Automated Documentation of Android Apps

Emad Aghajani, *Student Member, IEEE,* Gabriele Bavota, *Member, IEEE,*
Mario Linares-Vásquez, *Member, IEEE,* Michele Lanza, *Senior Member, IEEE*

Abstract—Developers do not always have the knowledge needed to understand source code and must refer to different resources (*e.g.,* teammates, documentation, the web). This non-trivial process, called program comprehension, is very time-consuming. While many approaches support the comprehension of a given code at hand, they are mostly focused on defining extractive summaries from the code (*i.e.,* on selecting from a given piece of code the most important statements/comments to comprehend it). However, if the information needed to comprehend the code is not there, their usefulness is limited.
We present ADANA, an approach to automatically inject comments describing a given piece of Android code. ADANA reuses the descriptions of similar and well-documented code snippets retrieved from various online resources. Our evaluation has shown that ADANA is able to aid the program comprehension process.

Index Terms—Program Comprehension, Documentation, Android.

1 Introduction

Software developers do not always possess the knowledge needed to comprehend the source code they are handling. This is especially true when code lacks documentation and comments [1], or when code and documentation do not co-evolve [2], [3], [4]. To make up for the lacking knowledge, developers often refer to teammates and other sources of information found on the Internet [5], such as Q&A websites like Stack Overflow. However, what developers often obtain are higher-level pieces of information, which are certainly useful, but do not always help to answer the question of what a specific chunk of source code is doing. This question has been tackled by automated summarization approaches [6], [7], [8], [9], [10], [11], [12], [13], and by creating extractive or abstractive summaries [14]. While in the former a subset of code/comment elements is selected from the code chunk to describe it, the latter includes information which is not explicit in the original document [14]. However, if the information to comprehend the code is simply not there, these approaches fall short.

We present ADANA, an approach to automatically generate and inject comments that describe a given piece of Android-related code. ADANA reuses the descriptions of similar and well-documented code snippets and is powered by a knowledge base of 64k well-described code snippets automatically retrieved and processed from Github Gist and Stack Overflow. ADANA also benefits from ASIA, a clone detector we tailored to identify Android-related code clones.

We evaluated ADANA in three studies. Results show that (i) ADANA can, on average, automatically document with code comments one third of the code composing a mobile app; (ii) the ASIA clone detector can find similar code snippets with good precision ($\sim$70%); and (iii) the comments injected by ADANA help in code comprehension both in terms of time needed and comprehension level.

2 ADANA

ADANA is implemented as a framework that includes an Android studio plug-in, a set of backend services for analyzing and extracting data from online repositories, and a knowledge base for storing snippets and descriptions. ADANA works as depicted in Figure 1. Dashed arrows represent dependencies (*i.e.,* ① and ⑦), full arrows indicate flows of information between components. Black arrows (*i.e.,* ① to ④) indicate operations performed with the goal of building the ADANA knowledge base; red arrows represent actions triggered by a request to document a selected piece of code through the ADANA Android Studio plug-in.

ADANA mines from the Web pairs composed of code snippets related to Android development and their description, which illustrates the task/feature implemented through a snippet (①). Quality checks ② are performed on the mined data to remove noise, such as pairs including non-Java code or unlikely to contain a meaningful description (*e.g.,* a single word description). The selected pairs are provided to the *description standardizer* ③ to convert the mined descriptions into a format suitable to document the related snippet of code and to store the processed pairs in the ADANA database ④ to form the knowledge base.

The developer using the ADANA Android Studio plug-in can select any snippet of code in the IDE and ask ADANA to describe it ⑤. The developer can also tune the "granularity level" of the description she desires (*e.g.,* describing every single block in the selected code, or getting an overall description of it). ADANA looks in the knowledge base for clones of the code snippet selected in the IDE by using ASIA (Android SImilarity Assessment).

ASIA is a clone detection approach tailored for Android ⑥ - ⑧ that we have designed to support ADANA. Once the code clones and their related descriptions are collected, the *descriptions ranker* component selects the best description(s) for the selected piece of code (⑨ - ⑩). Finally, the selected descriptions are pushed back to the IDE ⑪ where the developer can integrate them as code comments.

- *E. Aghajani, G. Bavota and M. Lanza are with the Università della Svizzera italiana (USI), Switzerland. E-mail: firstname.lastname@usi.ch*
- *M. Linares-Vásquez is with the Universidad de los Andes, Colombia. Email: m.linaresv@uniandes.edu.co*

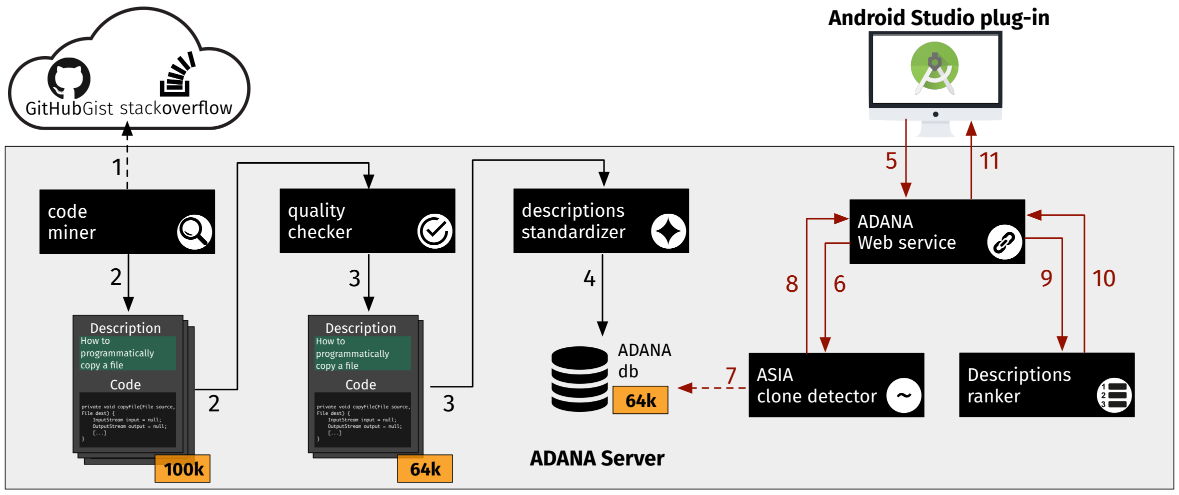


Fig. 1: The ADANA architecture.

2.1 Building the ADANA Knowledge Base

The ADANA knowledge base aims at containing snippets of code accompanied by their description (*e.g.*, "*downloads a file and shows the progress in a ProgressDialog*"). The knowledge base contains pairs of $\langle code, description \rangle$. We instantiated ADANA to the specific problem of documenting Android apps. We populate the knowledge base with code snippets (and related descriptions) relevant for Android apps.

Mining code snippets and related descriptions. ADANA's *code miner* component mines GitHub Gist [15] and Stack Overflow [16] to identify snippets of code with their related description to populate the knowledge base with pairs of $\langle code, description \rangle$.

Gist. A Gist can be a set of code files, a single code file, or a code snippet. Gists are particularly suited for our approach due to the fact that most of them are accompanied by a short description explaining their purpose (*e.g.*, the `RestartWifi` Gist [17] is accompanied by the description "*Code snippet to restart wifi interface (Android)*"). To identify relevant $\langle code, description \rangle$ pairs in Gist, we mined from the official Android documentation [18] all packages available in the Android APIs (*e.g.*, `android.bluetooth` or `android.support.v4.net`). Then, for each of the mined packages P_i, we used the Gist search feature to identify all code snippets containing an import statement `import Pi.*`, where the $.*$ acts as a wildcard (*i.e.*, it can represent a single class imported as well as the whole package).

We filter out all Gists not written in Java by using the filter provided by GitHub Gist. While the adopted search heuristic does not identify snippets of code not containing any import statement even if they are relevant to Android (false negatives), it is unlikely to select Gists that are not relevant to Android (false positives). In ADANA, we favor quality of data over quantity in the construction of the knowledge base. This has the drawback of limiting the amount of data available, reducing the number of code snippets that can be automatically documented.

Given a retrieved Gist, we create one knowledge base entry (*i.e.*, one pair $\langle code, description \rangle$) for every Java file composing it (remember that a Gist may have multiple files), using the Java file as the *code*, and the description provided by the user who shared the Gist as the *description*. By using this search heuristic, we extracted 22,864 pairs of $\langle code, description \rangle$ from Gist.

Stack Overflow (SO) is a well-known Q&A website. Besides mining the Q&A posts in SO, we also mined SO documentation, an initiative aimed at creating reference material for developers, collecting code examples showing how to deal with common tasks. Note that the SO documentation was recently shut down. However, we mined it (as detailed in the following) to extract the pairs $\langle code, description \rangle$ needed in the ADANA knowledge base when it was still online (May 2017). Since the extracted pairs still represent a precious source of information, we decided to keep them.

SO documentation is a straightforward resource from where to mine code snippets and their description since it includes pages related to a given topic (*e.g.*, *device display metrics*) showing snippets (and related descriptions) aimed at dealing with common tasks related to the topic (*e.g.*, programmatically capturing the size of the device display).

The *code miner* scrapes all Android-related topics that were already grouped together in the SO documentation. Each topic contains one or more examples (*i.e.*, pairs of $\langle code, description \rangle$), showing how to deal with tasks related to the topic (*e.g.*, the "Intent" topic, contained 19 pairs).

Some preprocessing was needed to identify in each example the related code and description. As for the code, we identify it as the text delimited by the $<$ `pre` $><$ `code` $>$ HTML tags. These tags are used in SO to format the code elements in the questions/answers and, in this case, in the examples reported in the SO documentation.

If multiple Java code snippets were present in the example, we merged them together to create the final code related to the example. Note that since we are focusing on Android, the related SO documentation posts could contain posts mixing code snippets written in Java, C++, and XML as well as makefiles. Since ADANA only supports the automatic documentation of Java code, we used a keywords-based heuristic to identify Java snippets. In particular, given a code snippet, we checked whether any of the regular expressions reported in Table 1 matched.

TABLE 1: The heuristic to identify Java code snippets.

#	Regular expression	Language
1	`"^\s*@Override"`	Java
2	`"^\s*<|>\s*"`	XML
3	`"^\s*\w+\s+:="`	Makefile
4	`"^\s*#ifdef|^\s*#ifndef|^\s*#include |^\s*#define|^\s*extern "C"|^\s* public:|^\s*private:|^\s*protected:"`	C++
5	otherwise	Java

The regular expressions were checked in the exact order reported in Table 1 and the process was stopped as soon as the first regular expression was matched. The basic idea behind this keywords-based approach is to exploit the unique features of each language (*e.g.,* the `@Override` keyword is only available in Java). A manual analysis of 660 code snippets from our knowledge base (details to follow) confirmed the validity of this simple filtering heuristic, since no non-Java snippets were found.

For the description, we extracted the text contained in the HTML tag having `class` = "`doc` − `example` − `link`". This text represented a short description of the code shown in the example (*e.g.,* "*Open a URL in a browser*"), and it is a good fit to concisely document similar snippets of code a developer will select in the Android Studio plug-in. Overall, we extracted 885 $\langle code, description\rangle$ pairs from SO documentation.

Concerning SO Q&A discussions, we mined the SO database dump dated June 2017, retrieving all questions:

1) *Tagged with a tag containing the word "android"*;
2) *Containing the word "how" in the title.* We use the question title as the description of the code snippets we mine from the answers. A sentence like "How to pass an object from one activity to another on Android" is easily converted into a short description to document a piece of code (*e.g.,* "Passes an object from one activity to another");
3) *Having at least one answer positively rated and/or accepted.* In SO, users can up- or down-vote answers. Also, the person who asked the question can "accept" a specific answer. Since we will use the code snippets reported in the answers as code documented by the question title, we want to make sure that the selected questions have at least one positively judged answer.

Once extracted the set of questions satisfying these constraints, we compose the pairs $\langle code, description\rangle$; the title of each question is used as the description (*e.g.,* "How to disable WiFi in Android"). The descriptions, as for the ones mined from the other repositories, are cleaned and standardized. As for the code, from each accepted/positively rated answer, we extract the code exploiting the < `pre` >< `code` > HTML tags, and use the same keywords-based approach exploited in SO documentation to only consider java code snippets (see Table 1). Thus, from a single question we can extract multiple implementations of the same task reported in different accepted/positively rated answers (*e.g.,* different snippets showing how to disable WiFi in Android). We extracted 76,769 $\langle code, description\rangle$ pairs by mining SO Q&A.

The overall mining process resulted in ∼100k $\langle code, description\rangle$ pairs coming from Gist (∼22k), SO Documentation (∼1k), and SO (∼77k) discussions. The larger amount of data coming from SO Q&A discussions is no surprise, considering its popularity.

TABLE 2: Original & standardized descriptions examples

Original Description	Standardized Description
How to pass an object from one activity to another on Android - API level 23+	Passes an object from one activity to another
Programmatically download a file with Android, and showing the progress in a ProgressDialog	Downloads a file and shows the progress in a ProgressDialog
A simple wrapper for Scalpel (https://github.com/JakeWharton/scalpel) that includes toggle controls accessible from a right-side navigation drawer.	A simple wrapper for Scalpel that includes toggle controls accessible from a right-side navigation drawer

Checking the quality of the mined data. The *quality checker* assesses the suitability of the collected $\langle code, description\rangle$ pairs for the ADANA knowledge base in two steps. The first one aims at removing from the descriptions unnecessary parts. In particular, in this step:

1) New lines are replaced with a space;
2) References to URLs are removed (see *e.g.,* 3rd description in Table 2);
3) Common adverbs indicating the need for performing a task programmatically are removed; these adverbs are often present in questions in which developers ask for help on Stack Overflow (*e.g.,* how do I programmatically [...])—see 2nd description in Table 2;
4) Expressions clarifying the Java and/or Android context of the task are removed (see 1st description in Table 2), since the mined descriptions certainly document Java/Android code thanks to the previous filters.

All the four steps described above are performed by using regular expressions. Table 3 reports the regular expressions used for the steps 2), 3), and 4). Note that the regular expression defined for step 4) uses other regular expressions we report in Table 4. For example, the first regular expression in Table 4, named *semanticVersion*, is used indirectly in the step 4) (last row in Table 3) to remove expressions clarifying the Java and/or Android context of the task.

After the cleaning process, the *quality checker* excludes all $\langle code, description\rangle$ pairs not satisfying a set of three requirements we defined to ensure the quality of the data stored in the ADANA knowledge base. The three requirements have been defined by the first author by manually analyzing a 99%±5% statistically significant sample (computed by using the Student's t-distribution) of the collected data (660 $\langle code, description\rangle$ pairs), looking for possible heuristics to discard low-quality descriptions/code snippets.

This process led to the exclusion from the knowledge base of all $\langle code, description\rangle$ pairs in which:

1) *The description is composed of less than four words.* We aim at removing code snippets accompanied by meaningless/useless descriptions (*e.g.,* "sample", "miniproject", "my application", *etc.*). In the manually analyzed sample (*i.e.,* the 660 instances), we found 110 descriptions having a description composed of less than four words, and only five of them were potentially useful for documenting the related code snippet (*e.g.,* "simple webview"). In the remaining 95.45% of cases, the descriptions were classified as useless to document the code.
2) *The description does not contain at least one verb.* Descriptions without verbs are unlikely to represent useful explanations and are too generic to properly document

TABLE 3: Regular expressions used by description quality checker for cleaning descriptions

Step	Regular expression	Goal
2)	"(?:at|in|on|here|@|see|check|check out|look|look at)?\s*[^a-zA-Z0-9_\]})]?\s*https?://\S+\s*[^a-zA-Z0-9_\[{(]?"	Removing references to an external resource (*e.g.,* url)
3)	"[(\[]?\s*,?\s*(?:programmatically|dynamically|through code|by code|using code|in code)\s*,?\s*[)\]]?"	Removing common unnecessary adverbs (*e.g.,* programmatically, through code, using code)
4)	"\s*(?:"+ android_pattern_withParentheses + "|"+ android_pattern_withoutParentheses + "|"+ java_pattern + "|"+ api_pattern_withParentheses + "|"+ api_pattern_withoutParentheses + "|"+ other_pattern + ")\s*,?\s*"	Removing expressions clarifying the Java and/or Android context of the task

code. In the 660 descriptions in our sample, we found 145 of them do not meet this requirement, and only 19 were classified as potentially useful (*i.e.,* 86.90% of descriptions with no verb were classified as not useful to document the code snippet). Thus, we defined this second heuristic, and use the Stanford CoreNLP toolkit [19] to identify the presence of verbs.

3) *The code snippet contained less than 50 characters (excluding white spaces) or more than 50 effective code lines (blank lines and comments excluded).* This removes very short and very long code snippets unlikely to represent the implementation of a well defined task. Indeed, in our manually analyzed sample we found 62 "too short" or "too long" snippets, with only 7 classified by the first author as implementing a well defined task. Moreover, not surprisingly, we observed that long snippets usually come with too generic descriptions such as "Custom DigitalClock" which makes them inappropriate for our purpose, *i.e.,* documenting a fine-grained piece of code.

During the manual analysis of the 660 $\langle code, description\rangle$ pairs, we also checked for the presence of non-Java snippets, to verify whether our keyword-based approach to isolate Java snippets works (see Table 1). All the 660 inspected snippets were in Java, confirming the validity of the defined approach. After cleaning the dataset by removing all $\langle code, description\rangle$ pairs matching one or more of the three above heuristics, we obtained 63,558 $\langle code, description\rangle$ pairs that represent the ADANA knowledge base.

Standardizing code descriptions. Before adding the $\langle code, description\rangle$ pairs to the ADANA knowledge base, the *description standardizer* converts, using the Stanford CoreNLP toolkit [19], the mined descriptions in a format suitable to document source code. We defined this process after manually analyzing the previously mentioned sample of 660 descriptions. Table 2 reports three example of code descriptions before and after the standardization process.

The *description standardizer* starts by splitting the description into sentences [19]. Then, it converts all the instances of *how-to* and *howto* (if any) to *how to*. For all sentences starting with *how* (*e.g., how to implement [...], how do you manage [...], etc.*), it: (i) removes the first two words (*e.g., how to, how do, etc.*) and (ii) removes the 3rd word if it is a personal pronoun (I, you, he, *etc.*). Then, in all sentences the *description standardizer* (iii) converts each infinitive verb not following a modal verb to third person, to give the developer the feeling that the description is referring to "the code" she selects in the IDE (see *e.g.,* the first description in Table 2); and (iv) converts each gerund verb following a conjunction to third person (2nd description in Table 2). Once the descriptions are standardized, they are stored together with the related code in the ADANA knowledge base (see Figure 1).

2.2 The ADANA Web Service

ADANA provides a Web service that can be exploited by a REST client, such as the ADANA Android Studio plug-in. The Web service expects from the client an HTTP post request containing a snippet of code. Then, it accesses the knowledge base to look for clones of the provided code snippet, to identify a suitable description provided to the client as an HTTP response. We detail the main steps behind this process as follows (red arrows in Figure 1).

The ASIA clone detector. The identification of code clones for the code snippet provided by the client is performed by running our ASIA clone detector on the knowledge base. It is worthwhile to explain why we decided to devise our own clone detector rather than reusing one of those existing in the literature [20], [21], [22]. We needed a clone detector able to run on incomplete, uncompilable code. Indeed, most of the code snippets we mined from SO are not complete compilation units. This excludes the use of efficient and well-known tree-based clone detectors such as Deckard [23]. The obvious choice in these cases is to use text-based clone detectors exploiting Information Retrieval (IR) techniques such as Simian [24], that can work on any given piece of code, compilable or not. However, they do not take advantage of the peculiar characteristics of Android code: native Android apps are highly dependent on the Android APIs [25], [26], [27]. This can substantially help in identifying whether two snippets of code implement the same feature (*i.e.,* whether they are clones). Indeed, snippets of code implementing the same feature in Android (*e.g.,* identifying the GPS location of the device) are basically "forced" to exploit the same APIs.

For these reasons, we defined ASIA, an approach built on top of standard IR clone detection and tailored for identifying clones in Android-related code. We show in Section 3.2 that ASIA achieves a better accuracy as compared to Simian [24].

ASIA is designed to detect not only exact clones (type-1 clone), but also clones differing for variable renaming (type-2), for the addition/deletion of few lines of code not changing the main feature implemented in the code (type-3), or even totally different implementations of the same functionality (type-4). Indeed, given the main goal of ADANA (*i.e.,* documenting a piece of code to explain what it implements), any type of code clone represents a valuable source of information.

To explain how ASIA computes the similarity between two code snippets S_i and S_j we introduce two similarity measures. The first is the standard Vector Space Model (VSM) cosine similarity [28] between the two vectors of words representing S_i and S_j. When applying VSM we (i)

TABLE 4: Regular expression used by the ones in Table 3

Pattern name	Regular expression (python regex)	Description
semanticVersion	`"\d+(?:[.]\d+)*(?:[.]X|[.][*])*[\b\W]"`	Semantic versioning format, *e.g.,* **4.***, **5.0**, or **3.7.x**
semanticVersion_widthOptionalParentheses	`"(?:"+semanticVersion+"|[(\[]\s*"+semanticVersion+"\s*[)\]])"`	Semantic versioning with optional parentheses, *e.g.,* **(4.*)**, **5.x.x**, or **[3.7.x]**
semanticVersion_withOptionalRange	`semanticVersion + "(?:\s*\+|\s*and above|\s*and later|\s*and higher|\s*and below|\s*and lower|\s*and up|\s*and further|\s*(?:-|to|and|or)\s*"+semanticVersion+")?"`	Semantic versioning with optional range, *e.g.,* **3.4+**, **4.x and above**, **3.0-6.x**, or **4.* and 5.***.
android_names	`"(?:pre-)?(?:Cupcake|Donut|Eclair|Froyo|Gingerbread|Honeycomb|Ice Cream Sandwich|IceCream Sandwich|ICS\b|Jelly Bean|JB\b|KitKat|Lollipop|android l\b|Marshmallow|android m\b|Nougat|android n\b|android tv\b)"`	List of Android Code names and their abbreviations, *e.g.,* **Nougat**, **pre-Cupcake**, or **JB**.
prefixWords	`"(?:^|-|\bin|\bon|\bwith|\bfor|\bfrom|\bor|\band|&)"`	A set of common prefix words, *e.g.,* **"from"** in "from android JB".
suffixWords	`"(?:beta|versions|version|application|app|devices|device|emulators|emulator|studio)"`	A list of words might appear after referring to an Android version to add more contextual information, *e.g.,* **version**, **application**, or **emulator**
android_fullname	`"(?:android(?!'s)|android\s*"+semanticVersion_widthOptionalParentheses+"|(?:android)?\s*"+android_names+")\s*"+suffixWords+"?"`	The usual way to refer to a specific Android versions, *e.g.,* ***Android 3.****, ***android honeycomb***, *or* ***android Icecream emulators***
android_fullname_withOptioanlRange	`android_fullname + "(?:\s*\+|\s*(?:\band|\bor|/|&|\bto|-|)\s*(?:higher|above|later|upper|up|further|below|lower|low|(?:android)?\s*(?:"+semanticVersion_widthOptionalParentheses+"|"+android_names+")))*\s*"+suffixWords+"?"`	Referring to a range of Android versions, *e.g.,* **Android 3.x emulators and higher**, **Android JB or higher devices**, or **Android 3.x and 4.x**
android_pattern_withParentheses	`"[(\[]\s*"+prefixWords+"?\s*(?:"+android_fullname_withOptioanlRange+"|android)\s*[)\]](?:\s*[|\-:])?"`	*e.g.,* **[Android 3.x and 4.x]:**, **[Android]**, or **(in android JB or higher)**
android_pattern_withoutParentheses	`prefixWords+"\s*"+android_fullname_withOptioanlRange+"(?:\s*[|\-:])?"`	Similar to *android_pattern_withParentheses*, but without parentheses, *e.g.,* **in Android ICS:**, **for android N or higher devices-**, or **with Android 3.x and 4.x**
api_pattern	`"(?:android)?\s*(?:api|sdk)\s*(?:level)?\s*\:?\s*(?:>|<|>=|<=)?"+semanticVersion_withOptionalRange+"\s*"+suffixWords+"?"`	*e.g.,* **Android API level >= 10.x**, **API level 23+**, or **Android SDK 21 emulators**
api_pattern_withParentheses	`"[(\[]\s*"+prefixWords+"?\s*"+api_pattern+"\s*[)\]]"`	*e.g.,* **(from Android API level >= 10.x)**, **[in API level 23+]**, or **(With Android SDK 21 emulators)**
api_pattern_withoutParentheses	`prefixWords+"?\s*"+api_pattern+"\s*"`	*e.g.,* **from Android API level >= 10.x**, **in API level 23+**, or **With Android SDK 21 emulators**
java_pattern	`"[(\[]\s*(?:in|for|by|using|with)?\s*java\s*[)\]]"`	*e.g.,* **(java)**, **[using java]**, or **(in java)**
other_pattern	`"(?:[(\[]\s*"+suffixWords+"\s*[)\]]|/java|with java|at runtime|during runtime)"`	*e.g.,* **[emulator]**, **/java**, or **during runtime**

normalized the snippets' text using identifier splitting (we also kept original identifiers), (ii) removed English words and reserved programming language keywords (used stop word lists available in [29]), and (iii) used the *tf-idf* weighting schema [28].

The second measure, that we named Android Similarity (AS), is a Jaccard similarity [30] between the Android-specific "objects" used in S_i and S_j.

We extracted from the Android documentation [31] the complete list of Android classes (*e.g.,* `Location`), API methods (*e.g.,* `distanceTo(Location)`), and constants (*e.g.,* `FORMAT_DEGREES`) in the Android framework API. We refer to the set of these Android-specific "objects" as ASO. We compute the AS between the two snippets as:

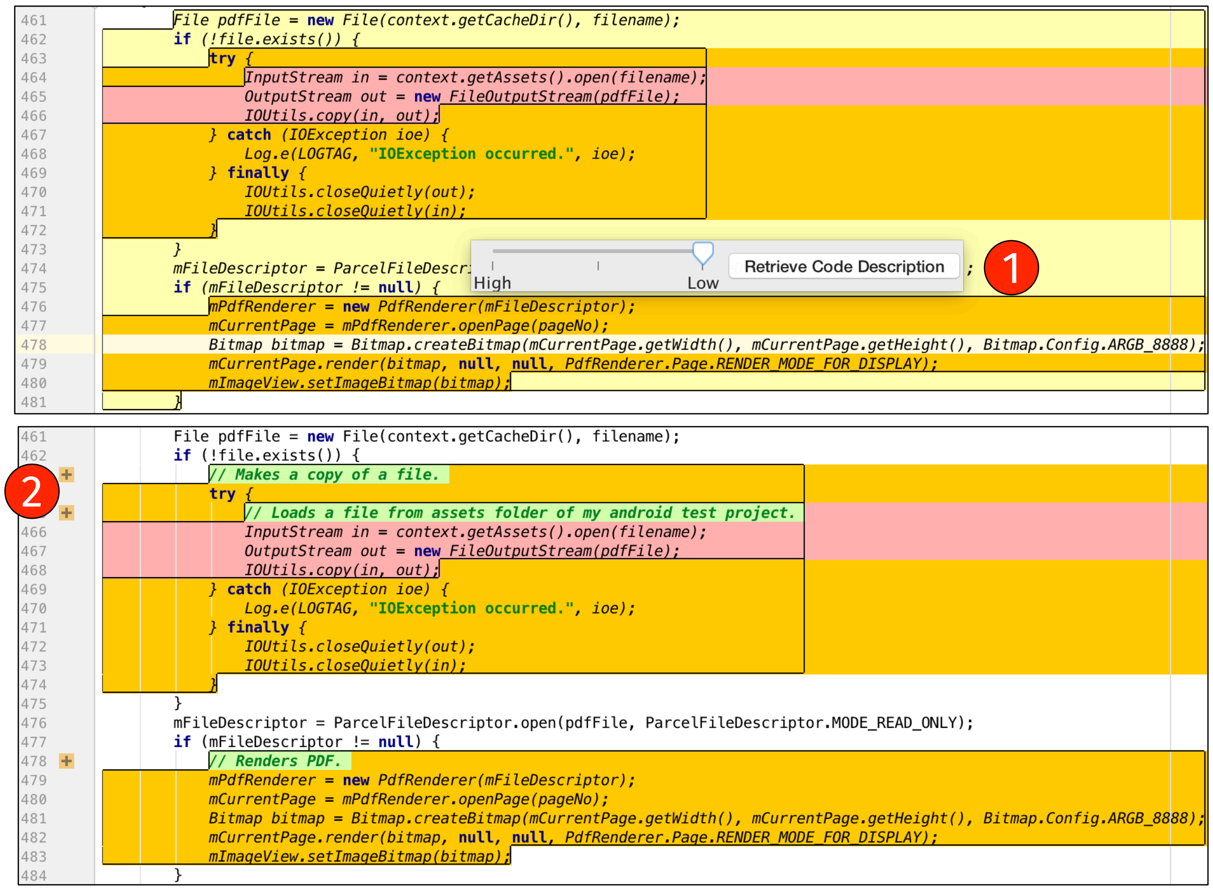
$$AS(S_i, S_j) = \frac{ASO_{S_i} \cap ASO_{S_j}}{ASO_{S_i} \cup ASO_{S_j}} \tag{1}$$

AS represents the percentage of Android-specific objects used by both snippets over the whole set of such objects they use. Given two snippets S_i and S_j, ASIA computes their similarity as:

$$sim(S_i, S_j) = \begin{cases} \alpha \cdot VSM(S_i, S_j) + \beta \cdot AS(S_i, S_j) & \text{if } |ASO| > 0 \\ VSM(S_i, S_j) & \text{otherwise} \end{cases} \tag{2}$$

If the two snippets do not contain ASO, their similarity is computed by relying on the VSM, otherwise it is calculated as a weighted sum of their VSM and AS similarity, both defined in [0, 1]. The two weights, α and β, are also both defined in [0, 1] and their sum must be equal to one, thus ensuring that $sim(S_i, S_j)$ is also in [0, 1]. ASIA detects the pair of snippets (S_i, S_j) as clones, if $sim(S_i, S_j) > t$. The tuning of α, β, and t is reported in Section 3.2.

Ranking descriptions. Once the list of clones for a code snippet is provided, three scenarios are possible. First, no clones have been found: the client is notified that ADANA is not able to document the snippet of code. Second, only one clone is identified: its description is returned to the client that will use it to document the code. Third, more than one



```
            File pdfFile = new File(context.getCacheDir(), filename);
            if (!file.exists()) {
                try {
                    InputStream in = context.getAssets().open(filename);
                    OutputStream out = new FileOutputStream(pdfFile);
                    IOUtils.copy(in, out);
                } catch (IOException ioe) {
                    Log.e(LOGTAG, "IOException occurred.", ioe);
                } finally {
                    IOUtils.closeQuietly(out);
                    IOUtils.closeQuietly(in);
                }
            }
            mFileDescriptor = ParcelFileDescr                                              ;
            if (mFileDescriptor != null) {
                mPdfRenderer = new PdfRenderer(mFileDescriptor);
                mCurrentPage = mPdfRenderer.openPage(pageNo);
                Bitmap bitmap = Bitmap.createBitmap(mCurrentPage.getWidth(), mCurrentPage.getHeight(), Bitmap.Config.ARGB_8888);
                mCurrentPage.render(bitmap, null, null, PdfRenderer.Page.RENDER_MODE_FOR_DISPLAY);
                mImageView.setImageBitmap(bitmap);
            }
```

```
            File pdfFile = new File(context.getCacheDir(), filename);
            if (!file.exists()) {
                // Makes a copy of a file.
                try {
                    // Loads a file from assets folder of my android test project.
                    InputStream in = context.getAssets().open(filename);
                    OutputStream out = new FileOutputStream(pdfFile);
                    IOUtils.copy(in, out);
                } catch (IOException ioe) {
                    Log.e(LOGTAG, "IOException occurred.", ioe);
                } finally {
                    IOUtils.closeQuietly(out);
                    IOUtils.closeQuietly(in);
                }
            }
            mFileDescriptor = ParcelFileDescriptor.open(pdfFile, ParcelFileDescriptor.MODE_READ_ONLY);
            if (mFileDescriptor != null) {
                // Renders PDF.
                mPdfRenderer = new PdfRenderer(mFileDescriptor);
                mCurrentPage = mPdfRenderer.openPage(pageNo);
                Bitmap bitmap = Bitmap.createBitmap(mCurrentPage.getWidth(), mCurrentPage.getHeight(), Bitmap.Config.ARGB_8888);
                mCurrentPage.render(bitmap, null, null, PdfRenderer.Page.RENDER_MODE_FOR_DISPLAY);
                mImageView.setImageBitmap(bitmap);
            }
```

Fig. 2: ADANA GUI

clone is retrieved: ADANA uses the *descriptions ranker* to identify the top clone with the most suitable description for documenting the selected code snippet.

The *descriptions ranker* assigns to the descriptions associated to each code clone a *Quality Score* (QS) indicating their suitability to document a code snippet. The QS for a description D of a clone C ($QS_{D,C}$) is computed by combining together three measures.

The first measure is the similarity (*i.e.*, the sim function in Equation 2) computed by ASIA between C (*i.e.*, the code described by D) and the selected snippet S. The higher the similarity between C and the code snippet S selected in the IDE, the higher the likelihood that D represents a good description for such a snippet.

The second measure is the *Comments Readability* (CR) proposed by Scalabrino *et al.* [32]. Extending the Flesch-Kincaid readability index [33], it captures the readability of code comments. We assume that descriptions having a high CR should be preferred over descriptions having a low CR, since the latter might be difficult to comprehend.

The third measure is the *Comments and Identifiers Consistency* (CIC_{syn}), proposed by Scalabrino *et al.* [32] to assess code readability. It computes the overlap of terms used in comments (in our case the description) and code identifiers (in our case, the code that we want to document). A high overlap of terms indicates that the comment describes the code well. CIC_{syn} takes into account synonyms (*e.g.*, "display" and "monitor"). CIC_{syn} computation is based on the Jaccard distance of terms used in the description and in the selected snippet.

Since the three measures can be all expressed in [0, 1], given a snippet S_i selected in the IDE, the *descriptions ranker* computes the *Quality Score* ($QS_{D,C}$) for a pair $\langle D, C\rangle$ (*i.e.*, $\langle description, code\rangle$) as:

$$QS_{D,C} = \frac{sim(S_i, C) + CR(D) + CIC_{syn}(D, S_i)}{3} \tag{3}$$

Once computed $QS_{D,C}$ for all clones retrieved for the selected snippets, the ADANA Web service returns to the client the description having the highest $QS_{D,C}$ value.

ADANA Android Studio plug-in. Figure 2 depicts the ADANA Android Studio plug-in. A developer using ADANA selects a snippet of code she is interested in comprehending, and then invokes ADANA by using the context menu (right click). ADANA requires the developer to select at least three code statements to automatically document (comment) it, to ensure that the ASIA clone detector has sufficient information to reliably identify code clones.

ADANA shows a *granularity slider* ① to set the granularity of the comments one is interested in retrieving: If the slider is to the left, ADANA looks for clones of the whole code selection and, in case of successful retrieval, only injects a single comment describing the selected code (*i.e.*, a single request is sent to the ADANA Web service). Moving the slider to the right, ADANA decomposes the selected code on the basis of the indentation level, as identified by parsing the AST representing the selection, and looks for clones of (i) the whole code selection, and (ii) the smaller code snippets obtained by decomposing the selection on the basis of the indentation levels. Each of the parts ADANA tries to document is shown in a different color. The maximum value of the *granularity slider* depends on the maximum indentation level of the selected code. Once the developer picks the granularity, she clicks on the "Retrieve Code Description" button close to the slider, obtaining the descriptions retrieved by ADANA for each of the highlighted code portions (see the bottom part of Figure 2). By using the code markers added by ADANA in the IDE ②, she can either accept it as is, modify and accept it, or reject it. If she accepts (before/after changing it), both code snippet and the associated comment are added to the ADANA knowledge base.

3 Study Design

The study addresses the following research questions (RQs):

RQ$_1$: *What percentage of Android apps' code can be automatically documented by* ADANA*?*

RQ$_2$: *What is the accuracy of* ASIA *in identifying clones for a given code snippet?*

RQ$_3$: *Does* ADANA *help developers during code comprehension activities performed on Android apps?*

3.1 Context Selection and Data Analysis

We describe for each research question its context, the data we collected, and the process adopted to analyze the collected data. The study dataset and the ADANA plug-in are publicly available [29].

3.1.1 What percentage of Android apps' code can be automatically documented by ADANA?

The focus of RQ$_1$ is not the correctness/usefulness of the provided comments, but on the *commented code coverage*. We expect the ADANA coverage to improve over time with the increase of data present in its knowledge base.

We selected 16 open source Android apps from the *open-source-android-apps* [34] GitHub project. The apps were randomly selected from 16 categories from Google Play, ensuring there is one app per category. The list of selected apps is available in Table 5. On average, the 16 apps have ∼10k ELOC (*i.e.,* Effective Lines Of Code, excluding blank and comment lines)—min=600, max=37k.

TABLE 5: The 16 apps selected for RQ$_1$

Category	Selected app (GitHub repository name)
Android TV	XiaoMi/android_tv_metro
Android Wear	romannurik/FORMWatchFace
Business	openshopio/openshop.io-android
Communication	VideoFly/VideoFly
Education	derekcsm/hubble_gallery
Finance	nothingmagical/coins-android
Game	snatik/memory-game
Health&Fitness	meghalagrawal/NightSight
LifeStyle	forezp/banya
Multi-Media	dkim0419/SoundRecorder
News	kinneyyan/36krReader
Personalization	ashutoshgngwr/10-bitClockWidget
Productivity	abhijith0505/CarbonContacts
Social Network	Jeffmen/Git.NB
Tools	cdeange/github-status
Travel	Swati4star/Travel-Mate

For each app, we simulate a developer selecting snippets of code and invoking the ADANA Web service to document them: given a class C implementing a set of methods M, we use a sliding window of length l to select snippets composed of l contiguous ELOC from the body of each method in M, until all the lines are covered by the sliding window. For example, given a method's body composed of six lines of code and assuming $l = 3$, we automatically extract four snippets of code S_i containing the following lines: S_1={1, 2, 3}, S_2={2, 3, 4}, S_3={3, 4, 5}, S_4={4, 5, 6}. This simulates a developer selecting snippets with three lines of code and asking ADANA to document them. Then, we keep track of the percentage of generated code snippets we were able to document by using ADANA. We experiment with values of l varying between 3 and 21 at steps of 3 (*i.e.,* 3, 6, . . ., 21). Our approach does not support selections shorter than three statements. While there is not always a correspondence between ELOC and number of statements, the three ELOC lower-bound ensures valid selections in most of the cases. Note that if a method in the apps considered in our study has less than three statements, we do not consider it.

We are assuming that the developer is not using the *granularity slider* (*i.e.,* she only wants an overall comment for the selected snippet of code). Indeed, given the various granularities we considered (from 3 to 21 ELOC), simulating the usage of the *granularity slider* is not needed, since the small snippets extracted from a method m (*e.g.,* those composed by 3 or 4 ELOC) are clearly contained into the larger snippets extracted from m (*e.g.,* those composed of 21 ELOCs). We ignore code not present in the method bodies (*e.g.,* import statements) since this is unlikely a real usage scenario for our approach.

To answer RQ$_1$ we show boxplots of the distribution of the *commented code coverage* obtained in the 16 apps for the considered values of l. Moreover, we present the *commented code coverage* in terms of (i) percentage of ELOC commented, and (ii) percentage of code snippets of length l commented.

3.1.2 What is the accuracy of ASIA in identifying clones for a given code snippet?

We randomly selected from our knowledge base 40 code snippets having between three and twenty ELOCs. We made sure that our approach was able to identify at least one clone for each of the selected snippets, otherwise we replaced it with another snippet from the knowledge base until all of them met the requirement.

Then, we asked study participants to assess whether the code clones identified by ASIA for each snippet were true or false positives. The choice of the upper-bound in the snippets' size was driven by the will of considering code snippets that are not too complex and, thus, limit the difficulty and effort required to participants in assessing the correctness of the identified clones. On average, our approach identified 4.8 clones per snippet (min=1, max=9). The set of 40 code snippets and the identified 192 clones are available in our replication package [29].

The participants were identified by using convenience sampling among the personal contacts of the authors. We invited developers and CS students/professors to take part in our study by using a Web application we developed to perform the following steps. First, we collected demographic data about participants (years of experience in programming, in Java, and in Android, their current position, *etc.*). Each participant was then required to assess the correctness of all clones identified by ASIA for eight snippets randomly selected from the 40 objects of this study. The Web application was designed to automatically balance the number of evaluations for each of the 40 snippets (*i.e.,* the number of participants assessing the correctness of each identified clone was roughly the same).

The eight snippets were presented individually (*i.e.,* each snippet in a different page) to participants, and each clone identified by ASIA for it was shown below the snippet with two radio buttons allowing the participant to express her assessment as: *it is a clone* or *it is not a clone*. We instructed participants to consider all types of clones (*i.e.,* from type-1 to type-4) as valid.

In total, we collected 534 evaluations across the 192 clones of the 40 snippets. We then removed the answers we collected from two participants with zero Android experience, and this resulted in (i) one snippet having no evaluations for its clones, and (ii) one snippet evaluated by only one participant. This latter had 11 clones reported by ASIA for which, only one was not assessed as a true positive. We removed these two snippets and corresponding clones from the analysis to have only clones that were evaluated by at least two participants. Thus, our analysis involves 490 evaluations related to 171 clones of 38 snippets. Each clone was evaluated, on average, by 2.87 participants (median = 2, Q3 = 3).

We analyzed questionnaires completed by 22 participants (11 professional developers, 4 PhD, 3 MSc, and 4 BSc students). Table 6 presents demographic information about the participants.

TABLE 6: Demographic of study participants (RQ_2)

#years experience in	Avg.	Median
Programming	7.7	7.0
Java programming	6.1	5.5
Android programming	1.9	1

We answer RQ$_2$ by reporting the percentage of true and false positives classified by the participants[1] as well as by discussing example cases of true and false positives, to highlight strengths and weaknesses of the ASIA clone detector (Section 4.2).

3.1.3 Does ADANA help developers during code comprehension activities performed on Android apps?

We asked 10 professional developers to comprehend a set of snippets of code with and without the help of the comments automatically injected by ADANA. Note that the scenario we aim at simulating in this study is that of developers comprehending code for which comments are not available.

We started by randomly selecting from the 16 mobile apps used in RQ$_1$ 16 methods (one per app) meeting the following criteria:

1. The percentage of true positives is equivalent to the precision measure (*a.k.a.,* clone detection rate) used by previous papers on clone detection [23], [35].

1) *Having between 10 and 50 ELOC*, to exclude methods that are too trivial or too complex to comprehend.
2) *Having at least one clone identified*, to ensure that ADANA added at least one comment to the methods.
3) *Being self-contained.* One of the authors manually analyzed the selected methods to ensure they were *self-contained*, *i.e.,* they could be comprehended without navigating additional source code (if not those of the Android API framework, available online). This resulted in the replacement of 3 methods.

We ran ADANA on each of the selected methods to have them with and without automatically injected comments. We moved the granularity slider to the right to add as many comments as possible. Also, we removed the original comments (if any) from the methods, to avoid a possible confounding factor and isolating the effect of the injected comments. Also, this is in line with our goal of simulating the real-life scenario in which the developer uses ADANA to understand a method which lacks comments. The comments removal was needed for 4 of the 16 methods (no comments were present in the remaining 12). We refer to the 16 original methods as the *uncommented* dataset, and to the 16 augmented with ADANA comments as the *adana* dataset.

We invited 10 participants via convenience sampling and ran this study via a Web application we developed. Demographic information about the participants are shown in Table 7.

TABLE 7: Demographic of participants (RQ_3)

#years experience in	Avg.	Median	Min.
Programming	8.4	7.5	1+
Java programming	7.0	6.5	1+
Android programming	2.1	2	1+

Each participant was required to comprehend a subset of eight methods randomly selected from the starting 32. Four of the eight snippets were selected from the *uncommented* dataset, and four from the *adana* dataset. We made sure that each participant comprehended eight **different** methods (*i.e.,* she did not comprehend twice the same method with and without the ADANA comments). The Web application was in charge of balancing the number of evaluations for each of the 32 methods. We collected 80 evaluations across the 32 methods (2.5 evaluations per method, on average).

The eight methods were presented individually and in a randomized order to mitigate learning and tiring effect. Participants were allowed to browse the Web to collect information about the types, APIs, data structures *etc.*, used in the methods. This was done to simulate the typical understanding process performed by developers. We asked participants to carefully read and fully understand each method. We clarified that "fully understand" can be read as "being able of explaining the method to another developer". Then, they were required to answer three *verification questions* about the method they inspected. The questions were defined, independently, by two of the authors for different sets of methods, with the goal of covering different areas of the method under analysis. This resulted in questions targeting both parts of the method commented and not commented by ADANA, and could represent a confounding factor. We preferred to focus our questions on the whole method rather than only on the parts commented by ADANA to not introduce a strong bias in our evaluation. Of the 48 formulated questions (3 questions $\times$ 16 methods), 23 questions explicitly targeted parts of the code commented by ADANA. This also includes *wrong* comments injected by ADANA and not just good comprehension hints, as we discuss in the results section. An example of a comprehended method together with its verification questions is provided in the result section (4.3).

The Web app we developed tracked the time needed by each participant to comprehend each of the eight methods and answer the three verification questions. Clearly, this included time spent by participants in browsing the Web looking for information needed to comprehend the snippet. We explicitly asked the participants to not interrupt the comprehension task in order to not introduce a bias in the tracked comprehension time and to report to us in case unexpected interruptions happened. None of the participants reported issues of this type.

To verify if ADANA helps developers in comprehending the code, we used the following two measures for code understandability, defined by Scalabrino *et al.* [36]:

Actual Understandability (AU). It is computed as the percentage of correct answers the participant provided to the three verification questions. Thus, the metric is defined in [0, 1] range, where 1 indicates high understanding.

Timed Actual Understandability (TAU). It is computed as:

$$TAU = AU\left(1 - \frac{Time}{\max Time}\right) \tag{4}$$

where $Time$ is the time needed to comprehend the method and answer the verification questions. The higher the AU is (*i.e.,* the percentage of correct answers), the higher is the TAU; and the higher the $Time$ is (*i.e.,* the time needed to understand the method), the lower is the TAU. Also, TAU is defined in [0, 1]. As done in [36], we considered the relative time ($\frac{Time}{\max Time}$) so that TAU gives the same importance to both the correctness achieved (AU) and the time needed ($Time$).

We computed these two proxies for each of the 80 evaluations by participants. Then, we compare their distributions for methods belonging to the *uncommented* and to the *adana* dataset. A normality check using the Shapiro-Wilk test indicated a statistically significant deviation from normal distribution (p-value< 0.05); hence we use non-parametric statistics. For all tests we consider a significance level $\alpha = 5\%$. We compare the results using the Wilcoxon signed-rank test. Since we do not know *a priori* in which direction the difference should be observed, we use a two-tailed test. We also assess the magnitude of the observed difference using Cliff's delta (d) effect size [37], suitable for non-parametric data. Cliff's d ranges in the interval $[-1, 1]$ and is negligible for $|d| < 0.148$, small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

Note that participants had no information about the purpose of the study and of the fact that comments were injected automatically. We revealed this information at the end of the experiment, and asked to comment their perceived usefulness in an (optional) open question. We report some of the representative comments left by participants.

3.2 Experimental Setting

To run our study we have to tune the ADANA's parameters and, in particular, the α, β, and t parameters used by the ASIA clone detector (see Section 2.2). To calibrate these parameters we created an oracle reporting true and false positive clones for a set of snippets. We randomly selected eight code snippets from the official Android development guide [38], making sure that the snippets:

- Were implementations of a well-defined task (*e.g.,* activate the WiFi network);
- Were implemented in Java, not involving any usage of files written in other languages (*e.g.,* XML files); and
- Made use of at least one "Android-specific object" (see Section 2.2).

The last rule was needed for the tuning of the α and β parameters. To properly set the weights of the VSM and of the AS similarities (Equation 1), we need to consider cases in which the AS can be computed. Then, we took the two longest snippets, and extracted from each of them one "sub-snippet", to simulate the situation in which the developer uses the granularity slider to obtain more fine-grained descriptions of the code. Thus, in total, we included in our oracle ten code snippets.

We identified in the ADANA knowledge base composed of 63,558 $\langle code, description \rangle$ pairs, clones of each selected snippet. In particular, for each snippet we created a set of candidate clones to manually validate by randomly selecting:

1) *Twelve candidate clones from the top-50 results returned by using the VSM similarity*. Thus, 12 clones that are in the top positions when using only textual information to identify clones. Selecting from the top of the ranked list, we expect to increase the likelihood of including true positives in our oracle.
2) *Twelve candidate clones from the top-50 results returned by using the AS similarity*. Thus, 12 clones that are in the top positions on the ranked list when using only information related to Android-specific objects (*i.e.,* classes, API methods, and constants of the Android framework). We made sure to not select in this stage candidate clones that were previously selected when looking at the top-50 results returned by the VSM similarity.
3) *Twelve candidate clones returned in position 51-to-500 by the VSM similarity or by the AS similarity (six for each similarity score)*. We expect these candidates to have a lower likelihood of being true positives, thus allowing us to obtain in our oracle a good mix of true and false positives for each snippet. We decided to not randomly pick from the whole ranked list (we considered up to position 500) to not make the classification of true and false positives trivial (*i.e.,* true positives have a very high similarity value, while false positives have very low similarity values), and thus to ensure a good tuning of the t parameter.

Our oracle includes 10 snippets and 36 candidate clones for each of them. Thus, we performed the tuning on a set of 360 candidate clones that represents a statistically significant sample of the 63,558 snippets present in the ADANA knowledge base with 95%$\pm$5% confidence interval (Student's t-distribution). This explains the choice of targeting 36 candidate clones per snippet. Once he obtained the dataset, the first author manually went through all the candidate clones marking each of them as a true or false positive.

The oracle we built is publicly available [29], and it includes 95 true positives and 265 false positives.

Finally, we run the ASIA clone detector on the built dataset, testing all 220 combinations of α, β, and t obtained varying α and β between [0, . . . , 1] at steps of 0.1 ensuring $\alpha + \beta = 1.0$ and t between [0.05, . . ., 1] at steps of 0.05. We evaluate each configuration in terms of (i) its precision, meaning the percentage of correct clones it identifies out of the returned clones, and (ii) its coverage, meaning the percentage of snippets for which it is able to identify at least one correct clone. Before discussing the results some clarifications are needed. First, we did not use recall, since for our specific application (*i.e.,* automatically identifying a description for a given code snippet) what we really care is to find at least one suitable description. This is why we rely on the snippets' coverage. Second, ADANA is the classic application in which precision is **much** more important than recall. Indeed, the scenario in which our tool will be used is that of a developer experiencing difficulties in comprehending a piece of code and, thus, asking for help to ADANA. In such a scenario it is better to just report to the developer a void result (*i.e.,* "*I am not able to document this code*") rather than providing a wrong description confounding the developer even more.

Figure 3 shows the results of the tuning process for (i) VSM similarity (*i.e.,* $\alpha = 1$ and $\beta = 0$), red line and bars (ii) AS similarity (*i.e.,* $\alpha = 0$ and $\beta = 1$), blue line and bars, and (iii) the best combination (in terms of high precision, good coverage compromise) of the two (*i.e.,* $\alpha = 0.5$ and $\beta = 0.5$) we identified. The results are shown for precision (y-axis, lines) and coverage (y-axis, bar chart) when varying the t threshold (x-axis). The results for all experimented combinations are available in our replication package [29].

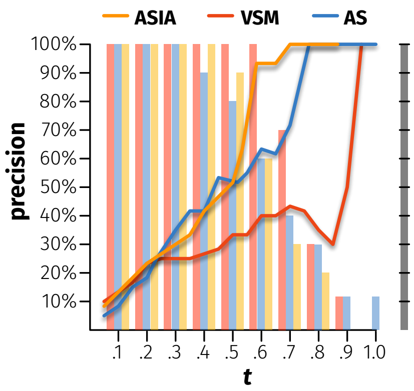


Fig. 3: Tuning of the α, β, and t ASIA parameters.

The VSM exhibits the lowest precision, with good values (100%) exhibited only when the coverage drops at 10%, meaning that it is able to identify true positives code clones for only one of the ten snippets in our oracle. The Android Similarity (AS) performs better, while still obtaining very low coverage (30%) when the precision values become acceptable ($> 75\%$). Finally, the best combination we identified for the ASIA clone detector is able to reach very high values of precision (93%) when the coverage still exhibits a good 60% (*i.e.,* we should be able to automatically document six out of ten snippets with such a level of precision) by using $t = 0.65$. $\alpha = 0.5$, $\beta = 0.5$, and $t = 0.65$ is the default ADANA configuration, and the one we will use in our study.

On the performance of the Simian [24] clone detector. As explained in Section 2.2, the obvious alternative to using ASIA in ADANA would have been to exploit a clone detector based on IR techniques. This is due to the need for running the clone detection on incomplete, uncompilable code. For this reason, before finalizing our decision of using ASIA, we also ran the Simian clone detector on the same dataset we used for the ASIA's tuning. Simian does not have a "similarity threshold" to tune, but simply returns the set of clones it is able to identify in a given dataset. However, it has a number of parameters to set (see [29] for the complete configuration we used). We set those parameters to make sure that Simian did not only look for exact (type-1) clones (*e.g.,* we ignore differences related to the name of the variables, constant values *etc.*). Simian was able to identify a correct clone for 50% of the 10 snippets (*i.e.,* coverage=50%) with a precision of 77%. ASIA, in the configuration we adopted, achieves a higher coverage (60%) accompanied by a higher precision (93%).

4 Results & Discussion

In the following sections we answer the three research questions formulated in Section 3.

4.1 What percentage of Android apps' code can be automatically documented by ADANA?

Figure 4 reports the *commented code coverage* achieved by ADANA on the 16 subject apps. The white box plots show the percentage of ELOC that were automatically documented by ADANA, while the grey ones report the percentage of automatically selected code snippets for which the ASIA clone detector was able to identify at least one clone. The results are shown when varying the length of the selected code snippet (l) between 3 and 21 at steps of 3 (x-axis). Finally, the black box plots show the average number of clones retrieved by ADANA for the snippets of code it was able to document. In total, this evaluation involved 114,499 code snippets of different length.

Looking at Figure 4, the first noteworthy finding is the stable coverage trend when varying the length of the selected snippets. Indeed, the median percentage of documented ELOC varies between 30% and 36%, while the percentage of documented code snippets between 38% and 45%. On the negative side, this indicates that ADANA generally cannot help developers in comprehending almost two-thirds of the apps' source code. While this might look like a negative result, it is worth remembering that the approach is fully automated and it is unrealistic to expect much higher coverage. Also, these percentages represent a lower-bound that is likely to increase over time with the growth in the size of the ADANA knowledge base.

Similar observations can be made for the average number of clones retrieved by ADANA for snippets it was able to document, with the median ranging between 14 and 18,

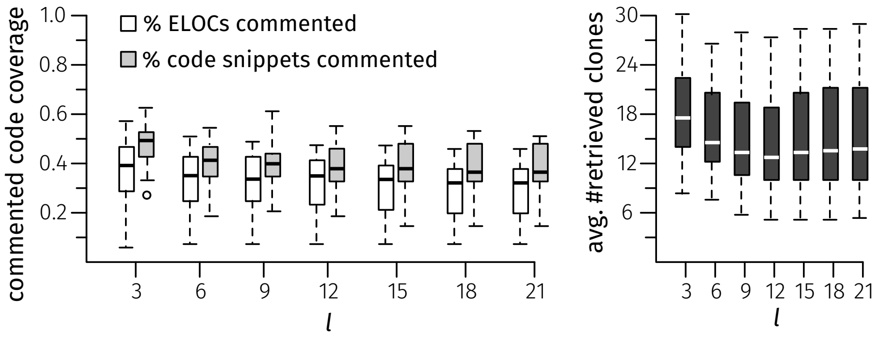


Fig. 4: Commented code coverage (left) and average number of retrieved clones (right)

indicating that ADANA is often able to retrieve several descriptions for a given snippet. This justifies the need for the *descriptions ranker* in our approach.

The app on which ADANA achieves the lowest coverage (∼7% ELOC and ∼12% snippets coverage) is FORM Watch Face for Android Wear [39]. It is developed for Android wearable devices (*i.e.,* watches) and, in the ADANA knowledge base, we only have 54 $\langle code, description\rangle$ pairs containing the word "*wear*": ADANA does not have enough relevant entries in its knowledge base. The code ADANA is able to document for this app is mostly standard Android code that can also be found in mobile phone apps.

10-bitClockWidget [40] is instead the app for which ADANA achieves highest coverage levels (∼48% ELOC and ∼50% snippets coverage). It implements a clock widget that can be embedded in the home screen. Android widgets strongly rely on the classes implemented in the `android.appwidget` package of the Android API framework, that does only contain five classes, thus promoting the use of similar code across different widgets.

Summarizing, the achieved results show a good coverage level exhibited by ADANA on the 16 subject apps, with almost one-third of the apps ELOC that could be automatically documented. Clearly, we did not focus on the correctness of the identified clones and, as a consequence, on the usefulness of the retrieved documentation, which is the object of the next research questions.

4.2 What is the accuracy of ASIA in identifying clones for a given code snippet?

As previously said, we collected 490 clones evaluations related to 171 clones of 38 code snippets. 381 of the 490 evaluations were marked as true positive, which accounts for a *percentage of true positives (a.k.a., precision)* of 77,76%.

Concerning the participants' agreement, for 118 clones out of 171 at least two-thirds of the evaluators classified the candidate clone as a true positive, while for 18 cases at least two-thirds of the evaluators agreed on classifying the candidate as a false positive. This means that moving the focus on the single code clones, by adopting a majority-voting schema (*i.e.,* by considering a clone as a true positive only if the majority of the evaluators classify it as a true positive), we obtain a precision of 69.00% (118/171). In more detail, for 99 out of 171 clones, all the participants evaluating the candidate clone agreed with ASIA (*i.e.,* answered "Yes, it is a clone"), while in 14 cases all the evaluators disagreed with our approach (*i.e.,* answered "No, it is not a clone").

We also investigated the types of code clones detected by ASIA. The first author manually analyzed the 118 clones classified by the majority of participants as true positives with the goal of classifying each of them as a type-1, type-2, type-3, or type-4 clone. We found no instances that can be considered as type-1 clones, four type-2 clones, 102 type-3 clones, and 12 type-4 clones. Thus, most of the clones identified by ASIA differ for the addition/deletion of few lines of code not changing the main feature implemented in the given code snippet.

Listing 2 reports an example of a type-3 clone detected by ASIA for the code snippet shown in Listing 1 classified by all participants as a true positive. The snippet and corresponding clone create a `Bitmap` object from a `URL`; note that the clone implements the same feature of the code snippet, but it has additional statements and there are changes in the identifiers.

```
URL url = new URL( "http://www.yourdomain/your/path/
    image.jpg");
HttpURLConnection connection = (HttpURLConnection)
    url.openConnection();
connection.setDoInput(true);
connection.connect();
final InputStream input = connection.getInputStream
    ();
Bitmap yourpic = BitmapFactory.decodeStream(input);
```

Listing 1: Code snippet creating a bitmap object from a URL

```
public Bitmap getBitmapfromUrl(String imageUrl)
{ try {
        URL url = new URL(imageUrl);
        HttpURLConnection connection = (
            HttpURLConnection) url.openConnection();
        connection.setDoInput(true);
        connection.connect();
        InputStream input = connection.getInputStream
            ();
        Bitmap bitmap = BitmapFactory.decodeStream(
            input);
        return bitmap;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    } };
```

Listing 2: Detected (true positive) clone for Listing 1

Conversely, Listing 4 depicts an example of a clone detected by ADANA for the snippet in Listing 3, classified by all the evaluators as false positive.

```
btn.setTag(textView.getText().toString());
btn.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      // TODO Auto-generated method stub
      String s =(String)v.getTag();
   } });
```

Listing 3: Code snippet declaring a listener for a button

```
textViewField.setOnLongClickListener(new
     OnLongClickListener() {
  @Override
     public boolean onLongClick(View v) {
        // TODO Auto-generated method stub
        return false;
     } });
```

Listing 4: Detected (false positive) clone for Listing 3

The snippet in Listing 3 (i) declares a click listener for a button and the corresponding `onClick` method, and (ii) shows how to set and get the button tag. Instead, the clone (Listing 4) implements a long click listener for a text view. This false positive is due to the extremely high VSM between the candidate clone and the snippet (*i.e.,* 0.79). Such a high value is due to several shared terms between the two snippets (*e.g.,* click, listener, text, view, auto, generated *etc.*). Some of these terms are due to the comment automatically generated by the IDE (*i.e.,* TODO [...]). These comments should be removed by matching them with regular expressions before computing the clones' similarity. This is something we implemented in ADANA after the results of this study and thanks to this example.

Another example of candidate clone classified by all participants as false positive is related to a code snippet which converts pixels to DIPs (Density Independent Pixels) for which ASIA identified a clone doing the opposite (*i.e.,* converting DIPs to pixels). Also, the identification of this false positive clone is partially due to the high VSM similarity between the candidate clone and the snippet, due to several terms shared between the two methods, as well as to the co-usage of Android constants such as `DisplayMetrics.DENSITY_DEFAULT`, needed in both methods.

Finally, for 35 clones the participants did not reach an agreement (*i.e.,* the answers were distributed equally towards true and false positives). For example, we had a snippet showing how to create a context menu and add items to it and a clone identified by ASIA for it implementing the same feature. However, in the snippet the items were statically added using String variables (*e.g.,* `menu.add`("Option1")), while in the clone the menu items were added dynamically according to an item selected on a list view. While at high-level the snippet and the clone implement the same feature (*i.e.,* create a context menu), the implementation differs in how the menu is populated.

Participants did not reach an agreement also for the code shown in Listing 6, reporting a clone detected by ASIA for Listing 5. While the two code snippets focus on very similar tasks, *i.e.,* writing/reading into/from `SharedPreferences`, the exact actions they perform on the `SharedPreferences` object are different. Three out of the six participants involved in the assessment of this clone marked it as a false positive.

```
SharedPreferences pref = PreferenceManager.
     getDefaultSharedPreferences(YourActivityName.
     this);
Editor edit1 = remembermepref.edit();
edit1.putInt(totalbalance_key,totalBalance);
edit1.commit();;
SharedPreferences pref = PreferenceManager.
     getDefaultSharedPreferences(YourActivityName.
     this);
int totalbalance = pref.getInt(totalbalance_key);
```

Listing 5: Code snippet writing/reading an int value into/from shared SharedPreferences

```
PreferenceManager.getDefaultSharedPreferences(this).
     edit().putInt(your_key, <Your_value>).commit();;
PreferenceManager.getDefaultSharedPreferences(this).
     getInt(your_key, <Default_value>);
```

Listing 6: Clone detected for Listing 5

Note that in this study we did not consider the false negatives (*i.e.,* clones of the considered code snippets that were present in our knowledge base but were not retrieved by ADANA) and, as a consequence, the ADANA's recall for two reasons. First, given a code snippet, it is practically impossible to manually identify all its clones in a database of 64k snippets. Thus, without having a complete oracle, it is not possible to identify false negatives. Second, considering the goal of our approach (*i.e.,* identifying clones to "reuse" code descriptions), what we care about is that the clones identified by ADANA are true positives, to avoid the injection of wrong comments in the code to document.

In summary, ADANA is able to detect clones for Android code snippets with a precision of $\sim 70\%$, by relying on lightweight textual analysis. Further work should be devoted to improving the precision of ADANA with static analysis techniques that might resolve the issues in the aforementioned examples.

4.3 Does ADANA help developers during code comprehension activities performed on Android apps?

Figure 5 reports the understandability (*i.e.,* correctness achieved in the verification questions—left side) and the TAU (*i.e.,* Timed Actual Understandability, taking both correctness and time needed for the comprehension into account—right side) achieved by developers when comprehending methods commented (*adana* group) and not commented (*uncommented* group) by our approach.

Participants obtained a better understanding of the method under analysis when comments injected by ADANA were present. Indeed, the average understandability was 0.92 (median = 1.00) in the *adana* group, and 0.77 (median = 0.83) in the *uncommented* group. This difference is statistically significant (p-value=0.03) with a medium effect size (d=-0.33).

The difference is even more marked when assessing the comprehension level by also considering the time participants spent understanding the methods and answering the verification questions. The average TAU is 0.66 in the *adana*

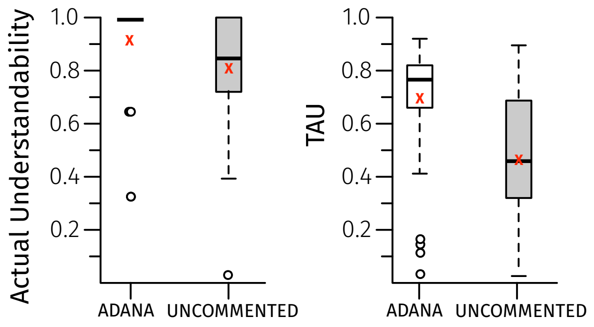


Fig. 5: Participants' understandability for methods commented (*adana*) and not (*uncommented*) by ADANA. The red x represents the average of distribution, while the circles show the outliers.

group (median=0.75), and 0.45 in the *uncommented* group (median=0.44). Such a strong difference is due in part to the higher understandability achieved by participants thanks to the comments injected by ADANA, but mostly to the time developers saved in comprehending the methods when working in the *adana* group. Indeed, on average, participants spent 99 seconds (median=87) per method when comments by ADANA were present as compared to 140 (median=126) when they were not present. Also, in this case, the difference is statistically significant (p-value<0.01) but with a large effect size (d=-0.48).

While ∼100 seconds looks insufficient to comprehend a method, it is worth remembering that we also had in our sample methods composed of only 10 ELOC. Also, the participants were all professional developers with Android experience, and we asked them to perform the comprehension activity in the shortest time possible to make sure they did not stop while performing a task (thus, introducing bias in the collected data). Given the maximum comprehension time we registered for a single method (*i.e.,* 318 seconds), we are confident that the developers did their best to understand the code and answer our questions in the shortest time possible.

```
public void onTextChanged(CharSequence s, int start,
     int before, int count){
  // Shows a Clear button after the first character
      pressed and hides it when the text is empty
  if(s.length() > 0){
    clear_button.setVisibility(View.VISIBLE);
    searchText = s.toString();
  }
}
```

Listing 7: Example of useful injected comment

Listing 7 shows (part of) one of the methods the developers were asked to understand. The comment in the method is automatically injected by ADANA and helped the participants in quickly understanding under which circumstances the clear button is visible on the screen (*When is the clear_button shown to the user?* was the first question we asked about this code snippet). While developers were able to fully comprehend and correctly answer all verification questions for this code snippet both with and without the injected comments, they saved, on average, two minutes of comprehension activity thanks to the ADANA comments.

```
// Resizes image before decoding it to bitmaps.
float widthRatio = ((float) rotatedWidth) / ((float)
     MAX_IMAGE_DIMENSION);
float heightRatio = ((float) rotatedHeight) / ((
    float) MAX_IMAGE_DIMENSION);
float maxRatio = Math.max(widthRatio, heightRatio);
BitmapFactory.Options options = new BitmapFactory.
    Options();
options.inSampleSize = (int) maxRatio;
srcBitmap = BitmapFactory.decodeStream(is, null,
    options);
[...]
// Rotates a bitmap.
Matrix matrix = new Matrix();
matrix.postRotate(orientation);
srcBitmap = Bitmap.createBitmap(srcBitmap, 0, 0,
    srcBitmap.getWidth(), srcBitmap.getHeight(),
    matrix, true);
```

Listing 8: Example of useful injected comment

Listing 8 reports another example of correct and useful comments injected by ADANA. In particular, Listing 8 shows two parts of a method subject of our study for which ADANA injected two comments (*i.e.,* "`//Resizes image before decoding it to bitmaps`" and "`//Rotates a bitmap`") both correctly describing the method behavior, helping participants to increase their average actual understandability (AU) by 22%, while still saving, on average, one minute of comprehension activity.

Interesting insights were also provided by participants when answering to the last open question in which we revealed that the comments were automatically injected and asked participants to provide their thoughts. One of the participants wrote "*comments were useful to comprehend at least parts of the snippets*", confirming the potential usefulness of ADANA. Another one said "*I noticed one case in which the comment was partially wrong, since it referred to loading JSON from a webpage, while the method was parsing the webpage but not as a JSON, other comments were good*". The developer is referring to a method in which ADANA injected this comment "`//JSON parser from web page`" in a method that stored the output of an HTTP request into a String for further analysis.

This clearly represents a case of "false positive" comment. We did not observe any strong impact of this comment on the participants' performance, likely due to the fact that it was compensated by the still useful context hint (*i.e.,* parsing a web page). A case in which the participants performed equally both with and without the comments injected by ADANA is represented by code snippet #8 (see replication package [29]). This case is interesting since, despite the fact that the injected comment ("`//Reads Distinct Contacts with Contact Number and Names`") correctly describes the snippet, it did not benefit the correctness achieved by participants nor the time they spent comprehending the code. Our conjecture is that the complexity of the code snippet, including two `while` loops, three `if` statements, and one `switch-case` statement, probably pushed the developers to carefully inspect the whole code in both scenarios, thus reaching a similar comprehension level in roughly the same amount of time ($\sim$200 seconds) when working with the two treatments.

Summarizing, ADANA seems to help developers in code comprehension activities performed on originally *uncommented* code snippets. This especially results in time saved for the code comprehension.

5 Threats to Validity

Construct validity. In RQ$_1$ we mimic developers selecting snippets of code to assess the ADANA code coverage. While we experimented with code snippets of different length, our simulation might not be realistic of typical snippets selected by developers. Also, we considered the "coverage" of all apps' ELOC as equally important, which is questionable (*e.g.*, the code implementing the application logic is likely the one most important to document).

Internal validity. To avoid bias in the experiments performed to answer RQ$_2$ and RQ$_3$, we made sure that participants were neither aware of the investigated research questions nor of the general goal of the tasks we required them to perform. Also, we decided not to include in our studies participants without Android experience to have a more homogeneous population and to avoid a strong influence of the participants' knowledge/skills as a confounding factor. Finally, we made sure to have multiple evaluators for each candidate clone (RQ$_2$) and for each method participants had to comprehend (RQ$_3$).

The three requirements used in the *quality checker* to exclude $\langle code, description \rangle$ pairs likely having a low quality have been defined by the first author, thus introducing possibly subjectivity bias. However, the requirements he defined have been discussed among all authors, also looking at the pairs discarded thanks to their application.

Also, in RQ$_3$ we limit our analysis to methods having a size between 10 and 50 ELOC, with the goal of excluding from our study methods that were too trivial or too complex to understand. However, these thresholds have been defined based on the authors' development experience, and experimenting with methods of different size could lead to different findings.

Conclusion validity. We address threats to conclusion validity by using appropriate statistical tests and effect size measures to support our claims. While we observed a positive effect of ADANA on code comprehension activities performed on *uncommented* code snippets, we are not claiming its usefulness in a scenario in which the code is commented, since a different study design would be needed to assess this.

External validity. The generality of our results is bounded by the limited number of apps (RQ$_1$), snippets (RQ$_2$), and methods (RQ$_3$) used in our study, as well as by the number of participants (RQ$_2$ and RQ$_3$). For example, it is possible that the coverage level observed on the apps selected for RQ$_1$ does not generalize to other apps.

6 Related Work

ADANA is related to other approaches for automatic documentation of software artifacts and mining of source code snippets/examples with recommendation purposes. Previous automatic documentation approaches rely on summarization techniques [14], [41], stereotypes [42], [43], [44], and mining of repositories with developers communications and Q&A Websites [45], [46]. These approaches automatically generate documentation at different granularities and for software artifacts such as unit test cases [13], [44], [47], [48], changes/commits [9], [49], [50], [51], database usages [12], release notes [52], [53], classes [6], [54], methods [11], [55], [56], code fragments [7], [10], [57], bug reports [58], software concerns [59], and mailing lists [60].

The closest work to ADANA, in terms of granularity, is the one by Ying *et al.* [7], [10], which also focus on automatic documentation of source code snippets. Ying *et al.* explored the usage of machine learning for selecting lines in a code fragment that should be in an extractive summary. As opposed to that, ADANA "freely" documents a snippet with descriptions mined from the Web. Wong *et al.* [57] also exploits the use of existing code comments for automated code comments generation with an approach called ColCom. It is not limited to Android, but it only works with type-1 and 2 clones found in a limited set of projects given as an input of the approach. Unlike ADANA, ColCom does not rely on a central persistent database which can improve over time and its performance depends on the input projects. Moreover, since the code comments are extracted from source code, ColCom relies on a set of heuristics to identify the code comments associated to a code snippet.

Automated extraction/recommendation of API usage examples is also related to ADANA. Such approaches [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71] mine software repositories to find representative API usage examples for assisting developers when trying to use a class or method, or when finding code examples showing how to implement a given task/feature. For instance, *MUSE* by Moreno *et al.* [66] uses static analysis techniques and clone detection methods for providing developers with usage examples for a given method. Compared to previous approaches, ADANA does not rely on source code repositories of software systems and does not generate code examples. Instead, it mines fragments that are in SO discussions and GitHub Gists to inject code comments in a given snippet. ADANA is therefore complementary to approaches like *MUSE*.

Finally, automatic documentation of source code based on examples from the crowd has been explored by Vassallo *et al.* [46] and Rahman *et al.* [45]. Both these works mine SO discussions to re-document a Java software system but at different granularities: method and code fragment respectively. The approach by Vassallo *et al.* searches on Stack Overflow for sentences related to the method the developer is interested in documenting. For example, if the developer is working on the Apache Lucene project (one of the two used in the evaluation), and she wants to document a method implemented in a class, the approach searches in SO using as search keys project name + class name + method name. This means that this approach can only support the documentation of very well known systems widely discussed on SO, while it cannot support the documentation of unknown software projects, like mobile apps still to be published on the Google Play store. ADANA can instead document, in part (see RQ$_1$ results), any Android mobile app. Concerning the work by Rahman *et al.* [45], the authors present CodeInsight, a tool pioneering the mining of insightful comments about a snippet of code from SO. CodeInsight looks for comments discussing bugs or improvement tips for code, while ADANA retrieves complementary information, mined from multiple sources, aimed at explaining what a snippet of code does.

7 Conclusion

We presented ADANA, an online-resources-mining approach and a tool to collect $\langle code, description \rangle$ pairs that can be reused to automatically document similar pieces of code — given a target snippet to inspect —, which are identified by using the ASIA clone detector we devised. ADANA is currently tailored to work on Android apps but could be adapted/extended to support the documentation of any software system. For example, assuming the will to extend the ADANA support to C++ systems, this would mostly require extensions to the (i) clone detector, by adding a detector designed to work on C++ code, and (ii) knowledge base, mining C++ code snippets.

While the results achieved in the performed evaluation are already encouraging, we believe that the strength of ADANA lies in the always increasing amount of data that it will be able to exploit in the mined online resources, making ADANA better and better over time.

Future work will be devoted to enlarging the ADANA knowledge base. This will be mainly due by defining techniques able to identify well-commented code snippets in open source software repositories (*e.g.,* by mining the change history to identify commits in which a snippet of code and its comments are added to the system), thus dramatically increasing the amount of information in our knowledge base. Our long-term vision is that of a tool able to exploit as much crowdsourced knowledge as possible to automatically document software systems. Also, we want to enlarge the ADANA support to Java applications in general. Finally, we are planning larger, more robust evaluations, and in particular, controlled experiments/case studies aimed at deeply investigating the usefulness of our approach during code comprehension activities.

Acknowledgments

The authors would like to thank the participants involved in the reported empirical studies. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project PROBE (SNF Project No. 172799).

References

[1] D. Spinellis, “Code documentation,” *IEEE Software*, vol. 27, no. 4, pp. 18–19, July 2010.

[2] B. Fluri, M. Wursch, and H. C. Gall, “Do code and comments co-evolve? on the relation between source code and comment changes,” in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

[3] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, “Analyzing the co-evolution of comments and source code,” *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s11219-009-9075-x

[4] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, “How do developers document database usages in source code?” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 36–41.

[5] T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: a study of developer work habits,” in *Proceedings of ICSE 2006*. ACM, 2006, pp. 492–501.

[6] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, “Automatic generation of natural language summaries for java classes,” in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 23–32.

[7] A. T. T. Ying and M. P. Robillard, “Code fragment summarization,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 655–658. [Online]. Available: http://doi.acm.org/10.1145/2491411.2494587

[8] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, “Autofolding for source code summarization,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1095–1109, 2017. [Online]. Available: https://arxiv.org/abs/1403.4503v5

[9] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, “On automatically generating commit messages via summarization of source code changes,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Sept 2014, pp. 275–284.

[10] A. T. T. Ying and M. P. Robillard, “Selection and presentation practices for code example summarization,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 460–471. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635877

[11] P. W. McBurney and C. McMillan, “Automatic source code summarization of context for java methods,” *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, Feb 2016.

[12] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, “Documenting database usages and schema constraints in database-centric applications,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 270–281. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931072

[13] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, “Automatically documenting unit test cases,” in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 341–352.

[14] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, “On the use of automated text summarization techniques for summarizing source code,” in *2010 17th Working Conference on Reverse Engineering*, Oct 2010, pp. 35–44.

[15] “https://gist.github.com,” 2017.

[16] “https://stackoverflow.com,” 2017.

[17] “https://gist.github.com/MBtech/37f2f3df5dfe5805adfd,” 2017.

[18] “https://developer.android.com/,” 2017.

[19] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, “The stanford corenlp natural language processing toolkit,” in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL’14)*, 2014, pp. 55–60.

[20] R. Koschke, “Survey of research on software clones,” in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/962

[21] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[22] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in *29th International Conference on Software Engineering (ICSE’07)*, 2007, pp. 96–105.

[24] S. Harris, “http://www.harukizaemon.com/simian/,” 2003.

[25] I. Mojica Ruiz, M. Nagappan, B. Adams, and A. Hassan, “Understanding reuse in the Android market,” in *20th IEEE International Conference on Program Comprehension (ICPC’12)*, 2012, pp. 113–122.

[26] R. Minelli and M. Lanza, “Software analytics for mobile applications – insights & lessons learned,” in *17th European Conference on Software Maintenance and Reengineering*, 2013, p. To appear.

[27] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, “Revisiting android reuse studies in the context of code obfuscation and library usages,” in *Working Conference on Mining Software Repositories (MSR’14)*, 2014, pp. 242–251.

[28] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[29] "Replication package for "automatic documentation of android apps". http://adana.si.usi.ch/," 2018.

[30] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[31] "https://developer.android.com/reference/packages.html," 2017.

[32] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[33] R. Flesch, "A new readability yardstick." *Journal of applied psychology*, vol. 32, no. 3, p. 221, 1948.

[34] "https://github.com/pcqpcq/open-source-android-apps," 2017.

[35] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sept 2016, pp. 87–98.

[36] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?" in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Press.

[37] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[38] "https://developer.android.com/samples," 2017.

[39] "https://github.com/romannurik/FORMWatchFace," 2017.

[40] "https://github.com/ashutoshgngwr/10-bitClockWidget," 2017.

[41] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568247

[42] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse engineering method stereotypes," in *2006 22nd IEEE International Conference on Software Maintenance*, Sept 2006, pp. 24–34.

[43] ——, "Automatic identification of class stereotypes," in *2010 IEEE International Conference on Software Maintenance*, Sept 2010, pp. 1–10.

[44] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Aiding comprehension of unit test cases and test suites with stereotype-based tagging," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. ACM, 2018, pp. 52–63.

[45] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2015, pp. 81–90.

[46] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "Codes: Mining source code descriptions from developers discussions," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 106–109.

[47] M. Kamimura and G. C. Murphy, "Towards generating human-oriented summaries of unit test cases," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 215–218.

[48] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 547–558.

[49] M. Kim, D. Notkin, D. Grossman, and G. Wilson, "Identifying and summarizing systematic code changes via rule inference," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 45–62, Jan 2013.

[50] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 709–712.

[51] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 320–323.

[52] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 484–495.

[53] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora, "Arena: An approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, Feb 2017.

[54] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 230–232.

[55] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859006

[56] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597149

[57] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 380–389.

[58] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, April 2014.

[59] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2011, pp. 103–112.

[60] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, pp. 63–72.

[61] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 643–652. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568313

[62] S. Jiang, A. Armaly, C. McMillan, Q. Zhi, and R. Metoyer, "Docio: Documenting api input/output examples," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 364–367.

[63] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 392–403.

[64] M. P. Robillard and Y. B. Chhetri, "Recommending reference api documentation," *Empirical Softw. Engg.*, vol. 20, no. 6, pp. 1558–1586, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-014-9323-y

[65] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 782–792.

[66] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 880–890.

[67] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 117–125.

[68] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 157–166. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882316

[69] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, Oct. 2013.

[70] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 102–111.

[71] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 94–105.

Emad Aghajani is a Ph.D. student in the Faculty of Informatics at the Universitá della Svizzera italiana (USI), Switzerland. He received his M.S. in Software Engineering from Sharif University of Technology, Iran. His research interests lie in the field of software evolution, software maintenance, and mining software repositories.

Gabriele Bavota is an Assistant Professor at the Universitá della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is the author of over 100 papers appeared in international journals, conferences and workshops. He received four ACM SIGSOFT Distinguished Paper awards at ASE 2013, ESEC-FSE 2015, ICSE 2015, and ASE 2017, the best paper award at SCAM 2012, and three distinguished reviewer awards at WCRE 2012, SANER 2015, and MSR 2015. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, FSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, and others.

Mario Linares-Vásquez is an Assistant Professor at Universidad de los Andes in Colombia, where he leads The Software Design Lab. He received his Ph.D. degree in Computer Science from the College of William and Mary in 2016. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. His research interests include software evolution and maintenance, software architecture, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks, and mobile development.

Michele Lanza is a full professor in the Faculty of Informatics at the Universitá della Svizzera italiana (USI), where he founded the REVEAL research group in 2004. He co-authored over 150 journal and conference publications and the book Object-Oriented Metrics in Practice. His activities span various international software engineering research communities. He has served on the program committees of ICSE, FSE, ICSME, ICPC, MSR and many other conferences, and as program co- chair of ICSM 2010, VISSOFT 2009, MSR 2008, IWPSE 2007, and MSR 2007. He was keynote speaker at MSR 2010, CBSOFT/SBES 2011, BENEVOL 2011, CSMR 2013, and SCAM 2016. He is a board member of CHOOSE (Swiss Group for Object-Oriented Systems and Environments), and vice-president of the Moose association.