CLEI ELECTRONIC JOURNAL, VOLUME 15, NUMBER 2, PAPER 6, AUGUST 2012

On the Analysis of Human and Automatic Summaries of Source Code

Laura Moreno
Universidad Nacional de Colombia, Departamento de Ingeniería de Sistemas e Industrial,
Bogotá, Colombia
lvmorenoc@unal.edu.co

and

Jairo Aponte
Universidad Nacional de Colombia, Departamento de Ingeniería de Sistemas e Industrial,
Bogotá, Colombia
jhapontem@unal.edu.co

Abstract

Within the software engineering field, researchers have investigated whether it is possible and useful to summarize software artifacts, in order to provide developers with concise representations of the content of the original artifacts. As an initial step towards automatic summarization of source code, we conducted an empirical study where a group of Java developers provided manually written summaries for a variety of source code elements. Such summaries were analyzed and used to evaluate some summarization techniques based on Text Retrieval.
This paper describes what are the main features of the summaries written by developers, what kind of information should be (ideally) included in automatically generated summaries, and the internal quality of the summaries generated by some automatic methods.

Keywords: Software comprehension, source code, summarization, text retrieval.

1 Introduction

As a broad concept, summarization is the process of reducing large volumes of information in entities like texts, speeches, or films, to short abstracts comprising the main points or the gist in a concise form [1]. Currently, one of the most promising applications of summarization is its use as a complement or a second level of abstraction for text retrieval tools, since they often return a large number of documents that overwhelm their users. For instance, automated summarizing tools are needed by internet users who would like to utilize summaries as an instrument for knowing the structure or content of the returned documents in advance, and eventually, be able to effectively filter out irrelevant results.

Recently, software engineering researchers have begun to explore the use of summarization technologies, mainly as a potential instrument for supporting software comprehension tasks. These tasks are key developer activities during software evolution, accounting for more than half of the time spent on software maintenance [2]. In the case of source code artifacts, developers are often faced with software systems with thousands or millions of lines of code, and before attempting any changes to those systems, they must find and understand some specific parts of them. We argue that offering developers descriptions of source code entities can reduce the time and effort needed to browse files, locate and understand the part of the system that they need to modify. Ideally, such summaries will be informative enough to be used for filtering irrelevant artifacts, and even, as substitutes of the detailed reading of full artifacts. Even when they do not convey enough information to replace the originals, they could be useful *indicative summaries*, a type of abstracts built with the purpose of indicating to the user which documents would be worth studying in more detail. In the worst scenario, developers would have to read the summary and the original artifact. Even in this case, this extra reading would be helpful, since the summary can provide a preview of the original document (e.g., its structure or an initial idea of its content).

The major challenge in automatic software summarization is to handle mixed software artifacts such as source code, where information is encoded in a different way than in natural text documents. One key issue that we need to address is determining what is relevant in source code documents, and therefore, should be included in the summaries. The answer may be different for various types of source code entities (e.g., class vs. method) [3], and also may differ between programming languages. Some feasible ways to address this issue are (1) to study how developers create summaries of source code artifacts, (2) to analyze the summaries generated by them, and (3) to use their expertise for determining what information should be included in the summaries of source code artifacts.

A second aspect of software summarization research is the evaluation of the quality and usefulness of automatically generated summaries. It is essential to know if the generated abstracts are useful supporting development or maintenance tasks and how this positive effect can be assessed. Moreover, some metrics are required to determine if the summaries convey the most relevant information in the original artifact. As a result, there have been considered two broad types of evaluation: *extrinsic* and *intrinsic* evaluation [4]. The former one aims at determining whether the summaries are good instruments to support real-user's work, whereas the latter one measures internal properties of the abstracts such as semantic informativeness, coherence, and redundancy.

From a research standpoint, intrinsic evaluation is important because it allows us to assess the results of a summarization system, compare the results of different approaches, and identify and understand the drawbacks of a particular summarization procedure. In this kind of evaluation, the quality of a summary can be established mainly through two approaches. In the first one, the *peer summary* (i.e. the summary being evaluated) is reviewed and rated by human judges, using some pre-established guidelines. The second alternative is to measure the similarity between the peer summary and some reference abstract given by experts, which is often called the *gold standard summary.*

In this paper, we present the results of an empirical study where a group of developers (1) generated two types of manually-written summaries for various kinds of source code artifacts, and (2) answered questions about what they think should be included in a summary. Additionally, we propose the use of these human-generated summaries, and some well-known text retrieval measures to carry out an intrinsic evaluation of several automatic summarization approaches.

The remaining of the paper is organized as follows. Section 2 presents the problem and research questions we want to answer. The details of the conducted empirical study are described in Section 3. Section 4 sums up the more important results of the study, and discusses possible explanations and their implications for automatic summarization of source code. Section 5 discusses related work, and Section 6 draws some conclusions and remarks regarding extensions for this research.

2 Definition and Research Questions of the Case Study

The goal of this experiment was to analyze code summaries written by developers, and use these summaries as a test-bed to carry out a comparative evaluation of Text Retrieval (TR) techniques, when they are used as automatic code summarizers. The quality focus was on improving tool support for software comprehension tasks, as well as, providing a stable evaluation framework to measure whether automatic generated summaries convey the most relevant information in the original software artifacts. The perspective was of researchers who need to gain insight into (1) how developers analyze and summarize various kinds of source code entities; (2) what structural elements they consider should be included in a summary; and (3) how intrinsic summary evaluation methods can be used to evaluate automatic code summarization approaches.

Therefore, within this case study the following research questions were formulated:

RQ1 How long are the summaries generated by developers?

RQ2 What type of structural information do developers include in their summaries?

RQ3 What are the main characteristics of the terms most selected by developers? Can they be considered as gold-standard summaries for generating and evaluating automatic summarization tools?

RQ4 How good are text retrieval techniques as automatic summarizers of source code artifacts? How much do these automatic summaries resemble the human-generated ones?

Answers to RQ1 and RQ2 will give us valuable information about the data that should be (ideally) included in automatically generated summaries, and how they can be created. RQ3 and RQ4 aim at exploring TR techniques as code summarizers, and intrinsic evaluation methods as suitable approaches to measure the quality of their outcomes.

3 Context of the Case Study

This section begins with a description of the resources selected to perform the empirical study, i.e., the system and the participants. Then, it presents the layout of the experiment.

3.1 Objects

The system selected to carry out the experiment was aTunes, an open source project that manages and plays audio files. It is a small-medium sized Java system whose application domain is easy to understand and often interesting for almost any developer. Table 1 sums up the main features of the selected version.

Table 1: Main characteristics of aTunes system

Feature	Description
Domain	Media player and manager
Version	1.6.0
Number of classes	221
Number of methods	1852
Number of non-blank lines of text	25000 (approximately 5000 lines of comments)
Comments existence and quality	Contains few leading comments
Identifiers quality	Full-length, descriptive identifiers
Codification style	CamelCase
Language of identifiers, comments and documentation	English

Since we are interested in analyzing how various types of source code entities are summarized, and we think the content and structure of summaries are affected by the size and type of the summarized artifact, we selected two methods, two classes, two groups of methods (each group of methods consists of three methods, which, as a calling sequence, implement a specific feature of the system) and one package.

We focused on entities dealing with the business logic (not GUI or data layer classes), and excluded too short entities, such as methods with less than 10 lines, classes with less than 3 attributes or a few number of short methods, and packages with only one class. Table 2 shows the basic features of the selected artifacts. It is important to point out that the aTunes version selected for the study contains very few heading comments; in fact, the artifacts used in the study do not contain such comments, but they do contain some inline comments.

Table 2: Features Of The Artifacts Selected From aTunes System

Artifact	Abv.	Features
Method 1	M1	42 LOC , 3 parameters
Method 2	M2	40 LOC, 1 parameter
Method sequence 1	S1	Three methods A, B and C, where A calls B, and B calls C. They are in the same class
Method sequence 2	S2	Three methods A, B and C, where A calls B, and B calls C. A and B are in the same class, and C belongs to a different class
Class 1	C1	6 attributes, 10 methods
Class 2	C2	4 attributes, 5 methods
Package 1	P1	3 classes

LOC Lines of Code

3.2 Subjects

The subjects of this study were twelve graduate and undergraduate students from the Computer Science Department at Universidad Nacional de Colombia. Senior undergraduate students were recruited mainly from two courses: Software Engineering and Software Architecture; both courses are part of the second half of the undergraduate degree program in Computer Science. Only two of the participants were master students who were also working as professional Java developers. We excluded from the analysis the subjects who did not complete the tasks, and those who made mistakes due to the misunderstanding of experiment instructions. This way, the conducted analysis is based on the responses of only nine of the subjects.

On average, the subjects had 4.6 years of programming experience. Regarding their knowledge of Java programming language, they reported an experience of 2.2 years on average, and considered their skills as satisfactory, good or very good, in all the cases. Just two of the subjects evaluated their experience in understanding and evolving systems as less than satisfactory. Since the experiment was carried out using the Eclipse IDE, we asked them about their abilities with this programming environment. Three of them considered their expertise with this particular IDE as poor or very poor. However, we think this issue did not have a major impact in the experiment's results because they used only searching and browsing commands, and they also mentioned extensive experience with NetBeans, a similar programming environment. Finally, subjects assessed their English proficiency at least as satisfactory, in all cases.

3.3 Experiment layout

As a preliminary experiment setup, Eclipse and aTunes were installed on each computer used in the experiment, and the following documents were created:

- A description of the main functionality of aTunes that was used as an introduction to the system.
- Slides for explaining the tasks to do within each stage of the experiment.
- A form to collect information related to the programming experience and skills of each participant.
- Forms to collect the summaries of each source code artifact.
- Forms to collect feedback at the end of each session of the experiment.

Participants attended three sessions; each one lasted around 2 hours. All sessions began with a brief explanation of the tasks to do. In particular, a general explanation of the whole experiment was given at the beginning of the initial session.

The first session was a training session, where participants filled out an individual form about their English and programming skills. After that, a 10-minutes presentation of the software system was given, which included a demo of its most important functionality. The remaining time was spent by the subjects reading documentation of the system, and getting familiar with the organization of the source code.

During the second session, subjects generated English sentence-based summaries for each artifact using the forms we prepared for this task. A *sentence-based* summary is an unrestricted natural language description of the software entity (*abstractive summary*), and within this experiment, it was used as a sanity-check instrument, i.e., a basic test to quickly evaluate that the answers of a participant did not contain elementary mistakes or impossibilities, or were not based on invalid assumptions. Each subject finished the session answering the post-experiment questionnaire regarding the tasks done and the kind of analysis performed.

During the last session, participants summarized the artifacts in Spanish and then, they created term-based summaries. A *term-based summary* is a set of unique words or identifiers selected from the source code of the software entity (*extractive summary*). They selected relevant terms for each artifact, by enclosing them within source code files, using a set of predefined tags. Finally, each subject answered the post-experiment questionnaire regarding the tasks done and the usefulness of various parts of the code when doing term-based summarization.

4 Results and Discussion

4.1 Brief description of artifacts' content

As stated in section 3.1, four types of artifacts were used in the experiment. Although all these entities are composed by a great number of identifiers and keywords, just some of them are useful to describe the content of the artifact. Actually, within a source code unit some terms are repeated constantly in different identifiers. For example, the method M1 is formed by 231 terms, but only 70 of those are unique. The same information for the other artifacts is presented in Table 3. The length in this case refers to the amount of terms in the artifact, including keywords and identifiers, after a process of splitting. For instance, the method name *timeInSeconds* is transformed into the terms *time In Seconds*. As another example, the variable name *DEFAULT_LANGUAGE_FILE*, is split in *DEFAULT LANGUAGE FILE*. The rationale behind splitting is that we want to know which is the exact vocabulary used within each artifact.

Table 3: Identifiers and unique terms of each selected artifact sorted by length

Artifact	Length (L)	Unique terms (UT)	UT : L
M2	219	82	0.37
M1	231	70	0.30
S2	308	81	0.26
S1	424	97	0.23
C2	426	104	0.24
C1	685	110	0.16
P1	791	129	0.16

At first glance, it can be observed that the ratio between unique terms and length is very low. Furthermore, this ratio decreases as the artifact increases its size. In detail, the ratio is higher for methods possibly because they encapsulate functions over specific objects. Sequences also perform specific functions but according to the classes which their methods belong to, the ratio of unique identifiers can increase: if the methods belong to the same class the ratio is lower, otherwise the ratio is higher. A possible explanation to this situation is that the greater amount of involved classes in an artifact, the more themes are touched by it, and by extension, there are more unique terms in its source code.

4.2 General description of developers' summaries

In this phase of the study, we focused the analysis on summaries written in English, in order to compare and contrast them with term-based summaries and also with the declaration of the artifacts, both of these written in English. Once again, the length of the summary is calculated as the amount of split terms it contains.

Regarding the sentence-based summaries, there is no significant relationship between their size and the length or type of artifact they describe (p-value for Pearson's correlation = 0.359). Even the average of unique terms in this kind of summaries is almost the same for all type of artifacts (Table 4). In the case of sequences, however, the length of the summaries is slightly greater, as well as their number of unique terms; this indicates that sequences are harder to describe or deserve more detailed descriptions.

Table 4: Length properties of sentence- and term-based summaries by artifact

	Sentence-based summaries			Term-based summaries		
Artifact	$\overline{L}$	$\overline{UT}$	$\overline{UT} : \overline{L}$	$\overline{L}$	$\overline{UT}$	$\overline{UT} : \overline{L}$
M1	37.22	26.67	0.38	10.89	7.78	0.11
M2	33.44	23.56	0.29	12.00	10.22	0.12
S1	40.67	28.00	0.29	19.22	14.00	0.14
S2	49.89	32.00	0.40	15.33	11.89	0.15
C1	32.00	23.22	0.21	27.56	18.56	0.17
C2	30.78	22.89	0.22	14.78	10.44	0.10
P1	34.33	25.22	0.20	22.00	12.89	0.10

$\overline{L}$	Average length
$\overline{UT}$	Average of unique terms

On the other hand, for term-based summaries we found direct relationships with respect to lengths (p-value for Pearson's correlation < 0.01). Specifically, developers tend to mark more terms when they are analyzing packages, fewer terms when the artifacts are classes or sequences, and even fewer terms when they are dealing with methods, as shown in Table 4. This situation indicates that all kind of artifacts cannot be summarized with the same fixed number of terms: as granularity level increases, the amount of terms needed to describe an artifact decreases. The high standard-deviation values obtained in this case indicate that developers hardly ever mark similar number of terms.

4.3 Origin of relevant terms

In a post-experiment questionnaire, we asked the developers about the usefulness of structural information from source code when doing term-based summarization. To that end, the participants rated different locations of source code (e.g. class name, attribute type, attribute name, etc.) on a 1-to-4 Likert scale [5],

Table 5: Percentage of terms' origins marked by developers

Methods		Sequences		Classes		Packages	
Origin of term	%	Origin of term	%	Origin of term	%	Origin of term	%
Var. name	31.07	Call name	40.13	Method name	26.51	Method name	24.24
Method name	21.84	Var. name	26.54	Call name	19.16	Attrib. name	22.73
Call name	13.59	Attrib. name	11.33	Var. name	18.64	Var. name	20.20
Par. name	9.22	Method name	7.77	Attrib. name	8.92	Class Name	16.16
Comment	8.25	Literal text	6.80	Class Name	7.61	Par. name	5.05
Var. type	7.77	Var. type	2.91	Attrib. type	6.56	Package name	4.04
Attrib. name	5.34	Par. name	2.91	Par. name	6.56	Attrib. type	1.52
Literal text	2.43	Class Name	1.29	Literal text	2.62	Call name	1.52
Return type	0.49	Attrib. type	0.00	Package name	1.31	Comment	1.01
Attrib. type	0.00	Comment	0.00	Comment	0.79	Var. type	1.01
Class name	0.00	Par. type	0.00	Var. type	0.26	Par. type	1.01
Par. type	0.00	Return type	0.00	Par. type	0.00	Return type	1.01
Package name	0.00	Package name	0.00	Return type	0.00	Literal text	0.51

where *1* represented *totally useless* and *4* represented *very useful*. We did not used the 5-level scale in order to avoid the tendency to mark non-committal answers (i.e., neither useful nor useless).

Not surprisingly, through the questionnaire we found that when summarizing methods, classes and packages, their respective names were considered as the most useful parts of code. Other locations equally useful when describing methods were invoked methods, which together with methods names, were assessed as very useful when dealing with classes. In the case of sequences (and similar to methods), the parts of code considered as most useful were the names of the invoked methods and the variables and parameters names. We also noticed two striking facts on the questionnaire:

1. Source code comments were not considered valuable, especially in packages and sequences cases, where they were rated as totally useless information.

2. Methods and classes names were considered useful when summarizing all four types of artifacts.

Furthermore, packages were recognized as more difficult to summarize than other artifacts, and only packages names, classes names and methods names were useful information, whereas the rest of the locations were marked as moderately or totally useless. This may be the cause of the tendency to mark a greater amount of terms within that type of artifacts, and additionally suggests that a multi-document approach, where each class of the package would be treated as an individual document, can be adequate for summarizing packages.

In order to go deeper into the origin of relevant terms and contrast the questionnaire answers, we identified where every marked term belonging to term-based summaries came from. These origins were classified in the same categories used in the questionnaire (Table 5). We discovered that developers constantly marked the local variable names when summarizing all artifacts, even for packages, where they had been classified as moderately useless information. We also noticed that terms from comments were hardly used when building summaries for sequences, classes and packages; this proves the uselessness of this location reported by developers. In the case of methods, however, terms in comments were even more frequently chosen than attributes names and parameter types, although these latter ones were ranked as very useful within this kind of artifact.

Additionally for methods summaries, the origin extraction confirmed the usefulness of method call names and method names given by developers to those parts of code; nevertheless, we observed that the names of their parameters were also frequently used when summarizing them. In contrast, well-ranked locations such as parameters types, classes names and attributes types, were not used at all when summarizing methods.

With regard to sequences, developers considered methods names, invoked methods names and local variables names as relevant, and actually, this was confirmed by the terms marked for their summaries. It is worth mentioning that although attributes types, parameters types and methods returns types were considered as useful in the questionnaire, they were never used when summarizing sequences.

In the case of classes summarization, developers rated their names as very useful information, which was confirmed by means of origins extraction. These same names were considered useful for all artifacts, but actually, they were rarely marked when summarizing methods and sequences.

Concerning packages, their names together with the names of classes and methods were constantly used to summarize this type of artifact, just as mentioned by developers in the questionnaire. Nonetheless, other locations such as the names of attributes, variables and parameters, which were ranked as useless by developers, were in fact often used in packages summaries.

Surprisingly, the type of variables, attributes and parameters were barely used in most of the cases, albeit they were considered useful when summarizing methods and sequences. Even so, sequences and classes kept the correlation between the scores given to parts of code by developers, and the real proportion of terms' locations in term-based summaries. This means that only such kinds of artifacts preserved a high coherence between developers' opinions about the usefulness of the locations, and their actual summarization choices.

Finally, additional categories not included in the questionnaire were used by developers, such as literal data allocated in string constants or systems logs, which were marked more times than other origins previously considered. For instance, the literal texts "Exporting process done" and "Exporting songs" were marked by some developers to describe the sequence S2.

4.4 Approximation to gold standard summaries

As shown in Table 6, from the set of unique terms in a sentence-based summary around 35% are provided by the declaration of the artifact, no matter its type. Apparently, this suggests that extractive approaches are not enough to generate the summaries automatically, given the great amount of new terms within the description provided by developers.

Nonetheless, the overlap between the term-based summaries (which are comprised exclusively by terms found in the declaration of the artifacts) and the terms in sentence-based summaries, reveals that relevant terms can be extracted from source code as the basis for a short description. Here, the relevancy of a term is defined by the amount of developers who chose it to be part of the summary, i.e., the agreement among subjects. Since in the experiment participated nine developers, the relevancy scale goes from 0 to 9, where 0 represents totally irrelevant and 9 means totally relevant.

Table 6: Average overlap between declaration, sentence- and term-based summaries by artifact

Artifact	% of unique terms in SBS $\bigcap$ AD	% of unique terms in TBS $\bigcap$ SBS	% of terms in SBS $\bigcap$ TBS with relevancy ≥ 5
M1	38.71	25.69	84.44
M2	31.27	19.62	72.96
S1	40.76	25.94	88.15
S2	35.35	53.20	80.65
C1	43.08	23.29	70.90
C2	42.28	32.40	64.44
P1	31.98	26.07	69.63

SBS	Sentence-based summaries
AD	Artifact declaration
TBS	Term-based summaries

For the terms in the last mentioned overlap, we found that about 75% of them were chosen by five or more developers. This percentage could increase if we take into account the use of synonyms in the free-form summaries; as a case in point, the term *encode* found in method M1, was replaced by words such as *transform* and *convert* in the sentence-based summaries. Thus, some text retrieval techniques might be suitable for identifying the most prominent terms within source code artifacts, as was proposed by [6].

In each single artifact, the relevant terms in the intersection between term- and sentence-based summaries could be considered as a *gold standard summary*, i.e., a reference or an ideal description that contains the important information of the entity under analysis. Nevertheless, the results show that the terms chosen by five or more developers in the term-based summaries form, in fact, a better approximation to those standard summaries. Some of their properties are presented in Table 7. It can be noticed that the length of these summaries depends on the type of artifact they describe, and also on the length of such artifact (p-value for Pearson's correlation < 0.01). Therefore, the gold summaries of packages and classes are larger than those that describe sequences and methods.

About the terms' origins, the proportions of term-based summaries remain stable for ideal summaries, with little exceptions. For example, in the case of sequences, attributes names and literal texts do not take part of gold summaries. The same situation occurs with variables names for classes and packages. The principal terms' locations for each artifact are presented in Table 7.

Since they represent the core of both types of abstracts, the gold standard summaries obtained in the experimental study represent the main target of the automatic summarizer we aim to achieve. Moreover, they are suitable to assess its results through intrinsic evaluation measures [4].

Table 7: Properties of gold standard, term-based summaries by artifact

Artifact	Length	Terms' origins	% of overlap with SBS
M1	6	Var. name, Method name, Par. name	100.00
M2	6	Method name, Var. name, Call name	50.00
S1	11	Call name, Var. name, Method name	81.82
S2	10	Call name, Var. name, Par. name	90.00
C1	15	Method name, Attrib. name, Class name	73.33
C2	7	Call name, Class name, Attrib. name, Attrib. type	100.00
P1	12	Attrib. name, Method name, Class name, Par. name	91.67

SBS Sentence-based summaries

4.5 Evaluating automatically generated summaries

Usually in text processing, the quality of summaries' content is determined by comparing the *peer summary* (i.e., the summary to be evaluated), with an ideal summary (i.e., a gold standard summary). In the specific case of extractive summaries, the primary metrics to perform such task are *precision* and *recall.* These metrics are based on the relevant content of the summary, and are defined as following:

$$precision = \frac{|summary_{gold} \cap summary_{peer}|}{|summary_{peer}|}$$

$$recall = \frac{|summary_{gold} \cap summary_{peer}|}{|summary_{gold}|}$$

The range of both metrics is $[0, 1]$. A precision value equal to one means that all the terms in the peer summary are relevant, although there could be relevant terms missing. On the other hand, a recall value equal to one means that the peer summary contains all the relevant terms, though it could also contain some irrelevant terms. In general, the lower the length of the peer summary, the higher the precision; whereas, the higher this length, the higher the recall.

A third metric, called F-score, measures the balance between precision and recall:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

The highest value reached by this harmonic mean is an indicator of the best achievable combination of the metrics it involves.

4.5.1 Text Retrieval Techniques in Source Code Summarization

In [3], three techniques from Text Retrieval were proposed to summarize source code artifacts: the Vector Space Model (VSM), Latent Semantic Indexing (LSI), and a combination of VSM and lead summarization. This latter approach is based on the hypothesis that the first sentences of a document are a good summary of it.

The *Vector Space Model* is one of the most common algebraic models in Text Retrieval for representing text corpora. It assumes a corpus as a set of documents D, from which is extracted the set of terms T, i.e., the vocabulary. Then, it represents this corpus as a matrix $M_{|T| \times |D|}$, where the row i corresponds to the term $t_i \in T$, and the column j corresponds to the document $d_j \in D$. In this sense, the value in the cell i, j is the weight of the term v_i in the document d_j.

The basic weighting scheme is the Boolean-based, which assigns one to the cell $m_{i,j}$ if the term t_i occurs in the document d_j, or zero otherwise. Other weighting schemes consider local and global weights, i.e., the contribution of the term t_i to the document d_j and to the entire set of documents D. For example, the popular scheme *tf-idf* determines the weight of the term t_i by multiplying its frequency of occurrence in the document d_j, by its inverse document frequency, as following:

$$m_{i,j} = f(t_i, d_j) \times idf(t_i)$$

where

$$idf\ (t_i) = log \left(\frac{|D|}{|\{d : t_i \in d \wedge d \in D\}|} \right)$$

When summarizing source code, the documents are code artifacts such as methods or classes. The k terms with the highest weight in the vector d_j are the ones conforming the summary of the document. This k value is usually called *constant threshold*.

The *Latent Semantic Indexing* (LSI) is based on a dimensionally reduced version of the vector space produced by VSM, in order to recover the underlying semantic in the corpus. Therefore, LSI uses *Singular Value Decomposition* (SVD) to decompose the matrix M into the left and right singular matrices U and V (which represent the terms and documents, respectively), and a diagonal matrix of singular values Σ. Then, $M = U\Sigma V^*$, where V^* is the transpose of V. The dimensions of these matrices can be reduced by choosing the C first columns of U and V, and the highest C singular values in Σ, which leads to $M_C = U_C \Sigma_C V_C^*$, i.e., to the approximation of the matrix M.

The corpus representation produced by LSI allows to compute the similarity between terms and documents. The summary of the document $d_j \in D$ is formed by the k terms in T with highest cosine similarity with the vector of the document d_j.

According to the results of an informal evaluation, where humans assessed the output of some TR-based summarizers, in [3] it was concluded that the combination of lead summarization and VSM (from now on called *lead+VSM*) produces better summaries than LSI and VSM by itself. Broadly, the *lead summaries* consist of the first k terms that appear in the target documents. In the case of source code artifacts, these first terms often contain the artifact type and name, which are rarely found in the VSM summaries. Thus, the combined summaries contain complementary information from both techniques.

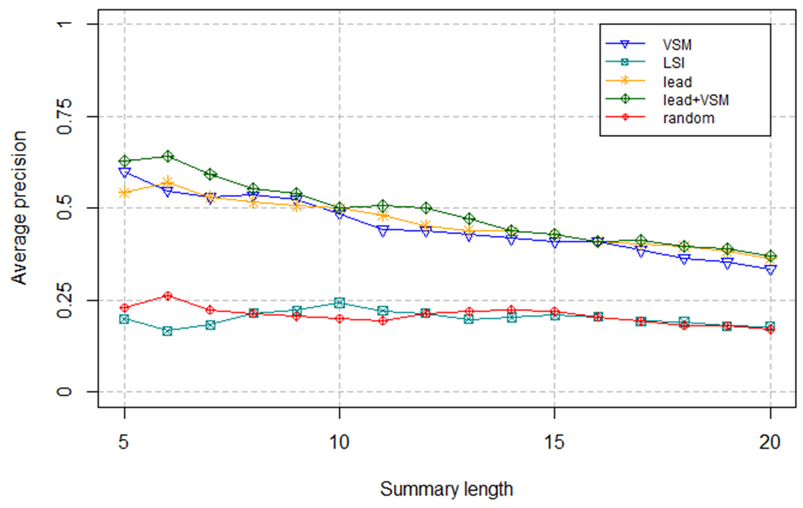

Figure 1: Average precision for VSM, LSI, lead+VSM, lead and random summaries. The x-axis represents the length of the summary, and the y-axis represents average precision values

4.5.2 Evaluating Text Retrieval Techniques through Intrinsic Measures

In order to evaluate the aforementioned techniques in software summarization, we computed the precision, recall and F-score metrics of their resulting summaries. For performing this task, we utilized the gold standard summaries described in section 4.4. Besides, we considered two baseline summarization methods, namely lead and *random*. This latter one generates summaries consisting of k terms randomly chosen from the target documents.

In [3], there are considered methods and classes summaries of length 5 and 10. In a different fashion, we evaluated summaries that vary its length from 5 to 20, since we were interested in analyze the influence of the length in the quality content metrics, and also, because we considered other kinds of code artifacts (i.e., sequences and packages). Thus, our objects of study were summaries composed by five to twenty terms, generated by VSM, LSI, lead+VSM, lead, and random methods, of each artifact described in Table 2.

For these summarization techniques, we observed that, as usual, the precision decreased as the recall and length of the summaries increased. However, in exceptional cases (artifacts M1, C2 and S2) there was an upward trend in the precision of LSI summaries, when increasing the length of the summaries. As expected, the precision of random summaries was low for all kind of artifacts and lengths, although in several cases LSI summaries had the lowest precision values, which confirms the results obtained in [3]. This fact was clearly observed in the artifacts C2 and P1, where LSI summaries had lower precision than random summaries. It was noticeable that in average, lead, VSM and lead+VSM had significantly higher precision and recall than random and LSI, no matter the kind of artifact that was being summarized. This fact can be observed in Fig. 1 and Fig. 2.

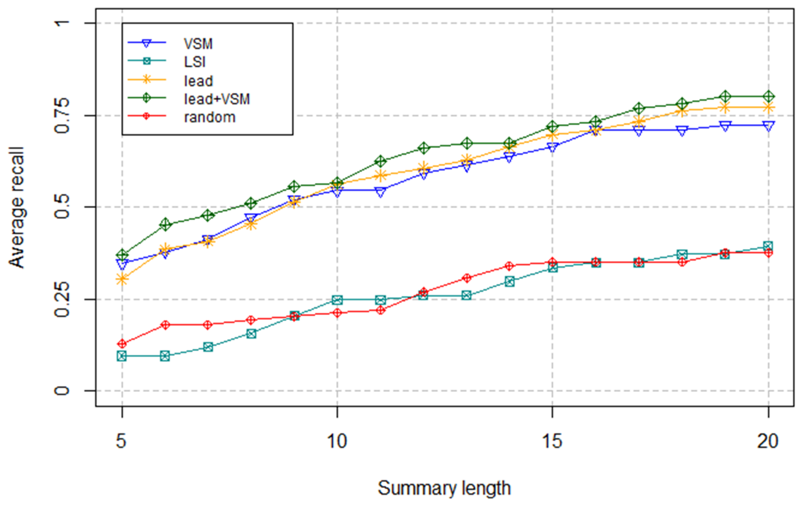

Figure 2: Average recall for VSM, LSI, lead+VSM, lead and random summaries. The x-axis represents the length of the summary, and the y-axis represents average recall values

When analyzing precision, we found that it was low for methods summaries having more than 10 terms, and for sequences and classes summaries having more than 15 terms. Considering the ranges where precision values were high, we found that in the method case, lead+VSM technique achieved the best results. This same technique together with lead got good precision values for sequences summaries. In the case of classes and packages, lead+VSM, lead and VSM summaries had similar precision, making it difficult to determine which technique was better for these kinds of artifacts. Furthermore, such techniques had a high precision (above 0.5), even for long summaries.

Regarding recall values, once again lead+VSM, lead and VSM outperformed LSI and random, and in some cases, random summaries achieved higher recall than LSI summaries (e.g., for artifacts C2 and P1). In addition, for every summarization technique, the recall values remained constant for the summaries consisting of more than 15 terms, with few exceptions, such as the lead+VSM summaries of the artifacts C1, S1 and P1, which continue increasing when $k > 15$.

All these results suggest that in order to get the gist of source code artifacts automatically, the length of a term-based summary should be in the interval $[10, 20]$. For methods, the number of terms in the summary is nearer to the lower bound (10), while for packages, this number is nearer to the upper bound (20). This means that automatic summaries are approximate 25% longer than the gold standard summaries, which is not an issue if they capture the intent of the code and remain shorter than the artifact they describe. These results were confirmed by the F-score values, which presented acceptable and stable values in the range $[10, 20]$ for all kinds of artifacts when the summaries were generated by lead+VSM, lead and VSM techniques. The average F-score values are presented in Fig. 3.

Additionally, the intrinsic evaluation also showed that LSI based on tf-idf and random are not appropriate techniques to summarize source code artifacts, which confirms the results in [3]. Although in average lead+VSM outperformed lead and VSM, none of these techniques is specially suitable or unsuitable for summarizing an specific kind of artifact. In fact, the performance of these three techniques is similar in all cases.

4.6 Threats to validity

As in any empirical study in software engineering, we cannot generalize the outcomes. Therefore, we consider these results only as useful heuristics to guide the development of automated summarization and documentation tools.

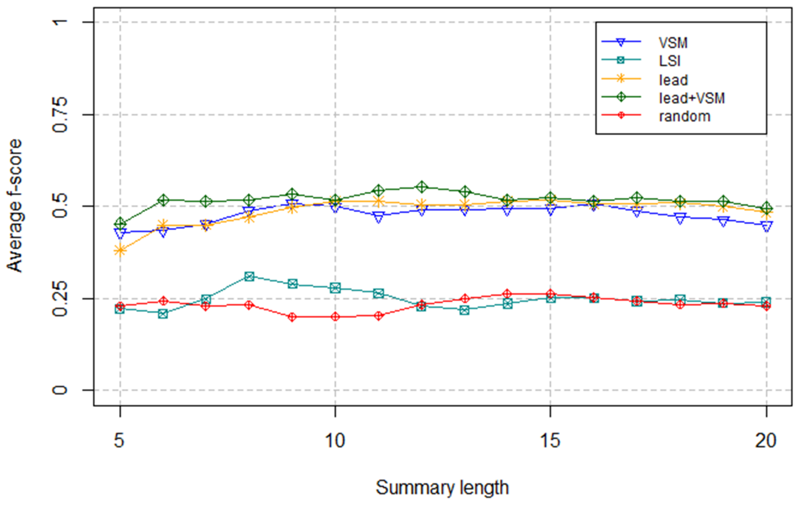

Figure 3: Average f-score for VSM, LSI, lead+VSM, lead and random summaries. The x-axis represents the length of the summary, and the y-axis represents average f-score values

The number of participants is always an issue for this type of experiment and in our case, nine developers is clearly a small group. Moreover, although subjects reported some experience in programming and evolving systems, they cannot be considered as professional developers. We plan to work with other research groups, and perform similar but larger studies involving more experienced subjects in order to gain more confidence in the results.

Equally important, we selected only two methods, two method sequences, two classes and one package from a single system. While we tried to vary their properties, they may not necessarily be the most representative of each type of artifact. Moreover, aTunes system has high quality, self-explanatory identifiers, and a very simple and clear domain. Therefore, we cannot estimate what would be the results for systems with poor identifier naming or a more complex domain.

During the summarization sessions carried out, developers had to study several times the same artifacts and write three different types of summaries for each of them. In consequence, the presence of a learning effect is possible. We did not try to measure or mitigate it.

5 Related Work

The automatic summarization of natural language text has been widely investigated by researchers and many approaches have been proposed, which are based mostly on Text Retrieval (TR), machine learning, and natural language processing techniques [1]. The summarization of software artifacts is only at the beginning, but there are promising results. For instance, abstracts of bug report discussions, generated using conversation-based classifiers, were proposed as a suitable instrument during bug report triage activities [7]; the summarization of the content of large execution traces was suggested as a tool that can help programmers to understand the main behavioral aspects of a software system [8].

Regarding automatic summarization of source code, in [9] it was proposed an abbreviated and accurate description of the effect of a software change on the run time behavior of a program, in order to help developers validating software changes and understanding modifications. High level descriptions of software concerns were designed for raising the level of abstraction and improving the productivity of developers, while working on evolution tasks [10]. The text retrieval based approaches for source code summarization, first introduced in [6], were applied for summarizing whole source code artifacts, with the purpose of aiding developers in comprehension tasks [3]. A form of structural summarization of source code has also been proposed in [11], which presented two techniques, i.e., the software reflection model and the lexical source model extraction for a lightweight summarization of software. These two techniques are complementary to the approaches we investigated and we envision combining them in the near future.

By the same token, some TR techniques are used in [12] to cluster source code and relevant terms from each cluster are extracted to form labels. A similar approach is used in [13], where TR is used to extract the most relevant set of terms to a group of methods returned as the result of a search. These terms are treated as attributes, which are used to cluster the methods. In each case, the labels and attributes can be considered as (partial) summaries.

Another related research thread is on source code tagging and annotations [14]. These mechanisms could support developers to create and represent manual summaries of the code (in addition to comments).

Although several alternatives have been explored to summarize various types of software artifacts, the evaluation of the generated summaries has been mostly informal. For example, [15] presents an approach to summarize methods by identifying and lexicalizing the most relevant units. The generated summaries in this case were evaluated by asking developers how much accurate, adequate and concise those descriptions were.

An exception to this informal situation is [7], where bug reports summaries were evaluated by using intrinsic measures such as precision, recall, F-score and pyramid precision, to assess the informativeness, redundancy, irrelevant content and coherence. Then, these results were compared against scores assigned by human judges to the same features. However, from a practical point of view, this study is considered as text-summarization, and therefore, its evaluation mode is not really novel.

In that sense, the term-based summaries generated by [6] from source code using information-retrieval techniques were evaluated using the Pyramid method. Also, the descriptions of source code produced in [3] underwent intrinsic-online evaluation for assessing the agreement between developers.

6 Conclusions and Future Work

The presented case study analyzed two kinds of summaries created by Java developers for several source code entities, with the purpose of studying how programmers create descriptions of source code. Besides, we asked developers to provide answers to questions about what they think should be included in a summary.

When developers create natural language descriptions of source code artifacts, the length is similar for all types of entities. We obtained slightly longer summaries for the case of sequences of calling methods. This result may indicate that this kind of artifact is harder to describe or deserves more detailed explanations.

On the other hand, the length of a term-based summary is correlated with the length of the artifact it summarizes. This result suggests that term-based summaries (extractive summaries) are inherently less informative than sentence-based summaries, and therefore, they are not enough to fully describe source code artifacts. Such fact is corroborated by the low percentage of words used in sentence-based summaries that correspond to terms selected within term-based summaries.

Consequently, despite textual information is essential, automatic code summarizers cannot exclusively rely on the identification of relevant terms contained in software entities. The precision, recall, and F-score values achieved by some TR-based techniques show that the semantic information by itself is not enough to generate high-quality code summaries. However, the outcomes of these techniques can be considered a good starting point for source code summarization, and they can be improved using structural information and natural language processing tools.

The experiment also gave us clues about what should be included in a summary. For instance, local variable names can be considered as useful pieces of information for describing all types of entities; names and invoked method names are quite relevant for summarizing methods; invoked method names and variable names are relevant for explaining sequences of calling methods; the name of a class is essential for describing its purpose. The results also suggest that summarization of packages is often problematic. This could indicate that packages cannot be considered as units, and in consequence, a multi-document approach, where a package would be conceived as a group of related documents (i.e., classes), is more appropriate.

Overall, the results obtained represent valuable information for building and evaluating automatic summarization tools. The gold-standard summaries characterize the main target of our envisioned summarizer, which will consider structural and textual information of artifacts. Since the text retrieval methods studied in this opportunity achieved only acceptable results, we plan to investigate and apply other text retrieval techniques in code summarization, and some multi-document summarization approaches for large artifacts as packages. Moreover, new user studies will be conducted to assess the effect of summaries on several development and maintenance tasks.

Acknowledgment

We express our gratitude to Andrian Marcus and Sonia Haiduc, members of the SEVERE[1] research group at Wayne State University, for their support and advice. Also, a special thanks to the students at Universidad Nacional de Colombia who are participating as Java developers within the user studies related to source code summarization research.

[1]http://www.cs.wayne.edu/~severe/

References

[1] K. S. Jones, "Automatic summarising: The state of the art," *Information Processing and Management: an International Journal*, vol. 43, no. 6, 2007.

[2] T. A. Corbi, "Program understanding: challenge for the 1990's," *IBM Systems Journal*, vol. 28, pp. 294–306, June 1989.

[3] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *WCRE '10: Proceedings of the 2010 17th Working Conference on Reverse Engineering.* Washington, DC, USA: IEEE Computer Society, 2010.

[4] J. Steinberger and K. Jeek, "Text summarization: An old challenge and new approaches," in *Foundations of Computational Intelligence*, ser. Studies in Computational Intelligence, A. Abraham, A.-E. Hassanien, D. Leon, and V. Snáel, Eds. Springer Berlin / Heidelberg, 2009, vol. 206, pp. 127–149.

[5] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 22, no. 140, pp. 1–55, 1932.

[6] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 2. New York, NY, USA: ACM, 2010, pp. 223–226.

[7] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering.* New York, NY, USA: ACM, 2010, pp. 505–514.

[8] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension.* Washington, DC, USA: IEEE Computer Society, 2006.

[9] R. Buse and W. R. Weimer, "Automatically documenting program changes," in *ASE '10: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering.* New York, NY, USA: ACM, 2010, pp. 33–42.

[10] S. Rastkar, "Summarizing software concerns," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering.* New York, NY, USA: ACM, 2010, pp. 527–528.

[11] G. C. Murphy, "Lightweight structural summarization as an aid to software evolution," Ph.D. dissertation, University of Washington, Washington, DC, USA, 1996.

[12] A. Kuhn, S. Ducasse, and T. G'ırba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[13] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–48.

[14] M. A. Storey, L. T. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer Supported Cooperative Work.* New York, NY, USA: ACM, 2006, pp. 195–198.

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