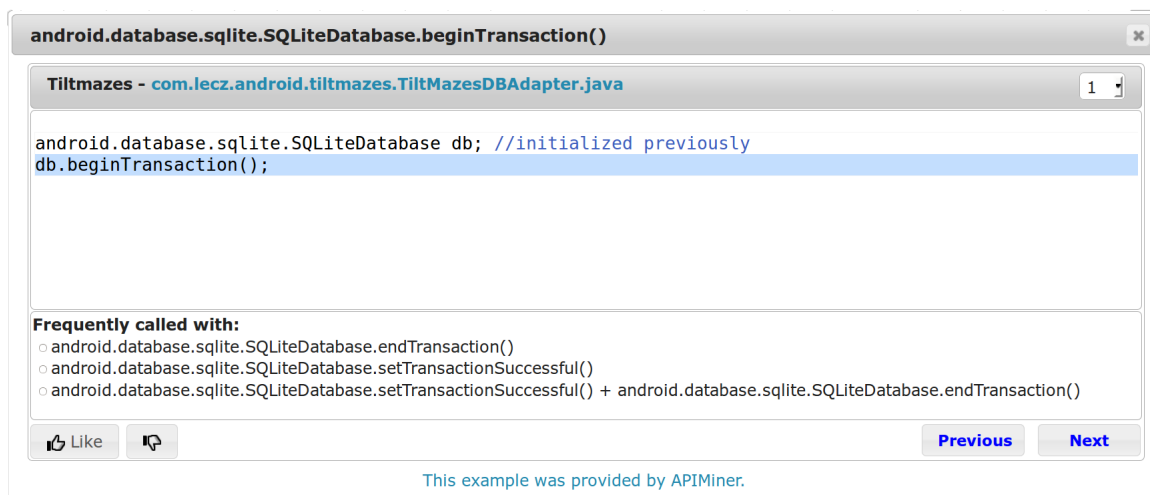
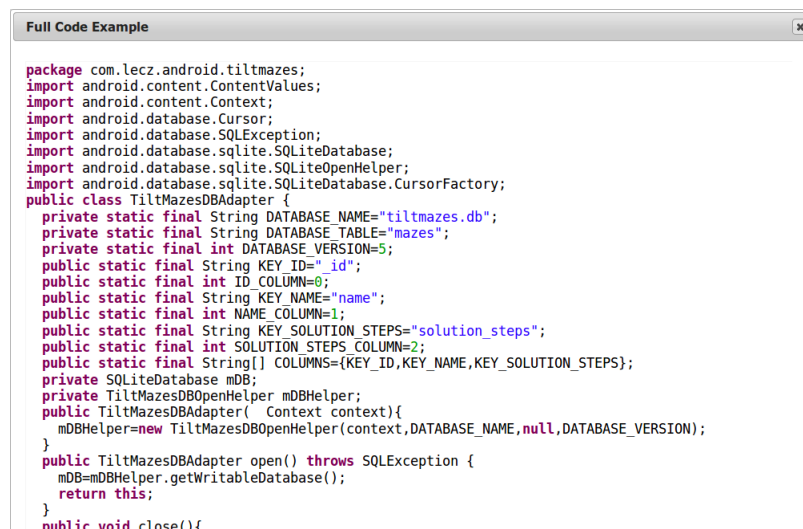


Figure 3.4: Example window in APIMiner 2.0

In the header of the dialog window, we present the method signature and an option to close the window. The body of the window is divided into four regions: (a) example header, which contains information about the current example; (b) example, which shows the example's source code; (c) usage patterns (detailed in Section 3.6.3); and (d) footer, which contains buttons for evaluation and navigation over the examples.

The window's header contains information about the project and source code file from where the example was extracted. On the right side, there is a select box that allows the user to navigate through the examples. Moreover, the user can view the full source code where the example was extracted by clicking in the file name (Figure 3.5).



```

package com.lecz.android.tiltmazes;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
public class TiltMazesDBAdapter {
    private static final String DATABASE_NAME="tiltmazes.db";
    private static final String DATABASE_TABLE="mazes";
    private static final int DATABASE_VERSION=5;
    public static final String KEY_ID="id";
    public static final int ID_COLUMN=0;
    public static final String KEY_NAME="name";
    public static final int NAME_COLUMN=1;
    public static final String KEY_SOLUTION_STEPS="solution_steps";
    public static final int SOLUTION_STEPS_COLUMN=2;
    public static final String[] COLUMNS={KEY_ID,KEY_NAME,KEY_SOLUTION_STEPS};
    private SQLiteDatabase mDB;
    private TiltMazesDBOpenHelper mDBHelper;
    public TiltMazesDBAdapter( Context context){
        mDBHelper=new TiltMazesDBOpenHelper(context,DATABASE_NAME,null,DATABASE_VERSION);
    }
    public TiltMazesDBAdapter open() throws SQLException {
        mDB=mDBHelper.getWritableDatabase();
        return this;
    }
    public void close(){

```

Figure 3.5: Full source code dialog window

The example is presented in the center of the dialog window and the code is formatted using a code syntax highlighter. Moreover, the seed statements are highlighted.

Finally, the footer contains buttons for evaluation (**like** and **dislike**) and navigation (**Previous** and **Next**). In the first case, the buttons are located on the left side and follows a strategy very similar to other websites such as YouTube, Facebook, and StackOverflow. In the second case, the buttons are located on the right side and allows the users to navigate through the examples.

### 3.6.3 Usage Patterns Interface

The usage patterns are presented between the example and the footer region. The interface starts by presenting the text “**Frequently called with:**”, indicating that the API method is frequently called with other API methods. Then, the interface shows the top three usage patterns, as computed by the **Ranking Engine** module.

By selecting one usage pattern, the instrumented documentation filters the examples and presents only those that include the methods in the selected one. For example, Figure 3.4 presents the first example for the method `android.database.sqlite.SQLiteDatabase.beginTransaction()` and three usage patterns. When selecting the third usage pattern, the result is as showed in Figure 3.6.

As we can observe, the methods involved in a usage pattern are highlighted (`beginTransaction()` + `setTransactionSuccessful` + `endTransaction()`). Moreover, the usage pattern is showed as selected and an option to clean the filter is enabled. We also emphasize that the navigation between the examples considers now the selected usage pattern.

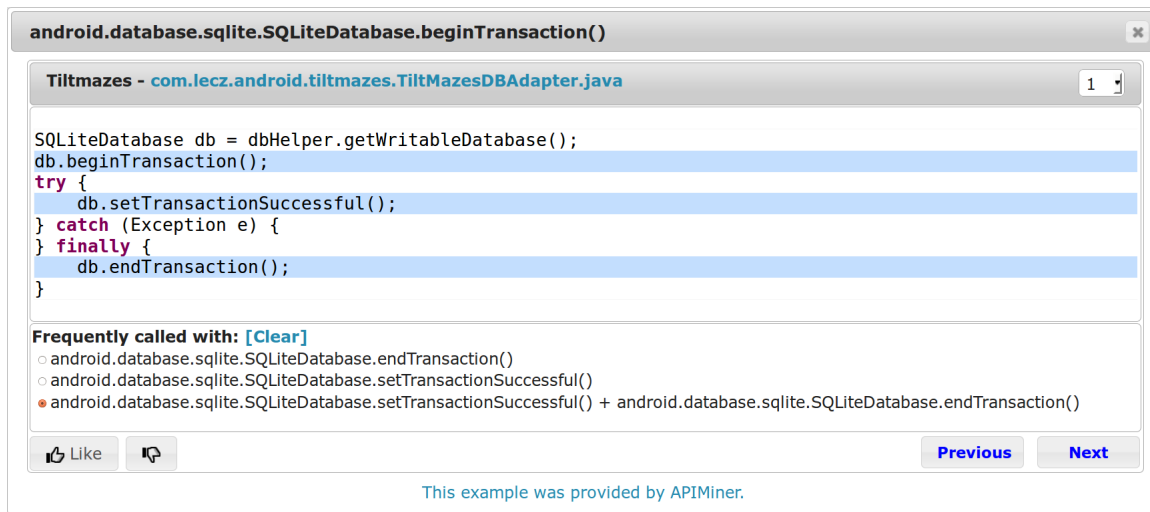


Figure 3.6: API usage pattern example

## 3.7 Final Remarks

This chapter presented in details the solution proposed in this master dissertation to extend the APIMiner 1.0 platform. This new version relies on an association rule mining algorithm to discover usage patterns for API methods. These patterns are used to extract source code examples including more than one API method call. Moreover, the examples have their readability improved based on properties usually suggested by the literature. We are calling this extended solution APIMiner 2.0.

# Chapter 4

## Evaluation

### 4.1 Overview

To evaluate the proposed platform, we implemented a particular instance for the Android API. Currently, Android APIMiner 2.0 provides around 100,000 examples for single API methods and 180,000 examples for the usage patterns. This new instance replaces the previous version and can be accessed in the following URL: <http://apiminer.org>. Figure 4.1 shows the new main page of this website.

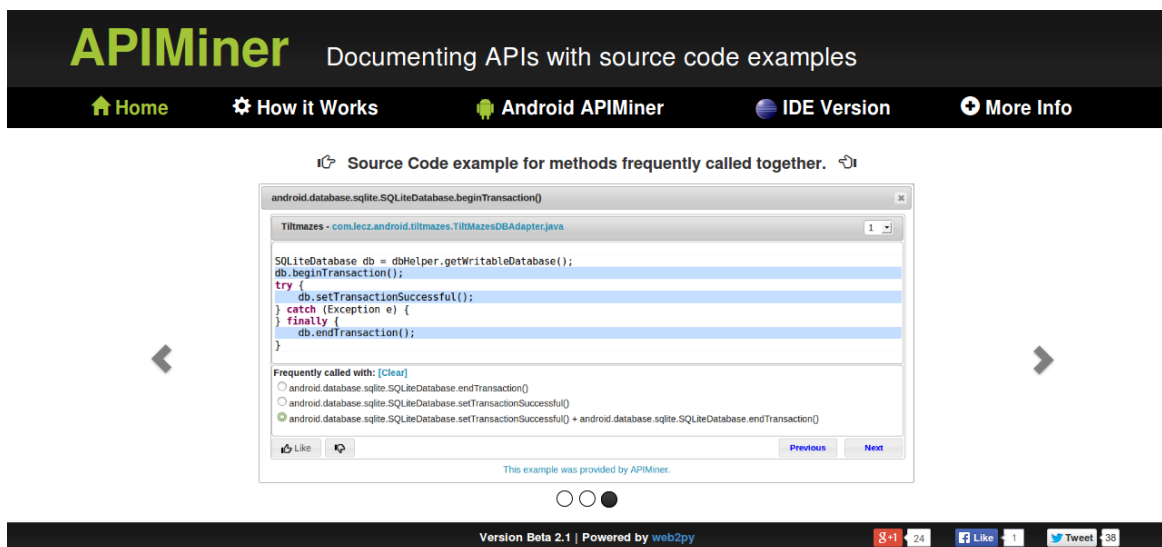


Figure 4.1: New main page of Android APIMiner 2.0

More specifically, in this chapter we report three studies conducted to evaluate the proposed solution. In the first study, we characterize an instantiation of our platform for the Android API. In the second study, we analyze the real data collected after the public usage of the platform. Finally, the third study reports a controlled experiment

performed with 29 subjects to evaluate Android APIMiner’s gains in comparison with the traditional Android documentation.

The remainder of this chapter is organized as follows. Section 4.2 presents some information on the Android API. Section 4.3 details some information on the instantiation of the proposed platform for the Android API. Finally, Section 4.4 and Section 4.5 present respectively an open study and a controlled experiment performed to evaluate Android APIMiner 2.0.

## 4.2 Android API

Android is an open-source operating system for mobile devices based on the Linux kernel. The first version (1.0) of the platform was released on 23 September 2008 and the system is currently on version 4.4.2 (released on 9 December 2013). Nowadays, the platform surpassed 1 billion device activations and every day more than 1 million Android devices are activated, in more than 190 countries. Moreover, more than 1.5 billion apps and games have been downloaded from Google Play each month<sup>1</sup>.

Google provides a Software Development Kit (SDK) for developing Android applications. Basically, this SDK provides a set of libraries and developer tools to build, test, and debug applications. Additionally, Google provides a modified version of the Eclipse IDE which includes the essential Android SDK components and the Android Developer Tools (ADT) to implement applications<sup>2</sup>.

In this dissertation, we relied on the version 4.1.2-r1 of the Android API to generate an instance of the proposed platform—called Android APIMiner 2.0. This Android version was released on 9 October 2012 and it presents only minor improvements and corrections regarding version 4.1. More specifically, this release has 24,934 public methods and 1,606 protected methods (both including class constructors) distributed over 2,920 public classes and 48 protected classes (including enumerators).

Furthermore, 17,303 methods (65.2%) and 2,575 classes (86.76%) are marked with JavaDoc annotations (denoting therefore artifacts subjected to automatic documentation generation). However, we detected an annotation `@Hide` to remove the node and all of its children from the documentation. In this case, the number of API methods and classes marked with the remaining annotations is 15,461 (58.26%) and 1,997 (67.28%), respectively. In our analysis, we considered all public and protected methods and classes available in the API, since developers can use them.

---

<sup>1</sup><http://developer.android.com/about>

<sup>2</sup><http://developer.android.com/sdk>



## 4.3 Android APIMiner 2.0

To better characterize Android APIMiner 2.0, we analyzed all data produced in the process of making this instance. Basically, the following sections reports and analyze data about the inputs and outputs of each module of the proposed platform.

### 4.3.1 Dataset

In this section, we describe the Android systems we used to extract the source code examples. As described in Section 3.1, APIMiner 2.0 relies on two repositories of client systems: (a) **Mining Dataset**, used to mine the API usage patterns, and (b) **Example Dataset**, used to extract the examples.

The selection of the systems in each repository followed different criteria, as discussed in Section 4.3.1.1 and Section 4.3.1.2. Additionally, we present some properties of the systems included in each repository.

#### 4.3.1.1 Mining Dataset

The selection of the systems in this dataset was more flexible to allow a large number of client systems. Particularly, we assume that systems not widely known by the Android community do not directly impact the quality of the mined patterns. To automate the construction of this dataset, we implemented a script that automatically searches and downloads Android client systems from GitHub. However, to assume a minimum level of quality, the following criteria were adopted:

- **Programming Language:** The projects should be marked as a Java project.
- **Commits:** The projects should have at least 50 commits. This criterion is followed to eliminate projects with very reduced programming or maintenance activity.
- **Forks:** The projects should not be a fork of another project. The inclusion of forked projects would duplicate information, therefore reducing the accuracy of the mined patterns.

In addition to the projects retrieved automatically from Github, we inserted some projects manually. These projects are well known by Android users and developers and their source code is publicly available for download. The list of all projects in this dataset is presented in Appendix A.1. Basically, we considered 396 projects, totalizing 57,658 classes and 450,762 methods, and 362,900 calls to API methods.

Among the considered projects, we can highlight projects such as CYANOGENMOD, ASTRID, and WORDPRESS. CYANOGENMOD is a customized firmware distribution for Android devices based on the Android Open Source Project. Basically, the project is divided in a core application and many customized applications. In our dataset, we include the core application and 23 customized applications.

ASTRID is a task organization tool with features like reminders, tagging, widgets, and integration with online synchronization services. The project contains more than 1,800 classes and 13,000 methods and it was recently acquired by Yahoo! Inc.. Finally, WORDPRESS is an Android application to write, edit, and publish posts to WordPress' websites. Currently, the application has between a million and five million downloads<sup>3</sup>.

To evaluate the usage of the Android API by client systems, we analyzed all projects in the mining dataset. Figure 4.2 shows the ratio of client's methods with at least one call to a method from the Android API. We can observe that in the system representing the median value, 41.38% of the methods have at least one call to an API method, which represents a high usage of the Android API by the client systems. Moreover, the data shows that the dataset contains projects using the API in different magnitude, contributing to the diversification of data.

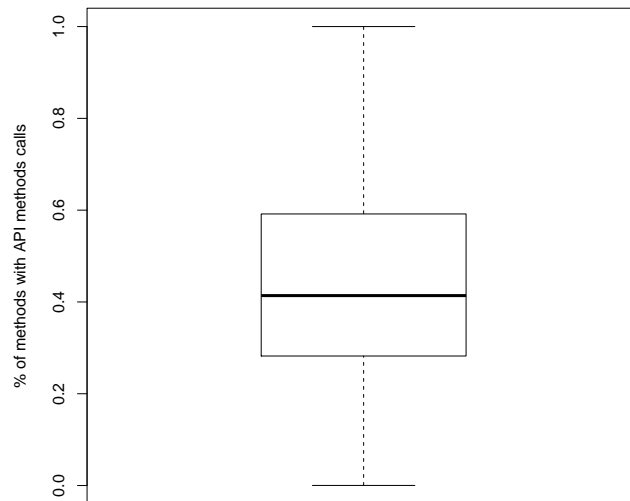


Figure 4.2: Ratio of methods with at least an API call, among the systems in the Mining Dataset

<sup>3</sup><https://play.google.com>

Figure 4.3 shows the ratio API calls/method in each project. More specifically, we calculate two ratios for each system: (i) the total number of API calls divided by the total number of methods, and (ii) the total number of API calls divided by the total number of methods with at least one call to API method. Regarding the first ratio, the system representing the median has 1.37 API calls/method. Furthermore, 75% of the systems had a ratio between 0.78 and 4.40 API calls/method, which demonstrates the importance of the API in such systems. We highlight that the system CMSCREENSOT has a ratio of 10.25 API calls/method. This happened because it is a small application that just saves the screen of the device and has only two classes and four methods that call 41 API methods.

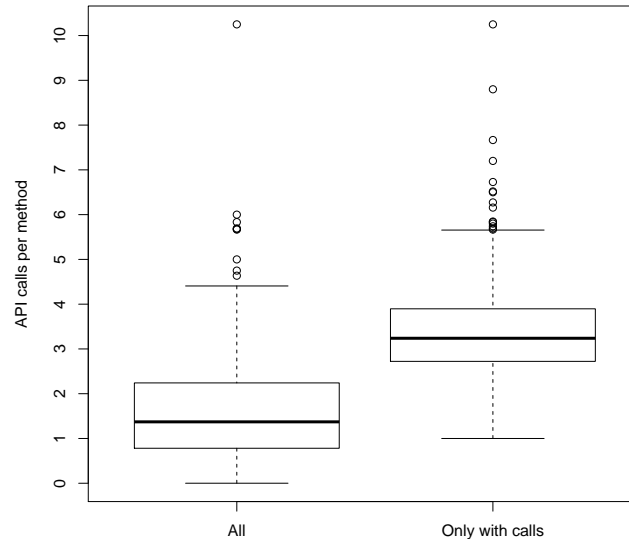


Figure 4.3: API calls/method, considering all methods of each system and only methods with at least a single API call

Regarding the second ratio, the system representing the median has 3.23 API calls/method. The data also shows that 50% of the projects have between 2.72 and 3.89 API calls/method, which reinforces our argument that real applications tend to call more than one API method to achieve their goals. Similarly to the first ratio, CMSCREENSOT is the system with the highest ratio because all methods call API methods.

Additionally, we performed a correlation test between the project's size (in terms of number of methods) and the usage of the API (in terms of number of calls to API methods). Initially, we checked whether the distribution of our data follows a

normal distribution, using the Shapiro-Wilk test. The test result was a *p-value* less than  $2.2E^{-16}$ , indicating that the data is not normally distributed. For this reason, we performed the Kendall Tau rank correlation test [Kendall, 1938] to measure the correlation between the defined variables. The result was a coefficient value equal to -0.43, which indicates that large projects tend to call API methods proportionally less than the smaller ones.

#### 4.3.1.2 Example Dataset

Different from the **Mining Dataset**, this second dataset follows more rigid criteria for selecting the projects. More specifically, the projects were manually selected based on their popularity among the Android developers community. Moreover, only open source projects and projects available for free download were considered.

A major goal of the proposed platform is to provide high-quality source code examples. However, to achieve this goal it is necessary that the source code used to extract the example present good quality, i.e., it should, preferably, follow good programming practices. In this sense, we claim that the selection of projects developed and maintained by larger groups of developers at least increase the chances of a data set with high quality code.

The projects selected for this dataset are presented in Appendix A.2. In short, we manually selected 151 projects that provided 287,263 examples (as detailed in Section 4.3.4). The main difference regarding the dataset described in the previous section consists of its more curated nature. Basically, contains less projects than the first dataset. In fact, this dataset is as a subset of the **Mining Dataset** since all projects from the **Example Dataset** are present in the **Mining Dataset**.

### 4.3.2 Transactions

As described in Section 3.3, the set of transactions (**Transactions Database**) is the main input to the association rules algorithm. We consider a transaction as a client method; moreover, the API methods called by such methods are the items.

However, the data presented in Section 4.3.1.1 shows that not all methods implemented by API clients include a call to an API method. Moreover, transactions with a single call to an API method are not relevant, for the particular purpose of mining association rules. For this reason, we first removed such methods from the database.

More specifically, we initially mined 512,993 transactions, where 397,183 (77.43%) are transactions that do not include calls to API methods and 47,150 transactions (9.19%) had a single call to an API method. Finally, 68,660 transactions (13.38%)

with least two calls to API methods were selected. We refer to transactions with more than one API method call as useful transactions.

Figure 4.4 shows the distribution of the number of method calls in the useful transactions. It is possible to observe that 37.66% of the useful transactions have a size equal to two and transactions greater than ten are very rare. Moreover, the median of the transactions is three (indicated by the first vertical line) and the mean is 4.6 (indicated by the second vertical line). In addition to the transactions shown in Figure 4.4, we retrieved 23 transactions with more than 40 API methods calls. Particularly, 10 of these transactions belong to the core application of the CYANOGENMOD project. For example, the method `onTransact(int, Parcel, Parcel, int)` in the class `android.app.ActivityManagerNative` from the project CYANOGENMOD CORE, has 137 calls to API methods. This method has 1,920 code lines and a major `switch` with multiples `case` statements.

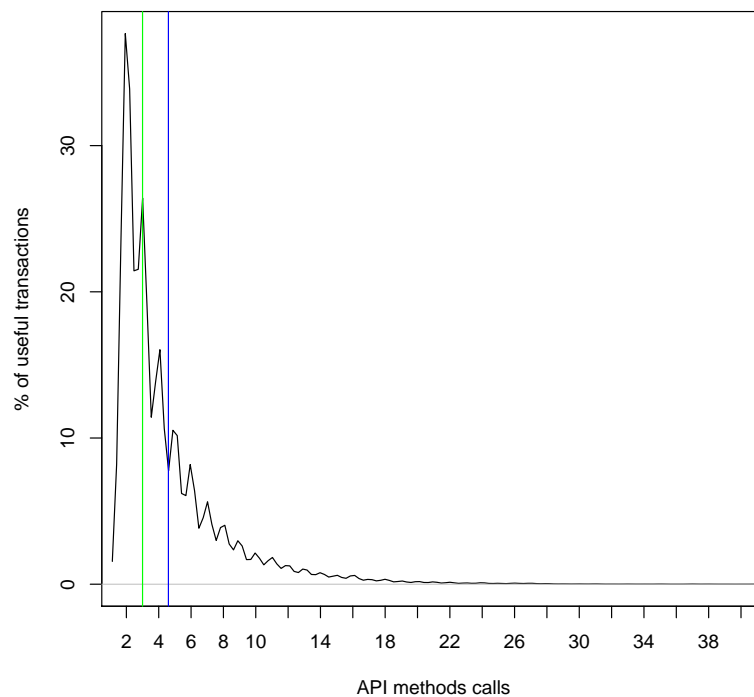


Figure 4.4: Distribution of the useful transactions

Figure 4.5 presents the percentage of useful transactions per project. On average, each project has 30.49% of useful transactions and the median is 26.53%. We can also observe that five projects have 100% of useful transactions (i.e., all methods implemented by these projects include at least two calls to API methods). A brief analysis

of such projects revealed that they are small projects, having between two and thirteen methods.

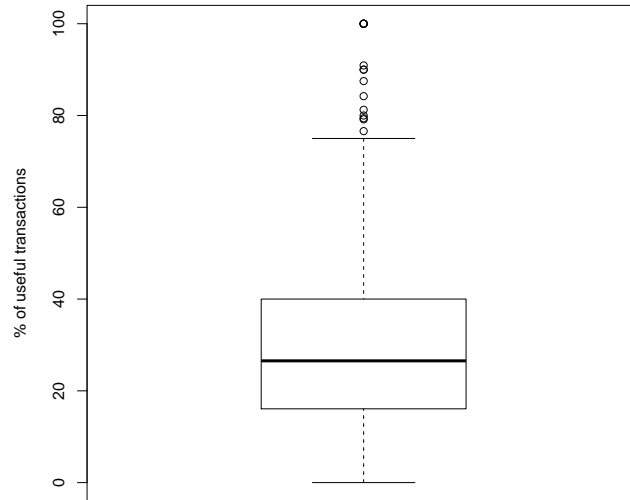


Figure 4.5: Percentage of useful transactions per project

Regarding the projects that more contributed with useful transactions, the CYANOGENMOD CORE project provided 11,322 useful transactions (16.49% of all useful transactions in the database). Table 4.1 presents the top five projects with more useful transactions. The high number of transactions in the CYANOGENMOD CORE project is due to the fact that it is the largest project in our database. In addition, it includes the complete code of the Android platform with their customizations. The second largest contributor project (ASTRID) has only 20% of the number of useful transactions provided by CYANOGENMOD CORE.

Table 4.1: Top five projects with more useful transactions

Project	Useful Transactions
CyanogenMod Core	11,322 (23.3%)
Astrid	2,307 (17.6%)
MiCode	2,290 (28.5%)
Openintents	1,228 (21.6%)
k9	1,080 (17%)

We also analyzed the most frequent transactions, as presented in Table 4.2. The most frequent transaction denotes calls to the

`Toast.makeText(Context, CharSequence, int)` and `Toast.show()` API methods, which were called together in 212 transactions from 76 projects. Analyzing the Android documentation, we can observe that `Toast.makeText(Context, CharSequence, int)` method is a static method that returns an object of the type `android.widget.Toast`. Basically, this type is used to create and show a view containing a quick message. Particularly, `Toast.show()` is called to show the view in the device.

Table 4.2: Top five most common transaction items

Frequent items	Frequency	# Projects
<code>Toast.makeText(Context, CharSequence, int)</code> <code>Toast.show()</code>	212	76
<code>ContentResolver.query(Uri, String[], String, String[], String)</code> <code>Context.getContentResolver()</code>	171	38
<code>Activity.onCreate(Bundle)</code> <code>Activity setContentView(View)</code>	161	47
<code>getMenuInflater()</code> <code>MenuInflater.inflate(int, Menu)</code>	142	81
<code>Log.d(String, String)</code> <code>Log.e(String, String)</code>	137	52

The first, second, and fourth items in Table 4.2 follow the same usage pattern, which consists of calling a method to obtain an object used as another call. The third item occurs when an application needs to create a new view. First the application sets the content of the view by calling the `Activity setContentView(View)` method and then calls the `Activity.onCreate(Bundle)` method from the super class. The fifth item is used to debug code in Android applications, using the `Log.d(String, String)` method to output debug messages and `Log.e(String, String)` method to output error messages. Under general terms, we perceive that these transactions are not restricted to few projects and for this reason they denote common patterns when using the involved API methods.

Another aspect we analyzed is the relationship of the API methods in the transactions. Basically, our objective was to identify whether the transactions are composed by methods from a single class or from different classes. The result of this analysis is showed in Table 4.3. We can observe that most transactions are composed by methods from more than one class (inter-class transactions). This result also emphasizes the complexity of real projects, since the relationship among methods from many classes demand more effort from developers, when understanding such calling patterns.

Finally, we analyzed how many API methods and API classes are covered by the transactions. As described in Section 4.2, the analyzed version of the Android API

Table 4.3: Intra-class and Inter-class calls in useful transactions

Type	Frequency
Intra-class	11,736 (17.09%)
Inter-class	56,924 (82.91%)

has 26,540 methods and 2,968 classes. From this total, 43.26% (11,483) of the API methods are present in the transactions and 38.86% (10,316) in useful transactions. The API classes have coverage of 48.85% (1,450) and 45.14% (1,340), respectively.

Tables 4.4 and 4.5 present the methods and classes most found in the transactions, respectively. In Table 4.4, we can observe that the most used API method is `Log.d(String, String)` with 6,729 calls, from 257 projects. However, the method present in more projects is `TextView.setText(CharSequence)`, with 5,028 calls from 310 projects. Basically, the first method outputs debug logs and the second one set a text for an object in an application view.

Table 4.4: Top five API methods most called in useful transactions

Method	Frequency	# Projects
<code>Log.d(String, String)</code>	6,729	257
<code>TextView.setText(CharSequence)</code>	5,028	310
<code>Context.getString(int)</code>	3,526	238
<code>View.findViewById(int)</code>	3,310	233
<code>Log.e(String, String)</code>	3,193	246

Regarding the classes, the most frequent used class is `android.view.View` with 40,967 uses in 343 projects. Basically, the classes listed in Table 4.5 reflect their importance in the API. The `android.view.View` class represents a basic way for building components in the user interface, the `android.app.Activity` class represents a basic class to interact with the users, the `android.util.Log` class provides methods to log events, the `android.content.Intent` class supports late operations, and finally `android.content.Context` class allows access to application-specific resources.

Table 4.5: Top five API classes most used in useful transactions

Class	Frequency	# Projects
<code>android.view.View</code>	40,967	343
<code>android.app.Activity</code>	27,633	371
<code>android.util.Log</code>	21,083	338
<code>android.content.Intent</code>	20,582	352
<code>android.content.Context</code>	16,389	345



### 4.3.3 Association Rules

In this section, we describe the decisions we made when selecting input values for the association rules mining algorithm and we also analyze the produced rules. As described in Section 3.3, APIMiner 2.0 does not provide an automated procedure to select automatically the best values from the input parameters. However, in this section we present the strategy we used to select these values.

The first step when mining association rules is selecting the minimum values of support and confidence (see Section 3.3). Our strategy aimed to select values that do not generate a very large number of rules, while maximizing the API methods coverage as possible. For that purpose, we performed several tests varying the values of support and confidence to observe the variation in the number of rules and their coverage.

Figure 4.6 shows the variation in the number of extracted rules by varying the support and confidence values. More specifically, we restrict the possible values of confidence to 70%, 80%, 90%, and 95% to keep a minimum quality in the rules. As we can observe, support values greater than 60 cause major reduction in the number of rules, despite the confidence threshold. On the other hand, support values less than 11 generate many rules (for example, 11,054 rules for a confidence value equal to 95% and 33,685 rules for a confidence value equal to 70%). Finally, we also observed that from support values equal to 21 the number of rules begins to grow more sharply. Thus, this support value represents a good candidate for selection according to our criteria.

Figure 4.7 shows the variation in the rules coverage. As we can observe, the variation in the support does not impact as much as the variation in the confidence value. Thus, we conclude that, from the viewpoint of coverage, the value of 70% for confidence covers a large proportion of methods without losing much in quality.

Based on the results and conclusions on the input values, we defined the minimum values of support and confidence as 21 transactions and 70%, respectively. As mentioned before, these values represent an interesting balance between the number of rules and the number of methods.

However, we emphasize that other support and confidence thresholds could be used. By using a minimum support value lower than 21 would increase the number of rules and the coverage of the API methods at the cost of obtaining of rules with a smaller degree of confidence. On the other hand, using a minimum support value greater than 21 would cover fewer API methods.

The execution of the **Patterns Analyzer** module returned 38,661 rules, including rules with antecedents having more than one method. Considering only rules whose antecedent size is equal to one, we obtained 1,952 rules. These rules are used to build

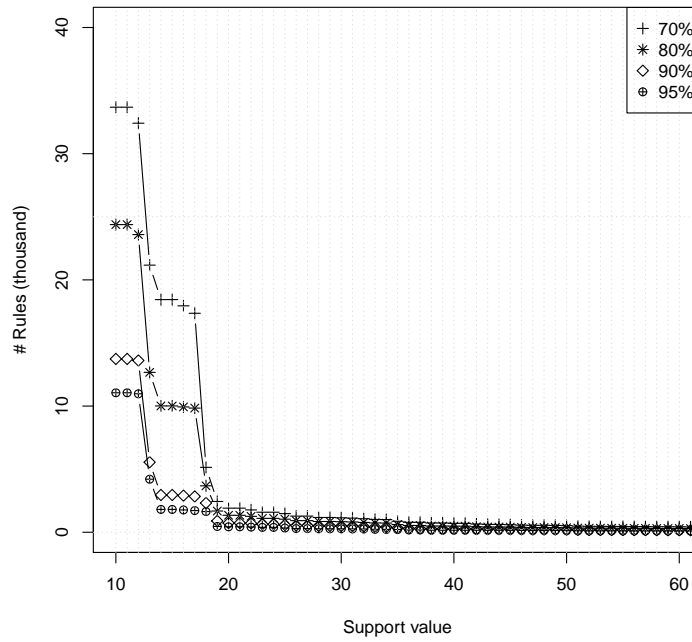


Figure 4.6: Number of rules by varying the minimum support value

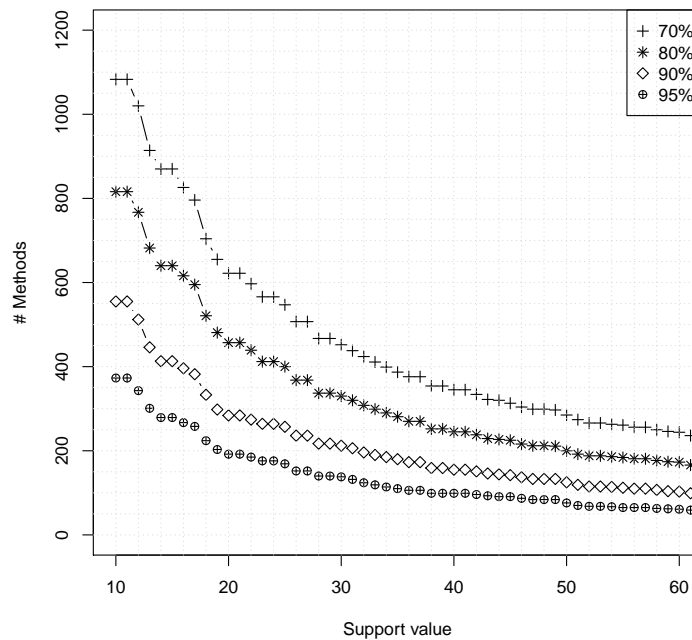


Figure 4.7: API methods coverage by varying the minimum support value

the final usage patterns. Moreover, the rules cover 624 distinct API methods, which represent coverage of 2.35% of all Android API methods and 6.05% of the Android API methods that appear in useful transactions. Furthermore, the rules cover 191 API classes, which represent 6.43% of all API classes and 13.17% of the API classes in useful transactions.

Table 4.6 shows the top five API methods with more rules (i.e., the antecedent of the rule contains only the cited method) and their average support. As we can observe, the API methods with more rules are not present in the most frequent transactions or among the most frequently called API methods (as showed in Tables 4.6 and 4.4, respectively). This fact is explained by the lower average support of the rules, which is close to the minimum value defined for the extraction.

Table 4.6: Top five methods with more rules and their average support

API Method	# of rules
<code>ViewGroup.addViewInLayout(View, int, ViewGroup.LayoutParams)</code>	63 (22.09)
<code>PackageManager.queryIntentActivityOptions(ComponentName, Intent[], Intent, int)</code>	41 (36.34)
<code>ListFragment.setEmptyText(CharSequence)</code>	41 (22.90)
<code>View.getWindowVisibleDisplayFrame(Rect)</code>	41 (41.97)
<code>View.resolveSizeAndState(int, int, int)</code>	41 (34.82)

Table 4.7 presents the five rules with higher support values. Initially, we can observe that the first and second rules include the same methods present in the most frequent transactions (Table 4.2). Moreover, we can also observe that the third and fourth rules include the same methods, but with the antecedent and consequent inverted. On the other hand, the methods in the fifth rule do not appear in the most frequent transactions, but they constitute the rule with the highest confidence among the listed rules. More specifically, from 1,044 occurrences of the method `AlertDialog.Builder.setTitle(CharSequence)` in useful transactions, in 1,019 transactions (97.60%) the method `AlertDialog.Builder.Builder(Context)` is also present.

#### 4.3.4 Examples

The example extraction process resulted in 287,263 examples. More specifically, 102,442 examples were extracted for single API methods and 184,821 examples for the usage patterns. Figure 4.8 shows the distributions of the examples according to the number of API methods called. We can observe that most examples include calls to only one API method (36%), and that the examples including calls to two API methods are minority (6%).

Table 4.7: Top five rules with higher support

Rule	Support	Confidence
Activity setContentView(View) ↓ Activity onCreate(Bundle)	1,362	75.08%
Toast.show() ↓ Toast.makeText(Context, CharSequence, int)	1,133	86.09%
ViewGroup.getChildCount() ↓ ViewGroup.getChildAt(int)	1,077	75.15%
ViewGroup.getChildAt(int) ↓ ViewGroup.getChildCount()	1,077	73.66%
AlertDialog.Builder.setTitle(CharSequence) ↓ AlertDialog.Builder.Builder(Context)	1,019	97.60%

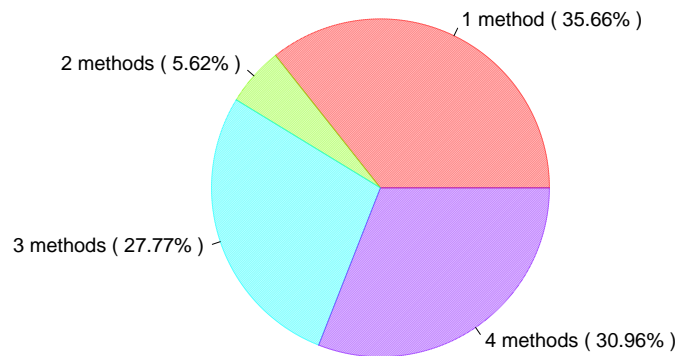


Figure 4.8: Number of API method calls in the examples

Moreover, the examples cover 3,915 Android API methods (14.75%) and 549 classes (18.5%). The five API methods with more examples are presented in Table 4.8. The `Activity.findViewById(int)` and `View.findViewById(int)` methods are used to retrieve an object in the view, the `TextView.setText(CharSequence)` method sets the text of the component, the `Context.getString(int)` method gets a localized string from the application, and the `Log.d(String, String)` method outputs debug messages.

Additionally, these five methods concentrate 11.75% of the examples for single methods.

Table 4.8: API methods with more examples

Method	# Examples
Activity.findViewById(int)	3,239
TextView.setText(CharSequence)	2,818
Log.d(String, String)	2,366
Context.getString(int)	2,126
View.findViewById(int)	1,489

Table 4.9 shows the five methods that most appeared in all examples, with emphasis on the `AlertDialog.Builder.Builder(Context)`, `AlertDialog.Builder.setTitle(CharSequence)`, and `AlertDialog.Builder.create()` methods. These methods are typically used together to construct dialogs that display up to three buttons.

Table 4.9: Methods that most appeared in all examples

Method	# Examples
AlertDialog.Builder.Builder(Context)	31,935
AlertDialog.Builder.create()	17,176
TextView.setText(CharSequence)	16,486
AlertDialog.Builder.setTitle(CharSequence)	15,803
View.findViewById(int)	13,886

The five classes that most appeared in the examples are presented in Table 4.10 together with their number of methods. The `android.view.View` class is the Android class with more methods in the API (511 methods). In this sense, it is natural that it is present in most examples. On the other hand, the `android.app.AlertDialog.Builder` class has fewer methods (38 methods), but it is also present in a large number of examples. More specifically, this class is present in 129,100 examples, which represents an average of 3,397 examples per method. Moreover, three methods of this class contribute with more than half of the presences (50.28%).

At the package level, the packages that most appeared in all examples are presented in Table 4.11. The `android.app` package contains high-level classes of the Android application model, the `android.view` package provides classes to handle screen layouts and interaction with users, the `android.content` package contains classes for accessing and publishing data on devices. Finally, the `android.widget` package contains UI elements to use on application screens, and the `android.database` package contains classes to manage databases.

Table 4.10: Classes that most appeared in all examples

Class	# Methods	# Examples
android.view.View	511	136,635
android.app.AlertDialog.Builder	38	129,100
android.app.Activity	187	48,444
android.content.Intent	140	32,930
android.content.Context	91	25,670

Table 4.11: Packages that most appeared in all examples

Package	# Examples
android.app	207,398
android.view	203,869
android.content	134,905
android.widget	54,330
android.database	37,691

We also extracted some metrics from the examples aiming to evaluate quality aspects. Table 4.12 presents the values for such metrics extracted from all examples. The `CONDITIONAL_STATEMENTS` metric returns the number of conditional statements (like `if`, `for`, and `while`) in the examples. The data shows that, on average, one out of three examples includes a conditional statement, which is an evidence that the examples have low cyclomatic complexity. The `LOC` (Line of Code) metric computes the size of the examples, which amounted 3,969,660 code lines. Declarations of variables that were not identified (`UNDISCOVERED_DECLARATIONS`) during the summarization process achieved an average of three declarations at each two examples. Finally, the `UNHANDLED_EXCEPTIONS` metric computes the number of exceptions that are not handled, including runtime exceptions. We highlight that the values measured for `UNDISCOVERED_DECLARATIONS` and `UNHANDLED_EXCEPTIONS` emphasize the importance of the readability improvements proposed in Section 3.4.2.

Table 4.12: Quality metrics (all examples)

Metric	Max	Min	Average	Sum
<code>CONDITIONAL_STATEMENTS</code>	30	0	0.31	90,615
<code>LOC</code>	290	1	13.82	3,969,660
<code>UNDISCOVERED_DECLARATIONS</code>	152	0	1.46	418,376
<code>UNHANDLED_EXCEPTIONS</code>	49	0	0.02	7,531

Regarding the size of the examples, Figure 4.9 presents the distribution of the examples by their size. More specifically, the first box plot shows the distribution

regarding single API method calls while the second one presents the size of the examples considered when mining for usage patterns. Moreover, outliers are not included to facilitate visualization.

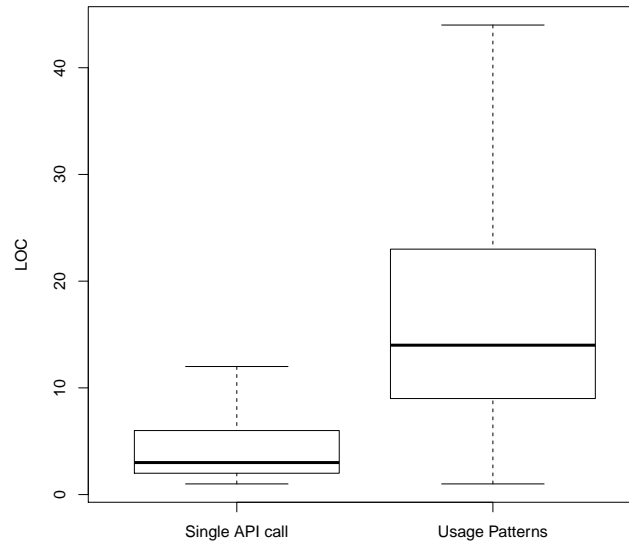


Figure 4.9: Distribution of the examples size

As we can observe in the first box plot, half of the examples have at most three lines of code and only 25% of the examples have six or more lines of code. Although the examples involve the invocation of a single API method, we consider such number of lines of code as good values because methods may require parameters. In this sense, the presence of variables in the example causes a natural increase in their size. On the other hand, the size of the examples considered while mining for usage patterns are considerably higher. More specifically, half of the examples have at most 14 lines of code and 25% have 23 lines of code or more. We consider such values adequate because 91.27% of the examples for usage patterns call at least three API methods.

### 4.3.5 Usage Patterns

As described in Section 3.5, the ranking engine supports basically two tasks: (i) ranking the examples including a single API call, previously stored in the **Examples Database**; and (ii) ranking examples for the already mined association rules. In the first task, the platform ranked the examples for 3,915 distinct API methods. In the second task, first the platform selected and ranked the examples for the association rules described in

Section 4.3.3. Next, the platform identified the association rules which had examples and generated and ranked the usage patterns.

In particular, for some usage patterns we did not identify examples because the **Examples Dataset** does not include all projects from the **Mining Dataset**. More specifically, we were not able to find examples for 280 association rules (14% of 1,952). Consequently, 88 API methods out of the 624 distinct API methods with rules were not included for not presenting examples.

In order to evaluate the rank criteria for single API methods, we analyzed the first 20 examples extracted for the `Activity.findViewById(int)` API method (which is the API method with more examples). We found that all examples have size equal to one and do not present compilation problems. Moreover, we identified four main ways of using this method: (i) by returning the result of the call (Listing 4.1); (ii) by assigning the value to a variable (Listing 4.2); (iii) by passing the result of the call to another method (Listing 4.3); and (iv) by performing a casting to specific types (Listing 4.4).

---

Listing 4.1: Using `Activity.findViewById(int)` to return a value

---

```
1 return findViewById(R.id.veecheck_no);
```

---



---

Listing 4.2: Using `Activity.findViewById(int)` in an assignment

---

```
1 View root = findViewById(R.id.form_root);
```

---



---

Listing 4.3: Using `Activity.findViewById(int)` as a target

---

```
1 findViewById(R.id.edit_name).requestFocus();
```

---



---

Listing 4.4: Using `Activity.findViewById(int)` in a casting

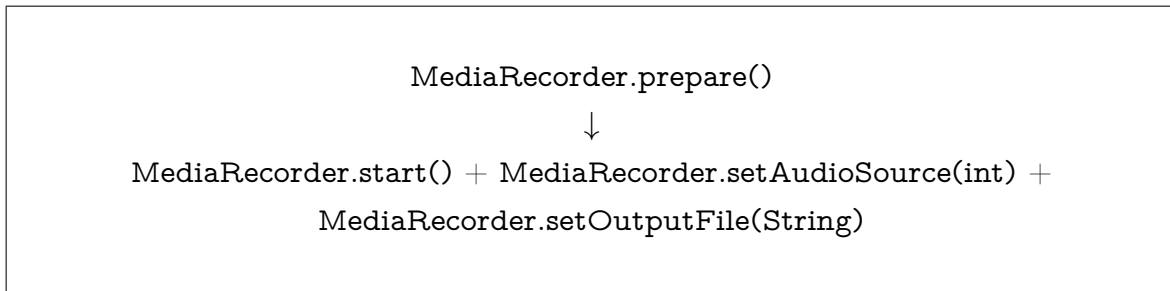
---

```
1 Button b = (Button) findViewById(R.id.clear);
```

---

To evaluate the ranking of usage patterns, we selected the `MediaRecorder.prepare()` method, used by many association rules. The first usage pattern for this method is:





with lift value of 1986.69. Moreover, another usage pattern has the same lift value but the number of methods in the rule (second criterion) is smaller.

Listing 4.5 presents the first example for the aforementioned usage pattern. As we can observe, the methods in the usage pattern are called in lines 3, 4, 6, and 12 and they are called in the same order as the example in the original documentation<sup>4</sup>. Moreover, the call to `MediaRecorder.prepare()` (line 6) happens within `try/catch` clause. On the other hand, we also identified that the example does not include other necessary API methods as `MediaRecorder.setOutputFormat(int)` and `MediaRecorder.setAudioEncoder(int)`.

Listing 4.5: First example of the first usage pattern for `MediaRecorder.prepare()` method

---

```

1 android.media.MediaRecorder mRecorder; //initialized previously
2 java.lang.String mFileName, AUDIO_DIR; //initialized previously
3 mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
4 mRecorder.setOutputFile(AUDIO_DIR + mFileName);
5 try {
6     mRecorder.prepare();
7 } catch ( IllegalStateException e) {
8     //do something
9 } catch ( IOException e) {
10    //do something
11 }
12 mRecorder.start();

```

---

Finally, we conclude that the ranking criteria followed by APIMiner 2.0 achieved satisfactory results because it resembles the example from the original Android API documentation. Moreover, it was possible to derive usage patterns for most of the association rules mined by the previous modules, even significantly reducing the number of projects in the **Examples Dataset**.

<sup>4</sup><http://developer.android.com/reference/android/media/MediaRecorder.html>

## 4.4 Field Study

To investigate how the community used this new version of the platform, we conducted a field study using Android APIMiner 2.0, available at [www.apiminer.org](http://www.apiminer.org). More specifically, our study reproduce the study of Montandon [2013], but aiming to evaluate the new features from this new version.

The evaluation was based on data collected in the period of 13th May 2013 to 11th October 2013 (five months). Similarly to the study conducted for the version 1.0, the data was obtained from a private logging service and from Google Analytics<sup>5</sup>. However, the private logging service was modified to log requests for usage patterns.

During the time frame considered in our study, Android APIMiner 2.0 received a total of 32,335 visits. This number represents 29.1% more visits per month than the version 1.0. Figure 4.10 presents the number of visits per week to our version. We emphasize that in the reported period we did not make any promotion in websites, only in academic events through publications and paper presentations.

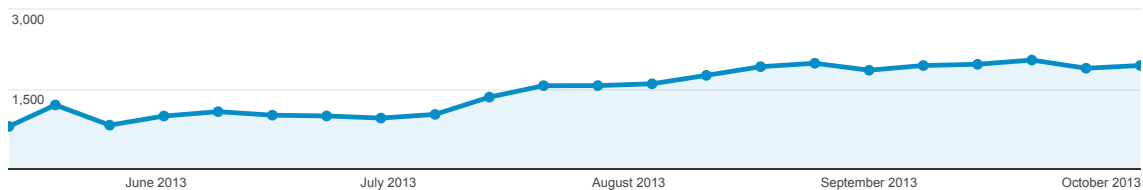


Figure 4.10: Number of visits per week

Regarding demographic information, Table 4.13 presents the top ten countries in number of visits and the their total participation in the number of visits. As we can observe, the country with more visits was India (5,089 visits) followed by United States (3,137 visits), and Brazil (1,692 visits). These three countries concentrate one third of all visits, and India had doubled its number of visits whereas the number of visits coming from the United States was maintained.

As result, these visits generated 5,545 requests for single API examples. More specifically, the number of requests by clicking in the **Example** button was 2,601 (46.9%). Therefore, 2,944 requests (53.1%) were made by navigating between the examples, which shows that users often seek for different examples for the same method.

Figure 4.11 presents the number of requests for examples—for single API methods—per week. We can observe that the number of requests increased in the period in the same way that the number of visits.

<sup>5</sup><https://www.google.com/analytics>

Table 4.13: Top ten countries in visits

Country	# Visits
India	5,089 (15.74%)
United Stated	3,137 (9.7%)
Brazil	1,692 (5.23%)
Japan	1,513 (4.68%)
South Korea	1,507 (4.66%)
Germany	1,100 (3.4%)
Taiwan	1,053 (3.26%)
United Kingdom	1,028 (3.18%)
France	856 (2.65%)
Canada	855 (2.64%)

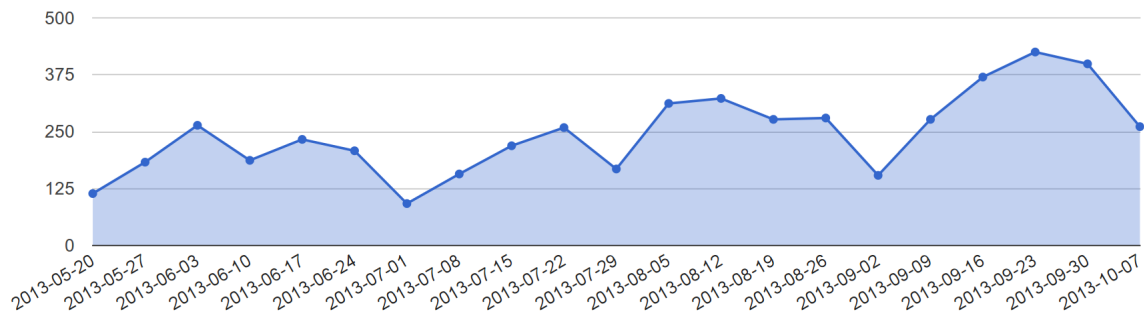


Figure 4.11: Number of examples provided per week

Table 4.14 presents the top ten methods with more requests of single examples and their number of available examples. The first method with more example requests was `SQLiteDatabase.beginTransaction()` (209 requests). However, this number is explained by the fact that we use the method as an example in the homepage of the platform.

Moreover, we can observe that most examples have some relationship such as:

- (A) `SQLiteDatabase.beginTransaction()`  
`SimpleCursorAdapter.SimpleCursorAdapter(Context,int,Cursor,String[],int[])`  
`SimpleCursorAdapter.setViewBinder(SimpleCursorAdapter.ViewBinder)`  
`SQLiteDatabase.insertOrThrow(String, String, ContentValues),`
- (B) `ViewPager.OnPageChangeListener.onPageSelected(int)`  
`ViewPager.OnPageChangeListener.onPageScrollStateChanged(int),`
- (C) `RectF.RectF(float, float, float, float)`

Table 4.14: Top ten methods with most requests for examples (#1) and their number of available examples (#2)

Method	#1	#2
SQLiteDatabase.beginTransaction()	209	36
ViewPager.OnPageChangeListener.onPageSelected(int)	101	4
SimpleCursorAdapter.SimpleCursorAdapter(Context,int,Cursor,String[],int[])	91	32
RectF.RectF(float, float, float, float)	57	71
Point.Point(int, int)	56	52
ViewPager.OnPageChangeListener.onPageScrollStateChanged(int)	55	4
SimpleCursorAdapter.setViewBinder(SimpleCursorAdapter.ViewBinder)	50	12
BitmapRegionDecoder.decodeRegion(Rect, BitmapFactory.Options)	50	1
SQLiteDatabase.insertOrThrow(String, String, ContentValues)	48	7
RectF.RectF()	44	24

Point.Point(int, int)

BitmapRegionDecoder.decodeRegion(Rect, BitmapFactory.Options)

RectF.RectF()

(A) which are used for manipulating databases and presenting queries results. (B) which refers to callbacks for changing states of pages. (C) which refers to codes to manipulate rectangle regions in images.

On the other hand, the methods in Table 4.14 do not have usage patterns, with the exception of `SQLiteDatabase.beginTransaction()`. In the Top 100 methods, 19 methods have at least one usage pattern.

Regarding the usage patterns, the visits generated 176 requests. Moreover, from 536 API methods with usage patterns, 135 API methods received at least one click on the “Example” button (event that allows the platform presents the examples and usage patterns). Table 4.15 presents the top ten methods with most requests for usage patterns, their number of single example requests, and usage patterns. Similarly to the previous table, `SQLiteDatabase.beginTransaction()` method concentrates the largest number of usage pattern requests, which is also explained by its usage the main page of Android APIMiner 2.0.

Figure 4.12 shows the relation between the number of example requests for usage patterns and for single API methods, considering only the methods with user requests. We can observe that the median has a ratio of 45% (i.e., from around two example requests for single API method is made an example request for an usage pattern). Particularly, `android.content.DialogInterface.cancel()` method received more example requests for usage patterns than single API methods. In the total, we obtained 31

Table 4.15: Top ten methods with most requests for usage patterns #1, their number of single example requests #2, and usage patterns #3

Method	#1	#2	#3
SQLiteDatabase.beginTransaction()	131	209	3
Toast.setGravity(int, int, int)	12	43	3
AlertDialog.Builder.create()	6	8	1
SQLiteDatabase.setTransactionSuccessful()	3	10	3
ContentValues.ContentValues(ContentValues)	3	3	1
DialogInterface.cancel()	3	2	5
Toast.makeText(Context, int, int)	2	9	1
AudioTrack.getState()	2	7	7
CursorWrapper.moveToNext()	2	5	4
Toast.setText(CharSequence)	2	5	1

evaluations of examples by end-users, distributed over 26 API methods. The methods with more evaluations are:

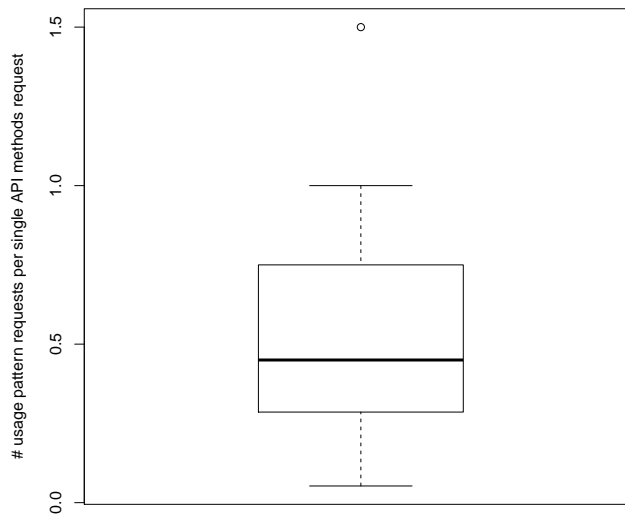


Figure 4.12: Relation between the number of examples requests for usage pattern and single API method

- `Color.rgb(int, int, int)` (three evaluations)
- `ViewPager.OnPageChangeListener.onPageSelected(int)` (two evaluations)
- `Intent.getStringArrayExtra(String)` (two evaluations)

- `NotificationCompat.Builder.setVibrate(long[])` (two evaluations)

Listing 4.6 and Listing 4.7 present the two examples of the `NotificationCompat.Builder.setVibrate(long[])` method evaluated by the end-users. The example in Listing 4.6 received a positive evaluation, while the example in Listing 4.7 received a negative one. We can observe that the first example is more complete (for example, all variables are initialized in the example’s scope). On the other hand, in the second example the variables are not initialized. This fact reinforces the findings of Buse and Weimer [2012], regarding the user’s preference examples for where all the variables are declared and initialized in the example’s scope.

Listing 4.6: Example for the method `NotificationCompat.Builder.setVibrate(long[])` (with a positive evaluation)

---

```

1 NotificationCompat.Builder notificationBuilder = new NotificationCompat.Builder(
2     context);
3 long [] vibratePattern = null;
4 vibratePattern = parseVibratePattern(vibrate_pattern_raw);
5 notificationBuilder .setVibrate(vibratePattern);

```

---

Listing 4.7: Example for the method `NotificationCompat.Builder.setVibrate(long[])` (with a negative evaluation)

---

```

1 long [] vibrationPattern; // initialized previously
2 android.support.v4.app.NotificationCompat.Builder builder; // initialized previously
3 builder .setVibrate(vibrationPattern);

```

---

## 4.5 User Study

To shed light on the benefits and drawbacks of our platform, we conducted a controlled experiment with 29 subjects who were invited to use both Android APIMiner, and the traditional Android documentation. More specifically, we proposed this study to answer the following question:

Do the recommendations provided by APIMiner 2.0 help developers to implement maintenance tasks with less effort?

To answer this question, we designed a study including two corrective maintenance tasks that were performed by the subjects. As a restriction, the subjects were asked to implement one task accessing the documentation provided by Android APIMiner 2.0, and the second task accessing Android's traditional JavaDoc. Moreover, we analyzed the relationships between the subjects' profile and the time they spent on each task.

This section is organized as follows. Section 4.5.1 presents the configuration of the study, which includes the target application, the proposed tasks, and the desired profile of the participants. Section 4.5.2 describes how the study was executed. Finally, Section 4.5.3 describes our findings and Section 4.6 discuss threats to validity.

### 4.5.1 Study Setup

In this study, we decided to use a strategy similar to the one followed to evaluate the first version of the platform. Basically, we developed three corrective maintenance tasks that the subjects were invited to perform. These tasks are related to the target system, **More Aqui** app, which consists of an Android application to help users who are looking for properties on sale. Basically, **More Aqui** allows users to register new properties or consult registered properties. **More Aqui**'s implementation relies on many resources of the Android platform such as: (i) database manipulation, (ii) localization services, (iii) user interface elements, and (iv) event handling.

Figure 4.13 presents three screens from **More Aqui**. Figure 4.13(a) presents the main screen of the application, which allow users to register new properties (Figure 4.13(b)) by clicking in the button **Novo** or to list all properties already registered (Figure 4.13(c)) by clicking in the button **Visualizar**.

Regarding the tasks, they were designed to represent real maintenance activities. In other words, each task represents an application functionality that as implemented do not follow certain requirements. The proposed maintenance tasks are described as follows:

1. **Obtaining location:** When a user carries out the inclusion of a property, the application collects the last registered location using the location provider service available on the Android devices. The initial problem is that one of the criteria for selecting the provider is wrong and the implementation does not address scenario where there are no available provider;
2. **Inserting new properties:** After the user fills out the information and clicks on the **Done** button, the data is validated and inserted into the database. However,

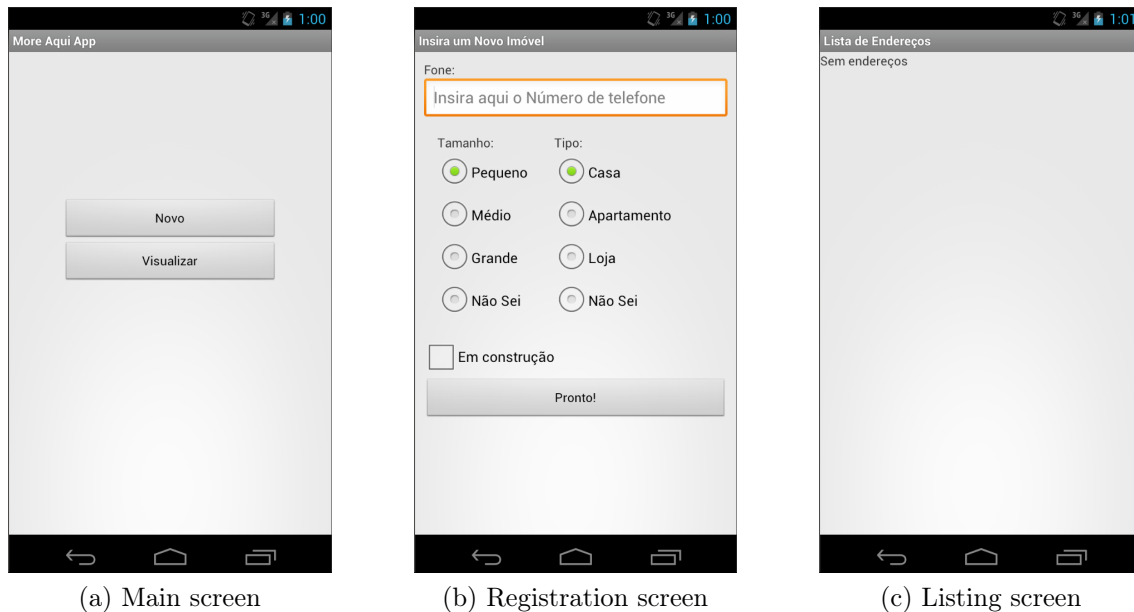


Figure 4.13: Screenshots from *More Aqui*

as initially implemented, the system does not handle possible problems when inserting the property, like connection problems;

3. **System messages:** For some user actions it is necessary to present a response to the user, using little messages. The proposed task demands a modification in the code so that the message appears centralized and with a long duration.

For each task, we presented to the subjects a detailed description of the requirements that the code should attend. Moreover, we developed an automated test that checks whether such requirements are implemented correctly.

Regarding their complexity, the tasks have the same difficulty considering that the number of lines of code and the number of presented problems are the same. The main aspect that can facilitate the tasks' implementation is prior knowledge on certain technologies. For instance, subjects with expertise in manipulating databases could perform better than other subjects without this knowledge. Furthermore, the **Obtaining location** task was used to explain the experiment to the subjects (as discussed in details in the next section) and the **Inserting new properties** and **System messages** tasks were used in the experiment. Particularly, the use of different domains in each task does not allow them to become simpler after the implementation of the previous task.

When selecting the subjects, we defined that they should have skills on the Java programming language, but they should not be experts in the Android platform. This



information was obtained through a characterization form, presented in Appendix B.1. At this point, we emphasize that the identification of the subjects was not requested in this form. However, we offered the subjects the opportunity to receive the compiled results of the experiments along with their performance in relation to other subjects.

In the experiment we followed a crossover methodology, which consisted of dividing the subjects in two groups (A and B). Group A did the first task accessing the traditional Android documentation and the second task accessing the documentation provided by Android APIMiner 2.0. In contrast, group B did the first task accessing the Android APIMiner 2.0 and the second task using the traditional Android documentation. Thus, each subject did a task in each evaluated approach and at the end we obtained an equal number of samples for each approach to be analyzed. The division of the subjects in groups was made randomly.

We prepared a laboratory with 20 computers for the experiment execution. Each computer has the following settings: Dell Optiplex 790 with Intel Core I3 3.30 GHz, 8 GB RAM, 1 TB of HD, and Windows 7 Professional 64 bits. Each station was configured with Eclipse IDE and the Android ADT version 22.3.0 (latest version) along with a pre-defined workspace.

## 4.5.2 Experiment Execution

The experiment included 29 subjects and was divided in three sessions. Basically, the first session was executed on 20th September 2013 with nine postgraduate students, the second session was executed on 14th October 2013 with nine postgraduate students, and the third session was also executed on 14th October 2013 with 11 graduate students.

A subject declared that he had no knowledge in Java and therefore he was eliminated from the experiment. Furthermore, four subjects dropped out before the time limit allowed and were also eliminated. Regarding the subjects distribution, group A stayed with 12 subjects and group B also stayed with 12 subjects. In other words, the elimination of the subjects did not affect the final distribution of the subjects.

Figure 4.14(a) summarizes the knowledge of the subjects in the Java programming language, according to the answers in the form. Most of the subjects rated their knowledge as “Basic” (nine subjects, 37.5%) and “Intermediary” (nine subjects, 37.5%). The other six subjects rated their knowledge as “Expert” (25%). Figure 4.14(b) summarizes the knowledge of the subjects on the Android platform. As expected, most subjects stated they had no knowledge in this platform (18 subjects, 75%). Five subjects rated their knowledge as “Basic” and one rated their knowledge as “Intermediary”.

Moreover, Figure 4.15 summarizes the professional experience of the subjects.

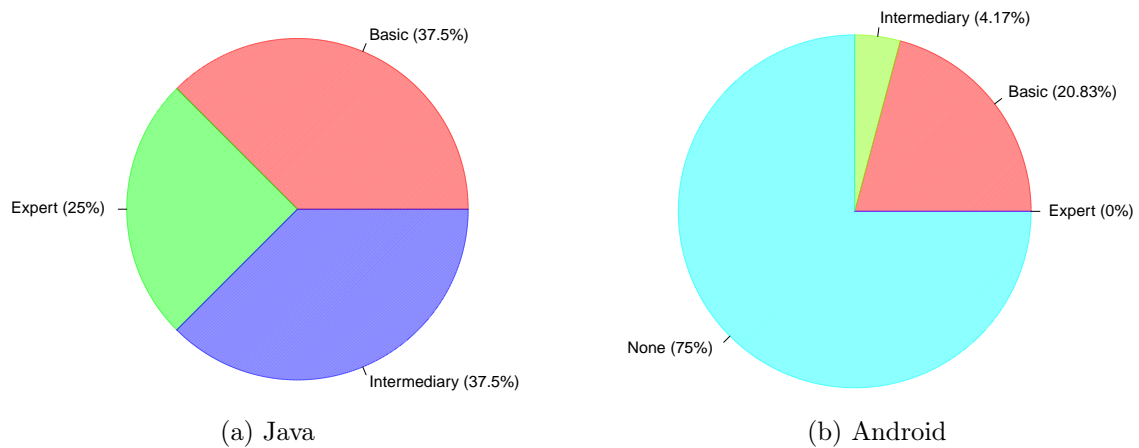


Figure 4.14: Subjects' expertise

Most subjects (10 subjects, 41.67%) reported more than three years of experience and eight subjects (33.33%) reported that never worked professionally. Furthermore, five subjects reported between one and three years of experience and one subject reported less than one year of experience. As we can observe, the experiment included a representative group of subjects regarding their professional experience.

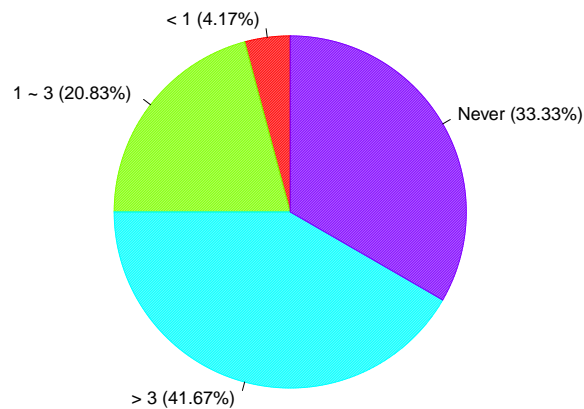


Figure 4.15: Professional expertise of the subjects (in years)

We started the experiment with a brief introduction on Android development, highlighting the main topics the subjects should know when implementing the tasks. Among these topics, we showed how to use the development tools and the **More Aqui** application. Next, we used the **Obtaining location** task as a training task. Basically,

the following methodology was recommended when implementing the tasks:

1. **Read the description:** First the subjects were instructed to read the entire description of the task. This description includes information on how the application should behave, the requirements that the code should attend, and specific instructions such as the documentation to be used (traditional or instrumented by APIMiner 2.0);
2. **Register the starting time:** With emphasis on this statement, we asked the subjects to fill the form with the current time;
3. **Run the automated test:** As a first step, the subjects were instructed to run the automated test to verify the current problem of the code, and to get initial context to start the task;
4. **Perform the task:** When implementing the tasks, the participants were instructed to access a specific documentation. The task is considered successfully completed when the code is accepted by the automated testing. If the time limit expires, the task is considered failed and the subjects were instructed to move to the next task;
5. **Register the ending time:** After finishing a task, the subjects were instructed to fill the form with the current time.

The tutorial task was used to illustrate this methodology, but we did not limit the maximum time for the tasks. After the initial presentation and the tutorial task, the subjects were informed that the maximum time for performing each task was 25 minutes. Because the tasks are independent from each other and checked by an automated testing, we did not follow each subject individually during the experiment. However, 25 minutes after starting the first task we warned those subjects who failed to move to the next task.

Besides informing the maximum time, we also informed the subjects that the tasks should be done individually and no questions would be answered, since all necessary information is available in the form distributed or in the source code.

### 4.5.3 Experiment Results

Table 4.16 presents the time spent (in minutes) in each task by the subjects, the session they participated and the group they were assigned to. In this table, the subjects are identified by the codes P1 to P29. Moreover, **Task 1** denotes the **Inserting**

**new properties** task and **Task 2** denotes the **System messages** task. The  $\infty$  symbol means that the subject failed to complete the task and the group indicates which documentation was used in each task. Group A means that the **Task 1** was conducted using the traditional documentation and the **Task 2** using the documentation provided by Android APIMiner 2.0. In the group B, the documentations used in each task are reversed.

Table 4.16: Performance of the subjects in the experiment

Subject	Session	Task 1	Task 2	Group
P1	1	24	16	A
P2		$\infty$	10	B
P3		$\infty$	3	B
P4		$\infty$	20	A
P5		22	$\infty$	B
P6		$\infty$	19	A
P7		$\infty$	14	A
P8		$\infty$	22	A
P9		19	24	B
P11	2	25	9	A
P12		$\infty$	7	B
P13		13	6	A
P15		25	16	B
P16		15	$\infty$	A
P17		$\infty$	15	A
P19	3	14	7	B
P20		10	11	B
P21		17	6	A
P22		$\infty$	15	A
P23		$\infty$	24	B
P25		$\infty$	$\infty$	B
P26		$\infty$	23	A
P27		$\infty$	$\infty$	B
P28		$\infty$	$\infty$	B

Table 4.17 shows the number of subjects that completed the tasks using Android APIMiner 2.0 and with the traditional documentation. Initially, we can observe that 10 participants (41.66%) finished the first task, including five subjects using the traditional documentation and five using Android APIMiner 2.0. On the other hand, 19 subjects finished the second task. Among the five participants who failed to complete this task, four used the traditional documentation and one used Android APIMiner 2.0.

After inspecting **Task 1** results, we concluded that the knowledge in Java and the experience of the subjects in each group is the same. The main difference among

Table 4.17: Number of subjects who completed the tasks

<b>Task</b>	<b>Android APIMiner 2.0</b>	<b>Traditional JavaDoc</b>
Task 1	5	5
Task 2	11	8
<b>Total</b>	<b>16</b>	<b>13</b>

the subjects is their knowledge on Android. Table 4.18 shows the Android’s knowledge of the subjects who completed the **Task 1** and the documentation they used in this task. As we can observe, the group using Android APIMiner has more subjects without knowledge on Android (3 subjects). Therefore, there are evidences that the documentation provided by Android APIMiner helped such participants in finishing this task.

Table 4.18: Expertise on Android development of the subjects who concluded the **Task 1**

<b>Documentation</b>	<b>None</b>	<b>Basic</b>	<b>Intermediary</b>
Android APIMiner	3	1	1
Traditional	2	3	0

Regarding **Task 2**, the group using the documentation provided by Android APIMiner 2.0 has also more subjects without knowledge on Android (Table 4.19). Moreover, the number of subjects who concluded this task using Android APIMiner 2.0 outperforms the number of subjects that concluded the task using the traditional documentation.

Table 4.19: Expertise on Android development of the subjects who concluded the **Task 2**

<b>Documentation</b>	<b>None</b>	<b>Basic</b>	<b>Intermediary</b>
Android APIMiner	7	4	0
Traditional	6	1	1

Regarding the time spent in the tasks, Table 4.20 presents the minimum time, the maximum time, the average time, and the number of subjects who concluded the tasks using each of the evaluated documentation. As we can observe in Table 4.16, P13 is the subject who concluded **Task 1** in less time (13 minutes), using the traditional documentation. Moreover, the maximum time spent by subjects who concluded the tasks in each group is 25 minutes (the maximal allowed time). Finally, the average time spent by the subjects using Android APIMiner was slightly lower than the average time

spent by the subjects who used the traditional documentation—18 minutes against 18 minutes and 48 seconds, respectively.

Table 4.20: Time spent in the tasks (in minutes)

Task	Documentation	Minimum	Maximum	Average	#(%)
#1	Android APIMiner	14	25	18	5 (41.67%)
	Traditional	13	25	18.8	5 (41.67%)
#2	Android APIMiner	6	24	15	11 (91.67%)
	Traditional	3	24	12.75	8 (66.67%)

Considering **Task 2**, P3 is the subject who concluded the task in less time (3 minutes), using the traditional documentation. Meanwhile, P21 concluded the task in 6 minutes, using Android APIMiner. However if we observe the other times we can classify P3 an outlier in this task. On the other hand, the average time spent by the subjects using the traditional documentation is less than the time spent using Android APIMiner 2.0. However, the difference between the number of subjects who completed the tasks is also considerable (3 subjects or 37.5% more subjects).

Regarding the example requests made by the subjects when performing their tasks, we registered 686 example requests for single API methods distributed over 48 API methods and 182 example requests for usage patterns distributed over 13 API methods. On average, each subject requested 23 examples for single API methods and 6 examples for usage pattern examples.

## 4.6 Threats to Validity

We organize threats to validity in four categories, as proposed by Wohlin et al. [2000]:

**Construct Validity:** The first decision of our study is over the set of client systems to compose the **Mining Dataset** and **Examples Dataset**. Regarding the second dataset, we followed a more rigid criteria when selecting the client systems to be used as example providers; this decision resulted in some usage patterns without examples. However, we observed that the usage patterns and the extracted examples are very close to the ones produced by real developers.

Also regarding the client systems, we claim that the selection of projects that are developed and maintained by larger groups of developers at least increases the chances of a dataset with high quality code. However, this assumption is hard to be formally proved. The analysis by an expert in each client system would be extremely

time-consuming, subjective, and non replicable. Moreover, Stamelos et al. [2002] investigated the structural code quality of well-known Linux applications and identified that the vast majority of program components are acceptable by the community of developers. Finally, in our user study we included only students (undergraduate and postgraduate). However, we also observed that most subjects had professional expertise. More specifically, 62.5% of all subjects had one or more years of expertise in professional software development.

**Internal Validity:** When creating the instance for the Android API, we decided to select not the highest possible values for support and confidence. As discussed in Section 4.3.3, lowest values of support and confidence can cover a highest number of API methods, although producing usage patterns that are less representative. Regarding the user study, the subjects were randomly assigned to groups and some were later removed, since they did not finish the tasks. However, the groups remained balanced regarding the number of subjects and therefore the analysis was not affected.

**External Validity:** When collecting the access data in Android APIMiner 2.0, we do not identify the profile of the visitors. We argue that this information is not important, since the documentation provided by APIMiner 2.0 is not restricted to any developer profile. Moreover, in our evaluation we always used the particular instance for the Android API. Thus, our results are specific for this API and may not be generalized to other APIs. To a better comprehension on the impact of APIMiner 2.0, we need to create other instances for APIs with different levels of documentations, from incomplete to the well-documented APIs.

**Conclusion Validity:** Regarding the user study, our tasks and target application may not be representative enough to reproduce real development scenarios. However, we noted that many API methods used by the tasks are among the methods most used by Android applications (Section 4.2).

## 4.7 Final Remarks

In this chapter, we presented a particular instantiation for the Android API. This instantiation, called Android APIMiner 2.0, provides 287,263 source code examples obtained from 151 open-source Android applications and 1,672 usage patterns mined

from 396 open-source Android applications. We presented a characterization study on the use of the Android API by such applications. We found, for example, that more than 40% of the client's methods in the **Mining Database** call at least one API method. On the other hand, only 38% of the API methods were used in the useful transactions.

A field study based on data collected after five months of use by real Android developers showed that the access to Android APIMiner 2.0 remained in constant growth, besides a greater number of requests for single API methods and for usage patterns examples. Moreover, a user study with 29 participants showed that APIMiner 2.0 was particularly useful for subjects who do not have knowledge on Android. When compared with APIMiner 1.0, we observed that the usage patterns of usage patterns helped the users in the implementation of the proposed maintenance tasks, specially the ones involving several API elements.



# Chapter 5

## Conclusion

### 5.1 Contributions

In this master dissertation, we tackled the problem faced by developers when learning new APIs. Particularly, APIs' documentations are the central learning resource accessed by API users [Robillard and DeLine, 2010]. However, the poor quality of this documentation is a major obstacle for their use. In order to make more productive the use of APIs, several studies propose the use of source code example as an essential resource in API documentations [McLellan et al., 1998; Robillard, 2009; Robillard and DeLine, 2010; Hou and Li, 2011; Buse and Weimer, 2012; Duala-Ekoko and Robillard, 2012]. Moreover, Nasehi et al. [2012] and Buse and Weimer [2012] argue that the example's readability is a critical factor for developers when searching for usage examples.

In this master dissertation, we presented the APIMiner 2.0 platform that provides examples for API usage patterns, which also consider some readability aspects of source code examples. Specifically, the contributions of the APIMiner 2.0 are as follows:

- Usage pattern examples: We proposed in this work a new approach to recommend usage patterns in API documentations in the JavaDoc format. First, the usage patterns are mined from a collection of API client systems using data mining algorithms. Moreover, the usage patterns are ranked and presented according to the strength of the subjacent association rules;
- Readability improvements: We implemented an algorithm that modifies the examples to consider some attributes of readability suggested by the recent literature. Among such improvements, we can mention the use of abstract initialization of variables and comments in empty code blocks;

- **Implementation:** We implemented the proposed solution and made it available for public use. Moreover, it provides a set of functions that support the users when using APIMiner 2.0, like functions to define the inputs for the **Patterns Analyzer** module;
- **Android APIMiner 2.0:** We implemented and configured a particular version of the APIMiner 2.0 platform for the Android API. As result, we obtained 1,952 usage patterns for 624 API methods and 287,263 examples. Android APIMiner is publicly available at <http://apiminer.org>;
- **Practical Evaluation:** We evaluated the Android APIMiner 2.0 instance under three dimensions: (i) by analyzing all data produced in the process of creating the instance, (ii) by analyzing the data obtained from its usage by real Android developers, and (iii) by conducting a controlled experiment with 29 subjects to evaluate the gains provided by Android APIMiner.

## 5.2 Comparison with Related Work

In Chapter 2 we presented the related works to our solution. In this section, we compare such works with APIMiner 2.0.

### 5.2.1 IDE-based Recommendation Systems

IDEs provide a set of features to support the development process. Particularly, IDE-based tools can explore the developer’s environment to provide more relevant recommendations. However, the recommendations provided by such tools are not focused on documentation. Typically, IDE-based tools recommend code snippets based on information gathered from the project in which the developer is working.

The VCC tool also collects information from the project that the developer is working and uses as input to an algorithm for sequential pattern mining [Silva Jr et al., 2012]. To make the recommendations, the approach collects existing statements in a method and searches for code sequences that match the one used as input. However, in API documentations, the usage of sequential patterns is not suitable because of the lack of contextual information. Typically, developers access API documents for learning how to use specific methods. Therefore, the recommendation of methods frequently called together, as supported by APIMiner 2.0, is more appropriate since the required input is just a target method.

The MACs tool recommends frequent API usage patterns mined from relevant source code files retrieved in real time from a code search engine [Hsu and Lin, 2011]. Basically, the tool receives as input one or more statements and provides related code snippets as result. Because it is an approach that builds recommendations in real time, the number of retrieved files from the code search engine is limited to 20 files. This restriction may limit the result of the mining process, and therefore it may contribute to the lack of representativeness of the mined usage patterns. On the other hand, APIMiner 2.0 is based on a private source code repository to mine and recommend usage patterns, therefore allowing full control over the data source.

In common, the examples provided by such tools are not focused on documentation and do not consider readability characteristics. On the other hand, the examples provided by APIMiner 2.0 are slices of code extracted from real projects improved with readability aspects.

## 5.2.2 API Documentation

Unlike IDE-based tools, JavaDoc-based tools are designed and implemented to be independent from IDEs and usually are publicly accessed from the web (as APIMiner). Moreover, this type of documentation depends on limited contextual information.

APIMiner 1.0 instruments the standard Java-based API documentation format with concrete source code examples, extracted from a private repository [Montandon, 2013]. Moreover, the code examples are summarized using a standard static slicing algorithm that does not address aspects of examples readability. Furthermore, the examples provided by the tool are focused on single API methods. On the other hand, APIMiner 2.0 supports examples for API methods frequently called together.

EXOADOCS is a tool that searches, summarizes, and automatically embeds API documents with code examples [Kim et al., 2009]. Like the MACs tool, EXOADOCS relies on code search engines to obtain the source code files used to extract the examples. Moreover, the tool uses a summarization algorithm to extract the examples and a clustering algorithm to identify different usages. However, the instrumented JavaDoc provided by EXOADOCS must be regenerated whenever a new source code example is processed. On the other hand, APIMiner 2.0 relies on data mining algorithms to identify and recommend API usage patterns and examples that illustrate such patterns.

Finally, both EXOADOCS and MACs do not consider issues related to the examples readability. In contrast, the examples provided by APIMiner 2.0 consider some attributes of readability presented in the literature.

## 5.3 Future Work

In Section 4.3.3, we presented our strategy to select the minimum values of support and confidence. Basically, we aimed to do not generate a large number of rules, while maximizing the API methods coverage as possible. As a result, we mined a reasonable number of rules for a small group of API methods. Therefore, we recommend the investigation of other strategies for selecting the support and confidence values. Particularly, we recommend the investigation of automatic strategies to choose such values, which do not depend on manual calibration.

As observed in Section 4.3.4, some API methods have a massive number of examples. Despite that, the proposed approach does not implement procedures to detect different usage types among the examples. In this sense, we recommend the investigation of techniques to detect such patterns in order to avoid the presentation of very similar examples.

Our current ranking algorithm is based on three criteria: (i) **Completeness**; (ii) **LOC**; and (iii) **Feedback score**. Particularly, the **Completeness** criterion was introduced in this new version of the APIMiner platform and has focus mainly on the readability of the examples. In this sense, we recommend the investigation of other ranking criteria, mainly based on their readability. Specifically, traditional metrics like McCabe’s Complexity may represent a good criteria for example ranking.

As observed from the results of the Field Study (Section 4.4), even using some workarounds to reduce readability problems, users still prefer examples where all variables and initializations are available. Therefore, we recommend the investigation of other static slicing implementations to consider these aspects. More specifically, we recommend an investigation concerning the scope delimitation of the slicing algorithm.

# Bibliography

- Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *Conference on Programming Language Design and Implementation (SIGPLAN)*, volume 25, pages 246--256.
- Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *12th International Conference on Management of Data (SIGMOD)*, volume 22, pages 207--216.
- Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases (VLDB)*, pages 487--499.
- Agrawal, R. and Srikant, R. (1995). Mining Sequential Patterns. In *11th International Conference on Data Engineering (ICDE)*, pages 3--14.
- Bayardo, Jr., R. J. and Agrawal, R. (1999). Mining the most interesting rules. In *5th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 145--154.
- Borges, H., Felix, D., Montandon, J. E., Costa, H. A. X., and Valente, M. T. (2013). APIMiner 2.0: Recomendação de exemplos de uso de apis usando regras de associação. *IV Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas)*, pages 1--5.
- Buse, R. P. L. and Weimer, W. (2012). Synthesizing API usage examples. In *34th International Conference on Software Engineering (ICSE)*, pages 782--792.
- Duala-Ekoko, E. and Robillard, M. (2012). Asking and answering questions about unfamiliar APIs: An exploratory study. In *34th International Conference on Software Engineering (ICSE)*, pages 266--276.
- Eclipse, F. (2013). Eclipse - The Eclipse Foundation open source community website.

- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software. *ACM SIGKDD Explorations Newsletter*, 11(1):10.
- Han, J., Kamber, M., and Pei, J. (2006). *Data Mining: Concepts and Techniques (2nd edition)*. Morgan Kaufmann.
- Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *19th International Conference on Management of Data (SIGMOD)*, volume 29, pages 1--12.
- Happel, H.-J. and Maalej, W. (2008). Potentials and challenges of recommendation systems for software development. In *2nd International Workshop on Recommendation Systems for Software Engineering (RSSE)*, page 11.
- Harman, M. and Hierons, R. (2001). An overview of program slicing. *Software Focus*, 2(3):85--92.
- Hou, D. and Li, L. (2011). Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *19th International Conference on Program Comprehension (ICPC)*, pages 91--100.
- Hsu, S.-K. and Lin, S.-J. (2011). MACs: Mining API code snippets for code reuse. *Expert Systems with Applications*, 38(6):7291--7301.
- Kendall, M. (1938). A new measure of rank correlation. *Biometrika*, 30(1/2):81--93.
- Kim, J., Lee, S., won Hwang, S., and Kim, S. (2009). Adding Examples into Java Documents. In *24th International Conference on Automated Software Engineering (ASE)*, pages 540--544.
- Kim, J., Lee, S., won Hwang, S., and Kim, S. (2010). Towards an intelligent code search engine. In *24th Conference on Artificial Intelligence (AAAI)*, pages 1358--1363.
- Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155--163.
- Mar, L. W., Wu, Y.-C., and Jiau, H. C. (2011). Recommending Proper API Code Examples for Documentation Purpose. In *18th Asia-Pacific Software Engineering Conference (APSEC)*, pages 331--338.
- McLellan, S., Roesler, A., Tempest, J., and Spinuzzi, C. (1998). Building more usable APIs. *IEEE Software*, 15(3):78--86.

- Mockus, A. and Herbsleb, J. D. (2002). Expertise browser: a quantitative approach to identifying expertise. In *24th international conference on Software engineering (ICSE)*, pages 503--512.
- Montandon, J. E. (2013). Documenting application programming interfaces with source code examples. Master's thesis, Federal University of Minas Gerais.
- Montandon, J. E., Borges, H., Felix, D., and Valente, M. T. (2013). Documenting APIs with Examples: Lessons Learned with the APIMiner Platform. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 401--408.
- Nasehi, S. M., Sillito, J., Maurer, F., and Burns, C. (2012). What makes a good code example?: A study of programming Q&A in StackOverflow. In *28th IEEE International Conference on Software Maintenance (ICSME)*, pages 25--34.
- Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M., and Gordon, M. (2002). What Programmers Really Want: Results of a Needs Assessment for SDK Documentation Janet. In *20th International Conference on Computer Documentation (SIGDOC)*, pages 133--141.
- Okazaki, N. and Tsujii, J. (2010). Simple and efficient algorithm for approximate dictionary matching. In *23rd International Conference on Computational Linguistics (COLING)*, pages 851--859.
- Oracle, C. (2013). Welcome to NetBeans. <https://netbeans.org/>.
- RecSys (2009). 3rd ACM International Conference on Recommender Systems.
- Robillard, M. (2009). What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27--34.
- Robillard, M. and DeLine, R. (2010). A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703--732.
- Robillard, M. and DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703--732.
- Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80--86.
- Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013). Recommending move method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232--241.

- Silva Jr, L. L. N. d., de Oliveira Alexandre Plastino, T. N., and Murta, L. G. P. (2012). Vertical code completion: Going beyond the current ctrl+ space. In *6th Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SB-CARS)*, pages 81--90.
- Stamelos, I., Angelis, L., Oikonomou, A., and Bleris, G. L. (2002). Code quality analysis in open source software development. *Information Systems Journal*, pages 43--60.
- Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2013). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1--28.
- Uddin, G., Dagenais, B., and Robillard, M. (2011). Analyzing temporal API usage patterns. In *26th International Conference on Automated Software Engineering (ASE)*, pages 456--459.
- Uddin, G., Dagenais, B., and Robillard, M. (2012). Temporal analysis of API usage concepts. In *34th International Conference on Software Engineering (ICSE)*, pages 804--814.
- Venkatesh, G. A. (1991). The semantic approach to program slicing. *ACM SIGPLAN Notices*, 26(6):107--119.
- WebStorm (2013). IntelliJ IDEA — The Best Java and Polyglot IDE.
- Weiser, M. (1981). Program slicing. In *5th International Conference on Software Engineering (ICSE)*, pages 439--449.
- Weiser, M. (1984). Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352--357.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). Experimentation in software engineering: An introduction. *The Kluwer International Series In Software Engineering*.
- Zaki, M. J. and Meira Jr, W. (2014). *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press.
- Zaki, M. J., Parthasarathy, S., Ogihara, M., and Li, W. (1997). New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, volume 20, pages 283--286.



- Zhong, H., Xie, T., Zhang, L., and Pei, J. (2009). MAPO: Mining and recommending API usage patterns. *ECOOP Object-Oriented Programming*, 5653:318–343.



# Appendix A

## Android APIMiner 2.0 Datasets

Table A.1: Mining Dataset

System	Description	Transactions
360 Engine for Android	360 Engine for Android	3,321
4Chan Image Browser	4Chan Image Browser	88
4Dnest	—	391
AA-C4C	Computer Academic Assoc	31
Absolute Android RSS	RSS Reader for Absolute Android	78
aCal	CalDAV calendar client	2,012
AccelerometerPlay	Android 16 Samples	19
ACRA	Application Crash Reports for Android	342
ActionBarCompat	Android 16 Samples	109
ActionBarSherlock	Library for action bar design pattern	2,721
ActionBarSherlock-v4	—	2,453
ADW Launcher	Launcher	1,145
Aerogear android	Android library for AeroGear	614
Agendatech android	agendatech para android	65
Agit	Agit - Git client for Android	1,017
Alien Blood Bath	2D platform shooter	205
Allplayers android	Android app for AllPlayers.com	298
Amarino	Interface to Arduino via Bluetooth	247
And Bible	Study the Bible on Your Android Mobile	2,720
AndAR	Augmented Reality on the Android platform	439
Andless	audio player	141
Andlytics	Android Market statistics app	3,555
AndNav	GPS navigation program	2,649
Andro Sens	Displays sensor data	10
Android actionbar	Android Action Bar Implementation	38
Android Archetypes	Maven Archetypes for Android development	58
Android async http	An Asynchronous HTTP Library for Android	152
Android autostarts	Tool to manage autostarts	160
Android BitmapCache	Specialised cache for use with Android Bitmap	101
Android database sqlcipher	SQLite API based on SQLCipher	581
Android delicious	—	290

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
Android File Manager	An open source file manager	101
Android GPUImage	Android filters based on OpenGL	394
Android Launcher Plus	Launcher	811
Android mapviewballoons	Information balloon annotation for MapView	71
Android menudrawer	A slide-out menu implementation	441
Android Metronome	Metronome application	15
Android Microblog	Microblogging client	363
Android network discovery	Android network tool	151
Android pedometer	Steps counter app	161
Android proxy library	Android Proxy Library (APL)	112
Android PullToRefresh	Pull-to-Refresh UI Pattern for Android	400
Android Simple Social Sharing	Reusable instrument for sharing	1,108
Android SlideExpandableListView	A better ExpandableListView	55
Android store	Open Code Project for In App Purchasing	453
Android support v4 googlemaps	A port of the Android Compatibility	2,499
Android Terminal Emulator		663
Android ui design pattern	Demo of Android UI Design Patterns	152
Android Universal Image Loader	Async image loading, caching and displaying	549
Android viewflow	A horizontal view scroller library	115
Android ViewPagerIndicator	Paging indicator widgets	264
Android VNC Viewer	VNC Remote Desktop	716
Android xbmcremote	Official XBMC Remote for Android	5,193
Android xbmcremote sandbox	Playing with new features	354
Android-ocr	Optical character recognition on Android	225
AndroidBeamDemo	Android 16 Samples	8
AndroidBillingLibrary	Market In-app Billing Library	301
Androidquery	AndroidQuery	1,085
AndroidRivers	Anxiety free news reader for Android	6,482
androidtracks	Android Tracks	128
AndroidUIFundamentals	—	146
Androkom	Android KOM client	244
Andorm	An Object Relational Mapper for Android	591
Andtweet	Twitter Client	417
AntennaPod	A podcast manager for Android	1,420
APG	OpenPGP for Android	435
ApiDemos	Android 16 Samples	2,102
Apktool	A tool for reverse engineering Android apk files	556
APM android	Android version of Universal Password Manager	298
APN	Mobile Data Switch for Android	6
AppNavigation	Android 16 Samples	24
Apps Organizer	Organize installed applications using labels	1,388
Aptoide Client	Alternative application installer	1,480
ARViewer	Augmented reality application	657
Asqare	Stone-swapping puzzle game	325
Astrid	Astrid: Android's #1 Task Management Application	13,104
Augmented Reality Framework	Reality App on Android	319
AutobahnAndroid	WebSocket WAMP for Android	276
BackupRestore	Android 16 Samples	18

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
Barcode Scanner	Barcode reader	429
BasicGLSurfaceView	Android 16 Samples	11
Battery Notifier	Battery level on notification area	91
Battery Widget	Shows battery percentage and temperature	12
Beem	XMPP (Jabber) IM Client	1,288
BeTrains	BeTrains for android	555
BetterBatteryStats	Advanced battery stats	431
Bistro Math	—	24
Bitcoin android	Send and receive bitcoins	5,396
Bluegps4droid	A branch of bluegps4droid	81
BluetoothChat	Android 16 Samples	38
BluetoothHDP	Android 16 Samples	24
Book Catalogue	A book cataloging tool for Android phones	2,951
Box android sdk	Box Android library	572
BuddyCloud client	buddycloud android client	1,110
Calculon	DSL for Android views and activities	185
Campyre	A Campfire client for Android	250
Catroid	A Catrobat IDE for Android	5,899
Caverns of Fire	Shooter Game	1,662
Ccalabash android	Automated Functional testing based on cucumber	6,773
Cell Finder	—	201
CIDR Calculator	Simple Android based IP subnet calculator	63
Cling	UPnP/DLNA library for Java and Android	6,471
cocos2d	cocos2d for android	5,505
Congress android	Learning about new bills and laws	971
ConnectBot	SSH Client	1,919
Contact Owner	Displays your (or a friend's) contact information	30
ContactManager	Android 16 Samples	19
Cordova android	Mirror of Apache Cordova Android	1,578
Corporate Addressbook	Exchange contact (GAL) lookup client	437
Countdown Alarm	Provides a basic countdown timer	69
CrossCompatibility	Android 16 Samples	22
Crouton	Context sensitive notifications for Android	142
Crowdroid	Twitter Client	2,535
csci498	Android Development	125
CubeLiveWallpaper	Android 16 Samples	39
CW advandroid	—	742
CW android	—	413
CyanogenMod Android dalvik	Android DalvikVM	49,862
CyanogenMod Browser	Android Web Browser	604
CyanogenMod Contacts	Android Contacts application	482
Cyanogen Updater	—	308
CyanogenMod ApplicationsProvider	—	35
CyanogenMod Bluetooth	—	890
CyanogenMod CMScreenshot	Screenshot activity for CM	4
CyanogenMod CMStats	—	30
CyanogenMod CMUpdateNotify	—	60
CyanogenMod ContactsProvider	—	1,180

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
CyanogenMod Core	—	48,566
CyanogenMod DSPManager	—	49
CyanogenMod Exchange	—	941
CyanogenMod FileManager	OI File Manager for CyanogenMod	288
CyanogenMod FM	—	367
CyanogenMod Gallery3D	Android Gallery3D Application	1,407
CyanogenMod LatinIME	—	1,857
CyanogenMod Launcher2	—	1,617
CyanogenMod LockClock	—	173
CyanogenMod MediaProvider	Android MediaProvider	62
CyanogenMod PackageInstaller	—	34
CyanogenMod Phone	Phone app for Android	1,473
CyanogenMod Settings	Android settings app	1,349
CyanogenMod SoundRecorder	—	57
CyanogenMod Stk	—	104
CyanogenMod Tag	—	667
CyanogenMod TelephonyProvider	—	110
CyanogenMod Torch	Nexus One Torch	49
CyanogenMod VoiceDialer	Android Voice Dialer	109
DeskClock	Backporting for Donut	184
Device Samsung d710	—	75
Dialer2	Alternative dialer with T9 search	91
Diaspora-Webclient	A simple android/ Diaspora-Webclient	41
DiskUsage	Storage card usage viewer	513
DownloadProvider	Android DownloadProvider	90
Drag-sort-listview	Android ListView with drag and drop reordering	294
Droid-fu	A utility library for your daily needs	793
Droidgiro android	Android scanner for swedish invoices	195
DroidDic	Spell checker / Crosswords cheater	127
DroidReader	Fast PDF Reader	105
EventBus	Android optimized event bus	337
Exchange OWA	Mail Client	354
FBReaderJ	e-book reader	5,404
FeedGoal	RSS/Atom Feed Reader	255
FFvideo Live Wallpaper	Video as the wallpaper	95
Floating Image	Continuous stream of floating images	1,674
Food-4-Thought	Android app for SDD	306
Frozen Bubble	Knock the bubbles	344
FTDriver	Android 3.x USB Host Serial Driver	48
gaeproxy	GAEProxy for Android	1,007
Gauges android	Gaug.es Android App	277
GCal Call Logger	Android Sync client	12
GCstar Scanner	—	9
GCstar Viewer	View your collections created with GCstar	136
Geeklist android	Android client for Geeklist	58
GeoBeagle	Search for geocaches and letterboxes near you	3,839
Geocaching4Locus	Download and import caches directly	504
GeocamMemoAndroid	Take geotagged notes and audio messages	105

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
GeocamMobileForAndroid	Take geotagged photos and upload	226
GeocamTalkAndroid	Send geotagged text and audio messages	238
GeoHunter	Find geocaches	1,233
Geoloqi Android SDK	Geoloqi example application	27
Gesture imageview	Implements pinch-zoom, rotate, pan as an ImageView	163
GestureBuilder	Android 16 Samples	35
Gmote	Computer Remote Control	233
Gnucash android	Gnucash companion application	372
Gobandroid	A Goban for Android	806
GPS Share	Minimalist location sharing	9
GreenDroid	GreenDroid development library	652
Greenhouse android	Greenhouse native Android client	350
HelloEffects	Android 16 Samples	21
Hellomap3d	Android 3D map SDK getting started	2,646
HideBar	Hides and restores the android systembar	45
Home	Android 16 Samples	71
HoneycombGallery	Android 16 Samples	69
Hot Death	Variant on the classic card game Uno	319
Hubroid	The Original Awesome GitHub App for Android	342
iFixitAndroid	Official iFixit Android App	591
Iglaset	App for iglaset.se	424
IrssiNotifier	Notifies from irc private messages	363
Jamendo android	Official Jamendo Android Player	749
JetBoy	Android 16 Samples	45
Jota Text Editor	Text editor for Android	1,233
K9	Email client	6,349
Keepassdroid	KeePass implementation for android	2,483
KeyChainDemo	Android 16 Samples	23
Lexic	Word-grid game - against the clock!	247
LibreGeoSocial	Geolocated Social Network	1,508
LibreSoft Gymkhana	Open source educational and geolocated game	672
libVoyager	OBD/CAN Vehicle network communications library	643
Life Saver	SMS and call log SD backup	46
LogAcceleration	Accelerometer values logger and grapher	39
Love android	Löve2d for android phones	411
LunarLander	Android 16 Samples	33
m2e-android	Android Connector for M2E Maven Integration	190
Madvertise android sdk	madvertise SDK for Android	150
MandelBrot	Fractal Viewer	197
Marine Compass	A nice looking 3D compass	26
Maven plugin samples	Usage examples for Maven Android Plugin	1,668
MemorizingTrustManager	SSL/TLS certificate management	31
MicDroid	Pitch correction effect made famous by T-Pain	203
Microlog4android	Microlog logging library to Android	322
MINDDroid	Controller for Lego Mindstorms NXT robots	222
Minitruco android	Port of miniTruco	459
Mixare	Augmented Reality Engine	895
Mobileorg android	MobileOrg for the Android platform	1,186

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
Motion Detection Framework	Motion detection	55
MultiResolution	Android 16 Samples	4
Mumble android	Android mumble client	2,486
Musicbrainz android	Official Android client to the MusicBrainz	1,151
Mustard	—	1,030
mWater Android App	Snaps a picture of a petrifilm	2,756
Net Meter	—	78
NetCounter	Network usage counter	400
Nethack Android	Rogue-like game NetHack	536
Newton’s Cradle	model the physics of Newton’s Cradle	48
NineOldAndroids	Android library for using the Honeycomb animation	644
Noise Alert	—	28
NotePad	Android 16 Samples	59
Novoda android	Examples of Android applications	771
Novoda ImageLoader	Library for async image loading and caching	501
Novoda RESTProvider	Automatically parse RESTful API responses	95
Novoda SQLiteProvider	Extended SQLite functionality for Android	480
Numberpicker	Example source of an number picker widget	84
Oauth for android	An OAuth Library/application	158
OI About	File manager	47
OneBusAway	OneBusAway	3,590
Onibus android	Encontre-se melhor no transporte coletivo!	2,481
Open Android Game	Open source game	94
Open WordSearch	a word search game for android	275
Openconferenceware	Open Source Bridge conference	56
openintents	OpenIntents applications	5,676
OpenMap Framework	OpenMap API	240
OpenSudoku	Sudoku numbers / maths puzzle game	527
Orbot	Tor proxy	207
OsmAnd	GPS navigation program	11,166
OsmTracker	GPS tracking with annotation	295
Osmtracker android	GPS tracking tool for OpenStreetMap	295
Otto	Enhanced Guava-based event bus	176
Password Hash	pwdhash implementation	61
Patchrom android	—	8,035
Pd for android	Pure Data for Android	326
Pedometer	Counts your steps	161
Pedometer remote service	—	40
Permissions	Shows permissions for installed android apps	96
Phonegap android	PhoneGap API	154
Picture Map	Show geotagged photos on a map	335
Play android	Play Android App	186
PMix	MPD client	175
Podax	Podcast client for Android	385
Pretty-Painter	Graphics editor for Android	138
Prey android client	Prey anti-theft software	644
ProgrammingAndroid2Examples	O’Reilly’s Programming Android	1,075
PullToRefresh ListView	Android ListView implementation	33

Continued on next page



Table A.1—continued from previous page

System	Description	Transactions
Pyload android	Android client for pyload	8,013
QPad	4-way breakout game for Android	56
QuasselDroid	Quasselclient for Android	561
Quran android	a quran reading application for android	765
RandomMusicPlayer	Android 16 Samples	70
RapidFTR	RapidFTR API	996
RenderScript	Android 16 Samples	3
Replica Island	a side-scrolling platformer for Android devices	1,562
Restful	RESTful Android application	139
Retrofit	Type-safe REST client	357
Ringdroid	Create ringtones with musics	249
Robospice	Writing asynchronous long running tasks	1,186
robotfindskitten	Android implementation of robotfindskitten	51
Robotic Space Rock		361
Robotium	Like Selenium	407
ROM Updater	Helps the user to maintain custom ROM updated	141
roman10 android tutorial	android tutorial source code	3,487
Root-tools	—	5,295
Ruboto-irb	IRB application for JRuby on Android	477
rVoix	Call recorder	184
SalesforceMobileSDK Android	Android SDK for Salesforce	961
SampleSyncAdapter	Android 16 Samples	121
Sanity	Call recorder/blocker/manager	810
Sat Info	—	93
Say My Name	reads out caller's name	61
SD Move	Move to SD Card manager	78
SD Watch	Move to SD card manager	22
SearchableDictionary	Android 16 Samples	31
Secrets	Application to securely store and manage passwords	187
SeriesGuide	Manage (re)watching favorite TV shows	1,339
ServeStream	HTTP(s) media server browser and stream player	779
Shortyz	A crossword puzzle client	939
Shuffle	Get Things done (GTD) application	1,337
Simon Tatham's puzzles	—	187
Simple App History Widget	App history widget	10
Simple Audio Widget	Audio volume widget	10
Simple Battery Widget	Battery Widget	9
Simple Cache Widget	Cache Widget	13
Simple CPU Widget	CPU Widget	9
Simple GPS Widget	GPS Widget	13
Simple Storage Widget	Storage space Widget	9
Simple Traffic Widget	Traffic Widget	10
Simple WLAN Widget	Wifi widget	10
SipDemo	Android 16 Samples	16
Sipdroid	SIP Client	2,606
SkeletonApp	Android 16 Samples	8
SL4A	Scripting Layer for Android (SL4A)	8,188
Slashdot	Slashdot reader	17

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
SlidingMenu	Create applications with slide-in menus	386
sls	Simple Last.fm Scrobbler	534
Sms backup plus	Backup SMS, MMS and call log	424
SMS Popup	SMS Messaging application	474
SMSSync	SMS gateway for Android powered phones	2,964
Snake	Android 16 Samples	34
Socialize sdk android	Socialize SDK for Android	6,455
SoftKeyboard	Android 16 Samples	87
Sokoban	Warehouse Puzzle Game	41
Solitaire Collection	Android Games	357
SparkleShare Android	Android Client for SparkleShare	213
Spell Dial	T9 search enabled dialer	60
SpellChecker	Android 16 Samples	16
Spinner	Android 16 Samples	13
SpinnerTest	Android 16 Samples	6
Spoon	Distributing instrumentation tests	460
Sri Lanka Train Schedule	Train Delays and Ticket Prices	99
StackMob Android	The official StackMob SDK	25
SubmatixBTAndroid4	SUBMATIX SPX42 Tauchcomputer	150
Subsonic	Home of the DSub Android client	4,111
SuperCollider Android	SuperCollider audio synthesis engine	58
SuperGenPass	Password hasher	189
SwallowCatcher	Podcast client for Android	134
SwiftP	FTP server for android	231
Taps of Fire	Frets on Fire (Guitar Hero like game)	1,073
Target	Anagram Word Puzzle	115
Taskwarrior androidapp	Taskwarrior for Android	2,853
Tea Timer	A straight-forward tea timer	96
TextSecure	Secure text messaging	2,876
TicTacToeLib	Android 16 Samples	34
TicTacToeMain	Android 16 Samples	2
TiltMazes	Logical puzzle game for the Android platform	96
Todo txt-touch	Managing your todo.txt file stored in Dropbox	3,426
Tomahawk android	Tomahawk's Android Music Player	790
TouchDB Android	CouchDB-compatible mobile database	712
ToyVpn	Android 16 Samples	10
Tram Hunter	Melbourne's Tram Tracker service	643
Transdroid	Android remote client	3,726
Trovebox	Trovebox mobile application for Android	4,295
TtsEngine	Android 16 Samples	18
Tumblife	Tumblr Client	422
TweetLanes	Tweet Lanes for Android	2,032
Twidere	Twitter client for Android	4,129
Twisty	Z-machine emulator	274
Twitli	Twitter Client	1,675
Twitterdroid	Twitterific for Android	103
Typographic Widgets	Weather, clock and battery widgets	178
USB	Android 16 Samples	52

Continued on next page

Table A.1—continued from previous page

System	Description	Transactions
Ushahidi Android	Stories on a maps	4,000
Vanilla	Vanilla Music Player	658
Veader	Chinese Ebook Reader	390
Vector-Pinball	Pinball game for Android	372
Vidiom	Mobile video blogging and web video publishing tool	5,321
VoicemailProviderDemo	Android 16 Samples	132
VoiceRecognitionService	Android 16 Samples	4
Voyager Connect	In-vehicle network diagnostic	24
VuDroid	PDF Reader	293
WeatherListWidget	Android 16 Samples	24
Webimageloader	Asynchronous image loading on Android	400
Weechat android	Weechat-Relay Android Client	417
Weiciyuan	Sina Weibo Android Client	3,397
WhatAndroid	The What.CD Android App	842
Wheelmap android	An android app for wheelmap	1,340
Wi-Fi Widget	Connect/disconnect Wi-Fi and show network SSID	14
WidgetPreview	Android 16 Samples	13
WiFiDirectDemo	Android 16 Samples	43
Wiktionary	Android 16 Samples	31
WiktionarySimple	Android 16 Samples	10
Wishlist	Android Utilities	224
WLANAudit-Android	Audit security of WLAN Access points	2,930
Word Seek	Word Search Game	190
Wordpress	WordPress CMS Client	1,146
XBMC Remote	Official XBMC Remote for Android	5,193
XmlAdapters	Android 16 Samples	87
Yaaic	Another Android IRC Client	3,317
YAXIM	Another XMPP Instant Messenger	362
Zandy	Zotero on Android	311

Table A.2: Examples Dataset

System	Description	Examples
4Chan Image Browser	4Chan Image Browser	138
aCal	CalDAV calendar client	6,044
ADW Launcher	Launcher	3,644
Agit	Android Git Client	1,134
Alien Blood Bath	2D platform shooter	346
Amarino	Interface to Arduino via Bluetooth	684
AndAR	Augmented Reality on the Android platform	350
And Bible	Study the Bible on Your Android Mobile	1,961
andless	audio player	918
AndNav	GPS navigation program	6,118
Android Launcher Plus	Launcher	2,713
Android Metronome	metronome application	55
Android Microblog	Microblogging client	762
Android Terminal Emulator	—	633
Android VNC Viewer	VNC Remote Desktop	1,038
Andro Sens	Displays sensor data	141
Andtweet	Twitter Client	1,797
APG	Android Privacy Guard	3,516
APN	Mobile Data Switch for Android	35
Apps Organizer	Organize installed applications using labels	1,658
Aptoid Client	Alternative application installer	3,249
ARViewer	Augmented reality application	2,345
Asqare	Stone-swapping puzzle game	708
Astrid	Task recording	29,285
Augmented Reality Framework	Reality App on Android	311
Barcode Scanner	Barcode reader	1,312
Battery Notifier	Battery level on notification area	335
Battery Widget	Shows battery percentage and temperature	101
Beem	XMPP (Jabber) IM Client	1,565
Bistro Math	—	82
Caverns of Fire	Shooter Game	42
Cell Finder	—	190
CIDR Calculator	Simple Android based IP subnet calculator	1,701
ConnectBot	SSH Client	2,306
Contact Ownder	Displays your (or a friend's) contact information	162
Corporate Addressbook	Exchange contact (GAL) lookup client	846
Countdown Alarm	Provides a basic countdown timer	344
Crowdroid	Twitter Client	10,588
Cyanogen Updater	—	1,437
Dialer2	Alternative dialer with T9 search	608
DiskUsage	Storage card usage viewer	1,400
DroidDic	Spell checker / Crosswords cheater	284
DroidReader	Fast PDF Reader	548
Exchange OWA	Mail Client	232
FBReaderJ	e-book reader	3,551
FeedGoal	RSS/Atom Feed Reader	1,906
FFvideo Live Wallpaper	Video as the wallpaper	168

Continued on next page

Table A.2—continued from previous page

System	Description	Examples
Floating Image	Continuous stream of floating images	3,549
Frozen Bubble	Knock the bubbles	403
GCal Call Logger	Android Sync client	239
GCstar Scanner	—	19
GCstar Viewer	View your collections created with GCstar	798
GeoBeagle	Search for geocaches and letterboxes near you	2,181
GeoHunter	Find geocaches	1,730
Gmote	Computer Remote Control	1,700
GPS Share	Minimalist location sharing	65
Hot Death	Hot Death is a variant on the classic card game Uno	356
K9	Email client	19,219
Keepassdroid	KeePass-compatible passphrase manager for Android	1,026
Lexic	Word-grid game - against the clock!	800
LibreGeoSocial	Geolocated Social Network	7,582
LibreSoft Gymkhana	Open source educational and geolocated game	3,538
libVoyager	OBD/CAN Vehicle network communications library	351
Life Saver	SMS and call log SD backup	126
LogAcceleration	Accelerometer values logger and grapher	170
MandelBrot	Fractal Viewer	293
Marine Compass	A nice looking 3D compass	198
MemorizingTrustManager	SSL/TLS certificate management	156
MicDroid	Pitch correction effect made famous by T-Pain	814
MINDDroid	Controller for Lego Mindstorms NXT robots	704
Mixare	Augmented Reality Engine	1,557
Motion Detection Framework	Motion detection	67
Mustard	—	3,286
NetCounter	Network usage counter	1,265
Nethack Android	Rogue-like game NetHack	1,114
Net Meter	—	159
Newton's Cradle	model the physics of Newton's Cradle	166
Noise Alert	—	158
OI About	File Manager	256
Openintents	PIM applications	20,959
OpenMap Framework	OpenMap API	66
OpenSudoku	Sudoku numbers / maths puzzle game	1,353
Open WordSearch	a word search game for android	2,440
Orbot	Tor proxy	671
OsmAnd	GPS navigation program	10,953
OsmTracker	GPS tracking with annotation	2,028
Password Hash	pwdhash implementation	71
Pedometer	Counts your steps	153
Pedometer remote service	—	209
Permissions	Shows permissions for installed android apps	681
Picture Map	Show geotagged photos on a map	37
PMix	MPD client	842
Replica Island	a side-scrolling platformer for Android devices	628
Ringdroid	Create ringtones with musics	681
robotfindskitten	Android implementation of robotfindskitten	250

Continued on next page

Table A.2—continued from previous page

System	Description	Examples
Robotic Space Rock	—	173
ROM Updater	Helps the user to maintain custom ROM updated	2,211
rVoix	Call recorder	1,116
Sanity	"Call recorder/blocker/manager	1,325
Sat Info	—	968
Say My Name	reads out caller's name	276
SD Move	"Move to SD Card" manager	199
SD Watch	Ultra-simple "Move to SD card" manager	56
Secrets	Application to securely store and manage passwords	386
ServeStream	HTTP(s) media server browser and stream player	4,376
Shortyz	A crossword puzzle client	1,122
Shuffle	Get Things done (GTD) application	1,720
Simon Tatham's puzzles	—	981
Simple App History Widget	App history widget	52
Simple Audio Widget	Audio volume widget	54
Simple Battery Widget	Battery Widget	37
Simple Cache Widget	Cache Widget	53
Simple CPU Widget	CPU Widget	54
Simple GPS Widget	GPS Widget	68
Simple Storage Widget	Storage space Widget	54
Simple Traffic Widget	Traffic Widget	56
Simple Wlan Widget	Wifi widget	52
Sipdroid	SIP Client	1,502
SL4A	Scripting Layer for Android (SL4A)	4,176
Slashdot	Slashdot reader	71
SMS Backup Plus	Backup and restore Android SMS/MMS	1,090
SMS Popup	SMS Messaging application	2,107
Sokoban	Warehouse Puzzle Game	439
Solitaire Collection	Includes Klondike, Spider Solitaire, and Freecell	242
Spell Dial	T9 search enabled dialer	120
Sri Lanka Train Schedule	Train Delays and Ticket Prices	249
SuperCollider Android	SuperCollider audio synthesis engine	131
SuperGenPass	Password hasher	728
SwallowCatcher	Podcast client for Android	1,769
Swiftp	FTP server for android	439
Taps of Fire	Frets on Fire (Guitar Hero like game)	616
Target	Anagram Word Puzzle	786
Tea Timer	A straight-forward tea timer	477
TiltMazes	Logical puzzle game for the Android platform	396
Tram Hunter	Melbourne's Tram Tracker service	1,814
Transdroid	Android remote client	16,506
Tumblife	Tumblr Client	353
Twisty	Z-machine emulator	492
Twitli	Twitter Client	3,490
Typographic Widgets	Weather, clock and battery widgets	659
Veader	Chinese Ebook Reader	3,830
Vector Pinball	Pinball Game	129
Vidiom	Mobile video blogging and web video publishing tool	1,300

Continued on next page

Table A.2—continued from previous page

<b>System</b>	<b>Description</b>	<b>Examples</b>
Voyager Connect	In-vehicle network diagnostic	99
VuDroid	PDF Reader	433
Wi-Fi Widget	Connect/disconnect Wi-Fi and show network SSID	42
Wordpress	WordPress CMS Client	6,257
Word Seek	Word Search Game	1,818
XBMC Remote	Official XBMC Remote for Android	7,123
Yaaic	IRC Client	14,599
Yaxim	Jabber/XMPP Client	2,234





# Appendix B

## Forms

### B.1 Subject Characterization Form

<b><u>Formulário – Experimento Controlado APIMiner 2.0</u></b>	
1. Marque os cursos/disciplinas que você já fez, ou então se tais conteúdos foram dados em conjunto com algum dos cursos que você fez.	
<input type="checkbox"/> Programação orientada a objetos	<input type="checkbox"/> Programação Java
<input type="checkbox"/> Programação Android	<input type="checkbox"/> Sistemas de banco de dados
<input type="checkbox"/> Desenvolvimento de aplicações Web	<input type="checkbox"/> Desenvolvimento de aplicações móveis
2. Você já trabalhou (ou está trabalhando) em empresa de desenvolvimento de sistemas? Se sim, por quanto tempo?	
<input type="checkbox"/> Não, nunca trabalhei em empresa de desenvolvimento.	
<input type="checkbox"/> Trabalhei menos que 1 ano.	
<input type="checkbox"/> Trabalhei entre 1 e 3 anos.	
<input type="checkbox"/> Trabalhei mais de 3 anos.	
3. Como você classifica seu conhecimento em relação aos seguintes tópicos?	
I. Desenvolvimento Java	
<input type="checkbox"/> Nenhum <input type="checkbox"/> Pouco <input type="checkbox"/> Moderado <input type="checkbox"/> Experiente	
II. Desenvolvimento Android	
<input type="checkbox"/> Nenhum <input type="checkbox"/> Pouco <input type="checkbox"/> Moderado <input type="checkbox"/> Experiente	
4. Se você deseja receber uma compilação dos resultados preencha o campo abaixo com seu email	
Email: _____	

## B.2 Tutorial Task Form

### **TAREFA TUTORIAL: OBTENÇÃO DE LOCALIZAÇÃO**

**1. Anote aqui o horário de início:** \_\_\_\_ : \_\_\_\_

**Arquivo da Tarefa:** com.dcc052.more.aqui.app.InsertActivity.java (More Aqui App)

**Arquivo do Teste:** com.dcc052.more.aqui.app.test.TarefaTutorialTest (More Aqui App Test)

**O contexto:**

No momento em que o usuário realiza a inserção de um imóvel, o aplicativo coleta informações de localização no serviço de localização provido pelo aparelho, como a última localização registrada pelo provedor. Mais especificamente, este aplicativo necessita somente dos últimos valores de latitude e longitude registrados pelo provedor.

**2. Anote aqui o horário de término:** \_\_\_\_ : \_\_\_\_

**3. Os exemplos providos pela plataforma foram úteis ?**

Sim     Não     Em parte

**4. As recomendações de padrões providas pela plataforma foram úteis ?**

Sim     Não     Em parte

**5. Observações (Opcional):**

---

---

---

---

---

**TAREFA 01: INSERINDO UM IMÓVEL**

1. Anote aqui o horário de início: \_\_\_\_ : \_\_\_\_

**Arquivo da Tarefa:** com.dcc052.more.aqui.app.InsertActivity.java (More Aqui App)

**Arquivo do Teste:** com.dcc052.more.aqui.app.test.TarefaUmTest (More Aqui App Test)

**O contexto:**

Após o usuário preencher as informações e clicar em "Pronto!", os dados são validados e inseridos. Neste processo é necessário que os dados sejam devidamente inseridos, evitando que os dados sejam corrompidos durante o processo. Nesse sentido, esta tarefa tem por objetivo garantir que os dados (já validados) sejam devidamente inseridos.

2. Anote aqui o horário de término: \_\_\_\_ : \_\_\_\_

3. Os exemplos providos pela plataforma foram úteis ?

Sim     Não     Em parte

4. As recomendações de padrões providas pela plataforma foram úteis ?

Sim     Não     Em parte

5. Observações (Opcional):

---

---

---

---

---

**TAREFA 02: EXIBINDO MENSAGENS AO USUÁRIO**

1. Anote aqui o horário de início: \_\_\_\_ : \_\_\_\_

**Arquivo da Tarefa:** com.dcc052.more.aqui.app.InsertActivity.java (More Aqui App)

**Arquivo do Teste:** com.dcc052.more.aqui.app.test.TarefaDoisTest (More Aqui App Test)

**O contexto:**

Para diversas ações dos usuários é necessário apresentar uma resposta para o usuário. Assim, esta atividade engloba os códigos necessários para apresentação de uma determinada mensagem ao usuário.

2. Anote aqui o horário de término: \_\_\_\_ : \_\_\_\_

**3. Observações (Opcional):**

---

---

---

---

---