DOCUMENTAÇÃO DE APIs USANDO EXEMPLOS DE CÓDIGO

JOÃO EDUARDO MONTANDON DE ARAUJO FILHO

DOCUMENTAÇÃO DE APIs USANDO EXEMPLOS DE CÓDIGO

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

Orientador: Marco Túlio Valente

Belo Horizonte

Março de 2013

JOÃO EDUARDO MONTANDON DE ARAUJO FILHO

DOCUMENTING APPLICATION PROGRAMMING INTERFACES WITH SOURCE CODE EXAMPLES

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio Valente

Belo Horizonte
March 2013

Montandon de Araujo Filho, João Eduardo

Documenting Application Programming Interfaces with Source Code Examples / João Eduardo Montandon de Araujo Filho. — Belo Horizonte, 2013

xxv, 79 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais

Orientador: Marco Túlio Valente

1. Application Programming Interfaces (APIs). 2. Software Documentation. 3. Software Reuse. 4. Program Slicing. 5. Recommendation Systems.
1. Computação – Teses. 2. Engenharia de software – Teses. I. Orientador. II. Coorientador. III. Título.

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

FOLHA DE APROVAÇÃO

Documentação de APIs usando exemplos de código (Documenting application programming interfaces with source code examples)

JOÃO EDUARDO MONTANDON DE ARAUJO FILHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Eduardo MP Figueiredo

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Leonardo Gresta Paulino Murta
Instituto de Computação - UFF

Belo Horizonte, 14 de março de 2013.

À todos que, de alguma forma, me inspiram a ser mais do que sou.

Agradecimentos

Gostaria de agradecer a todos que, de alguma forma, me ajudaram a construir este caminho nos últimos dois anos.

Em especial, agradeço a Deus e suas variáveis, por me guiar sabiamente durante esses 25 anos.

Agradeço à minha família—em especial aos meus pais João Eduardo e Marilene, e ao meu irmão Pedro Henrique—pelo imenso apoio, dedicação e compreensão.

Agradeço ao meu mentor e orientador, prof. Marco Túlio Valente, pela oportunidade que certamente mudou minha vida, e pelos cinco anos de ensinamentos, conselhos e aprendizado.

Agradeço à minha grande amiga e companheira, Míriam Cristina, por ser meu porto seguro onde encontrei motivação e apoio incondicionais a todo instante.

Agradeço à minha nova família—em especial à Isabel Rainha e Gilberto Eustáquio—por me incentivarem desde o início a trilhar este caminho.

Agradeço aos meus amigos—em especial os membros do LLP—pelo prazer da companhia, convivência e respeito.

Agradeço à secretaria do DCC por todo o suporte, tanto financeiro quanto profissional.

Agradeço ao CNPq pelo apoio financeiro.

"That day, for no particular reason, I decided to go for a little run. So I ran to the end of the road. And when I got there, I thought maybe I'd run to the end of town [...]. For no particular reason I just kept on going. I ran clear to the ocean. And when I got there, I figured, since I'd gone this far, I might as well turn around, just keep on going [...]. When I got tired, I slept. When I got hungry, I ate. When I had to go...you know... I went."

(Forrest Gump)

Resumo

O desenvolvimento moderno de software depende cada vez mais do reuso de *Application Programming Interfaces* (APIs) para garantir produtividade e qualidade. No entanto, devido ao seu tamanho e complexidade, o aprendizado de novas APIs exige um esforço não trivial por parte dos desenvolvedores. Para facilitar esse processo, os criadores de APIs normalmente fornecem recursos para auxiliar os desenvolvedores, geralmente na forma de uma documentação Web. Contudo, as informações contidas nesse tipo de documentação geralmente são insuficientes para o domínio de uma API. Visando ajudar a preencher essa lacuna, propõe-se nesta dissertação de mestrado uma ferramenta, chamada APIMiner, para instrumentação automática de documentações de APIs com exemplos de código fonte. Esses exemplos são extraídos de um repositório privado de sistemas e sumarizados por meio de um algoritmo de *slicing* estático. Além disso, foi implementada uma versão da ferramenta para a API do sistema operacional Android, chamada Android APIMiner. Para avaliar essa implementação, um estudo de campo foi realizado no qual a plataforma foi disponibilizada publicamente para uso por quatro meses. Para esse estudo, a plataforma extraiu 79,732 exemplos de uso de um repositório com 103 aplicações de código aberto. Ainda, a plataforma foi visitada 20,038 vezes, gerando mais de 40,000 visualizações de páginas e forneceu mais de 2,100 exemplos para seus usuários. Além disso, um experimento controlado foi conduzido envolvendo 17 participantes e incluindo a realização de duas tarefas de manutenção em uma pequena aplicação Android. Com esse experimento, observou-se que os exemplos providos pelo APIMiner ajudam a concluir tarefas de programação que envolvem poucos elementos da API. Por outro lado, os exemplos providos atualmente se mostraram menos úteis para resolver tarefas mais complexas, como as que requerem protocolos de chamadas de métodos mais elaborados.

Palavras-chave: Interfaces de Programação de Aplicações (APIs), Documentação de Software, Reuso de Software, Slicing de Programas, Sistemas de Recomendação.

Abstract

Nowadays, software development increasingly relies on Application Programming Interfaces (APIs) to improve quality and to increase productivity. However, learning to use new APIs in many cases is a non-trivial task given their size and complexity. For this purpose, API creators usually provide resources to assist developers in the understanding process, often in the form of a web-based documentation. However, the content available in this kind of documentation is frequently insufficient for mastering a new API. As a result, most developers face serious difficulties when trying to use modern APIs. To help developers during API learning process, we propose in this master dissertation a platform—called APIMiner—that instruments the standard Java-based API documentation format with concrete examples of usage. The examples provided by APIMiner are extracted from a private source code repository—composed by real systems—and summarized using a static slicing algorithm. We also describe a particular instantiation of our platform for the Android Software Development Kit, called Android APIMiner. To evaluate the proposed solution, we first performed a large scale field study where the platform has been used by professional Android developers during four months. For this study, Android APIMiner extracted 79,732 source code examples from 103 open source Android applications. Moreover, the platform was visited 20,038 times (from 130 different countries), generating more than 40,000 page views, and provided more than 2,100 examples to the users. Furthermore, we have conducted a controlled experiment with 17 subjects, including the implementation of two maintenance tasks in a small Android application. We observed that the examples provided by APIMiner helped to solve specific programming tasks, which comprise few and connected API elements. On the other hand, the current examples provided by APIMiner are less useful to solve more complex tasks, that require the implementation of more complex programming protocols.

Palavras-chave: Application Programming Interfaces (APIs), Software Documentation, Software Reuse, Program Slicing, Recommendation Systems.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

The use of APIs is increasingly common in modern software development projects [Zhong et al., 2009]. APIs adoption brings numerous benefits, for instance: (a) APIs contribute to reduce development time, and consequently the cost of software products, as developers do not need to implement the services provided by the API; (b) APIs also contribute to reduce the costs of the maintenance phase, because the API's maintenance and evolution is outsourced to its creators; and (c) APIs promote reuse, since developers do not need to re-implement the services provided by an API.

According to Robillard et al. [2012] an API is "*the interface to a reusable software entity used by multiple clients [...] that can be distributed separately.*". This definition is illustrated in Figure 1.1. Because APIs specifications are defined at source code level, developers can access procedures, data structures, variables, and any other programming structure provided by their creators. As examples of successful APIs, we can mention the Java API[1], .NET Framework Class Library[2], `C++` Standard Template Library (STL)[3], and the Android API[4].

However, the use and application of current APIs generally require a nontrivial effort. Most APIs provide resources to assist developers in the understanding process, often available in the form of a web-based documentation. However, due to modern APIs size and complexity, this kind of documentation is, in many scenarios, insufficient for mastering a new, large, and complex API [Robillard, 2009; Robillard and DeLine,

[1]`http://docs.oracle.com/javase/6/docs/api`
[2]`http://msdn.microsoft.com/en-us/library/gg145045.aspx`
[3]`http://www.cplusplus.com/reference`
[4]`http://developer.android.com/reference`

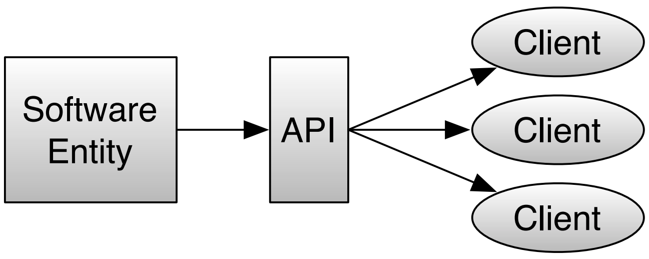


Figure 1.1: APIs definition

2011]. For instance, the Java API provided by the JDK 5 has more than 27,000 methods distributed in 32 packages [Kim et al., 2009]. As a result, most developers face serious difficulties when trying to use modern APIs. For this reason, questions on APIs usually represent the most common threads in programming forums. For example, at Stack Overflow there are more than 290,000 threads discussing questions related to the Android platform[5].

Robillard [2009] identified three major points where API's documentation can be improved: (a) the availability of a high-level description of the API architecture; (b) the availability of source code examples that can help to explain the API's functions; and (c) detailed information that could explain the API's unexpected behavior. Recent empirical studies also indicate that the poor quality of the documentation—which usually is limited to describe the signature of the methods—represent one of the major obstacles in the API learning process [McLellan et al., 1998; Robillard, 2009; Robillard and DeLine, 2011; Buse and Weimer, 2012]. In common, such studies share the finding that source code examples are a central instrument to facilitate and to make more productive the use of APIs.

1.2 Problem Description

Currently, most APIs documentation are based on the JavaDoc format (or an equivalent web-based documentation, in the case of other languages). As we argued previously, this model of documentation does not fulfill its role, since it lacks an important information to assist developers when learning a new API (i.e., source code examples). On the other hand, providing source code examples manually is complex and inefficient, because it requires a huge and specialized human effort [Mar et al., 2011].

To tackle this problem, some approaches have been proposed to provide source code examples automatically. Essentially, these approaches aim to assist developers by

[5]`http://stackoverflow.com/questions/tagged/android`

recommending source code examples that could fit their needs. Such approaches are referred in this master dissertation as API recommendation systems, and they can be organized in two distinct groups: IDE-based recommendation systems and JavaDoc-based recommendation systems.

IDE-based recommendation systems—such as Strathcona [Holmes and Murphy, 2005; Holmes et al., 2006], MAPO [Zhong et al., 2009], and API Explorer [Duala-Ekoko and Robillard, 2011]—are implemented as IDEs' extensions (i.e., plug-ins). The main advantage of these systems is their ability to explore the syntactic context provided by the IDE to recommend examples more relevant to developers. On the other hand, the examples provided by these systems are not documentation-focused, as they are highly dependent of a particular development context. Furthermore, IDE-based systems are restricted to the IDE they have been implemented to.

On the other hand, JavaDoc-based recommendation systems—such as APIExample [Wang et al., 2011], eXoaDocs [Kim et al., 2009, 2010], and PropER Doc [Mar et al., 2011]—are implemented independently from any IDE and usually can be accessed from the web. 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). On the other hand, they are not able to provide the same level of precision as IDE-based recommendation systems.

Due to their characteristics, JavaDoc-based systems can seamless replace a traditional documentation by offering a new documentation instrumented with source code examples. Despite this fact, both APIExample and PropER Doc have not been designed to replace the traditional JavaDoc, as they discard the original JavaDoc description information. As an example of recommendation system that extends a traditional JavaDoc document with source code examples, we can mention eXoaDocs. However, the examples provided by eXoaDocs are usually not from production-quality code (because they are freely extracted from web pages, without any external judgment on their quality). Moreover, the JavaDoc generated by eXoaDocs is static (i.e., the instrumented JavaDoc must be regenerated whenever a new source code example is processed).

In summary, to the best of our knowledge, there is no recommendation system that extends original JavaDocs while holding their original information and that at the same time provides small, relevant, and curated source code examples.

1.3 Goals and Contributions

The central objective of this master dissertation is to design, implement, and evaluate in the field an approach to instrument current web-based API documentations with source code examples extracted from real systems.

The proposed approach—called APIMiner—receives as input a list of methods provided by an API of interest and a repository of real client systems that use this API. Based on this input, the platform extracts source code examples from the systems and stores them in an internal database. This database then feeds the instrumented documentation—which is available to the users—with the source code examples previously extracted. Figure 1.2 illustrates the proposed approach.

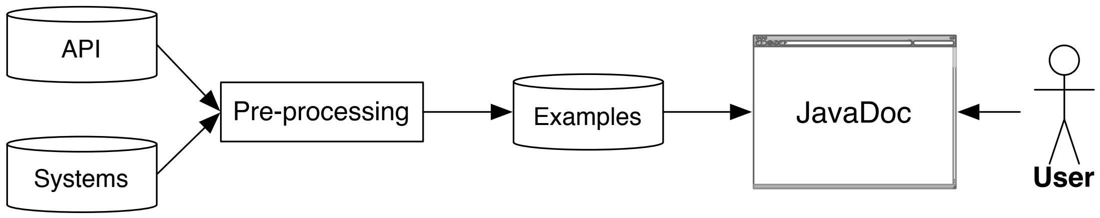


Figure 1.2: APIMiner overview

Basically, the solution proposed in this master dissertation has four main distinguishing characteristics:

1. The examples provided by APIMiner are automatically extracted from a repository of private systems. That is, the examples are not directly extracted from the Web, as happens with other solutions (like APIExample, eXoaDocs, and PropER-Doc). This characteristic allows us to control the quality of the examples. In addition, a private repository deals better with privacy issues. For instance, it opens the possibility to adopt the solution inside corporations and intranets.

2. In order to provide small but relevant source code examples, we have implemented an algorithm—based on static slicing—that summarizes the raw examples extracted from real systems. Basically, the goal is to select only the statements that have a structural dependency with the API elements that interests the user.

3. After the summarization step, the examples are ranked according to their relevance, using a combination of source code, process, and usage metrics.

4. The original API documentation is instrumented in order to include a link to the examples extracted and pre-processed by the platform. Therefore, it is important to highlight that the preprocessing phase—which is responsible for extracting the examples—can be executed independently from the querying phase.

To demonstrate and evaluate our approach, we have implemented a particular instance of the platform for the Android API, with almost 80,000 source code examples extracted from a repository of 103 open-source Android applications. Finally, we have performed two studies to evaluate this configuration of the platform for the Android API. More specifically, we have conducted a large-scale field study when the platform was used during four months by professional Android developers. We also conducted a controlled experiment involving 17 subjects that implemented maintenance tasks in a small Android application with the help of the source code examples provided by APIMiner.

1.4 Organization

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

- Chapter 2 presents the main works related to the central theme of this master dissertation. More specifically, we covered basic concepts related with recommendation systems and how they are being used in software engineering domains. Also, we explain in details how these systems can help developers in API comprehension tasks. We also present the major approaches proposed in the literature to tackle this problem.

- Chapter 3 presents the solution proposed in this dissertation. Basically, we describe the methodology and the decisions adopted in APIMiner's design. We also explain the summarization algorithm, which is a central component of our approach.

- Chapter 4 presents the evaluation performed on a particular instance of APIMiner for the Android API, called Android APIMiner. To evaluate Android APIMiner, we conducted a field study and a controlled experiment.

- Chapter 5 concludes this dissertation, highlighting the contributions of our approach and the limitations we have identified during the work. Furthermore, we present directions for future research.

Chapter 2

Background

2.1 Introduction

The following text fragment, extracted from the 9th ACM International Conference on Recommender Systems website (RecSys), and cited by Robillard et al. [2010], gives a general definition of a Recommendation System [ACM, 2009]:

> "Recommender 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. Recommender systems help overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance. Recommender technology is hence the central piece of the information seeking puzzle."

Basically, a recommendation system aims to help the user when he is dealing with a large volume of data, suggesting the information that best fit into the provided context. Similarly to people that needs to analyze a large amount of data, software developers meet the same difficulties when trying to perform every day program development tasks, such as understanding the behavior of a class from a large API [Robillard et al., 2010].

For this reason, various recommendation systems are emerging in order to assist developers in software engineering activities, such as recommending conceptually related artifacts or in API usage examples. Such systems are usually referred as Recommendation Systems for Software Engineering (RSSEs) [Robillard et al., 2010].

In a few words, an RSSE can be viewed as a software application that provides information items estimated to be valuable for a software engineering task according to a determined context. By helping developers to find information they should consider when evaluating alternative decisions, RSSEs provide a wide range of alternatives that can be helpful to execute a software engineering task. For example, RSSEs can help developers to find the right code, can provide examples of API usage, can rank bug reports, etc.

More specifically, most RSSEs support developers while programming. For example, CodeBroker analyzes—using textual similarity techniques—comments in the source code to recommend possible class library elements that could help in the implementation of a given functionality [Ye and Fischer, 2005]. The eRose plug-in mines past changes from version control systems to discover functionally related files [Zimmermann et al., 2004]. The Suade plug-in assists developers during the task of exploring a codebase. Basically, developers specify a set of relevant fields and methods and Suade retrieves the source files related to the provided parameters [Nagappan et al., 2006].

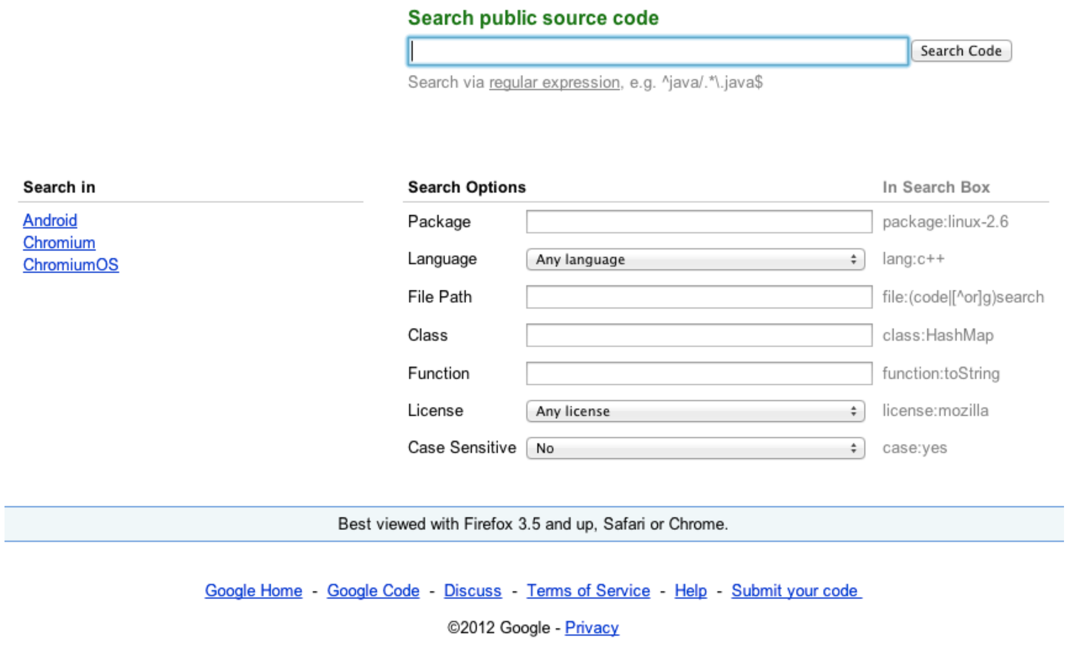
However, few RSSEs provide recommendations that are relevant in the process of API comprehension, for example by recommending relevant code snippets that make use of the desired API or by suggesting a sequence of API calls that perform a given programming task. The following sections focuses on this kind of recommendation systems.

2.2 Code Search Engines

Essentially, search engines index the content of a repository according to predefined keywords [Baeza-Yates and Ribeiro-Neto, 2008]. Later, the user provides the keywords of his interest and then the search engine returns the artifacts related to these keywords. Nowadays, Web Search Engines (WSEs) represent the most known implementation use of search engines concepts, as supported by platforms like Google, Yahoo, and Bing.

Following an architecture based on WSEs, some vertical search engines have been created to search specifically for source code publicly available in the Web. This kind of search engine is known as Code Search Engine (CSE). Differently from a Web Search Engine, where the repository is the whole Web, CSEs index only source code files available in open source repositories, like Google Code, GitHub, and SourceForge.

As examples of CSEs, we can mention Ohloh [Black Duck Software, 2004],


Search public source code

Search Code

Search via regular expression, e.g. ^java/.*\.java$

Search in

Android
Chromium
ChromiumOS

Search Options		In Search Box
Package		package:linux-2.6
Language	Any language	lang:c++
File Path		file:(code|[^or]g)search
Class		class:HashMap
Function		function:toString
License	Any license	license:mozilla
Case Sensitive	No	case:yes

Best viewed with Firefox 3.5 and up, Safari or Chrome.

Google Home - Google Code - Discuss - Terms of Service - Help - Submit your code

©2012 Google - Privacy

Figure 2.1: Google Code Search Interface

Krugle [Aragon Consulting Group, 2006], Codase [Codase, 2005], Jexamples [Jexamples, 2005], and Google Code Search [Google, 2006]. Basically, in such CSEs the user provides a set of keywords he is interested on, which typically represent a method or class name. Then, the CSE searches the repository for documents that contain the text provided by the user. Finally, the CSE returns a list of source files with the provided text.

Figure 2.1 presents the query interface of Google Code Search. As we can observe, the keywords are provided in textual format, and the user can customize the programming languages the query should be performed on. It is important to highlight that CSEs perform a textual search based on the provided keywords. In other words, CSEs do not consider the source code structure.

2.3 API Recommendation Systems

Some recommendation systems have been proposed to assist developers during API understanding tasks. In general, the goal behind these systems is to assist developers on how to use the API effectively and correctly. For example, techniques have been proposed to infer API usage patterns. In general terms, this kind of system is referred as API Recommendation Systems.

In this master dissertation, we decided to divide API Recommendation Systems into two distinct groups. The first group is referred as IDE-based Recommendation Systems, which contains systems developed as extensions of standard IDEs. The second group represents the JavaDoc-based Recommendation Systems, which contains systems implemented independently of any IDE, generally in a JavaDoc format.

The distinguished feature of the systems implemented as IDEs extensions is the possibility of using the information provided by IDEs during the recommendation process. With this information, such systems can provide a more precise recommendation based on the context that the developer is working on. However, this kind of tool is restricted to developers who are using IDEs.

On the other hand, JavaDoc-based Recommendation Systems have a wider reachability, because they are independent of IDEs and available in the Web. On the other hand, due to the lack of contextual information, this kind of system usually do not provide recommendations that are as specific to the user needs as IDE-based Recommendation Systems.

In the following sections we present in details both kinds of systems. Section 2.4 describes IDE-based Recommendation Systems and Section 2.5 discusses JavaDoc-based Recommendation Systems.

2.4 IDE-based Recommendation Systems

In this section, we will describe recommendation systems implemented as components of Integrated Development Environments (IDEs). In general, such systems can take advantage of the information available through the IDE to provide accurate and specific recommendations API users. However, the users are restricted to the IDE supported by the recommendation system.

This section presents three tools that rely on information provided by IDEs to give recommendations related to API usage.

2.4.1 Strathcona

Holmes et al. [2006] have proposed an approach that relies on the structure of the source code to find relevant examples in a repository [Holmes and Murphy, 2005]. Basically, the approach first extracts a set of structural facts about the context of a source code fragment. Then, the proposed approach relies on this structural context to search in a pre-processed repository for code examples with similar structure. Finally, the best results from the search are returned to developer for analysis.

The authors have built a tool that implements the proposed approach, called Strathcona, which has two main modules: Client and Server. The Client module, a plug-in for the Eclipse IDE, extracts the structural context from the Java source code fragments the developer would like to get examples of use. To create a structural context, the tool takes into account the following characteristics, called structural facts: (1) the method signatures called within the fragment; (2) the types that declare each of those methods, called declaring types; (3) the types of the fields declared by the declaring types; (4) the full qualified names of the types, methods, and fields referenced by the fragment, and (5) the supertypes of the declaring types. After the extraction process, the Client module sends the collected structural facts to the server for processing. Finally, the Client receives from the Server the list of similar examples.

The Server module has a repository of sample source code and a database of structural facts that reference this repository. It is important to notice that before Strathcona can be used, the repository and the database must be populated with source code using the API of interest. For this purpose, this code must be provided as an input to a preprocessing application. Basically, this application traverses the AST of the provided code, extracts the structural context of the method declarations that contains a relationship with the API of interest (like in the Client side), and stores the context in the database. After the execution of this process, Strathcona is ready for use.

When the user performs a query to the Strathcona server, the server attempts to find structurally similar code in the repository. For this purpose, Strathcona relies on a set of heuristics to find relevant examples. Each heuristic is executed independently from the server and returns, as a result, a list of relevant examples based on its criteria. Then, the lists are merged into a single list and sorted by their occurrence number. After this sorting process, Strathcona selects the ten most relevant examples to retrieve more details, like source code file and structure. Finally, this list is returned to the Client module. The Strathcona Client plug-in presents the examples in the following views: simplified UML-like class diagram, a textual description, and the source code of the examples.

To evaluate the tool, the authors created two versions of the Strathcona repository. The first one was populated by projects using Eclipse 3.08M. The second one was created with projects based on Eclipse 3.2M5. According to the authors, the tool scales well, since the second repository has more than three million structural facts and the tool takes between 0.3 to 3 seconds to generate and return the first ten examples.

2.4.2 API Explorer

Duala-Ekoko and Robillard [2011] have proposed an approach that leverages structural relationships between API elements to reduce the effort in learning how to use an API. Basically, the approach aims to facilitate the discoverability in APIs by recommending method or types which, although not directly related, may be relevant in solving a particular programming task.

To investigate the proposed approach, the authors have built a recommendation system that provides recommendations based on the structural context that is being developed, called API Explorer. As one of its central characteristics, API Explorer relies on an API-based dependency graph to correlate API elements that are not directly related. More specifically, API Explorer extends the content assist by recommending relevant method call sequences that can be use to perform a programming task. Currently, API Explorer is implemented as an extension of the content assist feature of the Eclipse IDE.

API Explorer provides three options for discovering relevant methods. The first option helps the developer when he needs to execute a command, but does not know how to obtain the information required by this task. In this case, the system provides, through the content assistant, different ways to solve the problem. The second option can be used when the developer is trying to interact with two types that are not directly related. In this case, the system searches for a sequence of commands that makes the link between objects of such types. The third option handles the cases where a developer might search for a method prefix that does not exists. In this case, API Explorer combines structural analysis with synonym analysis to recommend methods with a similar name.

To support the mentioned recommendations, API Explorer relies on a specialized API-based dependency graph, called API Exploration Graph (XGraph), and on algorithms that use this graph to generate recommendations based on the structural context. Basically, the XGraph models the structural relationships between API elements, where the API elements are the nodes and their relations are an edge. API Explorer uses XGraphs to generate recommendations on how the API elements should be combined. By default, API Explorer maintains an XGraph of the entire Java Runtime Environment (JRE) and automatically links it to the XGraph of the API of interest.

In addition to the XGraph data structure, API Explorer uses three recommendation algorithms to trigger recommendations. The Object Construction Algorithm is responsible for discovering statements that create an object of a given API type. The

Relation Exploration Algorithm looks for possible combinations between two distinct types. Finally, the Code Generation Algorithm generates the code necessary to execute the desired operation.

The authors evaluated the approach through studies where eight participants replicated four programming tasks with several discoverability hurdles. Basically, this study compares how developers are effective with API Explorer supporting. The results show that API Explorer was frequently invoked by the participants, ranging from 5 to 15 times when they solved the tasks.

2.4.3 MAPO

Zhong et al. [2009] have proposed an API mining framework that mines API usage patterns from a large number of code snippets. Essentially, the proposed framework uses data mining techniques to identify sequences of API methods that are frequently called together—known as mined API usage patterns—and recommends them associated with their code snippet representation to developers.

The authors have developed a tool called MAPO (Mining API usage Pattern from Open source repositories) that implements the proposed framework. For this implementation, it was necessary to develop techniques to extract the code snippets, to identify the patterns and to recommend the results to software developers. As in other approaches, the architecture behind the MAPO follows a pipeline structure with three main modules that are executed in the following order: (1) code analyzer; (2) API usage miner; and (3) API usage recommender.

The code analyzer extracts the API usage information from code snippets that call API methods and organizes the information according to the methods from which the information is collected. Before this analysis, the code snippets are retrieved from a repository of open source systems. In the next step, this module parses each code snippet and extracts the API usage information. Finally, based on the collected information, the module links each code snippet with the API call sequence and sends it to the next module. The proposed source code analyzer has been implemented as an extension of the Eclipse's JDT compiler.

The API usage miner module groups the API call sequences collected by the previous module into clusters to reduce the interference between different API usage scenarios. Moreover, the module mines API usage patterns from each cluster separately. In the clustering process, MAPO calculates a similarity rate based on the average of three metrics: (a) class name; (b) method name; and (c) the intersection of called API methods. After the clustering process, a sequential pattern mining algorithm, proposed by Agrawal and Srikant [1995], is applied into each cluster and the API call sequences greater than a determined support threshold are classified by MAPO as frequent API method call sequences. Finally, the proposed call sequences and their associated code snippets are sent to the recommender for displaying.

The authors have conducted an experimental study, where they applied MAPO to 20 open source projects that use the Eclipse Graphical Editing Framework (GEF). As a result, they were able to gather 93 patterns. Specifically, the patterns include a total of 157 API method call sequences, covering 856 API methods.

2.5 JavaDoc-based Recommendation Systems

This section presents JavaDoc-based Recommendation Systems, i.e., systems that take advantage of the availability that is provided by the web. Once the system is available at the web, anyone with a device that has access to the Internet can navigate through the provided documentation. In general, as JavaDoc-based Recommendation Systems have previous knowledge of all possible queries, they usually can be based on a pre-processed database of examples, making them more scalable. On the other hand, the contextual information available through IDEs cannot be used anymore.

In this section, we will discuss three JavaDoc-based recommendation systems.

2.5.1 eXoaDocs

Kim et al. [2010] argue that most API documentation do not include enough information for developers to fully understand possible API usages [Kim et al., 2009]. They also argue that searching for good code examples usually implies in a non-trivial effort. In an attempt to address this problem, they have proposed an automatic technique that extracts suitable code examples from code repositories, and generates example-oriented API documents, with the extracted code examples. The resulting documentation obtained is called *eXoaDocs*.

Initially, a code search engine (Section 2.2) is used to search for source code files that contain references to any resource of the given API. After that, the technique summarizes such references into snippets and then extracts characteristics from each snippet for clustering and ranking. Finally, the clustered and ranked snippets are embedded in the original API documentation.

Regarding the technique's implementation, it has four modules: summarization, representation, diversification, and ranking. Each module is explained in details next:

- **Summarization:** This module searches and collects potential code examples for each API element. To achieve this, the process leverages the Koders [Black Duck Software, 2008] code search engine by querying it using the given API name, and collecting the first 200 code files for each API element. Next, these files are submitted to the summarization algorithm that transforms the collected code into a concise snippet.

  The summarization algorithm has two steps. First, the algorithm identifies and extracts the methods with at least one use of the API element. Then, the algorithm selects the semantically relevant lines related to this API element based on a program slicing algorithm. More specifically, a 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) initialize the class of the given API element; or (R4) calls the given API element.

- **Representation:** The representation module extracts information that will be used for clustering and ranking the examples. More specifically, this module extracts the element vectors from the snippet's AST using a clone detection algorithm and, computes the similarity between the extracted vectors.

- **Diversification:** In the diversification module, the code examples are clustered based on their extracted vectors. To perform the clustering process, the *k-means* algorithm was applied varying the number of clusters and then choosing the result with the best quality according to a predefined set of quality metrics.

- **Ranking:** This last module is responsible for ranking the clustered examples, using two different ranking procedures. First, the summarized code snippets are ranked in each cluster to select the most representative example from each one based on measures such as representativeness, conciseness, and correctness. Next, the selected examples are sorted based on the number of occurrences.

To evaluate the tool, the authors have generated a version of eXoaDocs for the Java API. As a result, the approach has found examples for 20,480 methods, which represents 74% of the considered API.

2.5.2 APIExample

With a different focus than the eXoaDocs platform, Wang et al. [2011] take advantage of existing information in the web to provide examples of API usage. The supporting tool, called APIExample, identifies and extracts usage examples directly from web pages. In addition to the extraction, the proposed tool retrieves possible descriptive texts related to the extracted examples.

Basically, given a Java API, the tool collects related web pages from the web, extracts the Java code snippets with descriptive texts that explain the respective snippets, and assembles them into usage examples. Furthermore, the tool clusters and ranks the collected examples and provides some information related to the API use. APIExample provides two interfaces to end-users: a plug-in for the Eclipse IDE, and a web site[1].

The approach behind APIExample consists of four main modules, respecting a pipeline architecture: (1) the web page collection module, (2) the usage example extraction, (3) the example clustering and ranking modules, and (4) the statistics analysis module.

The Web Page Collection module is responsible for gathering web pages related to the API of interest. APIExample collects pages by leveraging Google through a query of *API_FQN example java*, where API_FQN represents the full qualified name of the given API. The first 300 web pages obtained from the search are downloaded and linked to the given API.

After the downloading process, the web pages are passed to the second module: The Usage Example Extraction. Basically, in this module, the examples and associate descriptive text are extracted from the pages. To perform this process, the page content is firstly divided into segments based on specific HTML tags. Next, the segments are submitted to a classification algorithm to determine whether they represent a code snippet. Basically, the algorithm first relies on a set of textual heuristics to verify if the segment "looks like" a code snippet. As central elements of the proposed heuristics, the authors consider the presence of braces and parentheses, the presence of lines ending with semicolon and the presence of reserved keywords, like `public`, `protected`, and `private`. Then the tool verifies whether the snippet is syntactically correct by executing a parser on it. If a problem occurs during this process, the parser conducts some adjustments in an attempt to fix it. Also during the parsing phase, structural information on the snippet is recorded for use in the following stages.

Once an snippet is found, the Usage Example Extractor module relies on another algorithm to identify descriptive texts explaining the selected snippet. For this purpose, the algorithm considers text fragments located immediately before or after the snippet.

After the extraction step, the approach applies a clustering and ranking process

[1]http://www.apiexample.com

over the collected examples. To calculate the similarity between examples, the authors use the sequence of API method calls exposed by each example. That is, examples that invoke the same methods of the target API are clustered in the same category. Next, the examples are ranked based on two criteria: (a) inter-cluster ranking, where the clusters are ranked based on the number of examples in each cluster; and (b) intra-cluster ranking, where the examples are ranked based on a set of metrics, like the number of API methods that are invoked and the number of descriptive texts attached to the example.

Finally, the tool conducts an statistical analysis over the data produced during the aforementioned stages to collect information related to the usage of the API methods, like the distribution of different API's usages and the frequently co-used APIs. The calculated data is stored in a database as well as the extracted usage examples for retrieving.

2.5.3 PropER-Doc

Mar et al. [2011] have proposed a methodology that recommends proper code examples for documentation purpose. The methodology—named as Proper Example Recommendation for Documentation (PropER-Doc)—aims to recommend code examples that demonstrate a whole object interaction scenario during the use of an specific API type.

Basically, given an API type of interest, PropER-Doc collects code example candidates from code search engines (CSEs) and recommends candidates meeting the developer requirements. The candidates are evaluated based on their relationship with other API elements (by means of API element links), where candidates in a higher syntactical level are selected first. Finally, the examples are ranked according to three recommendation metrics: significance, density, and cohesiveness.

Internally, the PropER-Doc approach can be divided into three main stages. In the first stage, PropER-Doc constructs a set of API element links that models implicit API element usage dependences. Basically, a link between two API elements is represented by a tuple where the API elements included in this tuple are structurally or conceptually related. Structural links represent structural relations between API elements, such as relations due to inheritance or composition. On the other side, conceptual links represent relations found on the descriptive content of the API documentation. Once this database is populated, PropER-Doc is ready for use.

In the second stage, the developer provides the API type he is interesting on—known as *targetType*—to PropER-Doc. Once this *targetType* is accepted as a query, it is forwarded to a CSE. The CSE searches in the web and returns relevant source files according to the provided *targetType*. PropER-Doc then parses each file and extracts the method's implementations—known as example candidates—that contains at least one call to the desired *targetType*. After collecting candidates from the CSE, the API calls relevant to demonstrate *targetType* within each candidate are captured and annotated based on API element links. For each API call identified, its relevance to the *targetType* is evaluated. If this call is considered relevant, it is annotated with a level that indicates its importance.

Finally, in the third stage the candidates annotated as relevant are reorganized and presented to the developer. The candidates are divided into groups according to the API types they contain. Once the developer decides to select code examples from a group, the candidates are ranked based on three metrics: (a) significance, which evaluates the importance of all API calls used by a candidate; (b) density, which evaluates the portion of lines of code that are annotated; and (c) cohesiveness, which evaluates the aggregation level of annotated API calls within a candidate. The candidate list is sorted according to the sum of the three metrics and the ordered list is returned to the developer.

PropER-Doc has been implemented to support recommendation of code examples for a framework implemented in Java. The Google Code search engine is used as the default CSE for performing the code search. The Eclipse JDT AST parser is used to construct the structural API element links and to parse the example candidates. Moreover, the Jericho HTML parser is used to parse Javadoc pages in order to extract conceptual API element links.

2.6 Other Systems

This section presents other systems that aim to assist developers in the process of understanding APIs. As in the previously discussed systems, several techniques are supported by these tools, such as structural analysis, data mining algorithms, and ranking heuristics.

Thummalapenta and Xie [2007] have developed the PARSEWeb tool, which address the problems faced by programmers in reusing existing frameworks or libraries. Basically, the approach accepts queries in the form `Source` $\rightarrow$ `Destination` as input and suggests method invocation sequences that reach the `Destination` type from the `Source` type. To extract this path between types, the approach relies on code samples extracted from CSEs.

Actually, PARSEWeb is based on the approach proposed by Mandelin et al. [2005]. In this work, the authors have implemented a tool, known as Prospector, which accepts queries in the form of a tuple (T_{in}, T_{out}) and suggests paths between T_{in} and T_{out}. However, differently from PARSEWeb, the approach behind Prospector relies only on the information contained in the API signatures.

Michail [2001] has developed the CodeWeb tool, which uses data mining techniques to detect patterns of use of APIs. Specifically, CodeWeb mines a repository for patterns of use of APIs, i.e., the goal is to infer—through association rules techniques—relationships between API calls. More specifically, if the relationship between a set of API calls is frequent, it is classified as a pattern of use.

The Eclipse Foundation [2011] is developing a set of IDE extensions to support developers learning new APIs, called Eclipse Code Recommender. More specifically, Eclipse Code Recommender provides two main modules: (a) Intelligent Code Completion Engines, which assist developers by recommending more relevant code snippets; and (b) Extended Documentation Providers, which aggregate information related with the API available in the web.

2.7 Critical Assessment

Table 2.1 compares the systems described in this chapter. We have classified the systems using six different attributes: input, interface, output, repository, clustering, and ranking. Moreover, we have classified the available tools into three major groups: (a) Code Search Engines (CSEs); (b) JavaDoc-based tools; and (c) IDE-based tools.

The `Input` attribute represents the input parameter required by the tool to perform the search for an example. Considering this property, IDE-based tools hold the most fine-grained parameter, which is the `Statement` that the user is directly interested on. As JavaDoc-based tools do not have this information, they generally rely on `Type` and `Method` names to perform the recommendation. CSEs have the simplest parameters for the recommendation, i.e., just a fragment of `Text`.

The `Interface` denotes the end-user interface proposed by the recommendation system. Generally, IDE-based tools are implemented as IDE's extensions, usually *plugins*. On the other hand, JavaDoc-based tools can provide platform-independent interfaces with the user through the `Web`.

The `Examples` property defines the level of granularity used for extraction and recommendation of examples. In this case, the level of granularity is related directly with the strategy adopted when extracting the source code examples. The lowest

Category	**System**	**Input**	**Interface**	**Output**
CSEs	CSEs	Text	Web	File
IDE Tools	Strathcona	Statement	Plugin	Method
	API Explorer	Statement	Plugin	—
	MAPO	Statement	Plugin	Method
JavaDoc Tools	eXoaDocs	Method	Web	Slicing
	APIExample	Type	Web/Plugin	Text
	PropER-Doc	Type	Desktop	Method

Category	**System**	**Repository**	**Clustering**	**Ranking**
CSEs	CSEs	Web	—	✓
IDE Tools	Strathcona	Private	—	✓
	API Explorer	Private	—	—
	MAPO	Private	✓	✓
JavaDoc Tools	eXoaDocs	Web	✓	✓
	APIExample	Web	✓	✓
	PropER-Doc	Web	✓	✓

Table 2.1: API Recommendation systems

granularity level can be obtained by an *slicing* process, which selects code snippets that contain just the statements related to the API element of interest. The coarse-grained granularity is represented by a `File`, which returns the whole source code file that contains the desired API Element. Moreover, other solutions provide an intermediate granularity level, such as a `Method` or a `Text`.

The Repository attribute indicates the dataset from where the tools extracts the recommendations. Basically, the repositories can be divided into two major groups: `Private` and `Web`. Tools based on `Web` repositories extract examples from the Web directly or through a CSE. On the other hand, tools based on `Private` repositories have first to populate their repository with systems that use the desired API.

Finally, the `Clustering` and `Ranking` properties show whether the systems use an strategy of clustering and ranking for recommending examples. As we can observe, it is common to implement a ranking algorithm over the extracted examples. On the other hand, it is more rare to have tools that rely on clustering algorithms.

2.8 Program Slicing

Generally, the time spent to understand how a software works is directly related with its size and complexity. To tackle this problem, software developers usually attempt to limit their analysis to the lines of code that implement a desired feature. However, this process is expensive and complex to developers, once they need to manually discover which lines are related to the feature of interest.

Weiser [1981, 1984] has proposed an approach that automatically decomposes programs into independent code fragments, known as program slicing. In a few words, a slicing algorithm receives two arguments: (a) the source code that will be sliced; and (b) a statement that represents the seed statement. Then the slicing algorithm processes each statement and selects those related to the seed statement.

For statement selection, a slicing algorithm analyzes the data and control flow dependencies in which the statement is inserted. If a statement belongs to the same control/data flow of any statement related to the seed, it is included in the slice produced by the algorithm. A data flow dependency represents the data relations that exists between the statements of the program. A control flow dependency represents a relation between two or more statements where the first statement decides whether the other should be executed or not.

Program slicing techniques can be divided into two major approaches: static and dynamic slicing [Harman and Hierons, 2001]. The static slicing uses the information of the source code structure to discover dependencies in compilation time. For this purpose, a static slicing algorithm performs the slicing process over the dependency graph that represents the program. Unlike a static slicing, dynamic slicing techniques use the information of a particular instance of the program execution to discover dependencies between statements at runtime.

Moreover, the technique can be performed in two ways: backward and forward. Basically, a backward slicing detects all the statements that can affect the seed statement. On the other hand, a forward slicing detects all the statements that could be affected by the seed statement.

Listings 2.1 and 2.2 illustrate an example of an slicing. Listing 2.1 represents the code that will be sliced. As seed we selected the line represented by the assignment in line 5. With this seed as input, the static slicing technique selects the lines 1 and 10, based on their data dependency, and selects the `for` loop (lines 4 and 7), based on its control dependency. As a result of the static slicing process, we obtain the code in Listing 2.2.

Examples of program slicing applications include debugging, testing, program

Listing 2.1: Original code

```
int sum = 0;
int prod = 0;

for(int i = 1; i < 20; i++) {
        sum = sum + i; //seed
        prod = prod + i;
}

System.out.println(prod);
System.out.println(sum);
```

Listing 2.2: Sliced code

```
int sum = 0;


for(int i = 1; i < 20; i++) {
        sum = sum + i; //seed

}


System.out.println(sum);
```

comprehension, and program parallelization. In this master dissertation, we will rely on a static slicing algorithm to select relevant statements that illustrate examples of usage of the methods provided by an API.

2.9 Final Remarks

This chapter presented background work related with API comprehension area. More specifically, we described solutions that aim to ease the API learning process. As we can observe, recommendation systems have great potential to assist developers in their every day programming tasks.

We decided to divide the solutions in two main groups: (a) IDE-based recommendation systems; and (b) JavaDoc-based recommendation systems. As an advantage, IDE-based systems can use the IDE context to provide accurate recommendations. On the other hand, this kind of system is tightly coupled to a particular IDE. Unlikely, JavaDoc-based systems have wider reachability, since they are available on the web.

Finally, we have classified the main recommendations systems available in the literature based on six criteria: Input, Interface, Output, Repository, Clustering, and Ranking.

Chapter 3

Proposed Solution

3.1 Introduction

The central idea behind our solution is to provide additional information that extends traditional JavaDoc documentation. We claim that JavaDoc lacks information that are essential when developers are learning how to use APIs. In other words, traditional JavaDoc does not fulfill its purpose, which is to provide a reliable and centralized information base for understanding APIs.

Robillard [2009] has conducted a survey over 83 Microsoft developers to identify the problems that impact the API understanding process. Basically, the survey has identified three distinct issues that difficult the API learning process. First, a lack of API high-level design structures, which could help developers to decide what and when to use API resources. Second, a lack of source code examples that are usually indicated by developers as essential learning artifacts. Third, a lack of APIs' internal information that could explain possible unexpected behaviors.

Our approach extends traditional API documents by including real examples of use. More specifically, we implemented our approach as a JavaDoc-based system, called APIMiner. We decided to implement APIMiner for JavaDocs because this format represents the de facto standard for documenting APIs in Java. Moreover, we have structured APIMiner to provide an scalable solution, once the examples are first pre-processed and stored in an example's database. In other words, there is any kind of processing when developers are navigating through the extended JavaDoc provided by APIMiner, except straightforward database accesses.

Figure 3.1 presents the internal structure of the APIMiner platform. Regarding its architecture, APIMiner can be divided into two major phases: preprocessing and querying. The preprocessing phase is responsible for the computation that is necessary

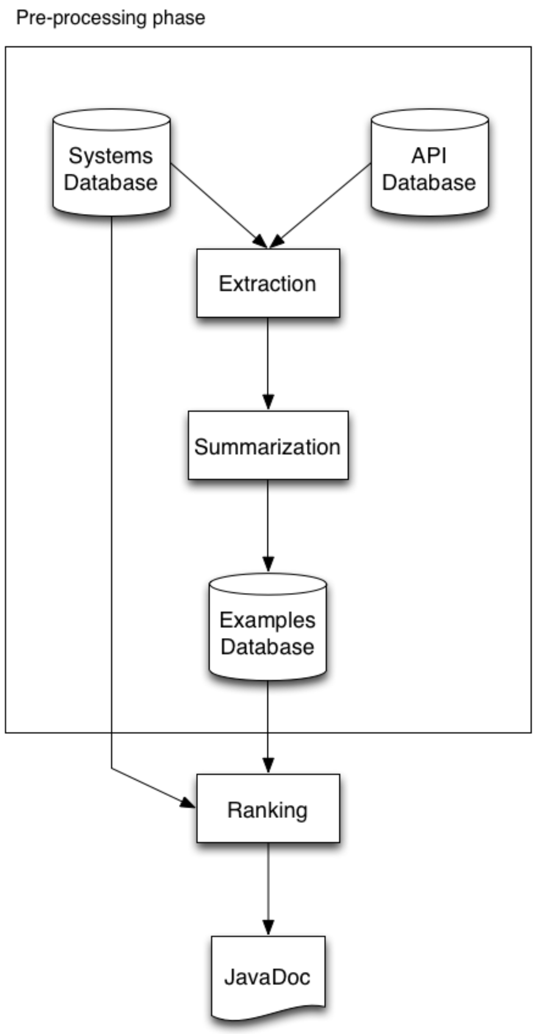


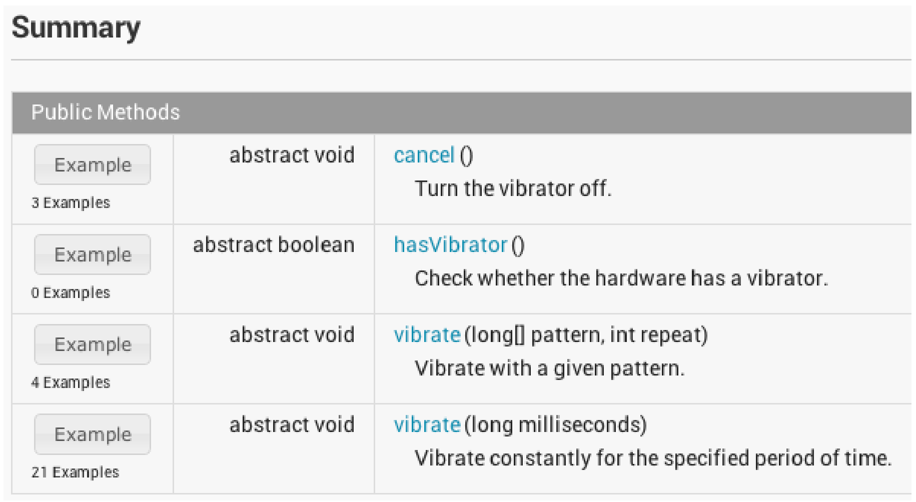
Figure 3.1: APIMiner architecture

for extracting the examples. This phase comprises steps ranging from the API/System Databases population to example extraction and summarization.

The preprocessing phase must be executed first, as it is responsible for extracting the examples. In this phase, the APIMiner administrator should populate the API Database, which contains a list with the signatures of the target API methods. Then, when a system is registered in the System Database, APIMiner parses its source code and extracts the examples based on the method signatures registered in the API Database.

The querying phase is responsible for the interaction between API users and the APIMiner system. In this phase, API users can navigate through the extended JavaDoc-based documentation generated by the APIMiner system. When end-users query for examples of a specific API method, APIMiner basically access the Examples

Database and returns a list of source code examples that invoke the informed method. Figure 3.2 illustrates how an original JavaDoc is instrumented, by showing the new documentation for the Vibrator class of the Android API[1].



Summary

Public Methods		
Example 3 Examples	abstract void	cancel () Turn the vibrator off.
Example 0 Examples	abstract boolean	hasVibrator () Check whether the hardware has a vibrator.
Example 4 Examples	abstract void	vibrate (long[] pattern, int repeat) Vibrate with a given pattern.
Example 21 Examples	abstract void	vibrate (long milliseconds) Vibrate constantly for the specified period of time.

Figure 3.2: JavaDoc for the Vibrator class produced by APIMiner

More specifically, APIMiner architecture has seven main components:

API Database: The API Database component stores the API Elements—i.e., API method signatures—provided by the API of interest.

Systems Database: The System Database stores the source code of systems that use the API of interest in their implementation. In other words, the examples provided by APIMiner come from the systems present in this repository.

Extraction: The Extraction Module parses each source code file available in the Systems Database and searches for API methods calls. When an API call is discovered, the module forwards the whole method declaration with the API call to the Summarization Module.

Summarization: This module receives the whole method declaration as input and filters the source code lines related with the selected call. For this purpose, we have implemented a static slicing algorithm that selects structurally related lines based on the variables used by the selected API method call.

[1]http://java.llp2.dcc.ufmg.br/apiminer/docs/reference/android/os/Vibrator.html

Examples Database: The Examples Database stores the examples that have been previously summarized. These examples represent code snippets that call methods of the target API.

Ranking: This module sorts the examples to provide the most interesting examples first. Currently, our ranking algorithm relies on four metrics: (a) number of lines of code of the example; (b) number of commits of the example's target source file; (c) number of downloads of the example's system; and (d) user feedback.

JavaDoc: This module represents the original JavaDoc instrumented by APIMiner to include source code examples. Users can navigate through this instrumented JavaDoc as usual and, when necessary, request examples for a given API method.

The following sections explain in details how these modules work internally. More specifically, Section 3.2 describes the API Database. Section 3.3 shows the internal structure of the Systems Database. Sections 3.4 and 3.5 describe how the example's Extraction and Summarization work. Finally, Section 3.7 and Section 3.8 document the main implementation decisions behind the Ranking and JavaDoc modules.

3.2 API Database

The API Database stores information regarding the API that will be instrumented by the APIMiner platform. That is, the proposed solution will provide examples for the methods registered in this database. With this approach, APIMiner administrators can easily customize the tool in order to define which API methods they are interested to provide examples.

To insert an API method in this database, the administrator needs to provide the method's signature and the full qualified name of the class that the method belongs to. Once these data are provided, the APIMiner platform creates a tuple (API_{class}, API_{method}) with the given information and stores it in the Database.

For example, if the administrator wants to register the method `substring(int beginIndex, int endIndex)` of the `String` class, he needs to inform the method signature, i.e., `substring(int,int)`, and the full qualified name of its class, i.e., `java.lang.String`.

3.3 Systems Database

The Systems Database is a repository that contains the source code of the systems that will be parsed to provide the examples proposed by our solution. In other words, the examples provided by the APIMiner platform are extracted from the source code of systems previously inserted in this repository.

The systems inserted in the Systems Database must attend two conditions. First, they must use the API of interest in their implementation. Second, they must be inserted through a specific protocol defined by APIMiner. Particularly, a system must be inserted in one of the following ways: (a) as a compressed file; or (b) as a system that is under a version control system (VCS). In both options, the user must provide a link to the system that will be inserted. For compressed files, the user must provide the file's download link. For systems under VCS, the user must provide the link that represents the main branch of the system.

Once the user informs a respective link, the APIMiner platform downloads the system into the Systems Database and builds it, if necessary. For systems under a VCS—more specifically SVN, in our current prototype implementation—the APIMiner platform collects historical information, like the number of modifications, on each source file and also stores this information into the database. If available, this information will be used as one of the criteria during the ranking process (Section 3.7).

3.4 Extraction

The Extraction Module represents the first computation step executed by the preprocessing phase. In a few words, this module is responsible locating and selecting code snippets that might represent an example of usage of a given API method. Basically, the module searches the System Database for API method calls and then sends the source code fragments with the APIs calls to the Summarization module, which is explained in details in Section 3.5.

To perform its task, the Extraction module first parses each source code file and analyzes each method call found during this process. For each method call, the module verifies whether it belongs to the API under analysis. We have implemented this verification by checking the method signature and the full qualified name of its class. When an API method call is found, the module marks the line that invokes the API method and sends it, along with the enclosing method declaration, to the Summarization Module.

In the implementation of this module, we have used the Eclipse Compiler for Java (ECJ), which is part of the Java Development Tools (JDT) of the Eclipse IDE. The ECJ compiler has a set of libraries for parsing, compiling, and analyzing Java source code files.

3.5 Summarization

The Summarization Module is a central component in the preprocessing phase. Basically, this module extracts source code lines representing an example of API usage and stores the extracted code in the Examples Database.

More specifically, this module receives as input two arguments: (a) a source code statement that contains an API method call; and (b) the whole method declaration containing the selected source code statement. First, this module selects the source code statements important for understanding the API method call. Then, the selected statements are extracted and grouped into a small source code fragment. This resulting fragment represents an example of usage of the selected API method.

For selecting the statements included in the example, we have implemented a slicing algorithm that extracts the statements structurally related with the API method call. Section 3.5.1 explains in details this algorithm.

3.5.1 Summarization Algorithm

Typically, good source code examples have some attributes that make them meaningful [Robillard and DeLine, 2011]. Basically, a good source code example must include its context, must have few lines of code, and must highlight the computation provided by the API. Further, a good example must be executed with minimal effort. In other words, a good example must be small, compact, readable, and expressive.

In order to obtain examples meeting the aforementioned conditions, we rely on a summarization process that extracts the source code lines structurally related with a given API method call. For this purpose, we have proposed a static slicing algorithm that performs the source code summarization.

By applying a slicing algorithm over a source code fragment, we obtain many of the attributes that features a good example. Because slicing algorithms extract statements that are structurally related, they ignore source code lines not connected with the context under extraction. For this reason, as result we have source code examples that are smaller and more readable. Moreover, the ignored lines do not impact in the execution of the example.

```
SUMMARIZATION($seed$, $body$)
$summarizedStmts = \emptyset$
$selectedStmts = seed$
while $selectedStmts \neq \emptyset$
    do
        $currentStmt$ = Pop($selectedStmts$)
        if $currentStmt \notin summarizedStmts$
            then
                $readableVars$ = GetReadableVariables($currentStmt$)
                $previousStmts$ = GetPreviousStmts($currentStmt$, $body$)
                $selectedStmts = selectedStmts \cup$
                BACKWARDSLICING($readableVars$, $previousStmts$)
                if $currentStmt = seed$
                    then
                        $writableVars$ = GetWritableVariables($currentStmt$)
                        $nextStmts$ = GetNextStmts($currentStmt$, $body$)
                        $selectedStmts = selectedStmts \cup$
                        FORWARDSLICING($writableVars$, $nextStmts$)
                if $currentStmt$ is a child of a control dependence statement
                    then $selectedStmts = selectedStmts \cup$ GetParentStatement($currentStmt$)
                $summarizedStmts = summarizedStmts \cup currentStmt$
return $summarizedStmts$
```

```
BACWARDSLICING($vars$, $statements$)
$result = \emptyset$
for $stmt \in statements$
    do
        $stmtVars$ = GetWritableVars($stmt$)
        if $vars \cap stmtVars \neq 0$
            then $result = result \cup stmt$
return $result$
```

```
FORWARDSLICING($vars$, $statements$)
$result = \emptyset$
for $stmt \in statements$
    do
        $stmtVars$ = GetReadableVars($stmt$)
        if $vars \cap stmtVars \neq 0$
            then $result = result \cup stmt$
return $result$
```

Listing 3.1: Summarization algorithm

Listing 3.1 presents the pseudo-code that describes our summarization algorithm. Basically, the algorithm is composed by three main functions: (a) `Summarization`; (b) `BackwardSlicing`; and (c) `ForwardSlicing`. In fact, the algorithm has other functions not showed in this pseudo-code because they have an auxiliary role. In a few words, `GetReadableVars` and `GetWritableVars` return the variables that a given statement reads or writes to. In a similar way, `GetPreviousStmts` and `GetNextStmts` functions return statements located before and after a given statement. Finally, `GetParentStatement` retrieves the statement that is the lexical parent of a given statement (considering that the statements are represented by an Abstract Syntax Tree (AST), as usual).

The `Summarization` method represents the entry-point of the summarization algorithm. The method receives as input two arguments: (a) `seed`, which is the statement with the API method call; and (b) `body`, representing all existing statements in the method declaration where the seed was found.

The method starts by declaring two local lists: `summarizedStmts`, which stores statements that have been processed and were considered relevant to the example (line 1); and `selectedStmts`, which stores statements that are relevant but that have not been processed by the algorithm (line 2). The algorithm then iterates over `selectedStmts` and executes the slicing process for each statement (lines 3-18). Then, the `summarizedStmts` list is returned with the relevant statements (line 19).

Inside the loop, the algorithm gets a statement from `selectedStmts` and verifies whether it has been processed (lines 5 and 6). If it has not, the slicing algorithm is executed based on the retrieved statement, which is stored in `currentStmt` (lines 8-18). Basically, first the algorithm retrieves the readable variables from `currentStmt` and the statements located before it (lines 8 and 9). Then, we call the `BackwardSlicing` function with `readableVars` and `previousStmts` as input, and after that we store the result in `selectedStmts` for subsequent iterations (line 10). If `currentStmt` and `seed` are the same, the algorithm retrieves the writable variables and the statements located after `currentStmt`. Then, we call the `ForwardSlicing` function with `writableVars` and `nextStmts` as input, and after that we store the result in `selectedStmts` for subsequent iterations (lines 11-15). Next, the algorithm verifies whether `currentStmt` is nested in a control dependence block and, if true, retrieves this block and inserts it in `selectedStmts` for further processing (lines 16 and 17). Finally, the statement that has been processed is stored in `summarizedStmts` (line 18).

Both `BackwardSlicing` and `ForwardSlicing` work in a similar way. They receive as input a list of variables, used to determine whether a statement is relevant, and a list of statements to analyze. Then, both methods iterate over the statements list (lines

2-6), extract the variables of each statement (line 4), and verify whether there is an intersection between the extracted variables and the variables received as parameter (line 5). If so, the statement is inserted in a list returned by the methods (lines 6 and 7).

The difference between both methods relies on the variables extracted from each statement. `BackwardSlicing` extracts only the writable variables of each statement, while `ForwardSlicing` extracts only the readable variables. This combination between the called slicing method and its respective parameters ensure that the summarization algorithm works as expected. That is, because `BackwardSlicing` receives `readableVars` and `previousStmts` as parameters, the method actually verifies whether any previous statement writes to a variable that the slice reads. In a similar way, `ForwardSlicing` receives `writableVars` and `nextStmts` as parameters and verifies whether any further statement reads a variable that is changed by the statements in the slice.

Although not showed for the sake of readability, our algorithm includes a last step where we remove from the returned slice any statement with an empty block (e.g., `for(...){}` or `if(...){}`).

Example: To illustrate the proposed summarization algorithm, we will rely on the method declaration presented in Listing 3.2. Basically, this code receives the name of a client and presents a welcome message with his name at the console. Further, the fragment gets the current time and prints the corresponding hour.

```
 1  public void welcome() {
 2          String client = "Smith, John";
 3  
 4          Calendar cal = Calendar.getInstance();
 5          int hour = cal.get(Calendar.HOUR_OF_DAY);
 6  
 7          int ini = 0;
 8          int end = client.indexOf(",");
 9  
10          if (end >= 0) {
11                  String surname = client.substring(ini, end);
12                  System.out.println("Welcome back Mr." + surname);
13                  System.out.println("Now, it is: " + hour + " hour(s)");
14          }
15  }
```

Listing 3.2: Method that calls a given API method (in this case `substring`, line 11).

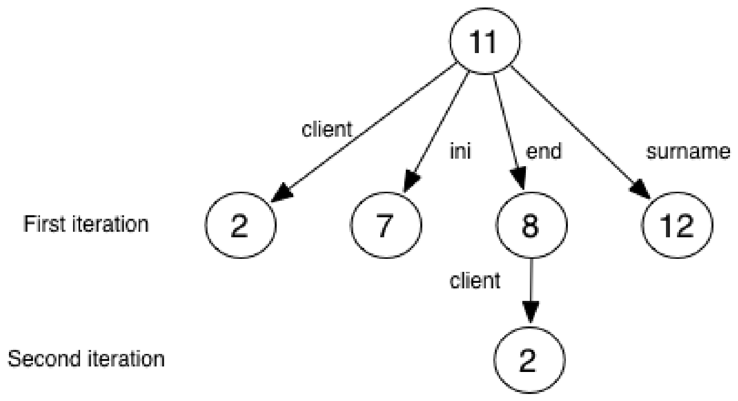


Figure 3.3: Dependency graph that illustrates the summarization algorithm

Suppose this method declaration has been retrieved by the Extraction Module, when searching for examples for the `java.lang.String.substring(int, int)` method. In other words, this method declaration was retrieved due to the existence of a call to `substring(int,int)` in line 11. Therefore, this line represents the seed of the summarization algorithm.

As we can observe, the seed call itself does not carry enough information to explain the use of the API method. First, it is important to include information on the variables used by the seed statement. More specifically, it is important to know the values of the variables `client`, `ini`, and `end`. Moreover, it would be interesting to know how the value returned by the function—in this example assigned to the `surname` variable—is used further. Finally, it is important to present the control dependences that impact the execution of the seed statement.

Considering the source code in Listing 3.2, we selected the call to `substring(int, int)` as our seed call (line 11). First, the algorithm extracts the readable variables (`ini`, `end`, and `client`), and searches for statements that write on them (lines 2, 7, and 8). Because we are iterating over the seed call, the algorithm also extracts writable variables (`surname`), and searches for statements that read these variables (line 12). Then we apply a new iteration of the algorithm for each statement previously selected.

Figure 3.3 illustrates this process using a dependency graph. Basically, each node in this graph represents a statement in the source code, and the node numbers denote their respective line. The edges represent data relationships characterized by the use of a given variable (reported as the edge's label). For instance, node 8 is related with node 2 through the use of the `client` variable.

Furthermore, as the seed statement is enclosed by an `if` statement (lines 10 - 14), we extracted the variables used by the `if`'s expression and apply the algorithm over

```
String client = "Smith, John";
int ini = 0;
int end = client.indexOf(",");

if (end >= 0) {
        String surname = client.substring(ini, end);
        System.out.println("Welcome back Mr." + surname);
}
```

Listing 3.3: Source code fragment returned by the summarization algorithm when applied to the method in Listing 3.2.

these variables. In this case, we consider only the `end` as a readable variable and select line 8, since it has an assignment to this variable.

Listing 3.3 shows the source code fragment generated by the proposed summarization algorithm for the method declaration in Listing 3.2. As we can observe, our algorithm extracted a source code fragment that represents an example of usage of the `java.lang.String.substring(int, int)` method. It is also important to mention that the algorithm ignored the lines responsible for printing the current time (line 13), because they are not related with the original seed statement.

3.6 Examples Database

The Examples Database represents the last component involved in the preprocessing phase. Basically, this database stores the examples extracted and summarized during the previous computation steps. Therefore, during the querying phase, the examples are not processed anymore. As a result, we have a large gain regarding the performance of the platform during the querying phase. That is, because the examples are recovered using a simple SQL query, our approach provides a scalable solution that fits the requirements of popular API documentations publicly available in the web.

When an example is generated by the summarization algorithm, the Examples Database stores four data: (a) the source code file from where the example was extracted; (b) the API method and class the example refers to; (c) the source code line used as seed by the summarization algorithm; and (d) the slicing representing the example.

When an end-user asks for examples for a particular method, the APIMiner platform queries this database for examples associated to this method and return them to the user. However, the Examples Database just contains the extracted examples, without any ranking. The ranking algorithm used by the APIMiner platform is presented in the next section (Section 3.7).

3.7 Ranking Algorithm

This module is responsible for ordering the examples of a given API method, when the user queries for them. For the sorting process, we have implemented a ranking criteria based on four metrics: (a) Lines of Code (LOC) of the example; (b) Number of commits of the example's source file; (c) Users feedback; and (d) Number of downloads of the system in a given software repository.

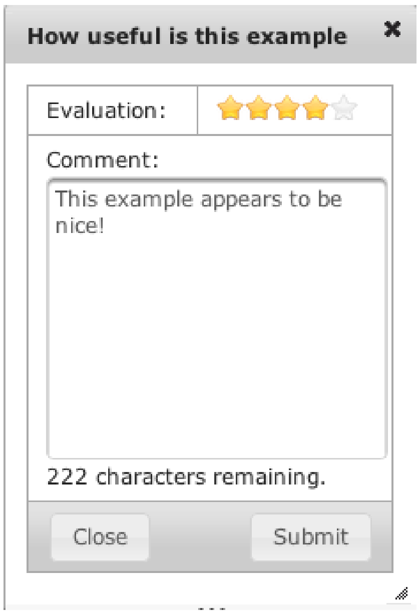
Lines of Code (LOC): The reasoning behind this metric is to give priority to concise and small examples.

Number of commits: This metric counts the number of commits of the source code file from where the example has been extracted. With this metric, we intend to give priority to examples originated from files that have been modified many times, because generally such files are very important in their system.

To calculate the value of this metric, we need historical information about the example's source code file. More specifically, we can apply this metric only to systems that have a public version control repository. From now, we have implemented this metric specifically for systems that have an SVN repository.

Number of downloads: This metric evaluates the popularity of the system from where the example was extracted. The intuition behind this metric is similar to that considered in the Number of Commits metric. A higher number of downloads indicates that the system is widely used and that it has an active community. With an active community around it, users frequently post bugs and request improvements. For this reason, these systems tend to come from well-known companies and organizations, with a well-defined software development process and skilled programmers. In summary, we believe that popular systems, in broad terms, tend to have high levels of external and internal software quality.

The number of downloads is available only for systems available in well-known software repositories (such as SourceForge, Google Play Store, etc.). Therefore, this metric needs to be manually retrieved from such repositories and inserted in our Systems Database.



This example appears to be nice!

222 characters remaining.

Figure 3.4: An example of user feedback

Users Feedback: This metric takes into consideration the feedback from the users of the APIMiner platform to rank the examples. Therefore, the value of the Users Feedback metric may vary overtime, since it is informed by the users of the platform. The dynamic nature of this metric allows that the community itself defines, along the time, which examples are more relevant.

When a user decides to provide a feedback for a specific example, he should fills a feedback form, which can be viewed in Figure 3.4. As we can observe, the feedback form has two fields: (a) Evaluation field, which is mandatory; and (b) Comment field, which is optional. The Evaluation field represents a rate the user can give to the example, ranging from one to five stars. The Comment field provides a comment that the user can associate to the provided rate.

Whenever an example is recovered, the APIMiner platform recovers the feedback rates related to it and calculates their average. This average represents the value of the Users Feedback metric.

To calculate the final ranking score of a given example, we first normalized the results returned by the mentioned metrics with a value that ranges from 0 to 10. Second, we apply a weight factor w to each metric value to give more relevance to specific metrics. With this weight factor, the APIMiner administrator can select which metric is more important in a particular instantiation of the APIMiner platform.

Third, we applied a simple weighted average over the values obtained from the metrics, as follows:

$$example_{rating} = \frac{(loc \times w_{loc}) + (comm \times w_{comm}) + (down \times w_{down}) + (feed \times w_{feed})}{(w_{loc} + w_{comm} + w_{down} + w_{feed})} \tag{3.1}$$

Examples with a high $example_{rating}$ value are presented first to the end-users.

3.8 JavaDoc Documentation

The JavaDoc component is the interface for communication between the API user and the APIMiner platform. Essentially, APIMiner's JavaDoc interface is similar to the original JavaDoc. The difference relies on the presence of Example buttons inserted by APIMiner in the original documentation to show the examples provided by the platform.

As observed in Figure 3.2, APIMiner instruments the original JavaDoc by adding a list of `Example` buttons next to the list of public methods of the target class. Also, there is a small label below each button, which indicates how many examples the platform provides for the associated method. Both components were automatically added using an HTML parser—called Jericho HTML parser[2].

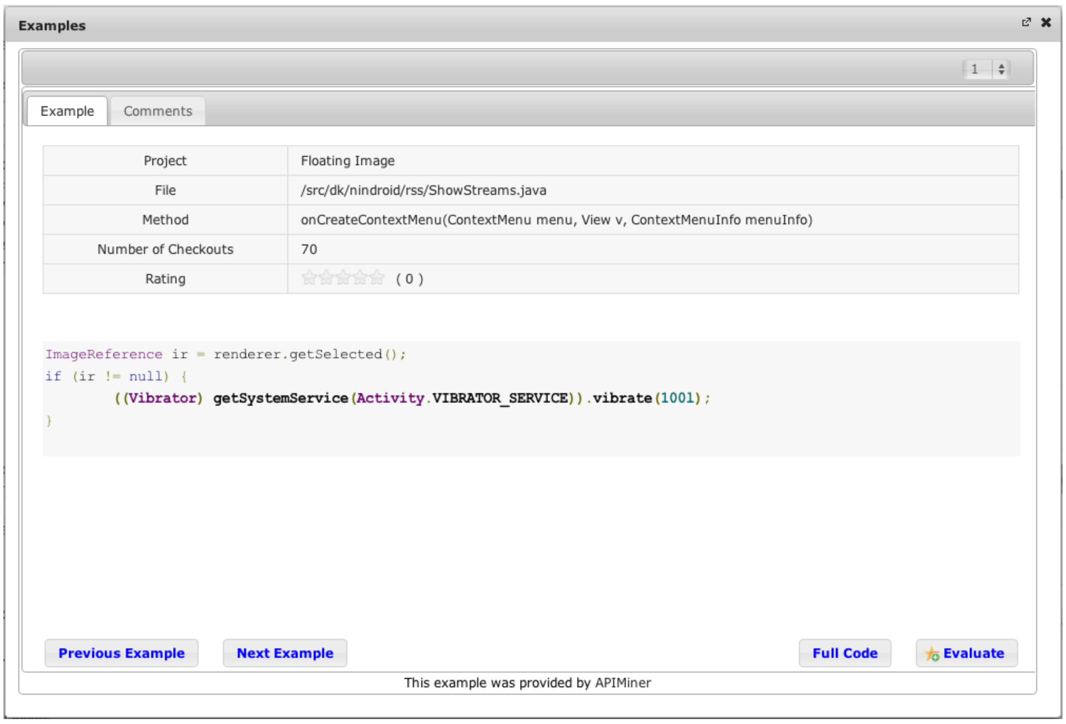
When the user clicks on an `Example` button, a pop-up window appears to show the examples provided by the platform. Figure 3.5 shows the example window after the user clicks on the `Example` button associated to `vibrate(long)` method.

As we can observe, the example pop-up window is divided in three regions: (a) header, which contains general information about the current example; (b) body, which shows the example source code; and (c) footer, which contains buttons for navigation in the list of examples.

The header region also has a second tab, called `Comments`, that shows in detail the ratings and comments associated to the current example. Also, in the right side of the window (next to the Close button), a dropdown list shows the ranking position of the current example. For instance, in Figure 3.5 we are showing the first example of the list.

The body region presents the examples extracted in the preprocessing phase. Additionally, we have used a syntax-coloring library to highlight the source code to ease its comprehension. The line in bold emphasizes the statement that contains the

[2]`http://jericho.htmlparser.net/docs/index.html`



Project	Floating Image
File	/src/dk/nindroid/rss/ShowStreams.java
Method	onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo)
Number of Checkouts	70
Rating	☆☆☆☆☆ (0)

```
ImageReference ir = renderer.getSelected();
if (ir != null) {
        ((Vibrator) getSystemService(Activity.VIBRATOR_SERVICE)).vibrate(100l);
}
```

This example was provided by APIMiner

Figure 3.5: Example popup window for the `vibrate(long)` method

API method call, which is also the statement used as seed by the Summarization process (Section 3.5).

The footer region contains four buttons. The left corner includes navigation buttons, which are used to browse the examples in the list. The right corner contains two buttons that provide additional functionalities. The `Full Code` button shows the complete source code file from where the example was extracted. The `Evaluate` button triggers another pop-up window, where the user can rate the current example (Figure 3.4).

3.9 Final Remarks

This chapter presented in details the solution proposed in this Master dissertation to instrument traditional JavaDocs with examples of usage. More specifically, we implemented a tool that extends traditional APIs documentation by providing real examples of usage for their public methods, called APIMiner.

As input, APIMiner receives a list of methods of a given API and a list of systems that use this API. As output, APIMiner provides a JavaDoc-based interface where users can request examples of usage for each API method. The design followed in APIMiner presents two distinguishing characteristics: (a) the tool was designed to achieve scalability, because the platform first pre-processes the given API by extracting and storing the examples in an internal database; and (b) APIMiner includes an implementation of a summarization algorithm—based on a static slicing algorithm—that extracts small and concise code snippets, that represent meaningful examples of usage.

Chapter 4

Android APIMiner

4.1 Overview

We have implemented a particular instance of the APIMiner platform for the Android API, called Android APIMiner. As argued by Syer et al. [2011], Android applications are widely dependent from services provided by the Android API. On average, 30% to 50% of the applications' source code rely on the Android API, i.e., contain at least a reference to an element (method, field, etc.) imported from the Android standard API.

Therefore, to implement Android applications, developers must know in details how the Android API works. In order to ease the learning process, Google provides a detailed JavaDoc that documents the API elements. However, due to JavaDoc complexity and lack of examples, the learning curve is still a problem for novice developers.

This context constitutes an interesting scenario for evaluating our APIMiner approach. More specifically, in this chapter we report two studies designed to evaluate an instance of APIMiner targeting the Android API. Basically, in the first study we have collected and analyzed usage data regarding an open version of APIMiner, such as number of access, number of page views, most queried examples, etc. The second study reports a controlled experiment performed with 17 subjects to evaluate Android APIMiner's effectiveness in comparison with the traditional Android documentation. More specifically, the subjects have implemented two Android-based maintenance tasks, one accessing the traditional JavaDoc, and another accessing the JavaDoc instrumented by APIMiner.

The remainder of this chapter is organized as follows. Section 4.2 introduces the Android API. Section 4.3 provides information about the Android APIMiner instance. Finally, Section 4.5 and Section 4.6 present two studies that have been performed to evaluate Android APIMiner.

4.2 Android API

Android is an open-source platform for mobile devices maintained by Google. The first version of the Android platform was launched in 2007, and currently the platform is on its 11th version. Nowadays, Android represents the most popular mobile platform, with around 700,000 registered applications on Google Play, and more than a million Android phones activations by day[1].

Internally, the Android platform relies on the Linux Kernel and on an specific virtual machine (VM), called Dalvik VM. While the Kernel manages the operating system, the Dalvik VM is responsible for executing the applications developed for the platform.

It is important to note that the Dalvik VM relies on a customized version of the Java language. Regarding the compilation cycle, when an application is built, its source code is firstly compiled into traditional Java bytecode files (i.e., `.class` files). However, before deploying, the bytecode files must be converted into Dalvik-compatible files (i.e., `.dex` files). Finally, the `.dex` files are compressed into a single `.apk` package. This package is the application binary file executed by the Android OS.

For developers who implement applications for Android devices, Google provides a set of libraries with several functionalities, which are available as part of the Android SDK. Basically, these libraries contain resources that implement typical mobile applications features, such as GUI, storage, security, and communication. Also, a set of testing and compiling tools is provided by the SDK.

Moreover, Google provides an Eclipse plug-in that facilitates the development process, called Android ADT. Through Android ADT, developers can access the Android SDK resources from the Eclipse IDE. For example, this plug-in automates many development steps, such as compiling, deploying, and testing.

Furthermore, the Android SDK provides an extensive API to access Android resources. In concrete numbers, the Android API has 1,814 classes, distributed in 84 packages. In terms of methods, the entire API includes 14,258 methods. These numbers were calculated for the version 4.1 of the Android API, which is the version used by Android APIMiner.

Table 4.1 shows the top 10 classes in number of methods. As we can observe, the top 10 classes have 1,566 methods, which represent around 11% of the total. Moreover, these classes are generally related to GUI functionalities. For example, the `View` class represents the basic element in a graphical interface component. The `TextView` class extends `View` by providing an specific text output component. The `GLES20`, `GLES10`

[1]According to `http://developer.android.com/about/index.html`

and `GLES11Ext` classes provide support to OpenGL. The `WebView` and `WebSettings` classes provide functionalities to display web pages. The `Activity` and `Intent` classes are responsible to manage the screens of an application. Finally, the `Parcel` class provides features for inter-process communication (IPC).

Class	**# Methods**
android.view.View	345
android.widget.TextView	202
android.opengl.GLES20	188
android.app.Activity	159
android.opengl.GLES10	123
android.content.Intent	122
android.opengl.GLES11Ext	120
android.webkit.WebView	109
android.webkit.WebSettings	99
android.os.Parcel	99
Total	**1,566**

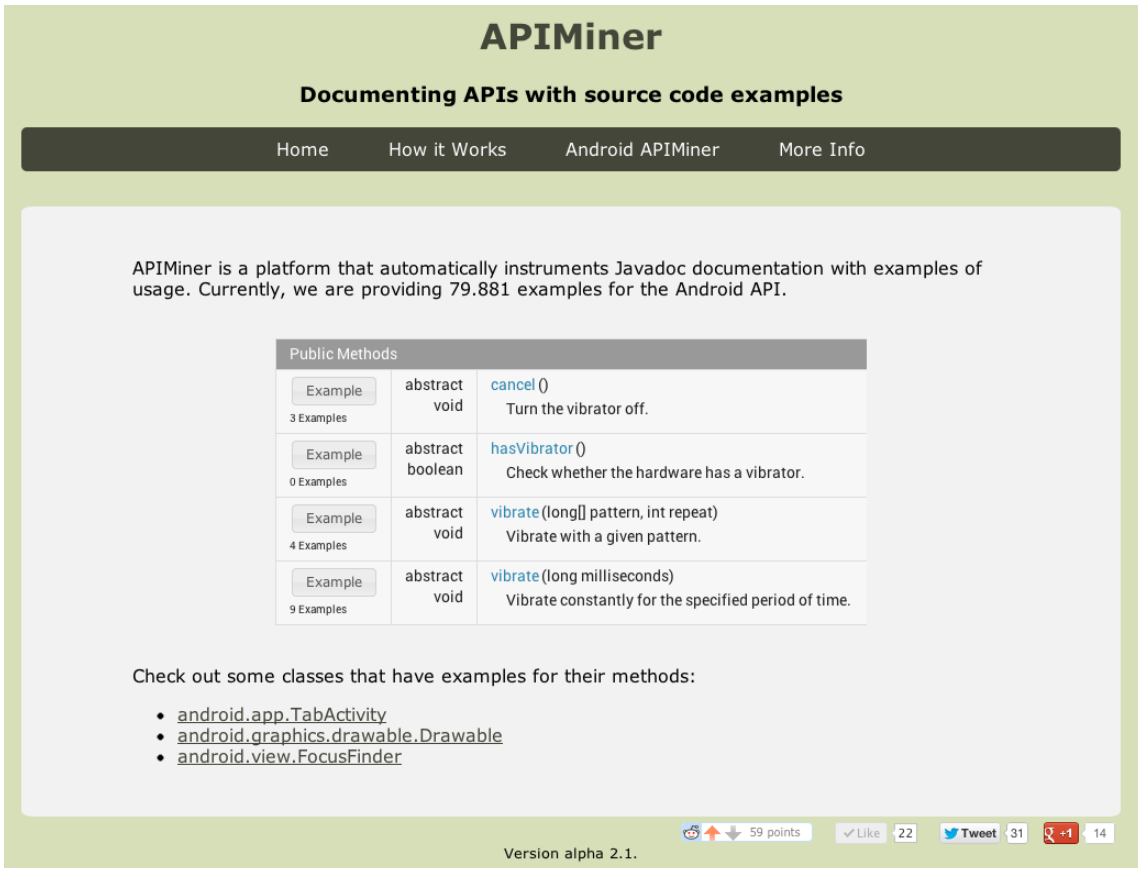
Table 4.1: Top 10 classes in number of methods

4.3 Android APIMiner

We have implemented a particular instance of the APIMiner platform for the Android API, called Android APIMiner. Currently, Android APIMiner provides almost 80,000 examples of use for Android API methods. The platform can be accessed publicly in the following URL: `http://apiminer.org`. Figure 4.1 shows the main page of this website.

More precisely, Android APIMiner provides 79,732 examples distributed in 2,494 methods (18%), and 349 classes (19%). These examples have been extracted from 103 popular open-source systems, such as Wordpress and ZXing Barcode Scanner. Appendix A shows a list with the systems used in the extraction process.

Regarding their size, 60,095 (75%) of the extracted examples have less than ten lines of code, and 7,964 (10%) have between eleven and fifteen lines of code. Listing 4.1 illustrates two source code examples extracted and summarized by Android APIMiner (respectively, for the methods `Vibrator.vibrate(int)` and `BluetoothAdapter.getDefaultAdapter()` methods). The lines in gray have been discarded by the summarization algorithm, while the lines in black represent the summarized example.


APIMiner

Documenting APIs with source code examples

APIMiner is a platform that automatically instruments Javadoc documentation with examples of usage. Currently, we are providing 79.881 examples for the Android API.

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

Check out some classes that have examples for their methods:

- android.app.TabActivity
- android.graphics.drawable.Drawable
- android.view.FocusFinder

Version alpha 2.1.

Figure 4.1: Main page of Android APIMiner

As a prerequisite, we selected systems that attend three conditions: (a) they are implemented under an open source license (such as GPL, Apache, etc.); (b) they have a public source code repository; and (c) they must build without errors. Part of the systems was selected from a public Android open source application list, available in the Wikipedia[2]. The remaining systems were selected from curated developer websites (such as `http://www.xda-developers.com`, `http://stackoverflow.com`, etc.) and specialized blogs (such as `http://sudarmuthu.com`).

Figure 4.2 shows the distribution of the source code examples extracted for the Android API. This distribution is represented as a treemap, whereas the size of the classes represents their number of methods. Also, the classes are colored according to the number of examples they provide: white classes have no examples while green classes have more examples. Additionally, a natural log scale is used to represent the number of examples in the visualization. The printed version of this treemap is restricted to the top-level packages from the Android API. An interactive visualization is available at: `http://apiminer.org/treemap-examples.html`. Using this visual-

[2]`http://en.wikipedia.org/wiki/List_of_open_source_Android_applications`.

```
 1 public boolean onLongClick(View view) {
 2     if (mIsSelecting) {
 3         return false;
 4     }
 5     Log.i(AnkiDroidApp.TAG, "onLongClick");
 6     Vibrator vibratorManager =
 7         (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);
 8     vibratorManager.vibrate(50);
 9     longClickHandler.postDelayed(startLongClickAction, 300);
10     return true;
11 }
```

```
 1 public void startDiscovery() {
 2     BluetoothAdapter bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
 3 
 4     if (bluetoothAdapter.isDiscovering()) {
 5         bluetoothAdapter.cancelDiscovery();
 6     }
 7 
 8     Set pairedDevices = bluetoothAdapter.getBondedDevices();
 9     for (BluetoothDevice device : pairedDevices) {
10         mListener.addBondedDevice(device.getName(), device.getAddress());
11     }
12 
13     final IntentFilter deviceFoundFilter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
14     mContext.registerReceiver(mReceiver, deviceFoundFilter);
15 
16     final IntentFilter discoveryFinishedFilter =
17         new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
18     mContext.registerReceiver(mReceiver, discoveryFinishedFilter);
19 
20     if (!bluetoothAdapter.isEnabled()) {
21         bluetoothAdapter.enable();
22     }
23 
24 bluetoothAdapter.startDiscovery();
25 }
```

Listing 4.1: Examples of source code examples for the Android API. The text in bold is included in the example; the text in gray was discarded. The seed of each example is in italic and bold (line 8 in the first example and line 2 in the second one).

ization, it is possible to navigate through the packages, using zoom-in and zoom-out features to visualize the treemap at different levels of granularity.

As we can observe, the treemap shows that the examples are highly concentrated. Popular packages—such as `android.app`, `android.view`, `android.content`, `android.os`, and `android.widget`—are well covered by examples, as indicated by the high presence of green blocks. On the other hand, we observe that some packages are poorly covered, even when they have a large number of classes. For instance, `android.test` has only 64 examples and `android.test.mock` has no examples. In comparison, the `android.app` package has 10,671 examples distributed in 1,010 methods. Similarly, despite the fact that `android.opengl` package has three classes among the first ten packages with more methods (Table 4.1), it has only 35 examples.

After investigating this fact deeper, we concluded that the packages with many examples—such as `android.app`, `android.content`, `android.view`—provide services that are frequently used by any mobile application. On the other side, packages with specific features—like `android.drm`, `android.test`, `android.renderscript`, etc.—have a lower number of examples. A similar result was obtained by Parnin et al. [2012], which confirms our observation. Basically, in this study the authors collected the posts related with Android in the Stack Overflow website and related them with elements in the Android API. As a result, the authors conclude that features that are commonly used by mobile applications are also largely mentioned on Stack Overflow's posts.

To verify this assumption, we analyzed in details the distribution of examples at three different levels: (a) package; (b) class; and (c) method. The results are summarized in Tables 4.2, 4.3, and 4.4.

Package	# **Examples**
android.content	15,446
android.view	11,664
android.app	10,671
android.widget	9,493
android.os	7,016
android.util	5,710
android.graphics	4,216
android.database	3,648
android.preference	3,038
android.content.res	1,998
Total	72,900

Table 4.2: Top 10 packages in number of examples

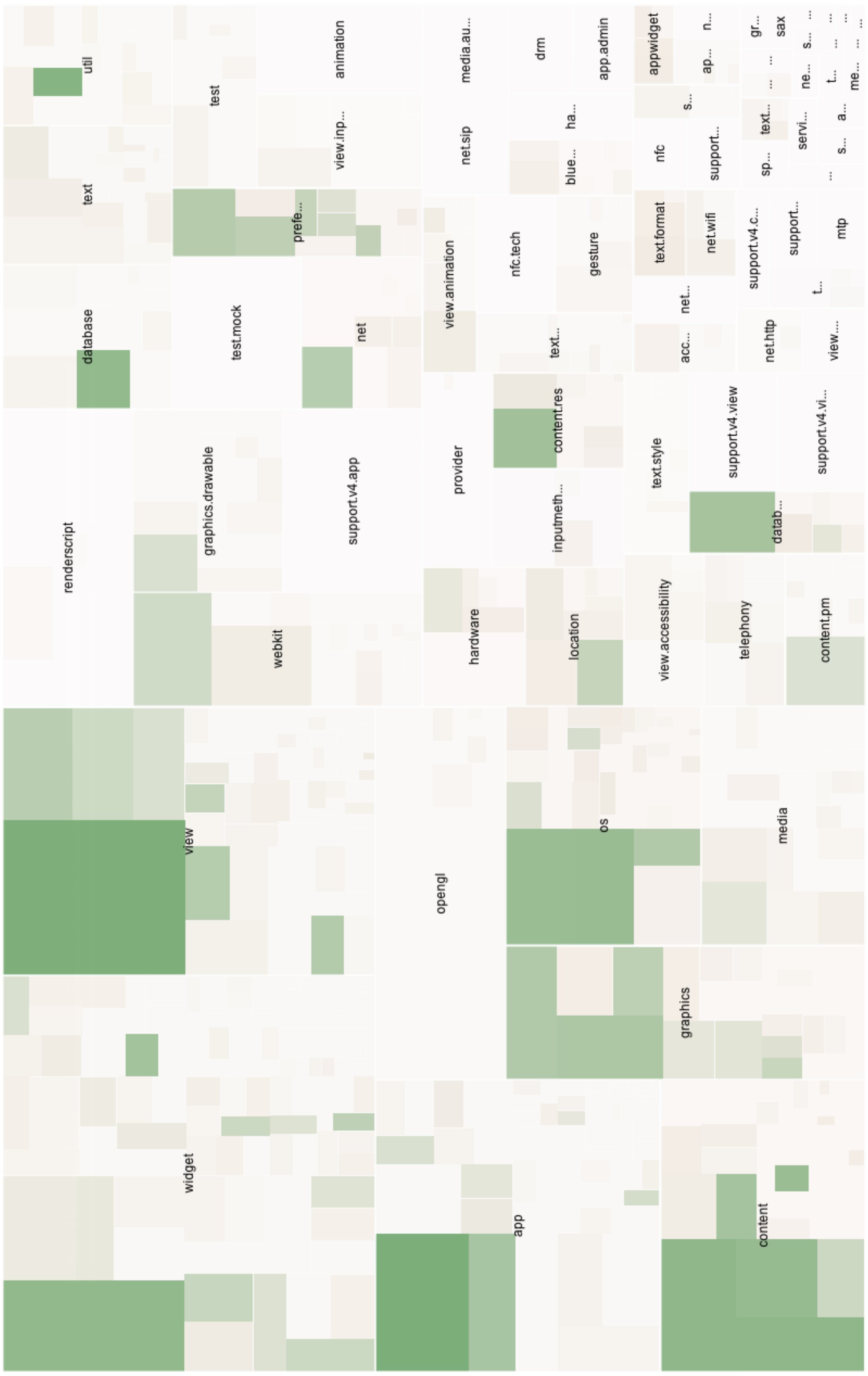


Figure 4.2: A treemap visualization showing the distribution of the extracted source code examples along the packages in the Android API

Table 4.2 shows the first 10 packages with more examples. As we can observe, the examples are highly concentrated, since the top 10 packages have 91% of the extracted examples (the remaining 9% are distributed in 74 packages). As expected, the packages in this list provide features commonly used when developing Android applications. For instance, `android.content` contains features related with content sharing and management. The packages `android.widget`, `android.view`, and `android.graphics` are responsible for features related with GUI concerns. Further, `android.database` provides services to manipulate databases. The `android.app` package contains high-level classes that abstract the typical Android application life-cycle. The `android.os` package provides basic services related to Android OS (like inter-process communication and power management). The `android.util` package contains common utility functions, for example, date manipulation, number conversions, etc. Finally, `android.content.res` contains methods for accessing application's resources, such as text, media, or other external files available as application resources.

Class	**# Examples**
android.app.Activity	7,883
android.view.View	7,171
android.util.Log	5,599
android.content.Intent	3,840
android.content.Context	3,729
android.database.Cursor	3,612
android.widget.TextView	3,578
android.content.ContextWrapper	2,817
android.content.SharedPreferences	2,569
android.os.Bundle	2,490
Total	43,288

Table 4.3: Top 10 classes in number of examples

Table 4.3 shows the top 10 classes in number of examples. As expected, the presented classes belong to the packages listed in Table 4.2. Also, the listed classes have a behavior similar to the one observed in Table 4.2: the top 10 classes concentrate 54% of the extracted examples. It is also important to mention that such classes provide widely used services, such as logging (`android.util.Log`), GUI (`android.view.View` and `android.widget.TextView`), and basic features (`android.app.Activity` and `android.content.Intent`).

The same behavior has been observed when we analyzed the methods with more examples, as presented in Table 4.4. From an universe of 14,258 methods, the top 10 methods are responsible for 19% of the source code examples. Similarly, the listed methods implement functionalities commonly used when developing Android applications. As observed, both `Activity.findViewById(int)` and `View.findViewById(int)` perform the same action: retrieve an interface component. The methods `Log.d(String, String)`, `Log.i(String, String)`, and `Log.e(String, String)` are used for logging purposes. The methods `TextView.setText(CharSequence)`, `View.setOnClickListener(OnClickListener)`, and `View.setVisibility(int)` define typical attributes in GUI components, such as their visibility and on-click events. Finally, `Context.getString(int)` and `ContextWrapper.getResources()` are used to retrieve application's resources.

Method	**# Examples**
android.app.Activity.findViewById(int)	2,900
android.content.Context.getString(int)	2,024
android.widget.TextView.setText(java.lang.CharSequence)	1,908
android.util.Log.d(java.lang.String,java.lang.String)	1,454
android.view.View.setOnClickListener(android.view.View.OnClickListener)	1,326
android.view.View.setVisibility(int)	1,279
android.view.View.findViewById(int)	1,250
android.util.Log.i(java.lang.String,java.lang.String)	1,172
android.content.ContextWrapper.getResources()	1,001
android.util.Log.e(java.lang.String,java.lang.String)	979
Total	**15,293**

Table 4.4: Top 10 methods in number of examples

4.4 Ranking Parameters

Table 4.5 presents the scores assigned for each range of values of the ranking metrics considered by APIMiner (i.e., LOC, downloads, commits, and users' feedbacks). For instance, examples between 4 and 8 lines of code have the highest score. On the other hand, examples with more than 10 lines of code have the lowest score. A similar approach was applied to the number of downloads (examples from systems with more than 5,000,000 downloads have the highest score) and for the number of commits (examples from files with more than 100 commits have the highest score).

The feedback of the examples was normalized based on a simple average of their values. Since such values range from 0 to 5, we calculated the sum of the feedback values for each example, multiplied the result by 2, and divided it by the number of feedbacks collected for each example. The resulting value represents the normalized score for this metric, which is a value that ranges from 0 to 10. The normalized scores

(a) Normalized score for the LOC metric

# LOC	Score
1	4
2 - 3	5
4 - 8	10
9 - 10	7
$\geq$ 10	0

(b) Normalized score for the Downloads metric

# Downloads	Score
$\leq$ 7,500	0
7,500 - 25,000	2
25,000 - 50,000	3
50,000 - 250,000	4
250,000 - 500,000	5
500,000 - 1,000,000	6
1,000,000 - 2,000,000	7
2,000,000 - 5,000,000	8
$\geq$ 5,000,000	10

(c) Normalized score for the Commits metric

# LOC	Score
$\leq$ 5	0
6 - 15	2
16 - 30	4
30 - 60	6
60 - 99	8
$\geq$ 100	10

Table 4.5: Normalized scores for the metrics values used by the ranking algorithm

obtained for each metric are then used in the final ranking algorithm, presented in Formula 3.1 (Chapter 3).

Regarding the ranking weights, we have defined the following weights: $w_{loc} = 4$, $w_{comm} = 2$, $w_{down} = 2$, and $w_{feed} = 2$. That is, we decided to give more priority to examples that fit our criteria for LOC values.

It is important to mention that the scores and weights used to rank the examples were given based on our experience and on general guidelines available in the literature [Robillard and DeLine, 2011]. Therefore, we acknowledge that more studies are needed to show the quality of our proposed ranking algorithm and constants.

4.5 Field Study

We have conducted a field study using the publicly available version of the Android APIMiner platform, which can be accessed at `http://apiminer.org`. Our goal was to identify how the community has used our platform. More specifically, we proposed the following questions to be answered:

1. **How many users accessed Android APIMiner?** How much time they spent in the platform? How many pages they visited?

2. **Which locations do the visits to Android APIMiner come from?** Which locations visited the site more frequently?

3. **How many examples Android APIMiner provided?** Does the number of examples provided by the platform increased over time? What were the most requested examples?

4. **Do developers search for source code examples?** Is there a demand for source code examples?

To answer these questions, we analyzed usage access data collected from September 14th to January 18th, in a total of four months. The data has been obtained from two distinct sources: (a) Google Analytics service[3], which collected the information related with APIMiner website access; and (b) a private logging service we have implemented in our platform. This service logs users events in the system, such as the examples requested by the users, the examples the users provided feedback to, etc.

Therefore, the data collection procedure used in our field study does not involve users directly. More specifically, the procedure is obliviousness to the users, since they are not aware of when and what data is collected. According to Lethbridge et al. [2005], this kind of procedure can be classified as a second degree data collection technique, which basically presumes an indirect involvement of the users of a field study during the data collection phase.

In the following sections we present answers for the aforementioned questions. Section 4.5.1 and Section 4.5.2 answer the questions related with the number of access and locations. Similarly, Section 4.5.3 and Section 4.5.4 answer our third and fourth questions, respectively.

[3]`http://www.google.com/analytics`

4.5.1 How Many Users Accessed Android APIMiner?

During the time frame considered in our field study, Android APIMiner received a total of 20,038 visits. As described in Table 4.6, 14,412 (72%) originated from organic search—i.e., search engine websites like Google, Bing, etc. Moreover, 3,393 (17%) of the visits originated from referral traffic, which means that the visitor has been redirected to Android APIMiner from another web site (i.e., from links in blogs, forums, etc). The remaining 2,233 (11%) visits come from direct access to our platform (i.e., visitors who typed the Android APIMiner URL directly in their browsers).

Traffic Origin	**# Visits**	**% Visits**
Organic search	14,412	72
Referral traffic	3,393	17
Direct access	2,233	11
Total	**20,038**	100

Table 4.6: Origin of APIMiner accesses

Figure 4.3 presents the number of visits to Android APIMiner by weeks. In a general way, we can observe that the number of visits increased consistently. Also, we observe two weeks with peaks of visits. In both cases, the peaks are due to two promotion posts on the Reddit website[4]. In the first case—in the second week of November—we posted a message about APIMiner in the Java development forum (`/r/java`), which has around 14,000 readers. As a result, we received more than 500 visits in this week.

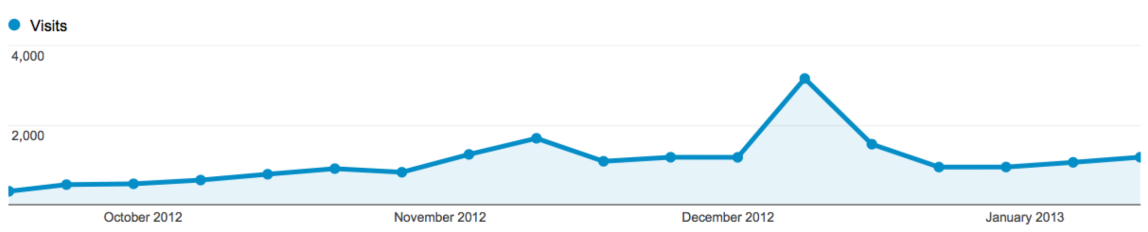


Figure 4.3: Number of visits per week

One month later, we posted a second message in the Reddit website[5], this time in the programming forum (`/r/programming`), which has around 415,000 readers. As

[4]`http://www.reddit.com`

[5]This message is available at: `http://www.reddit.com/r/programming/comments/14nsdo/apiminer_android_sdk_documentation_with_thousands`

a result, the number of visits originated from Reddit jumped from 5 to 1,300 in this week. In both cases, we received a positive feedback from the community, which provided interesting suggestions, most of them related to usability issues.

During our field study, Android APIMiner had 42,034 pageviews, resulting on an average of 2.10 pages/visit. The users remained on the site for an average of 1:28 minutes.

4.5.2 Which Locations do the Visits to Android APIMiner Come From?

Figure 4.4 depicts the number of visits by showing them in a world map. The level of the green color determines the number of visits from a given country, i.e., green countries visited the site many times, while white gray countries did not visit our site.

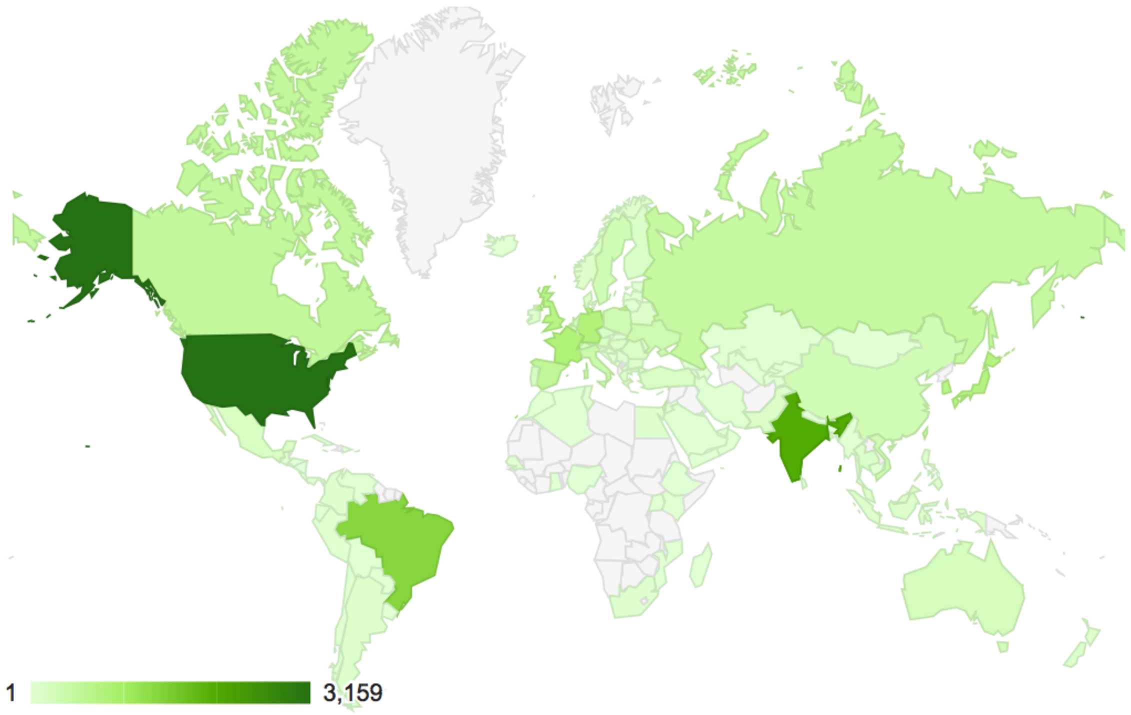


Figure 4.4: Number of access by country

Table 4.7 presents the top 10 countries in number of visits along with their Page/Visit ratio. These countries are responsible for 12,029 visits, which represent 60% of the total. As we can observe, Android APIMiner has been accessed from a large number of different locations. However, three countries concentrate the access: United States (3,162 visits), India (2,086 visits), and Brazil (1,743 visits). However, important countries also visited the site frequently, including France (855 visits), Germany (827 visits), Japan (777 visits), U.K. (749 visits), South Korea (719 visits), Spain (577 visits), and Canada (534 visits). In total, Android APIMiner has been accessed from 130 different countries (and from 3,087 different cities/provinces).

Country	**# Visits**	**Pages/Visit**
United States	3,162	2.50
India	2,086	1.62
Brazil	1,743	3.28
France	855	2.65
Germany	827	2.26
Japan	777	1.71
United Kingdom	749	2.07
South Korea	719	1.63
Spain	577	1.75
Canada	534	2.26

Table 4.7: Top 10 countries in visits

4.5.3 How Many Examples Android APIMiner Provided?

During the four months of our study, the users requested 3,910 examples from Android APIMiner. However, 1,753 requests (45%) have been made for methods where the platform has no examples. In other words, Android APIMiner has provided examples for 2,157 users requests (55%). Furthermore, the number of feedback scores we have received was not representative. During our field study, we just received feedback for nine examples.

Figure 4.5 shows the 2,157 example requests provided by Android APIMiner distributed by weeks. As expected, the distribution is similar to the one presented in Figure 4.3. Furthermore, the peaks in this figure are also due to the posts at Reddit. The highest number of examples was provided in the second week of December (due to our second post at Reddit). In this week, 722 examples have been provided by Android APIMiner.

We also generated another treemap to visualize the regions of the Android API with more example requests, as shown in Figure 4.6. As in the previous treemap (presented in Figure 4.2), the size of the classes are defined by their number of methods. However, in this second treemap the classes are colored according to the number of examples they have provided, i.e., green classes have provided more examples and white classes have provided fewer examples. A natural log scale is used to represent

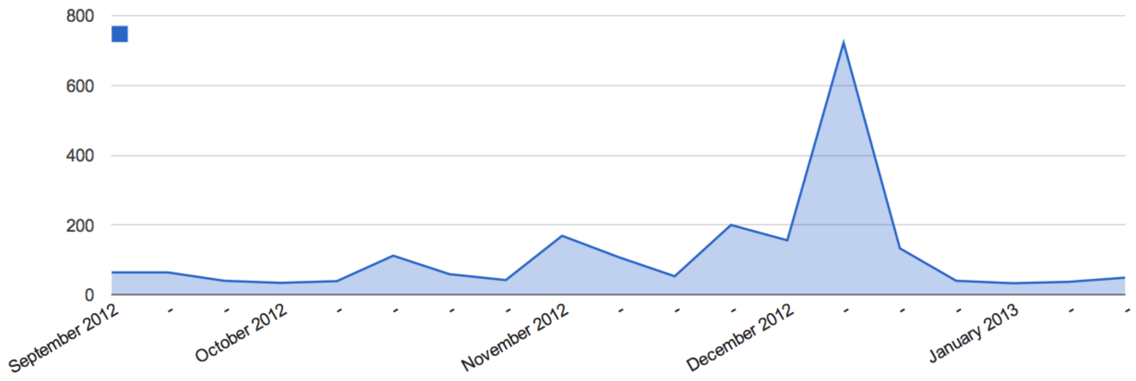


Figure 4.5: Number of examples provided per week

this number. The interactive visualization of this treemap is available at `http://apiminer.org/treemap-requests.html`

As we can observe, the distribution of the provided examples (Figure 4.6) does not follow the distribution of extracted examples (Figure 4.2). A restricted list of classes has been requested many times, while the remaining classes, even those with a large number of examples, have been requested few times. Stated otherwise, there are some packages with a large number of examples that have been requested many times (e.g., `android.app`, `android.database.sqlite`, `android.graphics`, and `android.widget`). However, there are also packages with a small number of examples that have been requested frequently, e.g., `android.location`, `android.hardware`, `android.bluetooth`, `android.appwidget`, and `android.accounts`. In common, all of these classes deal with Android peripherals and services.

This behavior remains the same when we analyze the provided examples at a fine-grained level. We analyzed the top 10 methods requested by the users, as presented in Table 4.8. As we can observe, none of the listed methods come from any of the top 10 classes with more examples presented in Table 4.3. Moreover, 30% of the examples have been requested for methods in the `android.widget.Toast` class, which is in 12th position in the number of examples. Also, we can observe that methods that access peripheral devices and services are often listed, like `Vibrator.vibrate(long)`, `Camera.autoFocus(Camera.AutoFocusCallback)`, `BluetoothAdapter.cancelDiscovery()`, and `BluetoothDevice.getName()`.

4.5.4 Do Developers Search for Source Code Examples?

In order to answer this question, we have analyzed the searching queries informed by the users when they reached Android APIMiner using a search engine. As mentioned

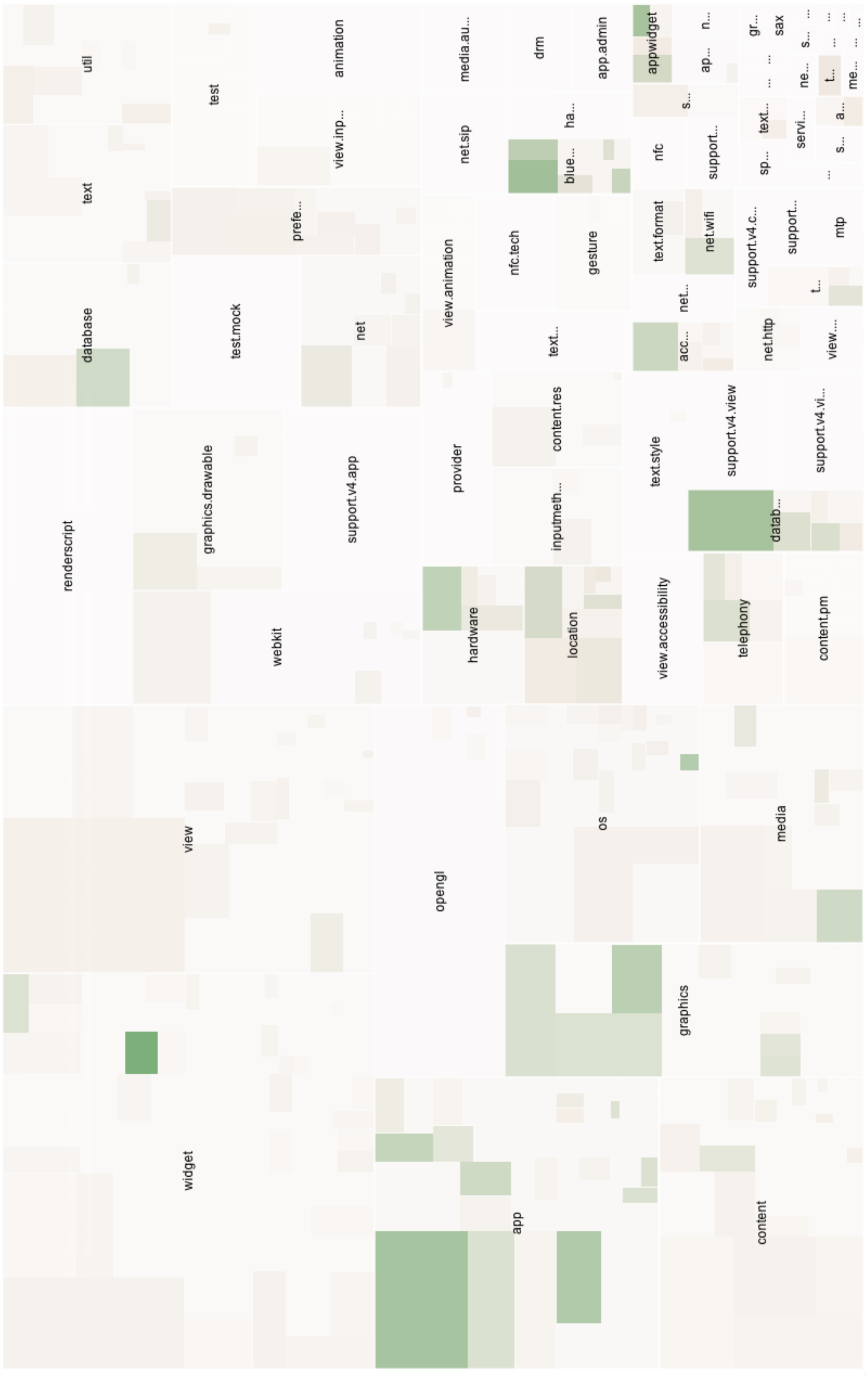


Figure 4.6: A treemap visualization showing the distribution of the examples requested by the users along the packages in the Android API

Method	**# Examples**	**# Requests**
Toast.cancel()	3	25
Toast.makeText(Context,int,int)	309	22
ActionBar.addOnMenuVisibilityListener (OnMenuVisibilityListener)	1	19
Toast.makeText(Context,CharSequence,int)	633	18
Vibrator.vibrate(long)	21	17
AppWidgetHost.allocateAppWidgetId()	6	16
Camera.autoFocus(Camera.AutoFocusCallback)	3	15
Bitmap.compress(CompressFormat,int,OutputStream)	46	15
BluetoothAdapter.cancelDiscovery()	5	14
BluetoothDevice.getName()	6	14

Table 4.8: Top 10 methods in terms of examples requested by the users

in Section 4.5.1, 14,412 visits originated from search engine queries. Due to privacy issues, Google does not provide search data from logged users[6]. For this reason, we had to discard 9,774 visits. The remaining 4,638 visitors have executed 3,660 different queries.

Keyword	**# Queries**
speechrecognizer wait timeout	53
apiminer	30
datepicker.keep_screen_on	16
eglquerysurface egl_width android resize	15
listpopupwindow **example** android	15
android.net.rtp **example**	14
gridlayoutanimationcontroller **example**	13
android notificationcompat **example**	12
notificationcompat.builder **example**	12
fragmentactivity **example**	11
Total	191

Table 4.9: Top 10 searching queries

Table 4.9 presents the top 10 most frequently used searching queries. As we can observe, the top 10 queries have been used 191 times (4%). Therefore, unlikely the results obtained for examples, the queries present a diversified behavior.

We also counted the number of queries containing the `example` keyword. As a result, 1,287 of the 3,660 available queries have the `example` keyword (35%). When analyzing the top 10 queries, the `example` keyword is present in six queries. In summary, the `example` keyword has been used frequently by the users, which reinforces our

[6]More details about this privacy issue is available at: `http://analytics.blogspot.co.uk/2011/10/making-search-more-secure-accessing.html`

initial claim that developers usually search for source code examples when accessing the documentation of an API.

4.6 Controlled Experiment

In order to better understand the benefits and the drawbacks of our platform, we conducted a controlled experiment where 17 subjects have used APIMiner. The objective of this experiment is to measure the gain provided by Android APIMiner in comparison with the traditional Android documentation. More specifically, we proposed this experiment in order to answer the following question:

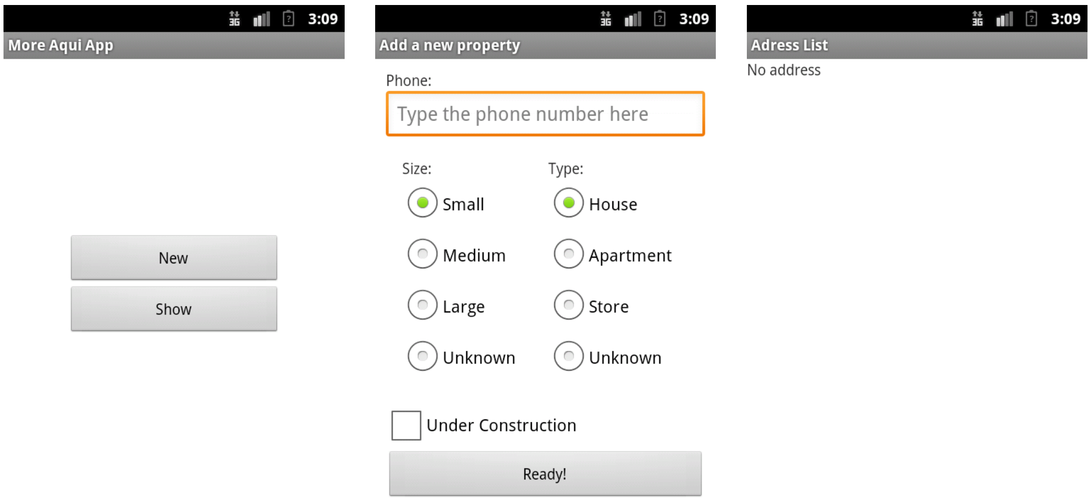
Do the examples provided by APIMiner help developers to implement maintenance tasks?

To shed light on this question we have created a simple Android application. However, we separated two features to be implemented by our subjects. As a restriction, the subjects have been asked to implement one task accessing only the documentation provided by Android APIMiner, and another task accessing the Android traditional JavaDoc. Basically, during the experiment we monitored the subjects and recorded the ones which successfully implemented the provided tasks.

This section is organized in the following way. Section 4.6.1 explains in details the application used in the experiment as well as the maintenance tasks assigned to the subjects. Section 4.6.2 shows how the experiment was configured and Section 4.6.3 describes how it was executed. Finally, Section 4.6.4 describes our findings after the experiment.

4.6.1 More Aqui

`More Aqui` is an Android application we have implemented to help users who are looking for properties on sale. Basically, the application works as an agenda that registers properties on sale which could interest the user. A typical scenario is as following: the user is walking in the city when he sees an ad for a property. The user opens the application and fills the information related to the property, such as the phone number and its size. The application then saves the information provided by the user and the property's location, which is collected automatically from the smartphone's GPS.



(a) Main screen (b) Registration screen (c) Listing screen

Figure 4.7: Screenshots from `More Aqui`

The application has three screens: (a) main screen, as presented in Figure 4.7(a); (b) registration screen, used to register a property of interest (as presented in Figure 4.7(b)); and (c) listing screen, which lists the properties that have been registered (as presented in Figure 4.7(c)).

We selected three maintenance tasks to be implemented by the subjects in the experiment. As prerequisites, we defined tasks that represent simple and objective programming tasks, and that require calling methods from the Android API with source code examples extracted by APIMiner. Moreover, we defined maintenance tasks with different levels of complexity. The selected maintenance tasks are described in the following:

- **Adding an OnClick event to the main screen's buttons:** This task aims to add OnClick events to the buttons `New` and `Show` in the main screen.
- **Screen's transition:** This task aims to implement the transition from the Main screen to the Registration and Listing screens.
- **Persisting a record in the database:** This task aims to implement the code that saves a property record in the database.

Regarding their complexity, the first and second tasks are considered easy tasks, because they involve few and simple interactions with the Android API. On the other hand, the third task is more difficult, because it involves a larger number of classes and methods. In fact, only the second and third tasks were used in the experiment. The first task was only used in a tutorial stage, i.e., used to explain the experiment to the subjects (as discussed in details in the next section).

4.6.2 Experiment Setup

We selected 17 subjects to conduct the experiment. The selected subjects had the following prerequisites: (a) good programming skills in the Java language; and (b) no proficiency in the Android platform. The selection was based on information provided by the subjects. We checked this information using a form distributed before the execution of the experiment. This form can be found at Appendix B.1.

The experiment was performed in a university laboratory, with 20 computers with the following configuration: Dell Optiplex 790 with Intel Core I3 3.30 GHz, 8 GB RAM, 1 TB of HD, and Windows 7 Professional 64 bits. We also configured the environment by installing an Eclipse IDE with the Android ADT and the Android SDK platforms. A pre-defined workspace was configured in each computer, with the Eclipse projects the subjects should work around.

We defined that the tasks should be executed in order, with a time limit of 20 minutes for each task. Additionally, we distributed a descriptive sheet with the information necessary to implement each task, including the variables, methods, and classes to use, and the link to the corrected documentation. It is important to mention that the subjects were instructed to access the documentation only using the link provided in the descriptive sheets. The sheets for the three tasks are included in the Appendix B.2.

Regarding the tasks implementation, we followed a crossover methodology. In this methodology, we divided the subjects in groups A and B. Group A implemented the first task accessing only the traditional Android documentation and the second task accessing the documentation provided by Android APIMiner. In contrast, group B implemented the first task accessing the Android APIMiner and the second task accessing the traditional Android documentation.

4.6.3 Experiment Execution

Due to subjects availability, we divided the experiment in two sessions. However, both sessions were executed in the same way. First, we gave a brief introduction on Android development, highlighting the main topics the subjects should know to implement the tasks. Among these topics, we showed how to use the development tools and discussed some basic concepts of Android development.

Also, we implemented the tutorial task—mentioned in Section 4.6.1—with the subjects. Basically, the following methodology was proposed to implement the tasks: (a) we initially showed how the application works before the task's implementation and how it should work after its implementation; (b) the subjects were instructed to read the task form to understand what methods should be used; (c) the subjects were instructed to access the provided documentation links to understand how the suggested methods work and how to call them; (d) the subjects were instructed to implement the task based on the information available in the documentation; and (e) they finally were instructed to test the application to check whether everything is correct.

After this introduction, we notified the subjects about the beginning of the experiment, which implies that no more questions were answered. For each task, we applied the aforementioned steps `(a)` and `(b)`. Also, we asked the subjects to fill the task's starting time in the task form. Then we start counting the time limit to implement the task. As the experiment is synchronized, we asked the subjects who finished the task before the time limit to fill the finish time field in the task form and to wait for new instructions. When the time limit was reached, we asked the remaining subjects to stop their implementation and to put a dash in the task's finish time. Then, we seated with each subject and checked their implementation. We also fixed the implementation in case of errors or missing code. In general, the sessions lasted for 1:40 hours approximately, including 30 minutes on the tutorial part and 50 minutes on the implementation part.

4.6.4 Experiment Results

From 17 subjects, only six implemented at least one task (35%). When investigating why few subjects were able to implement the proposed tasks, we figure out that many subjects have had difficulties in adapting to the programming model followed by mobile applications, which they considered very different from standard, non-mobile applications. Table 4.10 shows the number of subjects that implemented the tasks with Android APIMiner and with the traditional documentation.

Regarding the Screen Transition task, from five subjects who completed the task, four used Android APIMiner. Considering the Database Insertion task, from four subjects who completed the task, only one used APIMiner. After investigating this result, we discovered that the three subjects who completed the task without APIMiner are highly skilled Java developers, with a solid experience in frameworks and web

Task	**Android APIMiner**	**Traditional JavaDoc**
Screen Transition	4	1
Database Insertion	1	3
Total	**5**	**4**

Table 4.10: Number of subjects who implemented each task

development.

Furthermore, the Database Insertion task has proved itself to be more complex, because it requires calling various methods from distinct API classes. As illustrated in Listing 4.2, this task requires calling three methods in a well-defined sequence. First, developers need to call `SQLiteOpenHelper.getWritableDatabase`, to retrieve a writable database (line 1). Next, a sequence of `ContentValues.put` calls are used to fill the data that will be stored in the database (lines 4 - 9). Finally, the `SQLiteDatabase.insertOrThrow` method is used to persist the values previously stored (line 11). Therefore, examples for individual methods—such as those provided by APIMiner—did not help subjects who had not previous experience with Java-based persistence frameworks.

```
 1  SQLiteDatabase db = estatesDb.getWritableDatabase();
 2  ContentValues values = new ContentValues();
 3
 4  values.put(DBConstants.PHONE, estate.PHONE);
 5  values.put(DBConstants.SIZE, estate.SIZE);
 6  values.put(DBConstants.STATUS, estate.STATUS);
 7  values.put(DBConstants.TYPE, estate.TYPE);
 8  values.put(DBConstants.LAT, estate.LAT);
 9  values.put(DBConstants.LONG, estate.LONG);
10
11  db.insertOrThrow(DBConstants.TABLE_NAME, null, values);
```

Listing 4.2: Source code for the Database Insertion task

4.6.5 Threats to Validity

We have identified at least three issues that represent possible threats in this study. First, we split the experiment in two sessions. However, both sessions were executed in the same scenario and conducted by the same leaders. Second, the lack of knowledge related with Android's basic concepts may have impacted the subjects' performance. However, we gave an introduction covering all the necessary topics to implement the tasks. Third, our tasks and our target application may not be representative enough to cover how real-world developers use the Android API and its documentation.

4.7 Final Remarks

In this chapter, we presented a particular configuration of the APIMiner platform for the Android API, called Android APIMiner. In general numbers, Android APIMiner provides 79,732 source code examples extracted from 103 open-source Android applications. Moreover, we have conducted two studies to evaluate Android APIMiner. First, we have performed a field study based on usage data collected after four months of use by real users. By analyzing this data, we could understand the benefits and drawbacks of our current solution. Currently, 20,038 users have visited Android APIMiner, generating 42,034 page views. Also, the platform has provided 2,157 source code examples to such users. Second, we have conducted a controlled study to investigate Android APIMiner effectiveness, using a simple Android application that lacks two features the subjects have been asked to implement. As a result, we observed that Android APIMiner helps to solve specific programming tasks, which comprise few API elements. On the other hand, Android APIMiner may not help to solve more complex tasks, which comprise the call of several methods from different and not explicitly connected classes.

Chapter 5

Conclusions

5.1 Contributions

APIs constitute a reuse technology widely adopted in modern software development. However, the use and application of current APIs generally require a non-trivial effort. More specifically, the poor quality of APIs' documentation is a major obstacle for their use. Furthermore, recent empirical studies indicate that source code examples are an essential instrument to ease and to make more productive the use of APIs [McLellan et al., 1998; Robillard, 2009; Robillard and DeLine, 2011; Buse and Weimer, 2012]. However, there is still no consensus on how source code examples can be added to current API documentation formats, like JavaDoc.

To tackle this problem, we presented in this master dissertation the APIMiner platform, which relies on a private and curated database of source code examples to enrich APIs documentation in the JavaDoc format. The source code examples are extracted from a private source code repository and summarized using a static slicing algorithm. To demonstrate and evaluate our approach, we have implemented a particular instance of our platform for the Android API, and performed two studies to evaluate this instance: a large-scale field study and a small but controlled experiment.

More specifically, this master dissertation presents the following contributions:

1. We designed and implemented a platform that instruments APIs documentation in the JavaDoc format with source code examples extracted from a private repository. Furthermore, we have implemented a summarization algorithm—based on static slicing—that extracts small but relevant source code examples.

2. We have implemented and configured a particular version of the APIMiner platform for the Android API, with 79,732 source code examples extracted from 103

open-source applications. From such examples, 60,095 (75%) have less than 10 lines of code. Android APIMiner is publicly available at `http://apiminer.org`.

3. We have conducted a large-scale field study using the Android API version of the platform, when professional Android developers accessed the system during four months. After this period, the platform has been accessed 20,038 times from 130 different countries, generating more than 42,000 page views. Also, the platform has provided 2,157 source code examples to such users.

4. We have conducted a controlled experiment involving 17 subjects that implemented maintenance tasks in a small Android application with the assistance of the source code examples provided by APIMiner. We observed that the examples provided by APIMiner helped to solve specific programming tasks, which comprise few and connected API elements. On the other hand, the current examples provided by APIMiner are less useful to solve complex tasks, which require the implementation of more complex programming protocols.

5.2 Comparison with Related Work

Table 5.1 compares existing API recommendation tools with APIMiner. Code Search Engines (CSEs) represent the simplest approach regarding examples recommendation, as they provide examples without any structural preprocessing [Holmes et al., 2006; Zhong et al., 2009; Duala-Ekoko and Robillard, 2011]. IDE-based tools can explore the syntactic context provided by the IDE to recommend more relevant examples. However, the examples provided by such tools are not documentation-focused. Furthermore, IDE-based tools are restricted to the IDE they have been implemented to.

In counterpart, JavaDoc-based tools are designed and implemented to be independent from IDEs and usually are publicly accessed from the web (as APIMiner). Despite these characteristics, the available tools lack features that prevent them from replacing the original documentation. In fact, eXoaDocs is the tool with more resources similar to the ones proposed in APIMiner [Kim et al., 2009, 2010]. However, the process behind eXoaDocs' examples extraction—as well as the process used by the tool to instrument JavaDocs—can be improved in several aspects. For instance, eXoaDocs extracts examples from the web and summarizes them using only data dependencies. Moreover, the instrumented JavaDoc must be regenerated whenever a new source code example is processed. On the other hand, APIMiner relies on a curated source code repository and on a slicing algorithm that considers both data and control dependencies. Furthermore, in order to provide examples APIMiner requires the insertion of

Category	**System**	**Input**	**Interface**	**Output**
CSEs	CSEs	Text	Web	File
IDE Tools	Strathcona	Statement	Plugin	Method
	API Explorer	Statement	Plugin	—
	MAPO	Statement	Plugin	Method
JavaDoc Tools	ExoaDocs	Method	Web	Slicing
	APIExample	Type	Web/Plugin	Text
	PropER-Doc	Type	Desktop	Method
	APIMiner	**Method**	**Web**	**Slicing**

Category	**System**	**Repository**	**Clustering**	**Ranking**
CSEs	CSEs	Web	—	✓
IDE Tools	Strathcona	Private	—	✓
	API Explorer	Private	—	—
	MAPO	Private	✓	✓
JavaDoc Tools	ExoaDocs	Web	✓	✓
	APIExample	Web	✓	✓
	PropER-Doc	Web	✓	✓
	APIMiner	**Private**		**✓**

Table 5.1: Comparison between APIMiner and other API recommendation systems

a single button in a standard API documentation. Finally, we have evaluated our platform in the field, using a complex and widely popular API.

5.3 Future Work

As concluded from the results of the controlled experiment (Section 4.6.4), APIMiner provides less help in programming scenarios that require the call of several distinct and non-connected API methods. The reason is that in this case the examples have to reveal more complex API usage patterns, i.e., a sequence of multiple statements that must be called in a pre-defined order.

Therefore, we recommend the investigation of a solution that extends the current platform to identify and extract source code examples representing more complex APIs usage patterns. More specifically, we recommend the use of data-mining techniques—like association rules [Agrawal et al., 1993a,b]—to identify these patterns and to use an extended version of the summarization algorithm to extract them.

Furthermore, in some cases the examples extracted by APIMiner are not exactly legible, e.g., in some cases the examples have a large and complex set of statements.

Among other reasons, this happens mainly due to the nature of the slicing process. Because this process lacks information related with the underlying program semantics, the slicing technique can not distinguish relevant and irrelevant details of the examples. Therefore, we intend to investigate more sophisticated approaches to extract source code examples that might be more intuitive to developers, as the approach proposed by Buse and Weimer [2012].

Bibliography

ACM (2009). 9th ACM International Conference on Recommender Systems. http://recsys.acm.org/2009/.

Agrawal, R., Imieliński, T., and Swami, A. (1993a). Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, volume 22, pages 207–216.

Agrawal, R., Imieliński, T., and Swami, A. (1993b). Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216.

Agrawal, R. and Srikant, R. (1995). Mining sequential patterns. In *11th International Conference on Data Engineering (ICDE)*, pages 3–14.

Aragon Consulting Group (2006). Krugle open search. http://www.krugle.org/.

Baeza-Yates, R. and Ribeiro-Neto, B. (2008). *Modern information retrieval.* Addison-Wesley, USA, 2nd edition.

Black Duck Software (2004). Ohloh, the open source network. https://www.ohloh.net/.

Black Duck Software (2008). Koders code search engine. http://koders.com/.

Buse, R. P. L. and Weimer, W. (2012). Synthesizing API usage examples. In *34th International Conference on Software Engineering (ICSE)*, pages 782–792.

Codase (2005). Codase: Source code search engine. http://www.codase.com.

Duala-Ekoko, E. and Robillard, M. P. (2011). Using structure-based recommendations to facilitate discoverability in APIs. In *25th European Conference on Object-Oriented Programming (ECOOP)*, pages 79–104.

Eclipse Foundation (2011). Eclipse Code Recommenders. http://www.eclipse.org/recommenders/.

Google (2006). Google code search. http://code.google.com/codesearch.

Harman, M. and Hierons, R. (2001). An overview of program slicing. *Software Focus*, 2(3):85–92.

Holmes, R. and Murphy, G. C. (2005). Using structural context to recommend source code examples. In *27th International Conference on Software Engineering (ICSE)*, pages 117--125.

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

Jexamples (2005). Jexamples: Java examples in open source code. http://www.jexamples.com/.

Kim, J., Lee, S., won Hwang, S., and Kim, S. (2009). Automatic generation of example oriented API documents. In *24th International Conference on Automated Software Engineering (ASE).*

Kim, J., Lee, S., won Hwang, S., and Kim, S. (2010). Towards an intelligent code search engine. In *24th Conference on Artificial Intelligence (AAAI).*

Lethbridge, T. C., Sim, S. E., and Singer, J. (2005). Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341.

Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid mining: helping to navigate the API jungle. In *2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61.

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. G., Roesler, A. W., Tempest, J. T., and Spinuzzi, C. (1998). Building more usable APIs. *IEEE Software*, 15(3):78–86.

Michail, A. (2001). Code web: data mining library reuse patterns. In *23rd International Conference on Software Engineering (ICSE)*, pages 827–828.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *28th International Conference on Software Engineering (ICSE)*, pages 452–461.

Parnin, C., Treude, C., Grammel, L., and Storey, M.-A. (2012). Crowd documentation: exploring the coverage and the dynamics of API discussions on stack overflow. Technical report, Georgia Tech, College of Computing.

Robillard, M., Bodden, E., Kawrykow, D., Mezini, M., Tristan, and Ratchford (2012). Automated API property inference techniques. *IEEE Transactions on Software Engineering*, PP(99).

Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34.

Robillard, M. P. and DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732.

Robillard, M. P., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86.

Syer, M. D., Adams, B., Zou, Y., and Hassan, A. E. (2011). Exploring the development of micro-apps: A case study on the BlackBerry and Android platforms. *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 55–64.

Thummalapenta, S. and Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. In *22nd International Conference on Automated Software Engineering (ASE)*, pages 204–213.

Wang, L., Fang, L., Wang, L., Li, G., Xie, B., and Yang, F. (2011). APIExample: An effective web search based usage example recommendation system for Java APIs. In *26th International Conference on Automated Software Engineering (ASE)*, pages 592–595.

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*, 10:352–257.

Ye, Y. and Fischer, G. (2005). Reuse-conducive development environments. *Automated Software Engineering*, 12(2):199–235.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). MAPO: Mining and recommending API usage patterns. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343.

Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE)*, pages 563–572.

Appendix A

List of Android Systems

Table A.1 presents the list of open source Android systems that have been registered on Android APIMiner's repository. The following information is presented for each system: (a) the category of the system in Google Play, if applied; (b) a briefly description of the system; (c) number of downloads in Google Play; and (d) number of examples provided by the system.

Table A.1: List of Android systems used in the repository of Android APIMiner

System	**Category**	**Description**	**Downloads**	**Examples**
4Chan Image Browser	Entertainment	Image browser	25,000	84
aCal	Productivity	Calendar client	2,500	4,575
ADW Launcher	—	Launcher	—	2,796
Alien Blood Bath	—	Shooter game	—	312
Andless	Media/Video	Audio player	250,000	767
Android Launcher Plus	—	Launcher	—	1,238
Android Metronome	—	Metronome	—	52
Android motion detection	—	Motion detection framework	—	47
Android's Fortune	Entertainment	Based on "Fortune"	25,000	713
AndroSens	Lib/Demo	Data collector	7,500	169
Andtweet	Social	Twitter client	25,000	1,145
Ankidroid	Education	Flashcards spaced-repetition	250,000	3,914
Announcify	Business	Reads the caller's name	250,000	126
APG	Communication	Public key encryption tool	75,000	1,636
APNdroid	Tools	Mobile data Switcher	25,000	33
Aptoide Client	—	Application installer	—	3,175
Aptoide Uploader	—	Application uploader	—	487
ARViewer	Social	Augmented reality	2,500	1,136
Audalyzer	Tools	Audio analyzer	250,000	51
AR framework	—	AR framework	—	209
Banshee Remote	Media/Video	Remote control for Banshee	25,000	243
Barcode Scanner	Shopping	Barcode reader	75,000,000	801

Continued on next page

Table A.1—continued from previous page

System	Category	Description	Downloads	Examples
BatteryTracker	—	Battery sensor	—	43
Big Planet Tracks	—	Offline GPS tracker	—	699
Broadcast Dumper	—	—	—	19
Chime Timer	Tools	Timer	2,500	119
CIDR Calculator	Tools	IP subnet calculator	75,000	380
Clusterer	—	Cluster management module	—	67
ConnectBot	Communication	SSH client	2,500,000	1,437
Contact Owner	Tools	Contact info on widget	250,000	177
Corporate Addressbook	Communication	Address book	250,000	438
Countdown Alarm	Productivity	Timer	7,500	178
Crowdroid	Social	Twitter client	7,500	8,229
Cyanogen Updater	—	Cyanogen mod updater	—	1,140
Dazzle	Tools	Switcher widget	75,000	334
Dialer2	Productivity	T9 dialer search	75,000	453
Exchange OWA	—	Mail client	—-	148
FeedGoal	News	RSS reader	2,500	834
FFVideoLiveWallpaper	—	Live wallpaper	—	76
Floating Image	Entertainment	Image streamer	2,500,000	1,373
Formula	—	Calculaton	—	169
Frozen Bubble	Casual	Tetris style game	2,500,000	310
GCal Call Logger	Communication	Sync client	7,500	109
GCstar Scanner	Tools	Barcode scanner	2,500	21
GCstar Viewer	Tools	Personal collections manager	2,500	245
Hermit Android	—	—	—	666
Hot Death	Cards/Casino	Uno style game	25,000	454
K9 Mail	Communication	Email client	2,500,000	4,737
Keepassdroid	Tools	Passphrase manager	750,000	725
Lexic	Brain/Puzzle	Word-grid game	75,000	437
LibreGeoSocial	Social	Social network with AR	2,500	2885
MandelBrot	Lib/Demo	Fractal viewer	2,500	225
MemorizingTrustManager	—	SSL/TLS library	—	44
MINDroid	Communication	Lego Mindstorms controller	75,000	434
Mnemododo	Education	Flashcards spaced-repetition	750	306
Mustard	Social	Microblog client	25,000	2,093
Nethack Android	Brain/Puzzle	Nethack game	75,000	1,004
Newton's Cradle	Lib/Demo	Physics of Newton's Cradle	750,000	111
OI About	—	"About" box library	—	161
Open WordSearch	Brain/Puzzle	Word search game	2,500,000	635
OpenMap framework	—	OpenMap framework	—	51
OpenSudoku	Brain/Puzzle	Sudoku game	2,500,000	972
Orbot	Communication	Tor proxy	250,000	692
Password Hash	Tools	Password hash	2,500	61
Pedometer	Health	Counts your steps	250,000	174
Picture Map	Media/Video	Geotagged photos	25,000	57
Plughole	—	—	—	212
PMix	Media/Video	MPD client	25,000	457
Replica Island	Arcade/Action	2D side scrolling game	2,500,000	598
Ringdroid	—	Ringtone creator	—	609

Continued on next page

Table A.1—continued from previous page

System	Category	Description	Downloads	Examples
robotfindskitten	Casual	Port of robotfindskitten	25,000	95
Scrambled Net Full	Brain/Puzzle	Puzzle game	750,000	262
Secrets	—	Secret storage application	—	401
Shortyz	Brain/Puzzle	Crossword puzzle client	2,500,000	765
Shuffle	—	GTD application	—	1,417
Simon Tatham's puzzles	Brain/Puzzle	Puzzle game	250,000	549
Sipdroid	Communication	SIP client	750,000	1,441
SL4A	—	Scripting Layer	—	2,152
Slashdot	—	Slashdot reader	—	94
SMS Backup Plus	Tools	Backup application	2,500,000	735
Sokoban	Brain/Puzzle	Warehouse puzzle game	75,000	134
Solitaire Collection	—	Collection of card games	—	384
Spell Dial	Tools	T9 search dialer	250,000	114
Substrate	Personalization	Live wallpapers	250,000	211
SuperGenPass	—	Password hasher	—	251
Swallow Catcher	—	Podcast client	—	354
Swiftp	—	FTP server	—	171
Target	Brain/Puzzle	Anagram word puzzle	25,000	393
Test Card	—	—	—	40
Tippy Tipper	—	Tipping calculator	—	261
TouchTest	—	Touch test	—	85
Tricorder	—	Tricorder	—	431
Tumblife	—	Tumblr client	—	258
Twisty	Brain/Puzzle	Z-machine emulator	75,000	231
Twitli	—	Twitter client	—	2,737
Vector Pinball	Arcade/Action	Pinball game	75,000	63
Vidiom	Media/Video	Mobile video publishing tool	75	966
Voyager Connect	—	In-vehicle network diagnostic	—	23
Watch Aids	—	—	—	30
Wiki Dici	—	Wikitionary	—	42
Word Seek	Brain/Puzzle	Word search game	7,500	392
Wordpress	Social	Wordpress client	2,500,000	2,949
XBMC Remote	Media/Video	XBMC Remote control	750,000	1,794

Table A.2 shows the list of the considered Android systems that are under subversion control system (SVN) and their respective number of commits.

Table A.2: Android systems with SVN repository

System	# Commits
Alien Blood Bath	78
Andless	67
Android Launcher Plus	44
Android Metronome	2
AndroSens	34

Continued on next page

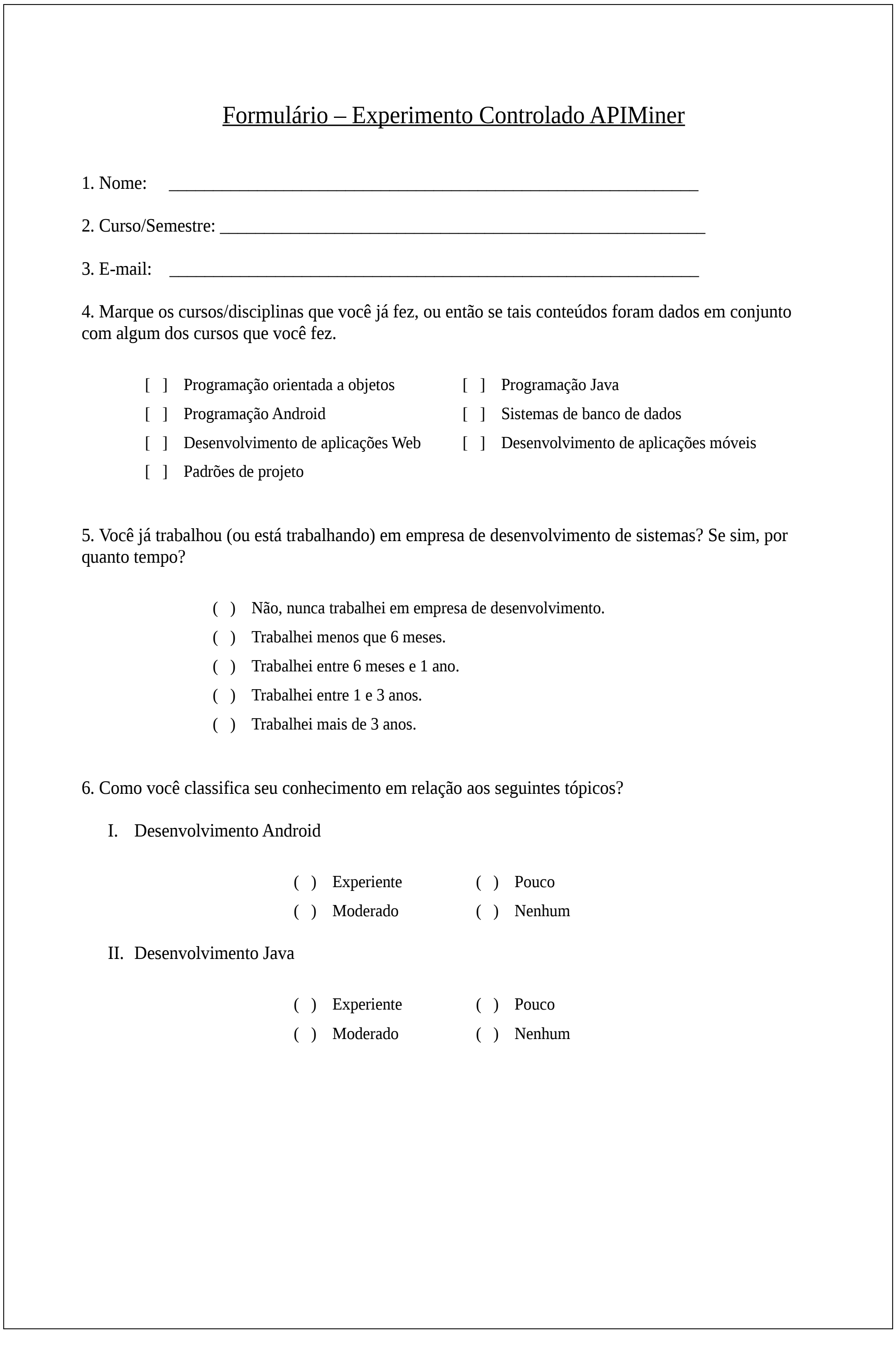
Table A.2—continued from previous page

System	# of Commits
APG	111
Audalyzer	47
Augmented Reality Framework	117
Barcode Scanner	628
BatteryTracker	3
CIDR Calculator	36
Clusterer	7
Contact Owner	4
Corporate Addressbook	3
Countdown Alarm	13
Cyanogen Updater	592
Dazzle	57
Exchange OWA	29
FeedGoal	159
Floating Image	124
Formula	9
Frozen Bubble	21
GCstar Scanner	3
GCstar Viewer	19
Hermit Android	32
Hot Death	24
LibreGeoSocial	932
MandelBrot	134
Motion Detection Framework	28
Nethack Android	333
Newton's Cradle	39
OpenMap Framework	13
OpenSudoku	10
Plughole	27
PMix	55
Replica Island	7
Ringdroid	33
Secrets	118
Shuffle	155
Sipdroid	279
Solitaire Collection	30
Spell Dial	15
Substrate	58
Target	36
TouchTest	18
Tricorder	119
Wordpress	772
Word Seek	17

Appendix B

Forms Used in the Controlled Study

B.1 Information Form



Formulário – Experimento Controlado APIMiner

1. Nome: ____

2. Curso/Semestre: ____

3. E-mail: ____

4. Marque os cursos/disciplinas que você já fez, ou então se tais conteúdos foram dados em conjunto com algum dos cursos que você fez.

[ ] Programação orientada a objetos	[ ] Programação Java
[ ] Programação Android	[ ] Sistemas de banco de dados
[ ] Desenvolvimento de aplicações Web	[ ] Desenvolvimento de aplicações móveis
[ ] Padrões de projeto	

5. Você já trabalhou (ou está trabalhando) em empresa de desenvolvimento de sistemas? Se sim, por quanto tempo?

( ) Não, nunca trabalhei em empresa de desenvolvimento.
( ) Trabalhei menos que 6 meses.
( ) Trabalhei entre 6 meses e 1 ano.
( ) Trabalhei entre 1 e 3 anos.
( ) Trabalhei mais de 3 anos.

6. Como você classifica seu conhecimento em relação aos seguintes tópicos?

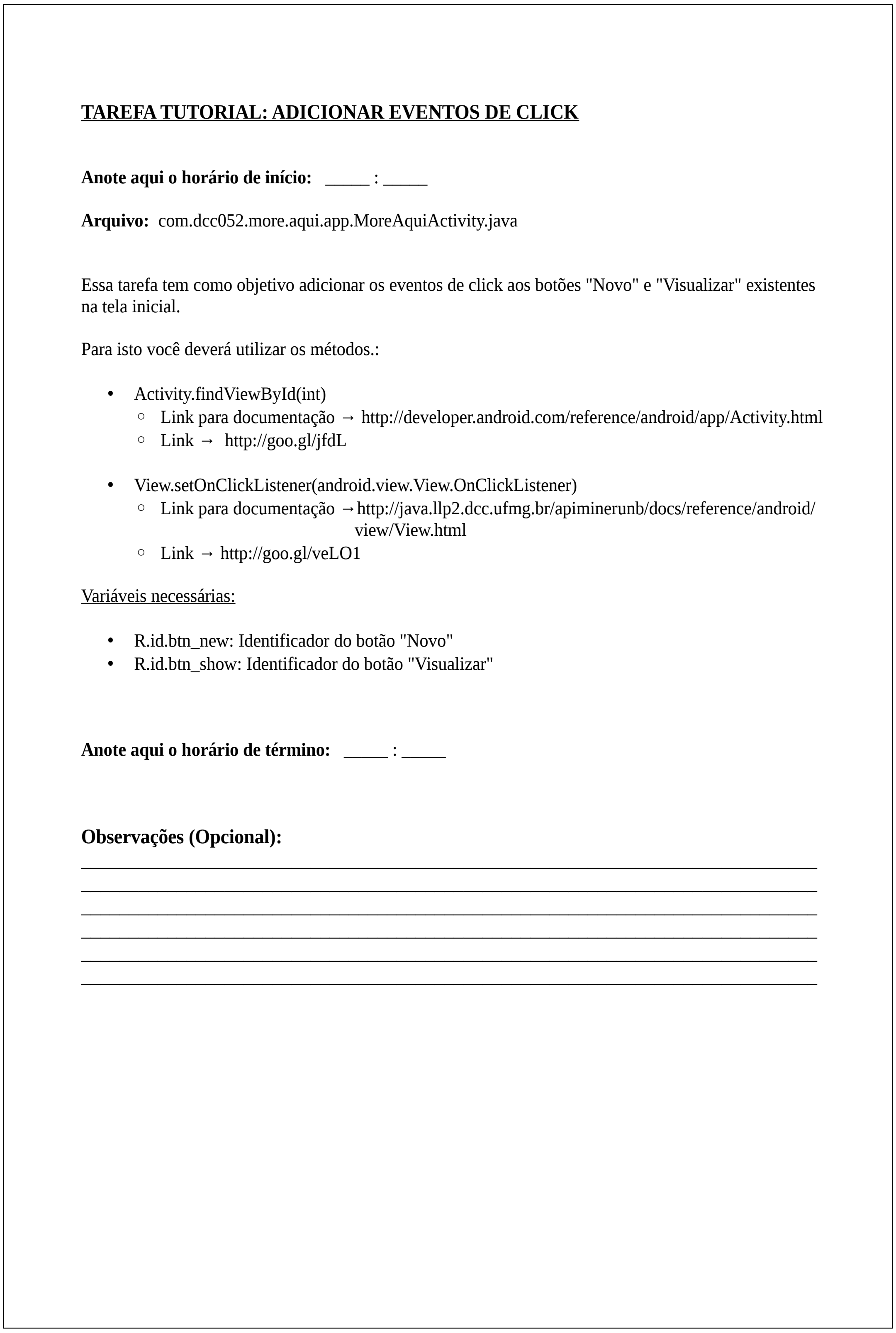
I. Desenvolvimento Android

( ) Experiente	( ) Pouco
( ) Moderado	( ) Nenhum

II. Desenvolvimento Java

( ) Experiente	( ) Pouco
( ) Moderado	( ) Nenhum

B.2 Task Descriptive Sheets



TAREFA TUTORIAL: ADICIONAR EVENTOS DE CLICK

Anote aqui o horário de início: _____ : _____

Arquivo: com.dcc052.more.aqui.app.MoreAquiActivity.java

Essa tarefa tem como objetivo adicionar os eventos de click aos botões "Novo" e "Visualizar" existentes na tela inicial.

Para isto você deverá utilizar os métodos.:

- Activity.findViewById(int)
  - Link para documentação → http://developer.android.com/reference/android/app/Activity.html
  - Link → http://goo.gl/jfdL

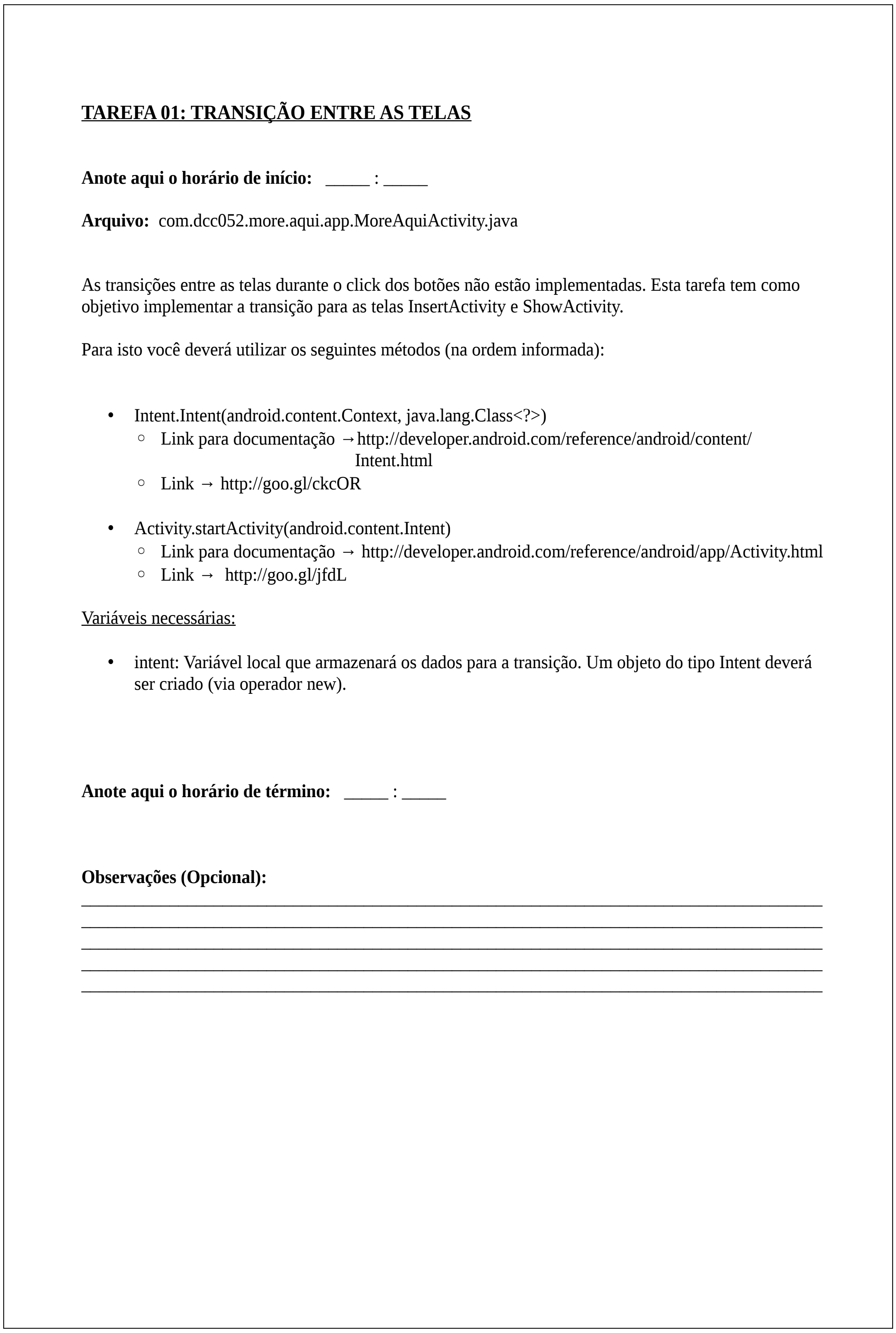
- View.setOnClickListener(android.view.View.OnClickListener)
  - Link para documentação → http://java.llp2.dcc.ufmg.br/apiminerunb/docs/reference/android/view/View.html
  - Link → http://goo.gl/veLO1

Variáveis necessárias:

- R.id.btn_new: Identificador do botão "Novo"
- R.id.btn_show: Identificador do botão "Visualizar"

Anote aqui o horário de término: _____ : _____

Observações (Opcional):



<u>TAREFA 01: TRANSIÇÃO ENTRE AS TELAS</u>

Anote aqui o horário de início: _____ : _____

Arquivo: com.dcc052.more.aqui.app.MoreAquiActivity.java

As transições entre as telas durante o click dos botões não estão implementadas. Esta tarefa tem como objetivo implementar a transição para as telas InsertActivity e ShowActivity.

Para isto você deverá utilizar os seguintes métodos (na ordem informada):

- Intent.Intent(android.content.Context, java.lang.Class<?>)
    - Link para documentação → http://developer.android.com/reference/android/content/Intent.html
    - Link → http://goo.gl/ckcOR
- Activity.startActivity(android.content.Intent)
    - Link para documentação → http://developer.android.com/reference/android/app/Activity.html
    - Link → http://goo.gl/jfdL

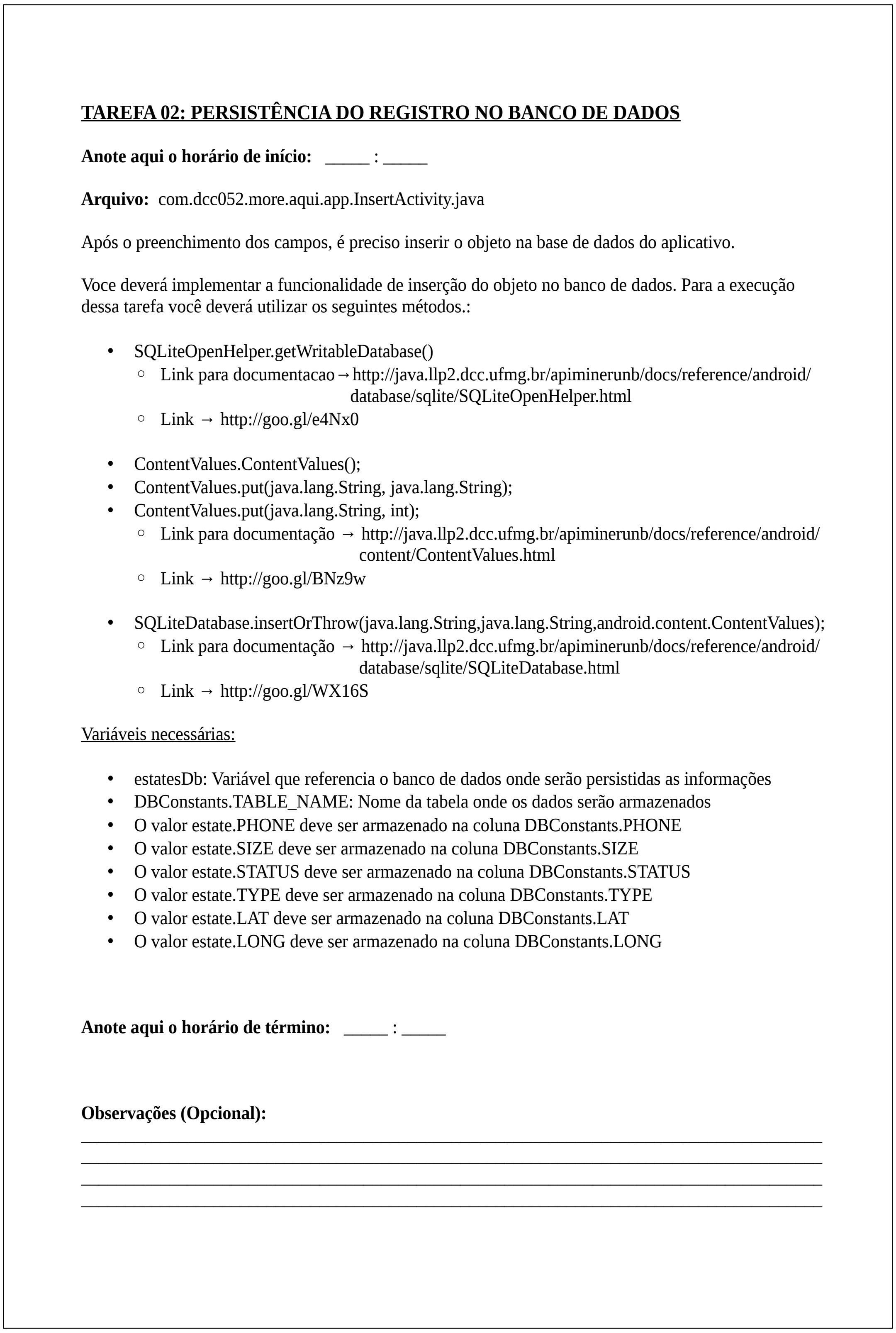
<u>Variáveis necessárias:</u>

- intent: Variável local que armazenará os dados para a transição. Um objeto do tipo Intent deverá ser criado (via operador new).

Anote aqui o horário de término: _____ : _____

Observações (Opcional):

____
____
____
____
____



TAREFA 02: PERSISTÊNCIA DO REGISTRO NO BANCO DE DADOS

Anote aqui o horário de início: _____ : _____

Arquivo: com.dcc052.more.aqui.app.InsertActivity.java

Após o preenchimento dos campos, é preciso inserir o objeto na base de dados do aplicativo.

Voce deverá implementar a funcionalidade de inserção do objeto no banco de dados. Para a execução dessa tarefa você deverá utilizar os seguintes métodos.:

- SQLiteOpenHelper.getWritableDatabase()
  - Link para documentacao→http://java.llp2.dcc.ufmg.br/apiminerunb/docs/reference/android/database/sqlite/SQLiteOpenHelper.html
  - Link → http://goo.gl/e4Nx0

- ContentValues.ContentValues();
- ContentValues.put(java.lang.String, java.lang.String);
- ContentValues.put(java.lang.String, int);
  - Link para documentação → http://java.llp2.dcc.ufmg.br/apiminerunb/docs/reference/android/content/ContentValues.html
  - Link → http://goo.gl/BNz9w

- SQLiteDatabase.insertOrThrow(java.lang.String,java.lang.String,android.content.ContentValues);
  - Link para documentação → http://java.llp2.dcc.ufmg.br/apiminerunb/docs/reference/android/database/sqlite/SQLiteDatabase.html
  - Link → http://goo.gl/WX16S

Variáveis necessárias:

- estatesDb: Variável que referencia o banco de dados onde serão persistidas as informações
- DBConstants.TABLE_NAME: Nome da tabela onde os dados serão armazenados
- O valor estate.PHONE deve ser armazenado na coluna DBConstants.PHONE
- O valor estate.SIZE deve ser armazenado na coluna DBConstants.SIZE
- O valor estate.STATUS deve ser armazenado na coluna DBConstants.STATUS
- O valor estate.TYPE deve ser armazenado na coluna DBConstants.TYPE
- O valor estate.LAT deve ser armazenado na coluna DBConstants.LAT
- O valor estate.LONG deve ser armazenado na coluna DBConstants.LONG

Anote aqui o horário de término: _____ : _____

Observações (Opcional):
____
____
____
____